Dynamic Programming: Matrix chain multiplication

(CLRS 15.2)

1 The 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 10×100 matrix A_1 with 100×5 matrix A_2 and 5×50 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.

2 Notation

- In general, let A_i be $p_{i-1} \times p_i$ matrix.
- 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 $m(1, n)$.

3 Recursive algorithm

- Assume that someone tells us the position of the **last** product, say k. Then we have to compute recursively the best way to multiply the chain from i to k, and from $k+1$ to j, and add the cost of the final product. This means that

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$$

- If noone tells us k, then we have to try all possible values of k and pick the best solution.

- Recursive formulation of $m(i,j)$:

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

- To go from the recursive formulation above to a program is pretty straightforward:

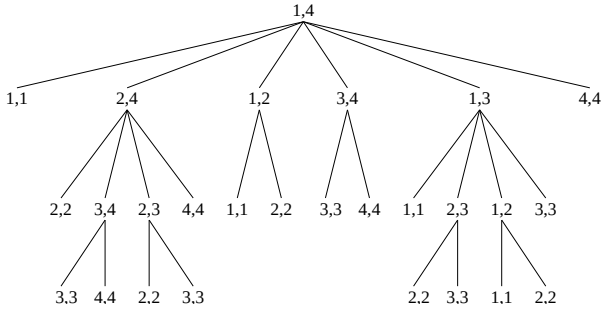
```
MATRIX-CHAIN(i, j)
    IF i = j THEN return 0
    m = ∞
    FOR k = i TO j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    Return m
END MATRIX-CHAIN

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

- Running time:

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-1}(T(k) + T(n-k) + O(1)) \\ &= 2 \cdot \sum_{k=1}^{n-1} T(k) + O(n) \\ &\geq 2 \cdot T(n-1) \\ &\geq 2 \cdot 2 \cdot T(n-2) \\ &\geq 2 \cdot 2 \cdot 2 \ldots \\ &= 2^n \end{aligned}$$

- Exponential is ... SLOW!
- Problem is that we compute the same result over and over again.
  - Example: Recursion tree for MATRIX-CHAIN$(1,4)$



For example, we compute MATRIX-CHAIN$(3, 4)$ twice.

4 Dynamic programming with a table and recursion

- Solution is to "remember" the values we have already computed in a table. This is called *memoization*. We'll have a table T[1..n][1..n] such that T[i][j] stores the solution to problem Matrix-CHAIN(i,j). Initially all entries will be set to ∞.

```
FOR i = 1 to n DO
    FOR j = i to n DO
        T[i][j] = ∞
    OD
OD
```

- The code for MATRIX-CHAIN(i,j) stays the same, except that it now uses the table. The first thing MATRIX-CHAIN(i,j) does is to check the table to see if $T[i][j]$ is already computed. Is so, it returns it, otherwise, it computes it and writes it in the table. Below is the updated code.

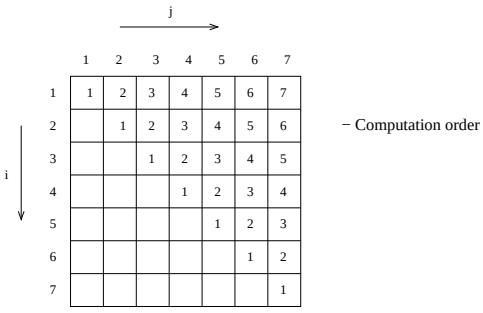
```
MATRIX-CHAIN(i, j)
    IF T[i][j] < ∞ THEN return T[i][j]
    IF i = j THEN T[i][j] = 0, return 0
    m = ∞
    FOR k = i to j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j)+p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    T[i][j] = m
    return m
END MATRIX-CHAIN

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

- The table will prevent a subproblem MATRIX-CHAIN(i,j) to be computed more than once.
- Running time:
  - $\Theta(n^2)$ different calls to MATRIX-CHAIN(i, j).
  - The first time a call is made it takes $O(n)$ time, *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.

5 Dynamic Programming without recursion

- Often dynamic programming is presented as filling up a table from the bottom, in such a way that makes recursion unnecessary. Avoiding recursion is perhaps elegant, and necessary 20-30 years ago when programming languages did not include support for recursion.
- Personally, I like top-down recursive thinking, and I think compilers are pretty effective at optimizing recursion.
- Solution for Matrix-chain without recursion: Key is that $m(i,j)$ only depends on $m(i,k)$ and $m(k+1,j)$ where $i \leq k < j \Rightarrow$ if we have computed them, we can compute $m(i,j)$
  - We can easily compute $m(i,i)$ for all $1 \leq i \leq n$ $(m(i,i) = 0)$
  - Then we can easily compute $m(i,i+1)$ for all $1 \leq i \leq n-1$
    $m(i,i+1) = m(i,i) + m(i+1,i+1) + p_{i-1} \cdot p_i \cdot p_{i+1}$
  - Then we can compute $m(i,i+2)$ for all $1 \leq i \leq n-2$
    $m(i,i+2) = \min\{m(i,i) + m(i+1,i+2) + p_{i-1} \cdot p_i \cdot p_{i+2}, m(i,i+1) + m(i+2,i+2) + p_{i-1} \cdot p_{i+1} \cdot p_{i+2}\}$
    $\vdots$
  - Until we compute $m(1,n)$
  - Computation order:



- Program:

```
FOR i = 1 to n DO
    T[i][i] = 0
OD
FOR l = 1 to n − 1 DO
    FOR i = 1 to n − l DO
        j = i + l
        T[i][j] = ∞
        FOR k = 1 to j − 1 DO
            q = T[i][k] + T[k + 1][j] + p_{i−1} · p_k · p_j
            IF q < T[i[j] THEN T[i][j] = q
        OD
    OD
OD
```

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

6 Extensions

Above we only computed the best way to multiply the chain (with the smallest number of operations). The algorithm can be extended to compute the actual order of multiplications corresponding to this optimal cost (we'll do this as homework or in-class exercise).

This is a simplification that is often done: focus on computing the optimal cost, and leave the details of computing the solution corresponding to that optimal cost for later.