Announcements

- Weekly Reading Assignment: CLRS, Chapter 15
- Homework #7 due on Monday, Dec. 12, 2005 (by 3:30pm in SN115)

Optimization Problems

- **In which a set of choices must be made in order to arrive at an optimal (min/max) solution, subject to some constraints. (There may be several solutions to achieve *an* optimal value.)**
- **Two common techniques:**
  - **Dynamic Programming (global)**
  - Greedy Algorithms (local)

Dynamic Programming

- Similar to divide-and-conquer, it breaks problems down into smaller problems that are solved recursively.

- In contrast, DP is applicable when the sub-problems are not independent, i.e. when sub-problems share sub-sub-problems. It solves every sub-sub-problem just once and save the results in a table to avoid duplicated computation.

Elements of DP Algorithms

- **Sub-structure:** decompose problem into smaller sub-problems. Express the solution of the original problem in terms of solutions for smaller problems.
- **Table-structure:** Store the answers to the sub-problem in a table, because sub-problem solutions may be used many times.
- **Bottom-up computation:** combine solutions on smaller sub-problems to solve larger sub-problems, and eventually arrive at a solution to the complete problem.

Applicability to Optimization Problems

- **Optimal sub-structure (principle of optimality):** for the global problem to be solved optimally, each sub-problem should be solved optimally. This is often violated due to sub-problem overlaps. Often by being “less optimal” on one problem, we may make a big savings on another sub-problem.

- **Small number of sub-problems:** Many NP-hard problems can be formulated as DP problems, but these formulations are not efficient, because the number of sub-problems is exponentially large. Ideally, the number of sub-problems should be at most a polynomial number.

Optimized Chain Operations

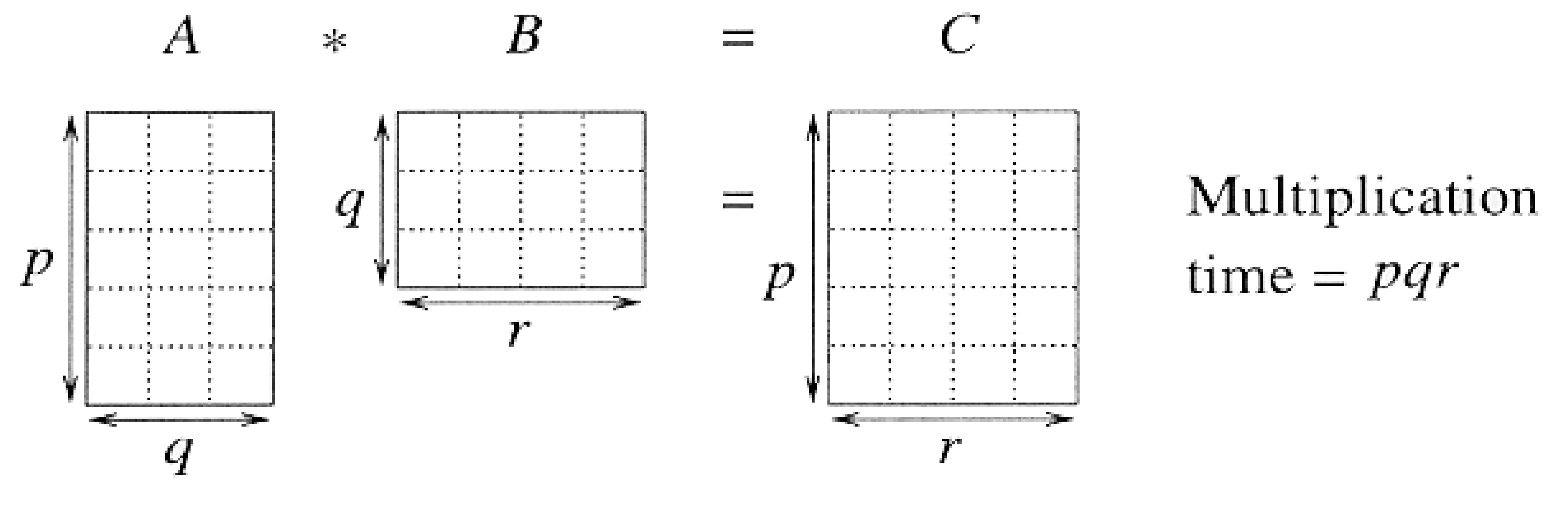
- **Determine the optimal sequence for performing a series of operations.** (the general class of the problem is important in compiler design for code optimization & in databases for query optimization)
- **For example: given a series of matrices: $A_1 \ldots A_n$, we can "parenthesize" this expression however we like, since matrix multiplication is associative (but not commutative).**
- **Multiply a p x q matrix A times a q x r matrix B, the result will be a p x r matrix C. (# of columns of A must be equal to # of rows of B.)**

Matrix Multiplication

- In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$,

$$C[i, j] = \sum_{k = 1 \text{ to } q} A[i, k]\, B[k, j]$$

- Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply 2 matrices is pqr.



Chain Matrix Multiplication

- **Given a sequence of matrices $A_1 A_2 \ldots A_n$, and dimensions $p_0 p_1 \ldots p_n$ where A_i is of dimension p_{i-1} x p_i, determine multiplication sequence that minimizes the number of operations.**

- **This algorithm does not perform the multiplication, it just figures out the best order in which to perform the multiplication.**

Example: CMM

- Consider 3 matrices: A_1 be 5 x 4, A_2 be 4 x 6, and A_3 be 6 x 2.

$$\text{Mult}[((A_1 A_2)A_3)] = (5\text{x}4\text{x}6) + (5\text{x}6\text{x}2) = 180$$

$$\text{Mult}[(A_1 (A_2 A_3))] = (4\text{x}6\text{x}2) + (5\text{x}4\text{x}2) = 88$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Naive Algorithm

- If we have just 1 item, then there is only one way to parenthesize. If we have n items, then there are n-1 places where you could break the list with the outermost pair of parentheses, namely just after the first item, just after the 2nd item, etc. and just after the (n-1)th item.
- When we split just after the kth item, we create two sub-lists to be parenthesized, one with k items and the other with n-k items. Then we consider all ways of parenthesizing these. If there are L ways to parenthesize the left sub-list, R ways to parenthesize the right sub-list, then the total possibilities is $L*R$.

Cost of Naive Algorithm

- The number of different ways of parenthesizing n items is

$$P(n) = 1, \qquad \text{if } n = 1$$

$$P(n) = \sum_{k = 1 \text{ to } n-1} P(k)P(n-k), \qquad \text{if } n \geq 2$$

- This is related to *Catalan numbers* (which in turn is related to the number of different binary trees on n nodes). Specifically $P(n) = C(n-1)$.

$$C(n) = (1/(n+1))\, C(2n, n) \in \Omega(4^n / n^{3/2})$$

where $C(2n, n)$ stands for the number of various ways to choose n items out of $2n$ items total.

DP Solution (I)

- Let $A_{i\ldots j}$ be the product of matrices i through j. $A_{i\ldots j}$ is a p_{i-1} x p_j matrix. At the highest level, we are multiplying two matrices together. That is, for any k, $1 \le k \le n-1$,

$$A_{1\ldots n} = (A_{1\ldots k})(A_{k+1\ldots n})$$

- The problem of determining the optimal sequence of multiplication is broken up into 2 parts:

  Q : How do we decide where to split the chain (what k)?

  A : Consider all possible values of k.

  Q : How do we parenthesize the subchains $A_{1\ldots k}$ & $A_{k+1\ldots n}$?

  A : Solve by recursively applying the same scheme.

  NOTE: this problem satisfies the "*principle of optimality*".

- Next, we store the solutions to the sub-problems in a table and build the table in a bottom-up manner.

DP Solution (II)

- For $1 \le i \le j \le n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i \ldots j}$.
- Example: Minimum number of multiplies for $A_{3 \ldots 7}$

$$A_1 A_2 \underbrace{A_3 A_4 A_5 A_6 A_7}_{m[3,7]} A_8 A_9$$

- In terms of p_i , the product $A_{3 \ldots 7}$ has dimensions ____.

DP Solution (III)

- The optimal cost can be described be as follows:
  - $i = j \Rightarrow$ the sequence contains only 1 matrix, so $m[i, j] = 0$.
  - $i < j \Rightarrow$ This can be split by considering each k, $i \le k < j$, as $A_{i\ldots k}$ (p_{i-1} x p_k) times $A_{k+1\ldots j}$ (p_k x p_j).
- This suggests the following recursive rule for computing $m[i, j]$:

$$m[i, i] = 0$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j) \text{ for } i < j$$

Computing *m*[*i*, *j*]

- For a specific *k*,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$$=$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j)$$

Computing $m[i, j]$

- For a specific k,

  $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$

  $= A_{i\ldots k}(A_{k+1} \ldots A_j)$ ($m[i, k]$ mults)

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j)$$

Computing $m[i, j]$

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$= A_{i\ldots k}(A_{k+1} \ldots A_j)$ ($m[i, k]$ mults)

$= A_{i\ldots k} A_{k+1\ldots j}$ ($m[k+1, j]$ mults)

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

Computing $m[i, j]$

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$$= A_{i\ldots k}(A_{k+1} \ldots A_j) \qquad (m[i, k] \text{ mults})$$

$$= A_{i\ldots k} A_{k+1\ldots j} \qquad (m[k+1, j] \text{ mults})$$

$$= A_{i\ldots j} \qquad (p_{i-1}\, p_k\, p_j \text{ mults})$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Computing $m[i, j]$

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$= A_{i \ldots k}(A_{k+1} \ldots A_j)$	($m[i, k]$ mults)
$= A_{i \ldots k} A_{k+1 \ldots j}$	($m[k+1, j]$ mults)
$= A_{i \ldots j}$	($p_{i-1}\, p_k\, p_j$ mults)

- For solution, evaluate for all k and take minimum.

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Matrix-Chain-Order(p)

$n \leftarrow length[p] - 1$
for $i \leftarrow 1$ to n // initialization: $O(n)$ time
    do $m[i, i] \leftarrow 0$
for $L \leftarrow 2$ to n // L = length of sub-chain
    do for $i \leftarrow 1$ to $n - L+1$
        do $j \leftarrow i + L - 1$
            $m[i, j] \leftarrow \infty$
            for $k \leftarrow i$ to $j - 1$
                do $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j$
                    if $q < m[i, j]$
                        then $m[i, j] \leftarrow q$
                            $s[i, j] \leftarrow k$
return m and s

Analysis

- The array $s[i, j]$ is used to extract the actual sequence (see next).
- There are 3 nested loops and each can iterate at most n times, so the total running time is $\Theta(n^3)$.

Extracting Optimum Sequence

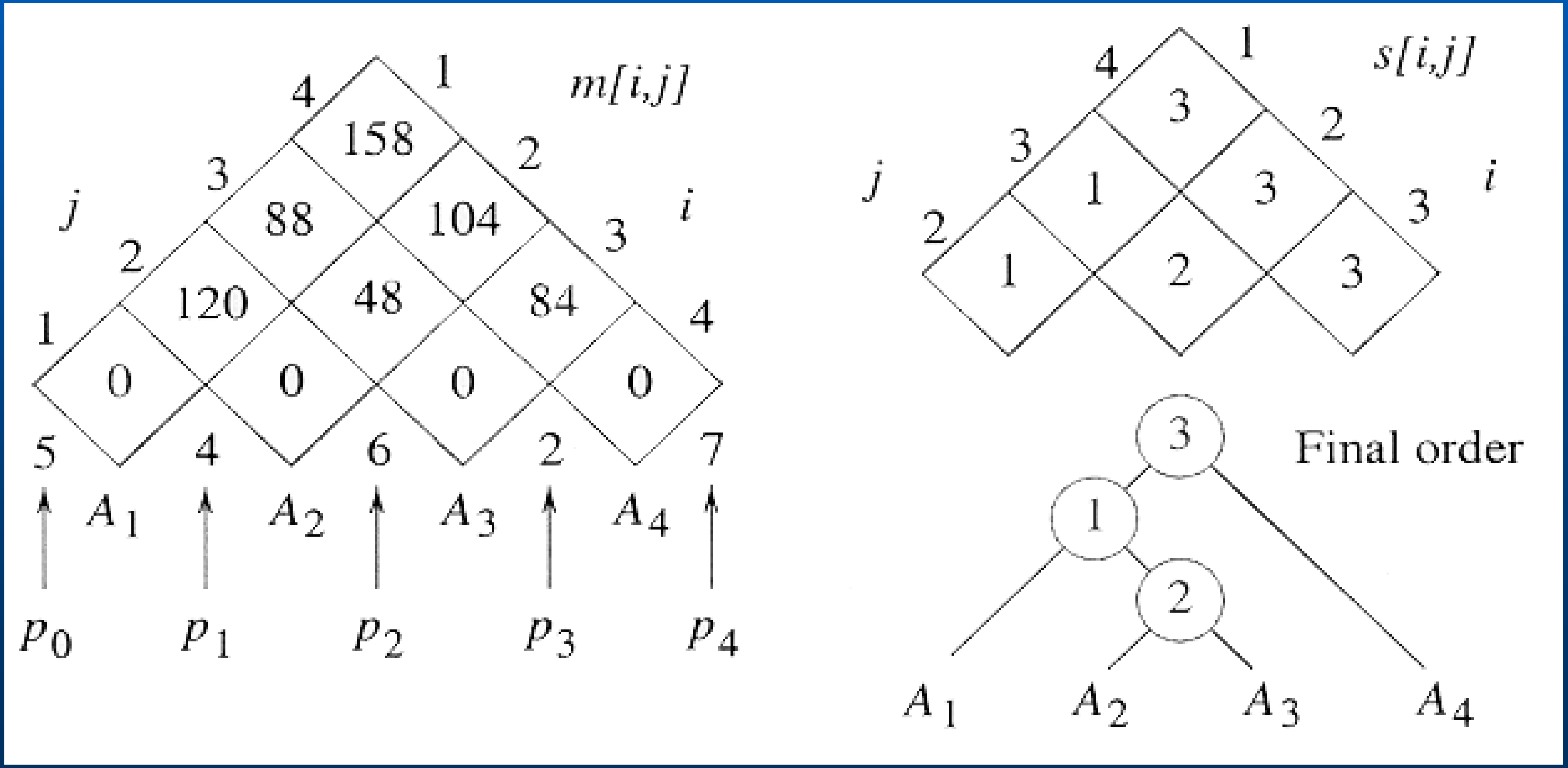
- **Leave a split marker indicating where the best split is (i.e. the value of k leading to minimum values of $m[i, j]$). We maintain a parallel array $s[i, j]$ in which we store the value of k providing the optimal split.**
- **If $s[i, j] = k$, the best way to multiply the sub-chain $A_{i\ldots j}$ is to first multiply the sub-chain $A_{i\ldots k}$ and then the sub-chain $A_{k+1\ldots j}$, and finally multiply them together. Intuitively $s[i, j]$ tells us what multiplication to perform *last*. We only need to store $s[i, j]$ if we have at least 2 matrices & $j > i$.**

Mult (A, i, j)

```
if (j > i)
    then k = s[i, j]
            X = Mult(A, i, k)           // X = A[i]...A[k]
            Y = Mult(A, k+1, j)         // Y = A[k+1]...A[j]
            return X*Y                  // Multiply X*Y
    else return A[i]                    // Return ith matrix
```

Example: DP for CMM

- The initial set of dimensions are <5, 4, 6, 2, 7>: we are multiplying A_1 (5x4) times A_2 (4x6) times A_3 (6x2) times A_4 (2x7). Optimal sequence is $(A_1\,(A_2 A_3))\,A_4$.



Finding a Recursive Solution

- Figure out the "top-level" choice you have to make (e.g., where to split the list of matrices)
- List the options for that decision
- Each option should require smaller sub-problems to be solved
- Recursive function is the minimum (or max) over all the options

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j)$$