1. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4\rangle$ then we can fully parenthesize the product in five distinct ways:

   - $\big(A_1, \big(A_2, \big(A_3, A_4\big)\big)\big)$
   - $\big(A_1, \big((A_2, A_3)A_4\big)\big)$
   - $\big((A_1 A_2)\big(A_3 A_4\big)\big)$
   - $\big((A_1\,(A_2 A_3))A_4\big)$
   - $\big(((A_1, A_2)A_3)A_4\big)$

   How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. For example, if we have three matrices with dimensions: $A_1[10 \times 100]$, $A_2[100 \times 5]$, $A_3[5 \times 50]$, then $((A_1A_2)A_3)$ takes 7500 scalar multiplications while $(A_1\,(A_2A_3))$ takes 75000 scalar multiplications.

   Given a chain of n matrices, $A_1A_2\ldots A_n$, where matrix A_i has dimension $r_i \times c_i$, provide a dynamic programming algorithm that finds the mininum number of scalar multiplications required to evaluate the product of the matrices.

Solution: Note the following before we discuss the solution:

(a) For any two consecutive matrices M_i and M_{i+1}, $c_i = r_{i+1}$.

(b) Multiplying two matrices M_iM_j requires $r_ic_ic_j (= r_ir_jc_j)$ scalar multiplications.

(c) Multiplying matrices $M_i, M_{i+1}, \cdots M_j$ results in a matrix with dimension $r_i \times c_j$.

Consider a matrix chain $M_1M_2\cdots M_n$. There are $n-1$ matrix multiplications to compute, and we want to order the $n-1$ matrix multiplications in a way that minimizes the number of scalar multiplications. Suppose we compute the k-th multiplication(the multiplication between M_k and M_{k+1}) at the very last. Then, the cost(the number of scalar multiplications) of the k-th multiplication would be $r_1c_kc_n$, since the matrices $M_1\cdots M_k$ and $M_{k+1}\cdots M_n$ would be multiplied separately before the k-th multiplication, giving the dimensions $r_1 \times c_k$ and $r_{k+1} \times c_n$ respectively. Therefore, the minimum cost we can get while computing the k-th multiplication at last would be $MSM(1,k)+MSM(k+1,n)+r_1c_kc_n$, where $MSM(i,j)$ is the minimum total cost of the multiplications on $M_i\cdots M_j$. Based on the above observation, we can construct the following recurrence.

$$MSM(i,j) = \begin{cases} 0 & \text{if } i >= j \\ \min\limits_{i\le k<j} MSM(i,k) + MSM(k+1,j) + r_ic_kc_j & \text{otherwise} \end{cases}$$

As mentioned, $MSM(i,j)$ represents the minimum number of scalar multiplications required to compute $M_iM_{i+1}\cdots M_j$, and therefore $MSM(1,n)$ would be the final answer we are looking for.

To get a dynamic programming algorithm, we use a 2D array $MSM[1\,..\,n, 1\,..\,n]$ for memoization, filling out the entries in decreasing i and increasing j order.

```
MSM(A[1..n]):
  for $i \leftarrow 1$ to $n$                ⟨⟨Base cases⟩⟩
      $MSM[i,j] \leftarrow 0$
  for $i \leftarrow n$ down to 1
      for $j \leftarrow i+1$ to $n$
          $min \leftarrow \infty$
          for $k \leftarrow i$ to $j-1$
              $c \leftarrow MSM[i,k] + MSM[k+1,j] + r_i c_k c_j$
              if $c < min$
                  $min \leftarrow c$
          $MSM[i,j] \leftarrow min$
  return $MSM[1,n]$
```

Since the amount of work at each iteration is constant, the runtime of the algorithm is $O(n^3)$. ■

2. Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1; l_2; \ldots; l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \Sigma_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm. Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1; l_2; \ldots; l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \Sigma_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Solution: Intuition: We must decide where to place line breaks so that the total penalty (defined as the cube of the extra spaces at the end of each line) is minimized. Greedily minimizing the extra spaces in each line does not account for the cumulative penalties of future lines, potentially leading to a higher total cost than an arrangement that strategically allows more extra space in some lines to minimize the overall penalty. Therefore, we will use DP to solve this question.

Function Definitions:

- M: The given maximum number of characters per line.
- $l[1...n]$: sequence of n words.
- minCost(j): The minimal total cost of neatly printing words from word 1 to word j. The solution is calculated by minCost(n).
- firstPosInLine(j): The position of the first word on the line ending at word j in the optimal arrangement. Used when reconstructing the paragraph.
- extras(i, j): The number of extra spaces at the end of a line containing words i to j. We calculate the number of extra spaces when words i to j are placed on a line using provided formula:

$$\text{extras}(i,j) = M - j + i - \Sigma_{k=i}^{j} l_k \tag{1}$$

- lineCost(i, j): The line cost (penalty) when words i to j are placed on a single line. lineCost(i, j) is defined as:

$$\text{lineCost}(i,j) = \begin{cases} \text{extras}(i,j)^3, & \text{if extras}(i,j) \geq 0 \text{ and } j \neq n \\ 0, & \text{if extras}(i,j) \geq 0 \text{ and } j = n \\ \infty, & \text{if extras}(i,j) < 0 \end{cases} \tag{2}$$

Recurrence Relation: The minimal total cost minCost(j) can be computed using the following recurrence relation:

$$\text{minCost}(j) = \begin{cases} 0 & \text{if } j = 0 \quad \text{(Base case)} \\ \min_{1 \leq i \leq j} \{\text{minCost}(i-1) + \text{lineCost}(i,j)\} & \text{if } j > 0 \quad \text{(Recursive case)} \end{cases} \tag{3}$$

This relation states that the minimal cost of arranging words 1 to j is the minimum cost of arranging words 1 to $i-1$ plus the cost of placing words i to j on a new line.

Filling Order: We use a length n 1D array as the memoization structure. Since j starts from 0 in our base case, we fill the array in order of increasing j from 1 to n. For each j, we need to consider all possible starting positions i for the line ending at j and choose the one that minimizes minCost(j).

Algorithm:

```
NeatlyPrintParagraph($l[1..n]$, $M$):
  ⟨⟨Step 1: Compute extras[i][j]⟩⟩
  ⟨⟨Precompute the extra spaces for all combinations of words from i to j⟩⟩
  For $i \leftarrow 1$ to $n$:
      extras[i][i] $\leftarrow M - l[i]$
      For $j \leftarrow i+1$ to $n$:
          extras[i][j] $\leftarrow$ extras[i][j - 1] $- l[j] - 1$
  ⟨⟨Step 2: Compute line costs lineCost[i][j]⟩⟩
  ⟨⟨Compute the line costs based on the extra spaces calculated⟩⟩
  For $i \leftarrow 1$ to $n$:
      For $j \leftarrow i$ to $n$:
          If extras[i][j] $\geq 0$:
              If $j = n$:
                  lineCost[i][j] $\leftarrow 0$ ⟨⟨No penalty for the last line⟩⟩
              Else:
                  lineCost[i][j] $\leftarrow$ (extras[i][j])$^3$
          Else:
              lineCost[i][j] $\leftarrow \infty$ ⟨⟨Words do not fit on one line⟩⟩
  ⟨⟨Step 3: Compute minimum cost minCost[j] and track breaks firstPosInLine[j]⟩⟩
  minCost[0] $\leftarrow$ 0
  For $j \leftarrow 1$ to $n$:
      minCost[j] $\leftarrow \infty$
      For $i \leftarrow 1$ to $j$:
          If minCost[i - 1] + lineCost[i][j] < minCost[j]:
              minCost[j] $\leftarrow$ minCost[i - 1] + lineCost[i][j]
              firstPosInLine[j] $\leftarrow$ i ⟨⟨Record the line break position⟩⟩
  ⟨⟨Step 4: Reconstruct the solution⟩⟩
  Initialize $lines \leftarrow$ empty list
  $k \leftarrow n$
  While $k > 0$:
      $i \leftarrow$ firstPosInLine[$k$]
      Prepend $(i, k)$ to $lines$
      $k \leftarrow i - 1$

  Return $lines$
```

- **Running Time Analysis:**
  (a) Computing extras[i][j] takes $O(n^2)$ time since we have nested loops over i and j.
  (b) Computing lineCost[i][j] also takes $O(n^2)$ time.
  (c) Computing $minCost[j]$ and $firstPosInLine[j]$ involves two nested loops, resulting in $O(n^2)$ time.
  (d) Reconstructing the solution takes $O(n)$ time.
  (e) **Total Time Complexity:** $O(n^2)$.
- Note: Because each word length is positive, each line cannot have more than M words, so each entry minCost[j] depends on at most n and at most M previous entries minCost[i]. Therefore, we can optimize the range of the loops to be the min(M, n) for the all functions used, and the resulting algorithm technically runs in $O(\min(n^2, nM))$ time, but it's not required for full credit.

- **Space Requirements:**
  (a) The arrays extras$[1..n][1..n]$ and lineCost$[1..n][1..n]$ require $O(n^2)$ space.
  (b) The arrays $minCost[0..n]$ and $firstPosInLine[1..n]$ require $O(n)$ space.
  (c) **Total Space Complexity:** $O(n^2)$**.**

■

3. Binomial coefficients are a family of positive integers that have a number of useful properties and they can be defined in several ways. One way to define them is as an indexed recursive function, C(n, k), where the "C" stands for "choice" or "combinations." In this case, the definition is as follows:

$$\begin{aligned} C(n,0) &= 1 \\ C(n,n) &= 1 \\ C(n,k) &= C(n-1,k-1) + C(n-1,k) \qquad 0 < k < n \end{aligned}$$

Describe a scheme for computing $C(n,k)$ using memoization.

Solution: For a 2D table, according to the recurrence, each time we look at the element at "upper left" and "upper" with respect to current element, and as we have k < n, we only need to fill the "bottom left" half of the memory table.

```
BinomialCoefficient($n, k$):
  Initialize a 2D array $dp$ of size $(n+1) \times (k)$
  for $i \leftarrow 0$ to $n$
      $dp[i][0] \leftarrow 1$ ⟨⟨$C(n,0) = 1$ for all n⟩⟩
      $dp[i][i] \leftarrow 1$ ⟨⟨$C(n,n) = 1$ for all n⟩⟩
  for $i \leftarrow 1$ to $n$
      for $j \leftarrow 1$ to $i-1$
          $dp[i][j] \leftarrow dp[i-1][j-1] + dp[i-1][j]$
      return $dp[n][k]$
```

Runtime: O(nk) ■

4. Suppose we are given a collection $A = \{a_1, a_2, ..., a_n\}$ of n positive integers that add up to N. Design an O(nN)-time algorithm for determining whether there is a subset $B \in A$ such that $\Sigma_{a_i \in B} a_i = \Sigma_{a_i \in A-B} a_i$.

Solution: Does there exist a subset that sums to $\frac{N}{2}$

English Description E(i, j):

Let $E(i, j)$ decide whether there exists a subset of $A[i \ldots n]$ that sums to j.

Recursive Relation:

$$E(i, j) = E(i+1, j) \vee E(i+1, j - A[i])$$

- **General Case:** The relation holds for all valid i and j.
- **Base Case:** $E(i, 0) =$ True for $i \in 1 \ldots n+1$.
- **Invalid Case:** $E(i, j) =$ False if $j < 0$.

Answer:

The final answer will be $E(1, \frac{N}{2})$.

Memoization:

- Use a 2D array to store results: decreasing i, increasing j.
- **Space Complexity:** $O(nN)$.

Time Complexity:

- O(n) options for i, O(N) options for j, and each takes O(1) comparisons.
- **Time Complexity:** $O(nN)$.

■