PRINT-TOUR(r, n)
print \( p_n \)
print \( p_{n-1} \)
\( k = r[n-1,n] \)
PRINT-PATH(r, k, n - 1)
print \( p_k \)

PRINT-PATH(r, i, j)
if \( i < j \)
\( k = r[i,j] \)
if \( k \neq i \)
print \( p_k \)
if \( k > 1 \)
PRINT-PATH(r, i, k)
else \( k = r[j,i] \)
if \( k > 1 \)
PRINT-PATH(r, k, j)
print \( p_k \)

The relative values of the parameters \( i \) and \( j \) in each call of PRINT-PATH indicate which subpath we’re working on. If \( i < j \), we’re on the right-to-left subpath, and if \( i > j \), we’re on the left-to-right subpath. The test for \( k \neq i \) prevents us from printing \( p_1 \) an extra time, which could occur when we call PRINT-PATH(r, 1, 2).

The time to run EUCLIDEAN-TSP is \( O(n^2) \) since the outer loop on \( j \) iterates \( n - 2 \) times and the inner loops on \( i \) and \( k \) each run at most \( n - 2 \) times. The sorting step at the beginning takes \( O(n \lg n) \) time, which the loop times dominate. The time to run PRINT-TOUR is \( O(n) \), since each point is printed just once.

---

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

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

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

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 \(\text{extras}[1 \ldots n, 1 \ldots n]\), \(\text{lc}[1 \ldots n, 1 \ldots n]\), and \(c[0 \ldots n]\) be new arrays

// Compute \(\text{extras}[i, j]\) for \(1 \leq i \leq j \leq n\).

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 \(\text{lc}[i, j]\) for \(1 \leq i \leq j \leq n\).

for \(i = 1\) to \(n\)
  for \(j = i\) to \(n\)
    if \(\text{extras}[i, j] < 0\)
      \(\text{lc}[i, j] = \infty\)
    elif \(j = n\) and \(\text{extras}[i, j] \geq 0\)
      \(\text{lc}[i, j] = 0\)
    else \(\text{lc}[i, j] = (\text{extras}[i, j])^3\)

// Compute \(c[j]\) and \(p[j]\) for \(1 \leq j \leq n\).

\(c[0] = 0\)

for \(j = 1\) to \(n\)
  \(c[j] = \infty\)
  for \(i = 1\) to \(j\)
    if \(c[i - 1] + \text{lc}[i, j] < c[j]\)
      \(c[j] = c[i - 1] + \text{lc}[i, j]\)
      \(p[j] = i\)

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 \(\text{lc}[i, j] = \infty\). We need only compute and store \(\text{extras}[i, j]\) and \(\text{lc}[i, j]\) for \(j - i + 1 \leq \lceil M/2 \rceil\). 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 \(\text{lc}\) and \(\text{extras}\) tables, and instead computing the value of \(\text{lc}[i, j]\) as needed in the last loop. The idea is that we could compute \(\text{lc}[i, j]\) in \(O(1)\) time if we knew the value of \(\text{extras}[i, j]\). And if we scan for the minimum value in descending order of \(i\), we can compute that as \(\text{extras}[i, j] = \text{extras}[i + 1, j] - l_i - 1\). (Initially, \(\text{extras}[j, j] = M - l_j\).) 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 \textsc{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\((p, j)\)

\[
\begin{align*}
i & = p[j] \\
\text{if } i & = 1 \\
\quad k & = 1 \\
\text{else } k & = \text{GIVE-LINES}(p, i - 1) + 1 \\
\text{print } & (k, i, j) \\
\text{return } & k
\end{align*}
\]

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 15-5**

\(a.\) Dynamic programming is the ticket. This problem is slightly similar to the longest-common-subsequence problem. In fact, we’ll define the notational conveniences \(X_i\) and \(Y_j\) in the similar manner as we did for the LCS problem: \(X_i = x[1..i]\) and \(Y_j = y[1..j]\).

Our subproblems will be determining an optimal sequence of operations that converts \(X_i\) to \(Y_j\), for \(0 \leq i \leq m\) and \(0 \leq j \leq n\). We’ll call this the “\(X_i -> Y_j\) problem.” The original problem is the \(X_m -> Y_n\) problem.

Let’s suppose for the moment that we know what was the last operation used to convert \(X_i\) to \(Y_j\). There are six possibilities. We denote by \(c[i, j]\) the cost of an optimal solution to the \(X_i -> Y_j\) problem.

- If the last operation was a copy, then we must have had \(x[i] = y[j]\). The subproblem that remains is converting \(X_{i-1}\) to \(Y_{j-1}\). And an optimal solution to the \(X_i -> Y_j\) problem must include an optimal solution to the \(X_{i-1} -> Y_{j-1}\) problem. The cut-and-paste argument applies. Thus, assuming that the last operation was a copy, we have \(c[i, j] = c[i-1, j-1] + \text{cost(copy)}\).
- If it was a replace, then we must have had \(x[i] \neq y[j]\). (Here, we assume that we cannot replace a character with itself. It is a straightforward modification if we allow replacement of a character with itself.) We have the same optimal substructure argument as for copy, and assuming that the last operation was a replace, we have \(c[i, j] = c[i-1, j-1] + \text{cost(replace)}\).
- If it was a twiddle, then we must have had both \(x[i] = y[j-1]\) and \(x[i-1] = y[j]\), along with the implicit assumption that \(i, j \geq 2\). Now our subproblem is \(X_{i-2} -> Y_{j-2}\) and, assuming that the last operation was a twiddle, we have \(c[i, j] = c[i-2, j-2] + \text{cost(twiddle)}\).
- If it was a delete, then we have no restrictions on \(x\) or \(y\). Since we can view delete as removing a character from \(X_i\) and leaving \(Y_j\) alone, our subproblem is \(X_{i-1} -> Y_j\). Assuming that the last operation was a delete, we have \(c[i, j] = c[i-1, j] + \text{cost(delete)}\).
- If it was an insert, then we have no restrictions on \(x\) or \(y\). Our subproblem is \(X_i -> Y_{j-1}\). Assuming that the last operation was an insert, we have \(c[i, j] = c[i, j-1] + \text{cost(insert)}\).
The outer for loop of the next block of code runs down from month \( n - 1 \) to 1, thus ensuring that when we consider month \( k \), we have already solved the subproblems of month \( k + 1 \).

The next inner for loop iterates through all possible values of \( f \) as described. For every choice of \( f \) for a given month \( k \), the total cost of \( (k, s) \) is given by the cost of extra labor (if any) plus the cost of inventory (if there is a surplus) plus the cost of the subproblem \( (k + 1, s + f - d_k) \). This value is checked and updated.

Finally, the required answer is the answer to the subproblem \( (1, 0) \), which appears in \( \text{cost}[1,0] \). That is, it is the cheapest way to satisfy all the demands of months \( 1, \ldots, n \) when we start with a surplus of 0.

The running time of INVENTORY-PLANNING is clearly \( O(nD^2) \). The space requirement is \( O(nD) \). We can improve upon the space requirement by noting that we need only store the solution to subproblems of the next month. With this observation, we can construct an algorithm that uses \( O(n + D) \) space.

**Solution to Problem 15-12**

Let \( p.\text{cost} \) denote the cost and \( p.\text{vorp} \) 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 \( X - p.\text{cost} \) 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 \( (i + 1, x - p.\text{cost}) \).
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 \( x - p_.cost \). 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:

\[
v[i, x] = \begin{cases} 
\max_{p \in S(N, x)} \{ p_.vorp \} & \text{if } i = N , \\
\max \left\{ v[i + 1, x], \max_{p \in S(i, x)} \{ p_.vorp + v[i + 1, x - p_.cost] \} \right\} & \text{if } i < N .
\end{cases}
\]

This recurrence lends itself to implementation in a straightforward way. Let \( p_{ij} \) denote the \( j \)th player who plays position \( i \).
**Problem:**

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 \(X\)

\(v[N, x] = -\infty\)

\(who[N, x] = 0\)

for \(k = 1\) to \(P\)

if \(p_{Nk}.cost \leq x\) 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 \(p_{ik}.cost \leq x\) 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 \(k \neq 0\)

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 \(N, P, X\), 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.