1. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to \( n \). During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, 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. Rewards may be positive, negative, or zero.

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:** Let \( \text{MaxR}(i, j) \) be the maximum possible reward if only the snails numbered \( i \) through \( j \) are allowed to find mates. We need to compute \( \text{MaxR}(1, n) \). This function obeys the following recurrence:

\[
\text{MaxR}(i, j) = \begin{cases} 
0 & \text{if } j \leq i \\
\max \left\{ \text{MaxR}(i + 1, j) \right\} & \text{otherwise}
\end{cases}
\]

\[
\text{MaxR}(i, j) = \max_{i < k \leq j} \left\{ M[i, k] + \text{MaxR}(i + 1, k - 1) + \text{MaxR}(k + 1, j) \right\}
\]

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 \( \text{MaxR}(i + 1, j) \). 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 function into a two-dimensional array \( \text{MaxR}[1..n, 0..n] \). Since each entry \( \text{MaxR}[i, j] \) depends only on entries \( \text{MaxR}[i', j'] \) with \( i' > i \), we can fill the array row by row from the bottom up (decreasing \( i \)). The resulting algorithm runs in \( O(n^3) \) time.

```python
MAXREWARD(M[1..n, 1..n]):
for i ← n down to 1
    MaxR[i, i-1] ← 0
    MaxR[i, i] ← 0
for j ← i + 1 to n
    MaxR[i, j] ← MaxR[i + 1, j]
    for k ← i + 1 to j
        tmp ← M[i, k] + MaxR[i + 1, k - 1] + MaxR[k + 1, j]
        if MaxR[i, j] < tmp
            MaxR[i, j] ← tmp
return MaxR[1, n]
```

**Rubric:** 10 points: standard dynamic programming rubric
2. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it’s his turn, Elmo always takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely hates it when grown-ups let him win.)

(a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do not follow the same greedy strategy as Elmo.

**Solution (full credit):** Suppose the initial cards are $[1, 2, 100, 3]$ and I play first.

Suppose my first move is to take the 1, which is not greedy. Then Elmo must take the 3, and then I take the 100 and win.

But if I play greedily, I must take the 3, and then Elmo takes the 100 and wins. ■

**Solution (full credit):** Suppose the initial cards are $[1, 2, 100, 3, 4]$ and Elmo plays first. Elmo will take the 4, leaving $[1, 2, 100, 3]$.

If my first move is to take the 1, which is not greedy, then Elmo must take the 3, and then I take the 100 and win. But if I play greedily, I must take the 3, and then Elmo takes the 100 and wins. ■

**Solution (extra credit):** Suppose the initial cards are $[5, 9, 2, 3, 0, 1, 4]$.

If I move first, I can win as follows: I take the 4 (not greedy); Elmo takes the 5; I take the 9; Elmo takes the 2; I take the 3; Elmo takes the 1; and I take the 0. I win 16–8.

If Elmo moves first, I can win as follows: Elmo takes the 5; I take the 9; Elmo takes the 4; I take the 1 (not greedy); Elmo takes the 2; I take the 3; and Elmo takes the 0. Elmo loses 11–13.

If Elmo and I both play greedily, the game ends in a tie, so I cannot win no matter who goes first. Player 1 takes the 5; player 2 takes the 9; player 1 takes the 4; player 2 takes the 2; player 1 takes the 3; player 2 takes the 1; finally, player 1 takes the 0. The final score is 12–12. ■

**Rubric:** 2 points = 1 for example + 1 for proof. +3 extra credit for an initial setup where optimal play beats Elmo no matter who goes first, but greedy-vs-greedy is a tie.
(b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

Solution: Suppose the initial cards are given in the array \( C[1..n] \). We define two functions:

- \( Me(i, j) \) is my maximum possible score if I go first, starting with the cards \( C[i..j] \)
- \( El(i, j) \) is my maximum possible score if Elmo goes first, starting with the cards \( C[i..j] \)

Depending on who starts the actual game, we need to compute either \( Me(1, n) \) or \( El(1, n) \). These two functions satisfy the following mutual recurrence:

\[
Me(i, j) = \begin{cases} 
0 & \text{if } j < i \text{ (no cards left)} \\
\max \left\{ C[i] + El(i+1, j), C[j] + El(i, j-1) \right\} & \text{otherwise}
\end{cases}
\]

\[
El(i, j) = \begin{cases} 
0 & \text{if } j < i \text{ (no cards left)} \\
Me(i+1, j) & \text{if } C[i] > C[j] \\
Me(i, j-1) & \text{otherwise}
\end{cases}
\]

We can memoize these functions into two two-dimensional arrays \( Me[1..n+1, 0..n] \) and \( El[1..n+1, 0..n] \). (We need row \( n+1 \) and column \( 0 \) to handle extreme base cases.) Each entry depends only on entries in the other table, immediately to the left or immediately below. Thus, we can fill both tables from the bottom up, each row left to right, alternating between \( Me \) and \( El \) at each step.

```plaintext
BeatKidElmo(C[1..n], mefirst):
    for i ← n + 1 downto 1
        Me[i, i - 1] ← 0
        El[i, i - 1] ← 0
    for j ← i + 1 to n
        Me[i, j] ← max \{ C[i] + El[i+1, j], C[j] + El[i, j-1] \}
        if C[i] > C[j]
            El[i, j] ← Me[i+1, j]
        else
            El[i, j] ← Me[i, j-1]
    if mefirst
        return Me[1, n]
    else
        return El[1, n]
```

The resulting algorithm runs in \( O(n^2) \) time.

Rubric: 4 points, standard dynamic programming rubric (scaled). This is not the only correct solution. An algorithm that assumes Elmo plays first, or that Elmo plays second, is sufficient for full credit.
(c) Five years later, Elmo has become a significantly stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a perfect opponent. [Hint: What is a perfect opponent?]

Solution (mutual recurrence): Suppose the initial cards are given in the array \( C[1..n] \). We define two functions:

- \( Me(i, j) \) is my maximum possible score if I go first, starting with the cards \( C[i..j] \)
- \( El(i, j) \) is my maximum possible score if Elmo goes first, starting with the cards \( C[i..j] \)

Depending on who starts the actual game, we need to compute either \( Me(1, n) \) or \( El(1, n) \). These two functions satisfy the following mutual recurrence:

\[
Me(i, j) = \begin{cases} 
0 & \text{if } j < i \text{ (no cards left)} \\
\max \left\{ C[i] + El(i + 1, j), C[j] + El(i, j - 1) \right\} & \text{otherwise}
\end{cases}
\]

\[
El(i, j) = \begin{cases} 
0 & \text{if } j < i \text{ (no cards left)} \\
\min \left\{ Me(i + 1, j), Me(i, j - 1) \right\} & \text{otherwise}
\end{cases}
\]

Again, we can memoize these functions into two two-dimensional arrays, which we can fill from the bottom up, each row left to right, alternating between \( Me \) and \( El \) at each step.

The resulting algorithm runs in \( O(n^2) \) time.

Solution (single recurrence for spread): Suppose the initial cards are given in the array \( C[1..n] \). We define a recursive function \( Spread(i, j) \), which is the maximum possible difference between my score and Elmo’s score, assuming I play first. I win if this difference is positive; Elmo wins if this difference is negative. Because the total value of the cards is fixed, maximizing my spread is the same as maximizing my score. Symmetrically, grown-up Elmo is trying to minimize the spread, which is the same as maximizing his score.

If I play first, we need to compute \( Spread(1, n) \). If I play second, we need to compute \( X - Spread(1, n) \), where \( X \) is the sum of the initial card values.
The `Spread` function obeys the following recurrence:

\[
Spread(i, j) = \begin{cases} 
0 & \text{if } i > j \text{ (no cards left)} \\
\max \left\{ C[i] - \text{Spread}(i + 1, j), C[j] - \text{Spread}(i, j - 1) \right\} & \text{otherwise}
\end{cases}
\]

We can memoize this function into a two-dimensional array, which we can fill bottom-up, each row left to right.

```
BEATGROWNUPELMO[C[1..n]]:
  for i ← n + 1 downto 1
    Spread[i, i - 1] ← 0
  for j ← i + 1 to n
    Spread[i, j] ← max \{C[i] − Spread(i + 1, j), C[j] − Spread(i, j - 1)\}
  return Spread[1, n]
```

The resulting algorithm runs in \(O(n^2)\) time.

**Rubric:** 4 points, standard dynamic programming rubric (scaled). These are not the only correct solutions. Again, an algorithm that assumes a particular player moves first is enough for full credit.
3. Describe and analyze an efficient algorithm to compute the maximum number of climbers that can play Alex's climbing game. More formally, you are given a rooted tree $T$ and an integer $k$, and you want to find the largest possible number of disjoint paths in $T$, where each path has length $k$. For full credit, do \textbf{not} assume that $T$ is a binary tree.

\textbf{Solution:} First, let's obtain a recursive definition for the problem. Consider the subtree of $G$ rooted at some arbitrary node $v$. In the optimal collection of disjoint length-$k$ paths, there are two possibilities for $v$:

(a) $v$ lies on one of the paths, which starts from from one of $v$'s ancestors. This path must continue into one of $v$'s children, and new paths may or may not begin at each of $v$'s other children.

(b) $v$ does not lie on any of the paths. In this case, new paths may or may not start at each of $v$'s children.

To formalize this intuition into a recursive formulation of the problem, we define the following values for each node $v$:

- For any integer $0 \leq \ell \leq k$, let $\text{MaxWith}(v, \ell)$ denote the maximum number of disjoint paths in the subtree rooted at $v$, where $v$ lies on a path of length $\ell$, but every other path has length $k$.
- Let $\text{MaxWithout}(v)$ denote the maximum number of disjoint length-$k$ paths in the subtree rooted at $v$, where none of the paths contain $v$.
- Finally, let $\text{MaxPaths}(v)$ denote the maximum number of disjoint length-$k$ paths in the subtree rooted at $v$, with no other restrictions.

We need to compute $\text{MaxPaths}(\text{root})$.

These functions obey the following mutual recurrences. I will use the non-standard notation $w \downarrow v$ to denote “$w$ is a child of $v$”.

\[
\text{MaxWith}(v, \ell) = \begin{cases} 
-\infty & \text{if } v \text{ is a leaf and } \ell > 0 \\
1 + \sum_{w \downarrow v} \text{MaxPaths}(w) & \text{if } \ell = 0 \\
\max_{w \downarrow v} \left\{ \text{MaxWith}(w, \ell - 1) + \sum_{x \downarrow v, x \neq w} \text{MaxPaths}(x) \right\} & \text{otherwise}
\end{cases}
\]

\[
\text{MaxWithout}(v) = \sum_{w \downarrow v} \text{MaxPaths}(w)
\]

\[
\text{MaxPaths}(v) = \max \{ \text{MaxWith}(v, k), \text{MaxWithout}(v) \}
\]

The recurrence for $\text{MaxWithout}$ does not need explicit base cases; we automatically have $\text{MaxWithout}(v) = 0$ when $v$ is a leaf, because $\sum \emptyset = 0$. Similarly, we automatically have $\text{MaxWithout}(v, 0) = 1$ when $v$ is a leaf.

In the interest of efficiency, we will remove the summations from our recurrence for
MaxWith(v, ℓ) as follows:

\[
\text{MaxWith}(v, ℓ) = \begin{cases} 
-\infty & \text{if } v \text{ is a leaf and } ℓ > 0 \\
1 + \text{MaxWithout}(v) & \text{if } ℓ = 0 \\
\text{MaxWithout}(v) + \max_{w\uparrow v} \{\text{MaxWith}(w, ℓ - 1) - \text{MaxPaths}(w)\} & \text{otherwise}
\end{cases}
\]

**Memoization:** We can memoize these functions into the tree itself, by adding three new fields to each vertex v: two integers v.MaxPaths and v.MaxWithout, and an integer array v.MaxWith[0..k].

**Evaluation order:** Each value v.MaxWith[ℓ] depends only on values stored at the children of v. Thus, we can fill the data structure by performing a post-order traversal of the tree, where at each node v, we compute v.MaxWithout, then fill the array v.MaxWith in arbitrarily order, and finally compute v.MaxPaths.

```plaintext
MAXPATHS(T, k):
    for all nodes v in T in postorder
        if v is a leaf
            v.MaxWithout ← 0
            v.MaxWith[0] ← 1
            for ℓ ← 1 to k
                v.MaxWith[ℓ] ← −∞
            v.MaxPaths ← 0
        else
            v.MaxWithout ← 0
            for all children w of v
                v.MaxWithout ← v.MaxWithout + w.MaxPaths
            for ℓ ← 0 to k
                localmax ← −∞
                for all children w of v
                    localmax ← max{localmax, MaxWith(w, ℓ - 1) - MaxPaths(w)}
                v.MaxWith[ℓ] ← v.MaxWithout + localmax
            v.MaxPaths ← max{v.MaxWith[k], v.MaxWithout}
    return T.root.MaxPaths
```

**Running time:** Assuming the input tree T has n vertices, this algorithm runs in \(O(nk)\) time. Ignoring the innermost loop (in red), the algorithm spends \(O(k)\) time in each iteration of the outer loop (that is, in each step of the postorder traversal). In addition, each node w spends \(O(k)\) time inside the innermost loop, all during its parent’s iteration of the outer loop. ■