Solution to Problem 15-4

This solution is also posted publicly

Note: We assume that no word is longer than will fit into a line, i.e., $l_i \le M$ for all i.

First, we'll make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a sequence of words fits in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

- Define $extras[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ to be the number of extra spaces at the end of a line containing words i through j. Note that *extras* may be negative.
- Now define the cost of including a line containing words i through j in the sum we want to minimize:

$$lc[i, j] = \begin{cases} \infty & \text{if } extras[i, j] < 0 \text{ (i.e., words } i, \ldots, j \text{ don't fit)}, \\ 0 & \text{if } j = n \text{ and } extras[i, j] \ge 0 \text{ (last line costs 0)}, \\ (extras[i, j])^3 & \text{otherwise}. \end{cases}$$

By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of lc over all lines of the paragraph.

Our subproblems are how to optimally arrange words $1, \ldots, j$, where $j = 1, \ldots, n$.

Consider an optimal arrangement of words $1, \ldots, j$. Suppose we know that the last line, which ends in word j, begins with word i. The preceding lines, therefore, contain words $1, \ldots, i-1$. In fact, they must contain an optimal arrangement of words $1, \ldots, i-1$. (The usual type of cut-and-paste argument applies.)

Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains words $i, \ldots, j$, then $c[j] = c[i-1] + lc[i, j]$. As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = lc[1, 1]$, which is what we want.

But of course we have to figure out which word begins the last line for the subproblem of words $1, \ldots, j$. So we try all possibilities for word i, and we pick the one that gives the lowest cost. Here, i ranges from 1 to j. Thus, we can define $c[j]$ recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0\,, \\ \min\limits_{1 \le i \le j} (c[i-1] + lc[i, j]) & \text{if } j > 0\,. \end{cases}$$

Note that the way we defined lc ensures that

- all choices made will fit on the line (since an arrangement with $lc = \infty$ cannot be chosen as the minimum), and
- the cost of putting words $i, \ldots, j$ on the last line will not be 0 unless this really is the last line of the paragraph ($j = n$) or words $i \ldots j$ fill the entire line.

We can compute a table of c values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel p table that points to where each c value came from. When $c[j]$ is computed, if $c[j]$ is based on the value of $c[k-1]$, set $p[j] = k$. Then after $c[n]$ is computed, we can trace the pointers to see where to break the lines. The last line starts at word $p[n]$ and goes through word n. The previous line starts at word $p[p[n]]$ and goes through word $p[n] - 1$, etc.

In pseudocode, here's how we construct the tables:

```
PRINT-NEATLY(l, n, M)
let extras[1..n, 1..n], lc[1..n, 1..n], and c[0..n] be new arrays
// Compute extras[i, j] for 1 ≤ i ≤ j ≤ n.
for i = 1 to n
    extras[i, i] = M − l_i
    for j = i + 1 to n
        extras[i, j] = extras[i, j − 1] − l_j − 1
// Compute lc[i, j] for 1 ≤ i ≤ j ≤ n.
for i = 1 to n
    for j = i to n
        if extras[i, j] < 0
            lc[i, j] = ∞
        elseif j == n and extras[i, j] ≥ 0
            lc[i, j] = 0
        else lc[i, j] = (extras[i, j])^3
// Compute c[j] and p[j] for 1 ≤ j ≤ n.
c[0] = 0
for j = 1 to n
    c[j] = ∞
    for i = 1 to j
        if c[i − 1] + lc[i, j] < c[j]
            c[j] = c[i − 1] + lc[i, j]
            p[j] = i
return c and p
```

Quite clearly, both the time and space are $\Theta(n^2)$.

In fact, we can do a bit better: we can get both the time and space down to $\Theta(nM)$. The key observation is that at most $\lceil M/2 \rceil$ words can fit on a line. (Each word is at least one character long, and there's a space between words.) Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i, j] = \infty$. We need only compute and store $extras[i, j]$ and $lc[i, j]$ for $j - i + 1 \le \lceil M/2 \rceil$. And the inner **for** loop header in the computation of $c[j]$ and $p[j]$ can run from $\max(1, j - \lceil M/2 \rceil + 1)$ to j.

We can reduce the space even further to $\Theta(n)$. We do so by not storing the *lc* and *extras* tables, and instead computing the value of $lc[i, j]$ as needed in the last loop. The idea is that we could compute $lc[i, j]$ in $O(1)$ time if we knew the value of $extras[i, j]$. And if we scan for the minimum value in *descending* order of i, we can compute that as $extras[i, j] = extras[i + 1, j] - l_i - 1$. (Initially, $extras[j, j] = M - l_j$.) This improvement reduces the space to $\Theta(n)$, since now the only tables we store are c and p.

Here's how we print which words are on which line. The printed output of GIVE-LINES(p, j) is a sequence of triples (k, i, j), indicating that words $i, \ldots, j$ are printed on line k. The return value is the line number k.

```
GIVE-LINES(p, j)
i = p[j]
if i == 1
    k = 1
else k = GIVE-LINES(p, i − 1) + 1
print (k, i, j)
return k
```

The initial call is GIVE-LINES(p, n). Since the value of j decreases in each recursive call, GIVE-LINES takes a total of $O(n)$ time.

Solution to Problem 15-12

Let $p.cost$ denote the cost and $p.vorp$ denote the VORP of player p. We shall assume that all dollar amounts are expressed in units of \$100,000.

Since the order of choosing players for the positions does not matter, we may assume that we make our decisions starting from position 1, moving toward position N. For each position, we decide to either sign one player or sign no players. Suppose we decide to sign player p, who plays position 1. Then, we are left with an amount of $X - p.cost$ dollars to sign players at positions $2, \ldots, N$. This observation guides us in how to frame the subproblems.

We define the cost and VORP of a *set* of players as the sum of costs and the sum of VORPs of all players in that set. Let (i, x) denote the following subproblem: "Suppose we consider only positions $i, i+1, \ldots, N$ and we can spend at most x dollars. What set of players (with at most one player for each position under consideration) has the maximum VORP?" A *valid* set of players for (i, x) is one in which each player in the set plays one of the positions $i, i+1, \ldots, n$, each position has at most one player, and the cost of the players in the set is at most x dollars. An *optimal* set of players for (i, x) is a valid set with the maximum VORP. We now show that the problem exhibits optimal substructure.

Theorem (Optimal substructure of the VORP maximization problem)
Let $L = \{p_1, p_2, \ldots, p_k\}$ be a set of players, possibly empty, with maximum VORP for the subproblem (i, x).

1. If $i = N$, then L has at most one player. If all players in position N have cost more than x, then L has no players. Otherwise, $L = \{p_1\}$, where p_1 has the maximum VORP among players for position N with cost at most x.
2. If $i < N$ and L includes player p for position i, then $L' = L - \{p\}$ is an optimal set for the subproblem $(i + 1, x - p.cost)$.
3. If $i < N$ and L does not include a player for position i, then L is an optimal set for the subproblem $(i + 1, x)$.

Proof Property (1) follows trivially from the problem statement.

(2) Suppose that L' is not an optimal set for the subproblem $(i+1, x - p.cost)$. Then, there exists another valid set L'' for $(i+1, x - p.cost)$ that has VORP more than L'. Let $L''' = L'' \cup \{p\}$. The cost of L''' is at most x, since L'' has a cost at most $x - p.cost$. Moreover, L''' has at most one player for each position $i, i+1, \ldots, N$. Thus, L''' is a valid set for (i, x). But L''' has VORP more than L, thus contradicting the assumption that L had the maximum VORP for (i, x).

(3) Clearly, any valid set for $(i+1, x)$ is also a valid set for (i, x). If L were not an optimal set for $(i+1, x)$, then there exists another valid set L' for $(i+1, x)$ with VORP more than L. The set L' would also be a valid set for (i, x), which contradicts the assumption that L had the maximum VORP for (i, x). ■

The theorem suggests that when $i < N$, we examine two subproblems and choose the better of the two. Let $v[i, x]$ denote the maximum VORP for (i, x). Let $S(i, x)$ be the set of players who play position i and cost at most x. In the following recurrence for $v[i, x]$, we assume that the max function returns $-\infty$ when invoked over an empty set:

$$v[i, x] = \begin{cases} \max\limits_{p \in S(N,x)} \{p.vorp\} & \text{if } i = N\ , \\ \max\Big\{ v[i+1, x], & \\ \qquad \max\limits_{p \in S(i,x)} \{p.vorp + v[i+1, x - p.cost]\} \Big\} & \text{if } i < N\ . \end{cases}$$

This recurrence lends itself to implementation in a straightforward way. Let p_{ij} denote the jth player who plays position i.

```
FREE-AGENT-VORP(p, N, P, X)
let v[1..N][0..X] and who[1..N][0..X] be new tables
for x = 0 to X
    v[N, x] = −∞
    who[N, x] = 0
    for k = 1 to P
        if p_Nk.cost ≤ x and p_Nk.vorp > v[N, x]
            v[N, x] = p_Nk.vorp
            who[N, x] = k
for i = N − 1 downto 1
    for x = 0 to X
        v[i, x] = v[i + 1, x]
        who[i, x] = 0
        for k = 1 to P
            if p_ik.cost ≤ x and v[i + 1, x − p_ik.cost] + p_ik.vorp > v[i, x]
                v[i, x] = v[i + 1, x − p_ik.cost] + p_ik.vorp
                who[i, x] = k
print "The maximum value of VORP is " v[1, X]
amt = X
for i = 1 to N
    k = who[i, amt]
    if k ≠ 0
        print "sign player " p_ik
        amt = amt − p_ik.cost
print "The total money spent is " X − amt
```

The input to FREE-AGENT-VORP is the list of players p and N, P, and X, as given in the problem. The table $v[i, x]$ holds the maximum VORP for the subproblem (i, x). The table $who[i, x]$ holds information necessary to reconstruct the actual solution. Specifically, $who[i, x]$ holds the index of player to sign for position i, or 0 if no player should be signed for position i. The first set of nested **for** loops initializes the base cases, in which $i = N$. For every amount x, the inner loop simply picks the player with the highest VORP who plays position N and whose cost is at most x.

The next set of three nested **for** loops represents the main computation. The outermost **for** loop runs down from position $N - 1$ to 1. This order ensures that smaller subproblems are solved before larger ones. We initialize $v[i, x]$ as $v[i + 1, x]$. This way, we already take care of the case in which we decide not to sign any player who plays position i. The innermost **for** loop tries to sign each player (if we have enough money) in turn, and it keeps track of the maximum VORP possible.

The maximum VORP for the entire problem ends up in $v[1, X]$. The final **for** loop uses the information in *who* table to print out which players to sign. The running time of FREE-AGENT-VORP is clearly $\Theta(NPX)$, and it uses $\Theta(NX)$ space.