Describe a dynamic programming algorithm to compute the minimum slop possible for the given words. Give asymptotic bounds for both the space and time required by your algorithm.

**Solution:** We can think about displaying a paragraph as displaying a sequence of lines, each defined by the words they contain. If we’ve already output the first \( i - 1 \) words, then our next decision is the last word \( j \) to appear on the line starting with word \( i \). All we need to remember from previous decisions is how many words we’ve already output, and our goal is minimize total slop for the remaining lines we need to print.

Let \( \text{MinSlop}(i) \) denote the minimum slop for printing a paragraph using words \( i \) through \( n \). Our goal is to compute \( \text{MinSlop}(1) \). Let \( \text{padding}(i, j) = L - j + i - \sum_{k=i}^{j-1} \) be the amount of extra white-space needed to fit words \( i \) through \( j \) on a single line. From the above discussion, we can define \( \text{MinSlop}(i) \) recursively as follows.

\[
\text{MinSlop}(i) = \begin{cases} 
0 & \text{if } \text{padding}(i, n) \geq 0 \\
\min \{ \text{MinSlop}(j + 1) + (\text{padding}(i, j))^3 \mid j \geq i \text{ and } \text{padding}(i, j) \geq 0 \} & \text{otherwise}
\end{cases}
\]

If all words \( i \) through \( n \) fit on a single line, then there is no slop, so the base case is correct. Otherwise, we’re taking the best choice of last word \( j \) to fit on the line with \( i \) where we suffer the slop of that first line plus the optimal slop of what’s left, computed correctly by induction on \( n - i \).

Function \( \text{MinSlop} \) takes a value \( i \) such that \( 1 \leq i \leq n \), so we can store results to our subproblems in an array \( \text{MinSlop}[1..n] \). Each subproblem depends upon those of larger index, so we can fill the array from right to left. Here is our algorithm.

```
MINIMIZE SLOP(1..n):
  for i ← n down to 1
    if padding(i, n) ≥ 0
      MinSlop[i] ← 0
    else
      MinSlop[i] ← ∞
    j ← i
    repeat
      if MinSlop[j + 1] + (padding(i, j))^3 < MinSlop[i]
        MinSlop[i] ← MinSlop[j + 1] + (padding(i, j))^3
      j ← j + 1
    while padding(i, j) ≥ 0
  return MinSlop[1]
```

Array \( \text{MinSlop}[1..n] \) has size \( O(n) \). For each entry of \( \text{MinSlop}[1..n] \), we check up to \( n \) values of \( j \), computing \( \text{padding}(i, j) \) for each of them. Evaluating the summation takes \( O(n) \) time, so the total running time is \( O(n^3) \). We can reduce the running time to \( O(n^2) \) by not evaluating the entire summation each time it is used but instead keeping a running total as we check different values for \( j \).

**Rubric:** 10 points total: 5 points for the recurrence and justification, 3 points for memoization, 1 point each for space and time analysis. The recurrence must be mostly correct to get the remaining points. Any correct polynomial time algorithm is worth full credit.
(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:** Consider the following set of four cards laid left to right, and suppose it is our turn:

\[
\begin{array}{ccc}
2 & 500 & 1 & 1 \\
\end{array}
\]

If we follow the greedy strategy, we'll take the 2, and then Elmo will take the 500 for the win. Instead, we should take the 1, forcing Elmo to pick the 2, leaving us with the 500. ■

**Rubric:** 2 points total: -0.5 for not describing the better strategy.

(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. You may assume that you get the first turn.

**Solution:** The natural backtracking strategy is to decide which of the two cards to take, and then optimally solve the same problem on what remains after Elmo takes his card. Since we're always picking cards from the extreme left or right, our recursive subproblems can be defined by stating which card is still leftmost and which is rightmost. Let $\text{GreedyPoints}(i, j)$ denote the maximum number of points we can collect if we begin our turn with cards $i$ through $j$ remaining, numbered left to right. Assuming we're initially dealt cards 1 through $n$, we should compute $\text{GreedyPoints}(1, n)$. Let $P[1..n]$ be an array where $P[i]$ stores the number of points for card $i$. For simplicity, we'll assume the point amounts are distinct.

\[
\text{GreedyPoints}(i, j) = \begin{cases} 
0 & \text{if } i > j \\
\max \left\{ P[i], \begin{array}{c}
P[i] + \text{GreedyPoints}(i+1, P[i+1] > P[j]), \\
\quad j - [P[j] \geq P[i+1]]
\end{array} \right\} & \text{if } i = j \\
\max \left\{ P[j] + \text{GreedyPoints}(i, P[i] > P[j-1]), \\
\quad j - 1 - [P[j-1] \geq P[i]] \right\} & \text{otherwise}
\end{cases}
\]

If $i > j$, then there are no cards and no points to collect. If $i = j$, there is exactly one card; we take it and the game ends. Otherwise, we get to decide on two cards. The combination of increasing $i$, decreasing $j$, and the rules in the $[\ ]$ notation tell us which cards will be left after Elmo takes his turn, and we return the correct maximum by induction on $j - i + 1$.

In any call to $\text{GreedyPoints}(i, j)$, we have $1 \leq i \leq n + 1$ and $0 \leq j \leq n$, so we can store results to our subproblems in a two-dimensional array $\text{GreedyPoints}[1..n+1, 0..n]$. Generic entry $\text{GreedyPoints}[i, j]$ depends upon entries in either lower rows or earlier columns, so we can fill the table row by row from bottom up, left to right within each row.
Rubric: 4 points total: 2 points for the recurrence and justification, 1.5 points for memoization, 0.5 points for running time analysis. The recurrence must be mostly correct to get the remaining points. Any correct polynomial time algorithm is worth full credit.

(c) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points you can collect playing against a perfect opponent. You may assume that you get the first turn.

Solution: We use the same backtracking strategy, but now we assume Elmo is running our own algorithm to maximize his score when his turn comes up. Therefore, we get an amount of points equal to the total minus what Elmo can get. Let \( \text{Perf ect Points}(i,j) \) denote the maximum number of points we can collect if we begin our turn with cards \( i \) through \( j \) remaining, numbered left to right. We should compute \( \text{Perf ect Points}(1,n) \). Again, \( P[i] \) is the number of points on card \( i \). Let \( \text{Total}(i,j) = \sum_{k=i}^j P[k] \).

\[
\text{Perf ect Points}(i,j) = \begin{cases} 
0 & \text{if } i > j \\
 P[i] & \text{if } i = j \\
\max \left\{ \text{Total}(i,j) - \text{Perf ect Points}(i+1,j), \text{Total}(i,j) - \text{Perf ect Points}(i,j-1) \right\} & \text{otherwise}
\end{cases}
\]

As in (b), we either get the 0 points possible or all the points on the one card in the base cases. Otherwise, we have a choice of two cards, and we may assume that Elmo collects the maximum possible number of points from what's leftover which is correctly computed by the recurrence by induction on \( j-i+1 \).
Again, we always have $1 \leq i \leq n + 1$ and $0 \leq j \leq n$, so we can store the subproblem results in the two-dimensional array \textit{PerfectPoints}[1..n + 1, 0..n]. We still depend upon later rows or earlier columns so we fill row by row from bottom up and left to right within each row.

\begin{verbatim}
MaxPointsPerfect(P[1..n]):
    for i ← n + 1 down to 1
        for j ← 0 to n
            if i > j
                PerfectPoints[i, j] ← 0
            else if i = j
                PerfectPoints[i, j] ← P[i]
            else
                takeLeft ← Total(i, j) − PerfectPoints[i + 1, j]
                takeRight ← Total(i, j) − PerfectPoints[i, j − 1]
                PerfectPoints[i, j] ← max{takeLeft, takeRight}
    return PerfectPoints[1, n]
\end{verbatim}

We again use $O(n^2)$ space to store our memoization structure. If we compute Total$(i, j)$ from scratch in each iteration, it takes $O(n)$ times to fill each entry for $O(n^3)$ time total. However, we can precompute all the Totals (using dynamic programming!) in $O(n^2)$ time so we can fill entries in constant time each for $O(n^2)$ time total.

\textbf{Rubric:} 4 points total: 2 points for the recurrence and justification, 1.5 points for memoization, 0.5 points for running time analysis. The recurrence must be mostly correct to get the remaining points. Any correct polynomial time algorithm is worth full credit.
Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. (An algorithm that merely computes the minimum amount of soda consumed is worth full credit.) Your input consists of the bus route (a list of exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit). Give asymptotic bounds for both the space and time required by your algorithm.

Solution: We’ll assume we’re given an array $TimeTo[1..n]$ where $TimeTo[i]$ is the total time in minutes to drive from exit 1 to exit $i$ (with exits ordered left to right), an array $Students[1..n]$ stating how many students get off at each exit, and the starting exit $s$. We can easily compute the $TimeTo[1..n]$ array in linear time if we’re given distances between adjacent stops. Given a list of exits yet to be visited and the current location of the bus, it is tempting to guess the next location, and send the now smaller list of unvisited exits to the Recursion Fairy. However, the number of subsets of exits they we may need to visit is $2^n$. Instead, we rely on a couple observations that significantly reduce the number of subproblems to consider.

First, we observe that we should never pass an exit without dropping off students there, since dropping off students is instantaneous. Therefore, the set of visited exits forms a contiguous set somewhere in the middle of the highway. In particular, the unvisited exits form a prefix and suffix of the set of all exits. Also, after every new visit, the bus either lies one exit to the right of the prefix or one exit to the left of the suffix. Let $MinSoda(i, j, side)$ denote the minimum amount of soda consumed if we still need to drop students at exits 1 through $i$ and $j$ through $n$ where the bus starts at exit $i + 1$ if $side = left$ and it starts at $j - 1$ if $side = right$. Since we can immediately drop all students off at exit $s$, our goal is to compute $MinSoda(s - 1, s + 1, left) = MinSoda(s - 1, s + 1, right)$. Let $remaining(i, j)$ denote the total number of students at exits 1 through $i$ and $j$ through $n$. We can do some linear time precomputation to know the total number of students at each prefix or suffix so $remaining(i, j)$ can be computed in constant time. Let $Time(i, j) = TimeTo[j] - TimeTo[i]$ be the time it takes to travel from stop $i$ to $j$ or vice versa. From the above discussion, we can define $MinSoda(i, j, side)$ as follows.

$$MinSoda(i, j, side) =$$

$$\begin{cases} 
0 & \text{if } i < 1 \text{ and } j > n \\
remaining(i, j) \cdot Time(i + 1, j) + MinSoda(i, j + 1, right) & \text{if } i < 1, j \leq n, \text{ and } side = left \\
remaining(i, j) \cdot Time(j - 1, j) + MinSoda(i, j + 1, right) & \text{if } i < 1, j \leq n, \text{ and } side = right \\
remaining(i, j) \cdot Time(i, i + 1) + MinSoda(i - 1, j, left) & \text{if } i \geq 1, j > n, \text{ and } side = left \\
remaining(i, j) \cdot Time(i, j - 1) + MinSoda(i - 1, j, left) & \text{if } i \geq 1, j \leq n, \text{ and } side = right \\
\min \left\{ \begin{array}{l}
remaining(i, j) \cdot Time(i, i + 1) + MinSoda(i - 1, j, left), \\
remaining(i, j) \cdot Time(i + 1, j) + MinSoda(i, j + 1, right)
\end{array} \right\} & \text{if } i \geq 1, j \leq n, \text{ and } side = left \\
\min \left\{ \begin{array}{l}
remaining(i, j) \cdot Time(i, j - 1) + MinSoda(i - 1, j, left), \\
remaining(i, j) \cdot Time(j - 1, j) + MinSoda(i, j + 1, right)
\end{array} \right\} & \text{otherwise}
\end{cases}$$

If there are no exits left, then there’s no more soda consumed. If there are no exits to the left, then our only choice is to move to exit $j$ and try to optimize from there, which we do by induction. A similar argument holds if there are no exits to the right. If there are exits on both sides, we need to decide which way to go. The students will drink soda on the way to the exit we
go to (either exit \(i\) or \(j\)), and then we need to optimize given the remaining exits, which we do by induction.

For the parameters to \(\text{MinSoda}\) we have \(0 \leq i < n - 1\) and \(2 \leq j \leq n + 1\) (since there’s at least one exit strictly between \(i\) and \(j\) for any subproblem we care about) as well as the two choices for \(\text{side}\). Therefore, we can store subproblem solutions in a two-dimensional array \(\text{MinSoda}[0..n-1,2..n+1]\) where each entry is a structure holding a \(\text{left}\) and \(\text{right}\) value. Generic entries depend upon those in an earlier row or later column, so we can fill the table row by row from top to bottom and right to left within each row.

```plaintext
SAVE_SODA(TimeTo[1..n], Students[1..n], s):
    for \(i \leftarrow 0\) to \(n - 1\)
        for \(j \leftarrow n + 1\) down to 2
            if \(i < 1\) and \(j > n\)
                \(\text{MinSoda}[i, j].left \leftarrow 0\)
                \(\text{MinSoda}[i, j].right \leftarrow 0\)
            else if \(i < 1\) and \(j \leq n\)
                \(\text{MinSoda}[i, j].left \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i + 1, j) + \text{MinSoda}[i, j + 1].right\)
                \(\text{MinSoda}[i, j].right \leftarrow \text{remaining}(i, j) \cdot \text{Time}(j - 1, j) + \text{MinSoda}[i, j + 1].right\)
            else if \(i \geq 1\) and \(j > n\)
                \(\text{MinSoda}[i, j].left \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i, i + 1) + \text{MinSoda}[i - 1, j].left\)
                \(\text{MinSoda}[i, j].right \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i, j - 1) + \text{MinSoda}[i - 1, j].right\)
            else
                \(\text{goLeft} \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i, i + 1) + \text{MinSoda}[i - 1, j].left\)
                \(\text{goRight} \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i + 1, j) + \text{MinSoda}[i, j + 1].right\)
                \(\text{MinSoda}[i, j].left \leftarrow \min\{\text{goLeft}, \text{goRight}\}\)
                \(\text{goLeft} \leftarrow \text{remaining}(i, j) \cdot \text{Time}(i, j - 1) + \text{MinSoda}[i - 1, j].left\)
                \(\text{goRight} \leftarrow \text{remaining}(i, j) \cdot \text{Time}(j - 1, j) + \text{MinSoda}[i, j + 1].right\)
                \(\text{MinSoda}[i, j].right \leftarrow \min\{\text{goLeft}, \text{goRight}\}\)
    return \(\text{MinSoda}[s - 1, s + 1].left\)
```

It takes \(O(n^2)\) \text{space} to store the memoization data structure. Assuming we did the preprocessing to compute \(\text{remaining}(i, j)\) and \(\text{Time}(i, j)\) in constant time, it takes constant time to fill each of the entries for \(O(n^2)\) \text{time total}. \[\blacksquare\]

\textbf{Rubric:} 10 points total: 5 points for the recurrence and justification, 3 points for memoization, 1 point each for space and time analysis. The recurrence must be mostly correct to get the remaining points. Any correct polynomial time algorithm is worth full credit.