Rubric (for all dynamic programming problems): For a problem worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - 1 point for a clear English description of the function
  - 1 point for base case(s).
  - 4 points for recursive case(s). −2 for one *minor* error, like a typo.
  - ★ **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 1 point for describing the memoization data structure
- 2 points for describing a correct evaluation order; a clear picture is sufficient for full credit.
- 1 point for analyzing the running time

Official solutions will always include pseudocode for the final dynamic programming algorithm, but this is ***not*** required for full credit. On the other hand, if correct pseudocode for the dynamic programming algorithm *is* included, it is not necessary to separately describe the recurrence, the memoization data structure, or the evaluation order.

The official solution will provide a target time bound. Algorithms faster than the official solution are worth extra credit, and algorithms slower than the official solution are worth fewer points, typically 2 or 3 points for each factor of n. For slower algorithms, partial credit is scaled to the lower maximum score. **Any correct algorithm, no matter how slow, is worth at least 2½ points**, assuming it is properly analyzed.

It is not necessary to state a space bound. There is no penalty for using more space than the official solution, but +1 extra credit for using less space to achieve the same (or better) running time.

This rubric also applies to dynamic programming problems on exams.

1. Suppose you are given an array $A[1..n]$ of positive integers. Describe and analyze an algorithm to find the *smallest* positive integer that is *not* an element of A in $O(n)$ time.

Solution (just do it): The following algorithm runs in $O(n)$ time, using $O(n)$ additional space. After the second for-loop, for any positive integer $i \le n$, the boolean $\mathit{Missing}[i]$ is equal to TRUE if and only if the integer i is not in the input array.

```
FirstMissing($A[1..n]$):
  for $i \leftarrow 1$ to $n$
      $\mathit{Missing}[i] \leftarrow$ TRUE
  for $i \leftarrow 1$ to $n$
      if $A[i] \le n$
          $\mathit{Missing}[A[i]] \leftarrow$ FALSE
  for $i \leftarrow 1$ to $n$
      if $\mathit{Missing}[i] =$ TRUE
          return $i$
  return $n+1$
```

We can actually reduce the space usage to $O(1)$ by using the sign of the integer $A[i]$ to encode the boolean $\mathit{Missing}[i]$. If we don't want the algorithm to change the array, we can easily reset all the signs to positive in a post-processing phase.

```
FirstMissing($A[1..n]$):
  for $i \leftarrow 1$ to $n$
      if $A[i] \le n$ and $A[A[i]] > 0$
          $A[A[i]] \leftarrow -A[A[i]]$
  for $i \leftarrow 1$ to $n$
      if $A[i] > 0$
          return $i$
  return $n+1$
```

■

Rubric: Max 10 points = 6 for algorithm + 2 for time analysis + 2 for proof of correctness. This is not the only correct solution. $O(1)$ space is not required for full credit. Leaving the input array unchanged is not necessary for full credit. Solutions that are correct only when the input integers to be distinct are still worth full credit ***if*** they explicitly assume that the input integers are distinct.

2. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..m, 1..n]$ whose entries are all 0 or 1. A *solid block* is a subarray of the form $M[i..i', j..j']$ in which every bit is equal to 1. Describe and analyze an efficient algorithm to find a solid block in M with maximum area.

Solution (canonical): Let's assume that $m > n$; otherwise, transpose the input array.

Let $Bottom(t, \ell, r)$ denote the maximum integer b such that $M[t..b, \ell..r]$ is a solid block; that is, the last possible row in a solid block with the given top row and left and right columns. $M[t..b, \ell..r]$ is a solid block if and only if $M[t, \ell] = 1$ and both $M[t+1..b, \ell..r]$ and $M[t..b, \ell+1..r]$ are solid blocks. Thus, we have the following recurrence:

$$Bottom(t, \ell, r) := \begin{cases} m & \text{if } \ell > r \text{ or } t > m \\ t - 1 & \text{if } M[t, \ell] = 0 \\ \min\{Bottom(t+1, \ell, r),\ Bottom(t, \ell+1, r)\} & \text{otherwise} \end{cases}$$

The first case considers solid blocks with zero columns or zero rows that end at the bottom row of the array. In the second case, there is a 0 in row t and column ℓ, and so the largest solid block has zero rows. We can memoize this function into a three-dimensional array $Bottom[1..m+1, 1..n+1, 1..n]$, which we can fill by considering the top rows t and left columns ℓ in decreasing order. We compute the area of each block $M[t..Bottom[t, \ell, r], \ell..r]$ in constant time as we fill in the array, and return the largest area found.

```
BIGGESTBLOCK($M[1..m, 1..n]$):
  $maxArea \leftarrow 0$
  for $r \leftarrow 1$ to $n$
      for $t \leftarrow m+1$ down to 1
          for $\ell \leftarrow n+1$ down to 1
              if $\ell > r$ or $t > m$
                  $Bottom[t, \ell, r] \leftarrow m$
              else if $M[t, \ell] = 0$
                  $Bottom[t, \ell, r] \leftarrow t - 1$
              else
                  $Bottom[t, \ell, r] \leftarrow \min\{Bottom[t+1, \ell, r],\ Bottom[t, \ell+1, r]\}$
              $maxArea \leftarrow \max\{maxArea, (Bottom[t, \ell, r] - t + 1) \cdot (r - \ell + 1)\}$
  return $maxArea$
```

The algorithm runs in ***$O(mn^2)$ time*** and uses $O(mn^2)$ space.

We can reduce the space to $O(mn)$ by keeping only a two-dimensional array indexed by t and ℓ, and then to $O(n)$ by maintaining a sliding window of two rows in this 2d array. The algorithm can be modified to return the actual largest block in the same time and space bounds (using Hirshberg's divide-and-conquer strategy). ■

Rubric: Max 10 points; standard dynamic programming rubric. This is not the only correct evaluation order. −½ if time analysis for reporting time as $O(n^3)$ instead of $O(mn^2)$ or $O(m^2n)$. Full credit for returning the *size* of the largest block instead of the block itself. Partial credit for slower algorithms: $O(n^4)$ is worth 7 points; $O(n^5)$ is worth 4 points; $O(n^6)$ or slower is worth 3 points. A correct $O(mn)$-time algorithm is worth +5 extra credit; yes, there is one.

3. Let T be a tree in which each edge e has a weight $w(e)$. A *matching* M in T is a subset of the edges such that each vertex of T is incident to at most one edge in M. The weight of a matching M is the sum of the weights of its edges. Describe and analyze an algorithm to compute a maximum weight matching, given the tree T as input.

Solution: let T denote the input tree, and let $w(e)$ denote the weight of any edge e. For any vertex v, let $MW(v)$ denote the weight of the maximum weight matching in the subtree rooted at v. To simplify notation, let $MWkids(v)$ denote the sum of $MW(u)$ over all children u of v.

$$MWkids(v) = \sum_{\text{child } u \text{ of } v} MW(u)$$

In particular, when v is a leaf, we have $MWkids(v) = 0$.

There are two possibilities; either the maximum weight matching in v's subtree uses no edges from v to a child of v, or it uses exactly one such edge. (If v is a leaf, only the first choice is actually possible.) In the first case, we have $MW(v) = MWkids(v)$. In the second case, if the maximum weight matching uses the edge uv, then it also includes the maximum weight matching in the subtrees of all v's children *except* w, and the maximum weight matching in the subtrees of all w's children. Thus, we have the following recurrence:

$$MW(v) = \max \left\{ \begin{array}{l} MWkids(v), \\ \displaystyle\max_{\text{child } u \text{ of } v} w(uv) + MWkids(v) - MMWV(u) + MWkids(u) \end{array} \right\}$$

Again, when v is a leaf, we vacuously have $MW(v) = 0$.

We can memoize this recurrence by recording both $MW(v)$ and $MWkids(v)$ in each node v; this requires $O(n)$ space. The data stored at each node depends only on the data stored at its children. Thus, we can then evaluate the recurrence in $\boldsymbol{O(n)}$ ***time*** using a postorder traversal of the tree. ■

Rubric: max 10 points: standard dynamic programming rubric.

4. Describe and analyze an algorithm that accepts three strings x, y, and z as input, and decides whether z is an interleaving of x and y.

Solution: Let the input strings be $X[0..\ell-1]$, $Y[0..m-1]$, and $Z[1..n]$. For any integer i, let $\boldsymbol{X \circlearrowleft i}$ denote the result of *rotating* X to the left by i characters:

$$X \circlearrowleft i \;:=\; X[i \bmod \ell \,..\, \ell-1] \cdot X[0 \,..\, i-1 \bmod \ell].$$

In particular, if i is a multiple of ℓ, then $X_i = X$. Define $Y \circlearrowleft j$ similarly for any integer j. For example, $\texttt{STRING} \circlearrowleft 4 = \texttt{STRING} \circlearrowleft 10 = \texttt{STRING} \circlearrowleft -2 = \texttt{NGSRTI}$. (0-indexing X and Y makes the modular arithmetic easier.)

We define a boolean function ***IsInt*(i, j, k)**, which is TRUE if and only if the suffix $Z[k..n]$ is an interleaving of the rotated strings $X \circlearrowleft i$ and $Y \circlearrowleft j$. This function obeys the following recurrence:

$$\mathit{IsInt}(i,j,k) := \begin{cases} \textsc{True} & \text{if } k > n \\ \big((Z[k] = X[i]) \wedge \mathit{IsInt}(i+1 \bmod \ell,\ j,\ k+1)\big) \\ \quad \vee\ \big((Z[k] = Y[j]) \wedge \mathit{IsInt}(i,\ j+1 \bmod m,\ k+1)\big) & \text{otherwise} \end{cases}$$

We can memorize this recurrence into an array $\mathit{IsInt}[0..\ell-1, 0..m-1, 1..n+1]$. Each entry $\mathit{IsInt}[i,j,k]$ depends only on entries of the form $\mathit{IsInt}[\cdot,\cdot,k+1]$, so we can fill the table from the bottom up, filling each 2d slice in any order we like.

```
INTERLEAVES($X[0..\ell-1], Y[0..m-1], Z[1..n]$):
  for $i \leftarrow 0$ to $\ell-1$
      for $j \leftarrow 1$ to $m-1$
          $\mathit{IsInt}[i,j,n+1] \leftarrow$ TRUE
  for $k \leftarrow n+1$ down to 1
      for $i \leftarrow 0$ to $\ell-1$
          for $j \leftarrow 1$ to $m-1$
              $\mathit{usex} \leftarrow (Z[k] = X[i]) \wedge \mathit{IsInt}[i+1 \bmod \ell,\ j,\ k+1]$
              $\mathit{usey} \leftarrow (Z[k] = Y[j]) \wedge \mathit{IsInt}[i, j+1 \bmod m,\ k+1]$
              $\mathit{IsInt}[i,j,n+1] \leftarrow \mathit{usex} \vee \mathit{usey}$
  return $\mathit{IsInt}(0,0,1)$
```

The algorithm runs in $\boldsymbol{O(n^3)}$ ***time*** and uses $O(n^3)$ space. The space can be reduced to $O(n^2)$ by keeping only two slices. ■

Rubric: max 10 points: standard dynamic programming rubric. This is not the only correct evaluation order.

5. Suppose there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward to be paid if snails i and j meet. Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

Solution: For any indices i and j, let $Reward(i,j)$ be the maximum possible reward if only snails numbered i through j find mates. We have the following recurrence:

$$Reward(i,j) = \begin{cases} 0 & \text{if } j \le i \\ \max \left\{ \begin{array}{l} Reward(i+1,j), \\ \max\limits_{i<k\le j} \big(M[i,k] + Reward(i+1,k-1) + Reward(k+1,j)\big) \end{array} \right\} & \text{otherwise} \end{cases}$$

If there is at most one relevant snail, no reward is possible. Otherwise, we recursively consider all ways of pairing up snail i. If snail i never finds a mate, the maximum reward is $Reward(i+1,j)$. If snail i meets snail k, the organizers immediately pay $M[i,k]$, and the two slime trails split the remaining snails into two independent subproblems: snails $i+1$ through $k-1$, and snails $k+1$ through j.

We can memoize this recurrence into an array $Reward[1..n, 0..n]$. Each element in the array depends only on elements in later rows. Thus, we can fill the array from the bottom up, filling each row in any order we like.

```
MaxReward($M[1..n, 1..n]$):
  for $i \leftarrow n$ down to 1
      $Reward[i, i-1] \leftarrow 0$
      $Reward[i, i] \leftarrow 0$
      for $j \leftarrow i+1$ to $n$
          $Reward[i,j] \leftarrow Reward[i+1,j]$
          for $k \leftarrow i+1$ to $j$
              $tmp \leftarrow M[i,k] + Reward[i+1,k-1] + Reward[k+1,j]$
              if $Reward[i,j] < tmp$
                  $Reward[i,j] \leftarrow tmp$
  return $Reward[1,n]$
```

The algorithm runs in $\boldsymbol{O(n^3)}$ ***time*** and $O(n^2)$ space. ■

Rubric: 10 points max: standard dynamic programming rubric. This is not the only correct evaluation order.