Dynamic Programming

Dynamic Programming

- Dynamic Programming is a design principle which is used to solve problems with overlapping sub problems
- It solves the problem by combining the solutions for the sub problems
- The difference between Dynamic Programming and Divide and Conquer is that the sub problems in Divide and Conquer are considered to be disjoint and distinct various in Dynamic Programming they are overlapping.

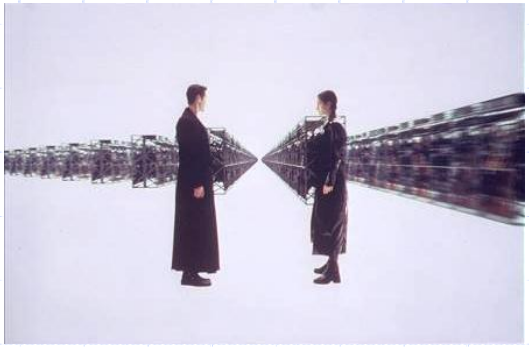
- DP is a method for solving certain kind of problems
- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

Properties of the problems that can be solved using DP

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

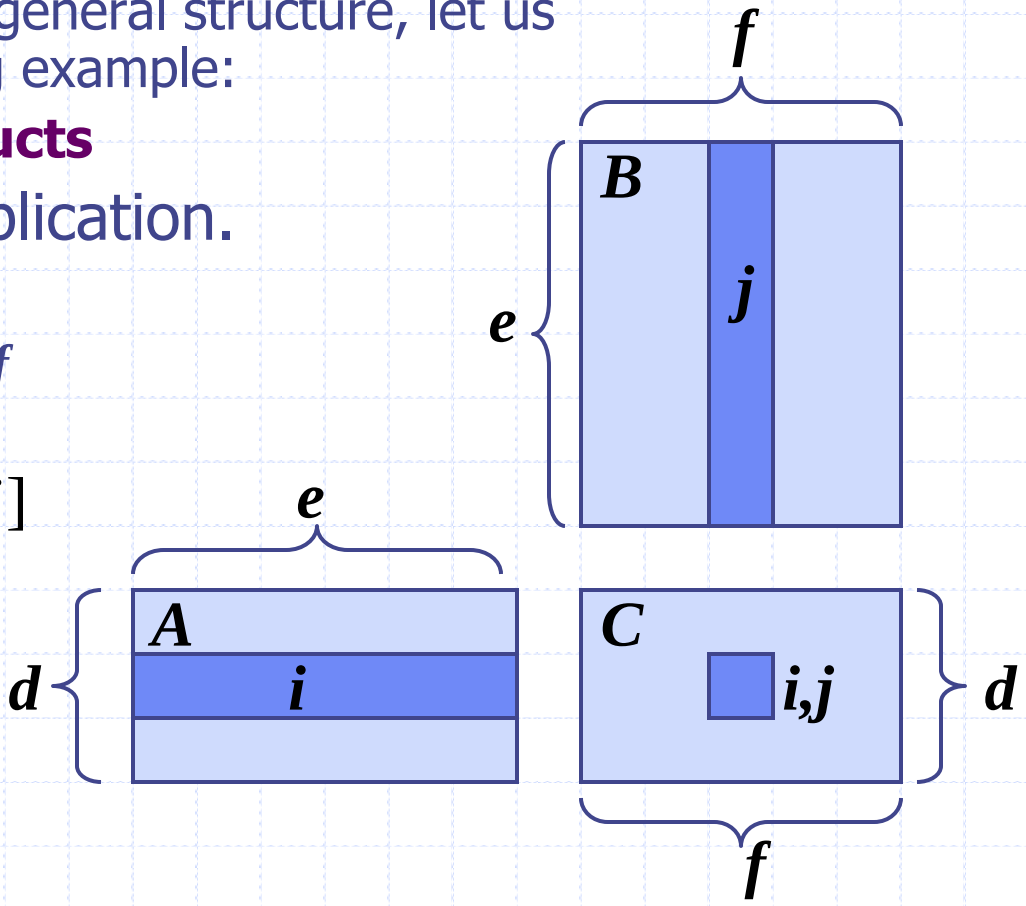
Matrix Chain-Products



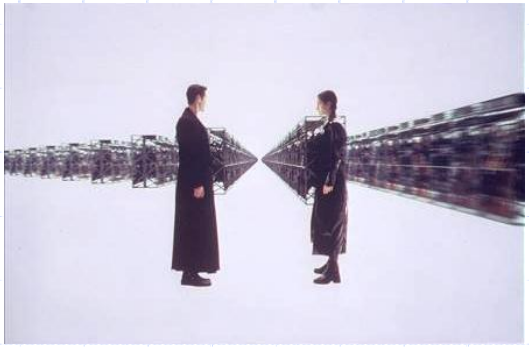
- Dynamic Programming is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**
- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

$$C[i, j] = \sum_{k=0}^{e-1} A[i,k] * B[k, j]$$

  - $O(def)$ time



Matrix Chain-Products



- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

Matrix chain product

- Want to multiply matrices
  A x B x C x D x E
- We could parenthesize many ways
  (A x (B x (C x (D x E))))
  ((((A x B) x C) x D) x E)…
- Each different way presents different number of multiplies!
- How do we figure out the wise approach?

Dynamic programming applied to matrix chain product

- Original matrix chain product
  A x B x C x D x E (ABCDE for short)
- Calculate in advance the cost (multiplies)
  AB, BC, CD, DE
- Use those to find the cheapest way to form
  ABC, BCD, CDE
- From that derive best way to form
  ABCDE

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of paranethesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Step1: the structure of an optimal parenthesization

- $A_{i...j}$: the product of matrices i through j.
  - $A_{i...j}$ 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,

    $A_{1...n} = (A_{1...k})(A_{k+1...n})$

- 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 $A_{1...k}$ & $A_{k+1...n}$?
  A : Solve by recursively applying the same scheme.

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

- ‘I’ is the subscript of the first matrix in the subproblem
- ‘j’ is the subscript of the last
- ‘k’ is the subscript for the way to break

$N_{i,j}$ into subproblems $N_{i,k}$ and $N_{k+1,j}$

Step 2: a recursive solution

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

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

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, $i \leq k < j$, as $A_{i...k}$ (p_{i-1} x p_k) times $A_{k+1...j}$ (p_k x p_j).

This suggests the following recursive rule for computing $m[i, j]$:

$$m[i, i] = 0 \quad \text{for } i = j$$

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

Step 3: Computing the optimal costs

- For a specific k,

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

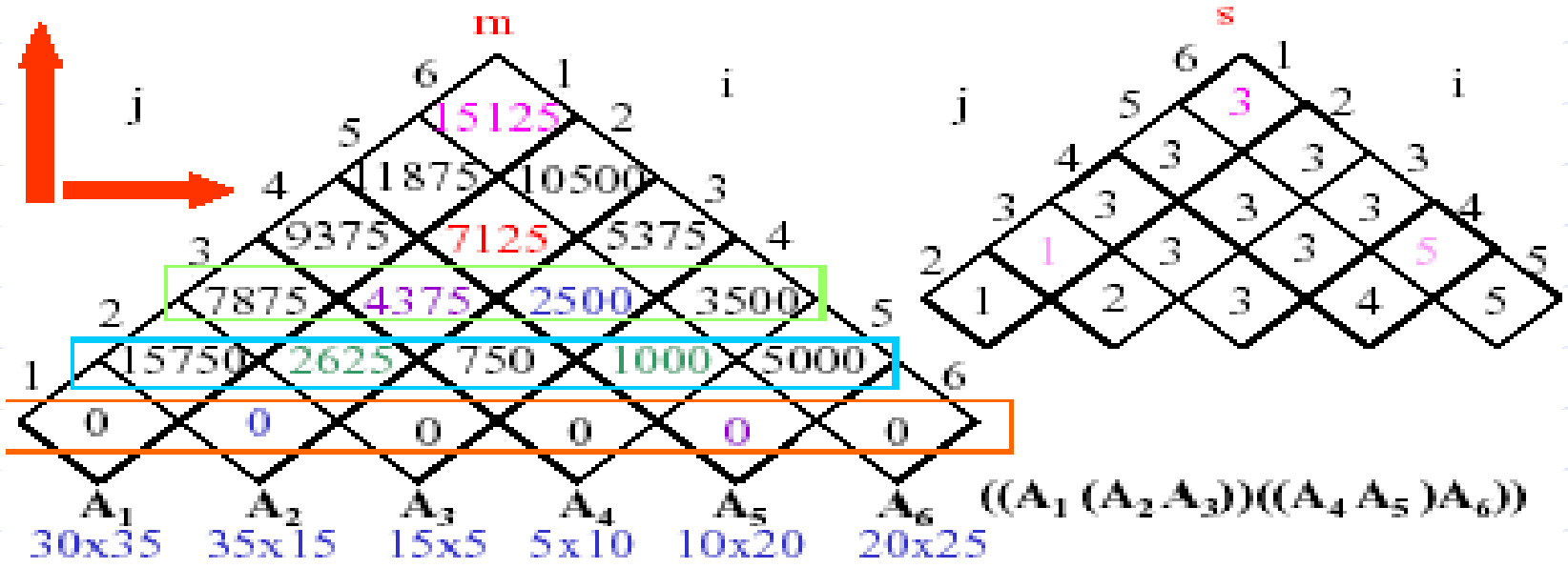
$$= A_{i \ldots k}(A_{k+1} \ldots A_j) \qquad (m[i, k] \text{ mults})$$

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

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

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

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



$$m[2,5] = \min_{k} \{ m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000,$$
$$m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125,$$
$$m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \}$$
$$= 7125$$

Matrix-Chain-Order(p)

```
1.  n ← length[p] - 1
2.  for i ← 1 to n                    // initialization: O(n) time
3.       do m[i, i] ← 0
4.  for L ← 2 to n                    // L = length of sub-chain
5.       do for i ← 1 to n - L+1
6.            do j ← i + L - 1
7.               m[i, j] ← ∞
8.               for k ← i to j - 1
9.                    do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
10.                        if q < m[i, j]
11.                         then m[i, j] ← q
12.                              s[i, j] ← k
13. 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 $\Theta(n^3)$.

Step 4: constructing an optimal solution

- 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 $A_{i..j}$ is to first multiply the sub-chain $A_{i..k}$ and then the sub-chain $A_{k+1..j}$, 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 & $j > i$.