Problem 14-1

14-1 Longest simple path in a directed acyclic graph

You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The ***weight*** of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What is the running time of your algorithm?

sol)

We will make use of the optimal substructure property of longest paths in *acyclic* graphs. Let u be some vertex of the graph. If $u = t$, then the longest path from u

to t has zero weight. If $u \neq t$, let p be a longest path from u to t. Path p has at least two vertices. Let v be the second vertex on the path. Let p' be the subpath of p from v to t (p' might be a zero-length path). That is, the path p looks like $u \to v \overset{p'}{\leadsto} t$.

We claim that p' is a longest path from v to t.

To prove the claim, we use a cut-and-paste argument. If p' were not a longest path, then there exists a longer path p'' from v to t. We could cut out p' and paste in p'' to produce a path $u \to v \overset{p''}{\leadsto} t$ which is longer than p, thus contradicting the assumption that p is a longest path from u to t.

It is important to note that the graph is *acyclic*. Because the graph is acyclic, path p'' cannot include the vertex u, for otherwise there would be a cycle of the form $u \to v \leadsto u$ in the graph. Thus, we can indeed use p'' to construct a longer path. The acyclicity requirement ensures that by pasting in path p'', the overall path is still a *simple* path (there is no cycle in the path). This difference between the cyclic and the acyclic case allows us to use dynamic programming to solve the acyclic case.

Let $dist[u]$ denote the weight of a longest path from u to t. The optimal substructure property allows us to write a recurrence for $dist[u]$ as

$$dist[u] = \begin{cases} 0 & \text{if } u = t \ , \\ \max\{w(u,v) + dist[v] : (u,v) \in E\} & \text{otherwise} \ . \end{cases}$$

This recurrence allows us to construct the following procedure:

```
LONGEST-PATH-AUX$(G, u, t, dist, next)$
  if $u == t$
      $dist[u] = 0$
      return $(dist, next)$
  elseif $next[u] \neq$ NIL
      return $(dist, next)$
  else for each vertex $v \in G.Adj[u]$
          $(dist, next) =$ LONGEST-PATH-AUX$(G, v, t, dist, next)$
          if $w(u,v) + dist[v] > dist[u]$
              $dist[u] = w(u,v) + dist[v]$
              $next[u] = v$
  return $(dist, next)$
```

(See Section 20.1 for an explanation of the notation $G.Adj[u]$.)

LONGEST-PATH-AUX is a memoized, recursive procedure, which returns the tuple $(dist, next)$. The array $dist$ is the memoized array that holds the solution to subproblems. That is, after the procedure returns, $dist[u]$ will hold the weight of a longest path from u to t. The array $next$ serves two purposes:

- It holds information necessary for printing out an actual path. Specifically, if u is a vertex on the longest path that the procedure found, then $next[u]$ is the next vertex on the path.
- The value in $next[u]$ is used to check whether the current subproblem has been solved earlier. A non-NIL value indicates that this subproblem has been solved earlier.

The first **if** condition checks for the base case $u = t$. The second **if** condition checks whether the current subproblem has already been solved. The **for** loop iterates over each adjacent edge (u, v) and updates the longest distance in $dist[u]$.

What is the running time of LONGEST-PATH-AUX? Each subproblem represented by a vertex u is solved at most once due to the memoization. For each vertex, its adjacent edges are examined. Thus, each edge is examined at most once, and the overall running time is $O(E)$. (Section 20.1 discusses how to achieve $O(E)$ time by representing the graph with adjacency lists.)

The PRINT-PATH procedure prints out the vertices in the path using information stored in the *next* array:

```
PRINT-PATH($s, t, next$)
  $u = s$
  print $u$
  **while** $u \neq t$
      print $next[u]$
      $u = next[u]$
```

The LONGEST-PATH-MAIN procedure is the main driver. It creates and initializes the *dist* and the *next* arrays. It then calls LONGEST-PATH-AUX to find a path and PRINT-PATH to print out the actual path.

```
LONGEST-PATH-MAIN($G, s, t$)
  $n = |G.V|$
  let $dist[1:n]$ and $next[1:n]$ be new arrays
  **for** $i = 1$ **to** $n$
      $dist[i] = -\infty$
      $next[i] =$ NIL
  $(dist, next) =$ LONGEST-PATH-AUX($G, s, t, dist, next$)
  **if** $dist[s] == -\infty$
      print "No path exists"
  **else** print "The weight of the longest path is " $dist[s]$
      PRINT-PATH($s, t, next$)
```

Initializating the *dist* and *next* arrays takes $O(V)$ time. Thus, the overall running time of LONGEST-PATH-MAIN is $O(V + E)$.

Alternative solution

We can also solve the problem using a bottom-up aproach. To do so, we need to ensure that we solve "smaller" subproblems before we solve "larger" ones. In our case, we can use a topological sort (see Section 20.4) to obtain a bottom-up procedure, imposing the required ordering on the vertices in $\Theta(V + E)$ time.

```
LONGEST-PATH-BOTTOM-UP$(G, s, t)$
  let $dist[1:n]$ and $next[1:n]$ be new arrays
  topologically sort the vertices of $G$
  for $i = 1$ to $|G.V|$
      $dist[i] = -\infty$
  $dist[s] = 0$
  for each $u$ in topological order, starting from $s$
      for each edge $(u, v) \in G.Adj[u]$
          if $dist[u] + w(u, v) > dist[v]$
              $dist[v] = dist[u] + w(u, v)$
              $next[u] = v$
  print "The longest distance is " $dist[t]$
  PRINT-PATH$(s, t, next)$
```

The running time of LONGEST-PATH-BOTTOM-UP is $\Theta(V + E)$.

Problem 14-2

14-2 Longest palindrome subsequence

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

sol)

We solve the longest palindrome subsequence (LPS) problem in a manner similar to how we compute the longest common subsequence in Section 14.4.

Step 1: Characterizing a longest palindrome subsequence

The LPS problem has an optimal-substructure property, where the subproblems correspond to pairs of indices, starting and ending, of the input sequence.

For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, we denote the subsequence starting at x_i and ending at x_j by $X_{ij} = \langle x_i, x_{i+1}, \ldots, x_j \rangle$.

Theorem (Optimal substructure of an LPS)
Let $X = \langle x_1, x_2, \ldots, x_n \rangle$ be the input sequence, and let $Z = \langle z_1, z_2, \ldots, z_m \rangle$ be any LPS of X.

1. If $n = 1$, then $m = 1$ and $z_1 = x_1$.
2. If $n = 2$ and $x_1 = x_2$, then $m = 2$ and $z_1 = z_2 = x_1 = x_2$.
3. If $n = 2$ and $x_1 \neq x_2$, then $m = 1$ and z_1 is equal to either x_1 or x_n.
4. If $n > 2$ and $x_1 = x_n$, then $m > 2$, $z_1 = z_m = x_1 = x_n$, and $Z_{2,m-1}$ is an LPS of $X_{2,n-1}$.
5. If $n > 2$, $x_1 \neq x_n$, and $z_1 \neq x_1$, then $Z_{1,m}$ is an LPS of $X_{2,n}$.
6. If $n > 2$, $x_1 \neq x_n$, and $z_m \neq x_n$, then $Z_{1,m}$ is an LPS of $X_{1,n-1}$.

Proof Properties (1), (2), and (3) follow trivially from the definition of LPS.

(4) If $n > 2$ and $x_1 = x_n$, then we can choose x_1 and x_n as the ends of Z and at least one more element of X as part of Z. Thus, it follows that $m > 2$. If $z_1 \neq x_1$, then we could append $x_1 = x_n$ to the ends of Z to obtain a palindrome subsequence of X with length $m + 2$, contradicting the supposition that Z is a

longest palindrome subsequence of X. Thus, we must have $z_1 = x_1$ $(= x_n = z_m)$. Now, $Z_{2,m-1}$ is a length-$(m-2)$ palindrome subsequence of $X_{2,n-1}$. We wish to show that it is an LPS. Suppose for the purpose of contradiction that there exists a palindrome subsequence W of $X_{2,n-1}$ with length greater than $m-2$. Then, appending $x_1 = x_n$ to the ends of W produces a palindrome subsequence of X whose length is greater than m, which is a contradiction.

(5) If $z_1 \neq x_1$, then Z is a palindrome subsequence of $X_{2,n}$. If there were a palindrome subsequence W of $X_{2,n}$ with length greater than m, then W would also be a palindrome subsequence of X, contradicting the assumption that Z is an LPS of X.

(6) The proof is symmetric to (2). ■

The way that the theorem characterizes longest palindrome subsequences tells us that an LPS of a sequence contains within it an LPS of a subsequence of the sequence. Thus, the LPS problem has an optimal-substructure property.

Step 2: A recursive solution

The theorem implies that we should examine either one or two subproblems when finding an LPS of $X = \langle x_1, x_2, \ldots, x_n \rangle$, depending on whether $x_1 = x_n$.

Let us define $p[i,j]$ to be the length of an LPS of the subsequence X_{ij}. If $i = j$, the LPS has length 1. If $j = i+1$, then the LPS has length either 1 or 2, depending on whether $x_i = x_j$. The optimal substructure of the LPS problem gives the following recursive formula:

$$p[i,j] = \begin{cases} 1 & \text{if } i = j \;, \\ 2 & \text{if } j = i+1 \text{ and } x_i = x_j \;, \\ 1 & \text{if } j = i+1 \text{ and } x_i \neq x_j \;, \\ p[i+1, j-1] + 2 & \text{if } j > i+1 \text{ and } x_i = x_j \;, \\ \max\{p[i, j-1], p[i+1, j]\} & \text{if } j > i+1 \text{ and } x_i \neq x_j \;. \end{cases}$$

Step 3: Computing the length of an LPS

The procedure LONGEST-PALINDROME takes a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ as input. The procedure fills table entries $p[i,i]$, where $1 \leq i \leq n$, and $p[i, i+1]$, where $1 \leq i \leq n-1$, as the base cases. It then starts filling entries $p[i,j]$, where $j > i+1$. The procedure fills the p table row by row, starting with row $n-2$ and moving toward row 1. (Rows $n-1$ and n are already filled as part of the base cases.) Within each row, the procedure fills the entries from left to right. The procedure also maintains the table $b[1:n, 1:n]$ to help us construct an optimal solution. Intuitively, $b[i,j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $p[i,j]$. The procedure returns the b and p tables. The entry $p[1,n]$ contains the length of an LPS of X. The running time of LONGEST-PALINDROME is clearly $\Theta(n^2)$.

```
LONGEST-PALINDROME$(X, n)$
  let $p[1:n, 1:n]$ and $b[1:n, 1:n]$ be new tables
  for $i = 1$ to $n - 1$
      $p[i, i] = 1$
      $j = i + 1$
      if $x_i == x_j$
          $p[i, j] = 2$
          $b[i, j] =$ "$\swarrow$"
      else $p[i, j] = 1$
          $b[i, j] =$ "$\downarrow$"
  $p[n, n] = 1$
  for $i = n - 2$ downto 1
      for $j = i + 2$ to $n$
          if $x_i == x_j$
              $p[i, j] = p[i + 1, j - 1] + 2$
              $b[i, j] =$ "$\swarrow$"
          elseif $p[i + 1, j] \geq p[i, j - 1]$
              $p[i, j] = p[i + 1, j]$
              $b[i, j] =$ "$\downarrow$"
          else $p[i, j] = p[i, j - 1]$
              $b[i, j] =$ "$\leftarrow$"
  return $p$ and $b$
```

Step 4: Constructing an LPS

The b table returned by LONGEST-PALINDROME enables us to quickly construct an LPS of $X = \langle x_1, x_2, \ldots, x_m \rangle$. We simply begin at $b[1, n]$ and trace through the table by following the arrows. A "$\swarrow$" in entry $b[i, j]$ means that $x_i = y_j$ are the first and last elements of the LPS that LONGEST-PALINDROME found. The following recursive procedure returns a sequence S that contains an LPS of X. The initial call is GENERATE-LPS$(b, X, 1, n, \langle\rangle)$, where $\langle\rangle$ denotes an empty sequence. Within the procedure, the symbol $\|$ denotes concatenation of a symbol and a sequence.

```
GENERATE-LPS$(b, X, i, j, S)$
  if $i > j$
      return $S$
  elseif $i == j$
      return $S \| x_i$
  elseif $b[i, j] ==$ "$\swarrow$"
      return $x_i \|$ GENERATE-LPS$(b, X, i + 1, j - 1, S) \| x_i$
  elseif $b[i, j] ==$ "$\downarrow$"
      return GENERATE-LPS$(b, X, i + 1, j, S)$
  else return GENERATE-LPS$(b, X, i, j - 1, S)$
```

Problem 14-8

14-8 Image compression by seam carving

Suppose that you are given a color picture consisting of an $m \times n$ array $A[1:m, 1:n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. You want to compress this picture slightly, by removing one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid incongruous visual effects, however, the pixels removed in two adjacent rows must lie in either the same column or adjacent columns. In this way, the pixels removed form a "seam" from the top row to the bottom row, where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, you are given a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Define the disruption measure of a seam as the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

sol)

a. Let us set up a recurrence for the number of valid seams as a function of m. Suppose we are in the process of carving out a seam row by row, starting from the first row. Let the last pixel carved out be $A[i, j]$. How many choices do we have for the pixel in row $i + 1$ such that the pixel continues the seam? If the last pixel $A[i, j]$ is in the first or last column ($j = 1$ or $j = n$), then there are two choices for the next pixel. When $j = 1$, the two choices for the next pixel are $A[i + 1, j]$ and $A[i + 1, j + 1]$. When $j = n$, the two choices for the next pixel are $A[i + 1, j - 1]$ and $A[i + 1, j]$. Otherwise—when the last pixel is not in the first or last column—there are three choices for the next pixel: $A[i+1, j-1]$, $A[i+1, j]$, and $A[i+1, j+1]$. Thus, for a general pixel $A[i, j]$, there are at least two possible choices for a pixel p in the next row such that p continues a seam ending in $A[i, j]$. Let $T(i)$ denote the number of possible seams from row 1 to row i. Then, we have $T(1) = n$ (since the seam can start at any column in row 1) and $T(i) \geq 2T(i - 1)$ for $i > 1$.

We guess that $T(i) \geq n2^{i-1}$, which we verify by direct substitution. We have $T(1) = n \geq n \cdot 2^0$, and for $i > 1$, we have

$$\begin{aligned} T(i) &\geq 2T(i-1) \\ &\geq 2 \cdot n2^{i-2} \\ &= n2^{i-1} . \end{aligned}$$

Thus, the total number $T(m)$ of seams is at least $n2^{m-1}$. We conclude that the number of seams grows at least exponentially in m.

b. As proved in the previous part, it is infeasible to systematically check every seam, since the number of possible seams grows exponentially.

The structure of the problem allows us to build the solution row by row. Consider a pixel $A[i, j]$. We ask the question: “If i were the first row of the picture, what is the minimum disruptive measure of seams that start with the pixel $A[i, j]$?”

Let S^* be a seam of minimum disruptive measure among all seams that start with pixel $A[i, j]$. Let $A[i + 1, p]$, where $p \in \{j - 1, j, j + 1\}$, be the pixel of S^* in the next row. Let S' be the sub-seam of S^* that starts with $A[i + 1, p]$. We claim that S' has the minimum disruptive measure among seams that start with $A[i + 1, p]$. Why? Suppose there exists another seam S'' that starts with $A[i + 1, p]$ and has disruptive measure less than that of S'. By using S'' as the sub-seam instead of S', we can obtain another seam that starts with $A[i, j]$ and has a disruptive measure which is less than that of S^*. Thus, we obtain a contradiction to our assumption that S^* is a seam of minimum disruptive measure.

Let $disr[i, j]$ be the value of the minimum disruptive measure among all seams that start with pixel $A[i, j]$. For row m, the seam with the minimum disruptive measure consists of just one point. We can now state a recurrence for $disr[i, j]$ as follows. In the base case, $disr[m, j] = d[m, j]$ for $j = 1, 2, \ldots, n$. In the recursive case, for $j = 1, 2, \ldots, n$,

$$disr[i, j] = d[i, j] + \min\{disr[i + 1, j + k] : k \in K\} \ ,$$

where the set K of index offsets is

$$K = \begin{cases} \{0, 1\} & \text{if } j = 1 \ , \\ \{-1, 0, 1\} & \text{if } 1 < j < n \ , \\ \{-1, 0\} & \text{if } j = n \ . \end{cases}$$

Since every seam has to start with a pixel of the first row, we simply find the minimum $disr[1, j]$ for pixels in the first row to obtain the minimum disruptive measure.

```
COMPRESS-IMAGE($d, m, n$)
  let $disr[1:m, 1:n]$ and $next[1:m, 1:n]$ be new tables
  for $j = 1$ to $n$
      $disr[m, j] = d[m, j]$
  for $i = m - 1$ downto 1
      for $j = 1$ to $n$
          if $j == 1$
              $low = 0$
          else $low = -1$
          if $j == n$
              $high = 0$
          else $high = 1$
          $min\text{-}neighbor\text{-}disruption = \infty$
          for $k = low$ to $high$
              if $disr[i + 1, j + k] < min\text{-}neighbor\text{-}disruption$
                  $min\text{-}neighbor\text{-}disruption = disr[i + 1, j + k]$
                  $next[i, j] = j + k$
          $disr[i, j] = min\text{-}neighbor\text{-}disruption + d[i, j]$
  $min\text{-}overall\text{-}disruption = \infty$
  $column = 1$
  for $j = 1$ to $n$
      if $disr[1, j] < min\text{-}overall\text{-}disruption$
          $min\text{-}overall\text{-}disruption = disr[1, j]$
          $column = j$
  print "The minimum value of the disruptive measure is "
          $min\text{-}overall\text{-}disruption$
  for $i = 1$ to $m$
      print "cut point at " $(i, column)$
      $column = next[i, column]$
```

The procedure COMPRESS-IMAGE is simply an implementation of this recurrence in a bottom-up fashion.

It first initializes the base cases, which are the cases when row $i = m$. The minimum disruptive measure for the base cases is simply $d[m, j]$ fpr column $j = 1, 2, \ldots, n$.

The next **for** loop runs down from $m - 1$ to 1. Thus, $disr[i + 1, j]$ is already available before computing $disr[i, j]$ for the pixels of row i.

The assignments to *low* and *high* allow the index offset k to range over the correct set K from above. The code sets *low* to 0 when $j = 1$ and to -1 when $j > 1$, and it sets *high* to 0 when $j = n$ and to 1 when $j < n$. The innermost **for** loop finds the minimum value of $disr[i + 1, j + k]$ for all $k \in K$. Then the code sets $disr[i, j]$ to this minimum value plus the disruption $d[i, j]$.

The *next* table is for reconstructing the actual seam. For a given pixel, it records which pixel was used as the next pixel. Specifically, for a pixel $A[i, j]$, if $next[i, j] = p$, where $p \in \{j - 1, j, j + 1\}$, then the next pixel of the seam is $A[i + 1, p]$.

The next **for** loop finds the minimum overall disruptive measure, which is over pixels in the first row. The procedure prints the minimum overall disruptive measure as the answer.

The rest of the code reconstructs the actual seam, using the information stored in the *next* array.

Noting that the innermost **for** loop runs over at most three values of k, we see that the running time of COMPRESS-IMAGE is $O(mn)$. The space requirement is also $O(mn)$.

Problem 14-12

14-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $\$X$ to spend on free agents. You are allowed to spend less than $\$X$, but the owner will fire you if you spend any more than $\$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.[10] Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

[10] Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate "positions," as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

To determine how valuable a player is going to be, you decide to use a sabermetric statistic[11] known as "WAR," or "wins above replacement." A player with a higher WAR is more valuable than a player with a lower WAR. It is not necessarily more expensive to sign a player with a higher WAR than a player with a lower WAR, because factors other than a player's value determine how much it costs to sign them.

For each available free-agent player p, you have three pieces of information:

- the player's position,
- $p.cost$, the amount of money it costs to sign the player, and
- $p.war$, the player's WAR.

Devise an algorithm that maximizes the total WAR of the players you sign while spending no more than $\$X$. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total WAR of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

sol)

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 WAR of a *set* of players as the sum of costs and the sum of WARs 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—yields the maximum WAR?" 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 WAR. We now show that the problem exhibits optimal substructure.

Theorem (Optimal substructure of the WAR maximization problem)
Let $L = \{p_1, p_2, \ldots, p_k\}$ be a set of players, possibly empty, with maximum WAR 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 WAR 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 WAR 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 $i, i+1, \dots, N$. Thus, L''' is a valid set for (i, x). But L''' has WAR more than L, thus contradicting the assumption that L had the maximum WAR 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 WAR more than L. The set L' would also be a valid set for (i, x), which contradicts the assumption that L had the maximum WAR for (i, x). ■

The theorem suggests that when $i < N$, we examine two subproblems and choose the better of the two. Let $w[i, x]$ denote the maximum WAR 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 $w[i, x]$, we assume that the max function returns $-\infty$ when invoked over an empty set:

$$w[i,x] = \begin{cases} \max\{p.war : p \in S(N,x)\} & \text{if } i = N\,, \\ \max\big\{w[i+1,x], & \\ \quad \max\{p.war + w[i+1, x - p.cost] : p \in S(i,x)\}\big\} & \text{if } i < N\,. \end{cases}$$

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

```
FREE-AGENT-WAR$(p, N, P, X)$
 let $w[1:N][0:X]$ and $who[1:N][0:X]$ be new tables
 for $x = 0$ to $X$
     $w[N, x] = -\infty$
     $who[N, x] = 0$
     for $k = 1$ to $P$
         if $p_{Nk}.cost \le x$ and $p_{Nk}.war > w[N, x]$
             $w[N, x] = p_{Nk}.war$
             $who[N, x] = k$
 for $i = N - 1$ downto 1
     for $x = 0$ to $X$
         $w[i, x] = w[i + 1, x]$
         $who[i, x] =$ NIL
         for $k = 1$ to $P$
             if $p_{ik}.cost \le x$ and $w[i + 1, x - p_{ik}.cost] + p_{ik}.war > w[i, x]$
                 $w[i, x] = w[i + 1, x - p_{ik}.cost] + p_{ik}.war$
                 $who[i, x] = k$
 print "The maximum value of WAR is " $w[1, X]$
 $spent = 0$
 for $i = 1$ to $N$
     $k = who[i, X - spent]$
     if $k \ne$ NIL
         print "sign player " $p_{ik}$
         $spent = spent + p_{ik}.cost$
 print "The total money spent is " $spent$
```

The input to FREE-AGENT-WAR is the list of players p and N, P, and X, as given in the problem. The table $w[i, x]$ holds the maximum WAR 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 the player to sign for position i, or NIL 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 WAR who plays position N and whose cost is at most x.

The next group 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. Initializing $w[i, x]$ as $w[i + 1, x]$ takes 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 enough money remains) in turn, and it keeps track of the maximum WAR possible.

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