Today we discuss a technique called "Dynamic programming". It is neither especially 'dynamic' nor especially 'programming' related. We will discuss dynamic programming by looking at an example.

1 Matrix-chain multiplication

- Problem: Given a sequence of matrices $A_1, A_2, A_3, ..., A_n$, find the best way (using the minimal number of multiplications) to compute their product.
  - Isn’t there only one way? $((\cdots ((A_1 \cdot A_2) \cdot A_3) \cdots) \cdot A_n)$
  - No, matrix multiplication is associative.
    e.g. $A_1 \cdot (A_2 \cdot (A_3 \cdot (\cdots (A_{n-1} \cdot A_n) \cdots)))$ yields the same matrix.
  - Different multiplication orders do not cost the same:
    * Multiplying $p \times q$ matrix $A$ and $q \times r$ matrix $B$ takes $p \cdot q \cdot r$ multiplications; result is a $p \times r$ matrix.
    * Consider multiplying $10 \times 100$ matrix $A_1$ with $100 \times 5$ matrix $A_2$ and $5 \times 50$ matrix $A_3$.
      - $(A_1 \cdot A_2) \cdot A_3$ takes $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ multiplications.
      - $A_1 \cdot (A_2 \cdot A_3)$ takes $100 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 100 = 75000$ multiplications.

- In general, let $A_i$ be $p_{i-1} \times p_i$ matrix.
  - $A_1, A_2, A_3, ..., A_n$ can be represented by $p_0, p_1, p_2, p_3, ..., p_n$

- Let $m(i, j)$ denote minimal number of multiplications needed to compute $A_i \cdot A_{i+1} \cdots A_j$
  - We want to compute $m(1, n)$.

- Divide-and-conquer solution/recursive algorithm:
  - Divide into $j - i - 1$ subproblems by trying to set parenthesis in all $j - i - 1$ positions.
    (e.g. $(A_i \cdot A_{i+1} \cdots A_k) \cdot (A_{k+1} \cdots A_j)$ corresponds to multiplying $p_{i-1} \times p_k$ and $p_k \times p_j$ matrices.)
  - Recursively find best way of solving sub-problems. (e.g. best way of computing $A_i \cdot A_{i+1} \cdots A_k$ and $A_{k+1} \cdot A_{k+2} \cdots A_j$)
  - Pick best solution.
• Algorithm expressed in terms of $m(i, j)$:

\[
m(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{if } i < j 
\end{cases}
\]

• Program:

```plaintext
MATRIX-CHAIN(i, j)
    IF i = j THEN return 0
    m(i, j) = \infty
    FOR k = i TO j - 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j
        IF q < m(i, j) THEN m(i, j) = q
    OD
    Return m(i, j)
END MATRIX-CHAIN

Return MATRIX-CHAIN(1, n)
```

• Running time:

\[
T(n) = \sum_{k=1}^{n-1} (T(k) + T(n - k) + O(1))
\]

\[
= 2 \cdot \sum_{k=1}^{n-1} T(k) + O(n)
\]

\[
\geq 2 \cdot T(n - 1)
\]

\[
\geq 2 \cdot 2 \cdot T(n - 2)
\]

\[
\geq 2 \cdot 2 \cdot 2 \cdots
\]

\[
= 2^n
\]

• Problem is that we compute the same result over and over again.

  - Example: Recursion tree for $\text{MATRIX-CHAIN}(1, 4)$
We for example compute \textsc{Matrix-chain}(3, 4) twice.

- Solution is to "remember" values we have already computed in a table—\textit{memoization}

\begin{verbatim}
\textbf{Matrix-chain}(i, j)
  IF \( i = j \) THEN return 0
  IF \( m(i, j) < \infty \) THEN return \( m(i, j) \) // This line has changed */
  FOR \( k = i \) to \( j - 1 \) DO
    \( q = \text{Matrix-chain}(i, k) + \text{Matrix-chain}(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j \)
    IF \( q < m(i, j) \) THEN \( m(i, j) = q \)
  OD
  return \( m(i, j) \)
\end{verbatim}

\begin{verbatim}
\textbf{Matrix-chain}(1, n)
\end{verbatim}

- Running time:
  - \( \Theta(n^2) \) different calls to \textsc{Matrix-chain}(i, j).
  - The first time a call is made it takes \( O(n) \) time, \textit{not} counting recursive calls.
  - When a call has been made once it costs \( O(1) \) time to make it again.
    \[ \Downarrow \]
    \( O(n^3) \) time
Another way of thinking about it: $\Theta(n^2)$ total entries to fill, it takes $O(n)$ to fill one.

2 Alternative view of Dynamic Programming

- Often (including in the book) dynamic programming is presented in a different way; As filling up a table from the bottom.

- Matrix-chain example: Key is that $m(i, j)$ only depends on $m(i, k)$ and $m(k + 1, j)$ where $i \le k < j \Rightarrow$ if we have computed them, we can compute $m(i, j)$

  - We can easily compute $m(i, i)$ for all $1 \le i \le n$ ($m(i, i) = 0$)
  - Then we can easily compute $m(i, i + 1)$ for all $1 \le i \le n - 1$
    \[
    m(i, i + 1) = m(i, i) + m(i + 1, i + 1) + p_{i-1} \cdot p_i \cdot p_{i+1}
    \]
  - Then we can compute $m(i, i + 2)$ for all $1 \le i \le n - 2$
    \[
    m(i, i + 2) = \min\{m(i, i) + m(i + 1, i + 2) + p_{i-1} \cdot p_i \cdot p_{i+2}, m(i, i + 1) + m(i + 2, i + 2) + p_{i-1} \cdot p_{i+1} \cdot p_{i+2}\}
    \]
    ...
  - Until we compute $m(1, n)$
  - Computation order:

    \[
    \begin{array}{cccccccc}
    & & & & & & & \\
    & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    \hline
    1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
    2 & & 1 & 2 & 3 & 4 & 5 & 6 \\
    3 & & & 1 & 2 & 3 & 4 & 5 \\
    4 & & & & 1 & 2 & 3 & 4 \\
    5 & & & & & 1 & 2 & 3 \\
    6 & & & & & & 1 & 2 \\
    7 & & & & & & & 1
    \end{array}
    \]
  - Computation order

- Program:
FOR \( i = 1 \) to \( n \) DO
\[
m(i, i) = 0
\]
OD
FOR \( l = 1 \) to \( n - 1 \) DO
  FOR \( i = 1 \) to \( n - l \) DO
    \( j = i + l \)
    \( m(i, j) = \infty \)
    FOR \( k = 1 \) to \( j - 1 \) DO
      \[
      q = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j
      \]
      IF \( q < m(i, j) \) THEN \( m(i, j) = q \)
    OD
  OD
OD

- Analysis:
  - \( O(n^2) \) entries, \( O(n) \) time to compute each \( \Rightarrow O(n^3) \).

- Note:
  - I like recursive (divide-and-conquer) thinking.
  - Book seems to like table method better.