Dynamic Programming (I)

Textbook, Chapter 16, Sections 16.1

CSCE310: Data Structures and Algorithms
`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
`choueiry@cse.unl.edu`, Tel: (402)472-5444

Dynamic Programming

solves problems by combining solutions to subproblems

Divide-&-conquer:

partition problems into independent subproblems
solve subproblem recursively
combine solutions to solve initial problem

Dynamic Prog.:

subproblems are not independent, share subsubproblems

Divide-&-conquer would solve common subsubproblems repeatedly
Dynamic Prog. would solve each subsubproblem once and save answer in a table ⟶ savings

Dynamic Programming: Typical use

Optimization problems (vs. decision problems):

- A problem has many solutions
- Each solution has a value (e.g., price, preference)
- Optimization: find solution that has best value (maximization/minimization)
- There could <u>several</u> optimal solutions
- Goal: find <u>one</u> optimal solution

How to develop: a Dynamic Programming algorithm

1. Characterize structure of optimal solution
2. Recursively define value of optimal solution
3. Compute value of optimal solution in a bottom-up fashion
4. Construct optimal solution from computed information

Step 4 can be omitted if you are looking only for <u>value</u> of optimal solution

Outline

- $\surd$ Dynamic programming for Matrix multiplication
  Goal: minimize number of scalar multiplications
- $\surd$ 2 key characteristics an optimization problem must satisfy to be solvable with Dynamic Programming
- ? How to find longest common subsequence of 2 sequences
- $\times$ Use of Dynamic Progr. to find an optimal triangulation of a convex polygon (fundamental in Computational Geometry)

Matrix-Chain multiplication

Given: a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices
Compute: their product $A_1 A_2 \ldots A_n$

How?

- One way:
  Compute A_1A_2, then $(A_1A_2)A_3$, then $(((A_1A_2)A_3)A_4)$, etc.
- Another way:
  Compute A_1A_2, and A_3A_4, then $(A_1A_2)(A_3A_4)$, etc.
- ... 5 distinct ways, full parenthesizations (FP)
  full parenthesization: single matrix or product of two FP products with parenthesis

All equivalent? Not in terms of running time

Matrix multiplication: number of scalar multiplications

```
MATRIX-MULTIPLY(A, B)
if columns[A] ≠ rows[B]
    then error "incompatible dimensions"
    else for i ← 1 to rows[A]
             do for j ← 1 to columns[B]
                    do C[i, j] ← 0
                       for k ← 1 to columns[A]
                           do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
         return C
```

Matrices: A $(p \times q)$, B $(q \times r)$
Product: AB $(p \times r)$ requires pqr scalar multiplications

Matrix multiplication: Example

Consider: $\langle A_1, A_2, A_3 \rangle$, with
dimensions of A_1 are 10×100
dimensions of A_2 are 100×5
dimensions of A_3 are 5×50

How many scalar multiplications:
in $((A_1A_2)A_3) : \ldots\ldots$
in $(A_1(A_2A_3)) : \ldots\ldots$

$\longrightarrow$ Sequence matters

Matrix-Chain Multiplication Problem (MCM)

Given: a sequence $\langle A_1, A_2, \ldots, A_n\rangle$ of n matrices each A_i has dimensions $p_{i-1} \times p_i$

Task: Fully parenthesize their product $A_1 A_2 \ldots A_n$

Objective: while minimizing the number of scalar multiplications

MCM Problem: Brute-force solution

- Compute all possible full parenthesizations (exhaustive search)
- Compute number of scalar multiplications for each
- Choose (any) one with the minimum such value

Is this a feasible solution?
Number of possible full parenthesizations (by solving recurrence)

$$P(n) = \frac{1}{n}\begin{pmatrix} 2(n-1) \\ n-1 \end{pmatrix} = \Omega(4^n/n^{3/2})$$

Exponential in n, thus avoid!

MCM Problem: Brute-force solution no good :-(

Check applicability of DP:

Step 1: Characterize structure of an optimal solution
Check whether an optimal solution contains optimal solutions to subproblems

Step 2: ...

MCM Problem: optimal solution

Notation: result of $A_i A_{i+1} \ldots A_j$ is matrix $A_{i \ldots j}$

An optimal solution:

- splits $\langle A_1, A_2, \ldots, A_n\rangle$ in $\langle A_1, \ldots, A_k\rangle$ and $\langle A_{k+1}, \ldots, A_n\rangle$ with $1 \leq k < n$
- parenthesize each of $\langle A_1, \ldots, A_k\rangle$ and $\langle A_{k+1}, \ldots, A_n\rangle$ (by splitting them somehow) to compute the 2 portions $A_{1\ldots k}$ and $A_{k+1\ldots n}$ and
- Compute product by multiplying the two matrices $A_{1\ldots k}$ and $A_{k+1\ldots n}$

MCM Problem: structure of optimal solution

An optimal solution by splitting $\langle A_1, A_2, \ldots, A_n \rangle$ in $\langle A_1, \ldots, A_k \rangle$ and $\langle A_{k+1}, \ldots, A_n \rangle$ with $1 \leq k < n$

Observation: parenthesization of each portion must be optimal

Justification: If there were a better FP $A_1 A_2 \ldots A_k$, choosing it would yield a better FP of $\langle A_1, A_2, \ldots, A_n \rangle$, which is an optimum. Contradiction

Conclusion: optimal solution contains within it optimal subsolutions

Retain: First hallmark of applicability of Dynamic Prog.
→ optimal substructure within optimal solution

Dynamic Programming: structure of optimal solution (I)

- To compute an optimal solution, compute all optimal subsolutions
- Start with all optimal subsolutions of size 1
- Compute all optimal subsolutions of size 2 using the optimal subsolutions of size 1
- Repeat until getting an optimal subsolution of size n, which is an optimal solution

Dynamic Programming: structure of optimal solution (II) (restatement)

To compute an optimal solution of a problem of size k

- Compute all solutions to problem using the optimal subsolutions of subproblems of size $k-1$
- Determine the optimum

Key question: define value of optimal solution

Step 2: Define value of an optimal solution recursively in terms of the optimal solutions to subproblems

MCM Problem: A recursive solution (I)

- Consider the chain: $\langle A_1, A_2, \ldots, A_n \rangle$
- Consider a subproblem: $A_i A_{i+1} \ldots A_j$ with $1 \leq i \leq j \leq n$
- Let $m[i,j]$ be cost of computing $A_{i \ldots j}$

Task: define $m[i,j]$ recursively

- $i = j$, $A_{i \ldots i} = A_i$ no scalar multiplication required
  $\Rightarrow m[i,i] = 0$ for $i = 1, 2, \ldots, m$
- $i < j$, exploit structure of optimal solution (**Step 1**)
  Assume $A_i, \ldots, A_j$ is split in $A_i, \ldots, A_k$ and $A_{k+1} \ldots A_j$ with $i \leq k < i$
  Then, $m[i,j] =$ min cost of computing $A_{i \ldots k}$ + that of $A_{k+1 \ldots j}$ + cost of multiplying $A_{i \ldots k}$ and $A_{k+1 \ldots j}$
  So, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ with $k = i, i+1, \ldots, j-1$
  Check $m[i,j]$ for all k ($(j-i)$ possibilities, choose the minimum

MCM Problem: A recursive solution (II)

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Let $s[i,j]$ be the value of k corresponding to the minimum

Now, $s[i,j]$ tells us where to split problem $A_iA_{i+1}\ldots A_j$ so that the cost $m[i,j]$ of computing $A_iA_{i+1}\ldots A_j$ is optimal (i.e., minimal)

MCM Problem: number of possible subproblems

Each $A_iA_{i+1}\ldots A_j$ is possible subproblem

How many subproblems, knowing $1 \leq i \leq j \leq n$?

1. for $i < j$, $\begin{pmatrix} n \\ 2 \end{pmatrix} = \frac{n(n-1)}{2}$ possibilities
   and $i = j$, n possibilities
   Thus $\frac{n(n+1)}{2} = \Theta(n^2)$ subproblems
2. Alternatively, $\sum_{x=1}^{n} x = \frac{n(n+1)}{2} = \Theta(n^2)$

We have to compute $\Theta(n^2)$ in total

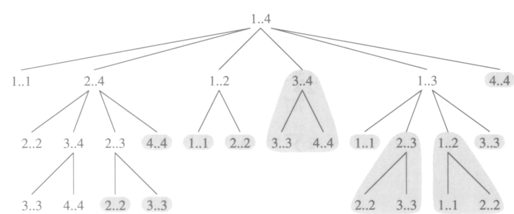
MCM Problem: optimal solution

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Two alternatives:

1. Compute $m[1,n]$ with a recursive algorithm (i.e., top-down)
   → Recursion tree
2. Compute $m[i,j]$ in a bottom-up fashion
   → **Step 3**

MCM Problem: recursion tree, example: $m[1,4]$



Recursion algorithms may encounter each subproblem many times in different branches. Its complexity is:

$$T(1) \geq 1$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \text{ for } n > 1$$

Solution: $T(n) \geq 2^{n-1} = \Omega(2^n)$, using substitution method

Exponential :-(

Optimized thanks to memoization :-)

Memoization

A technique of Computer Science to speed up programs by saving the results of computation.

The basic idea of memo functions is to accumulate a database of input/output pairs; when the function is called, it first check the database and see if it can avoid solving the problem from scratch.

Adapted from:
Artificial Intelligence: A Modern Approach
Russel & Norvig

MCM Problem: recursion algorithm

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i = j
    then return 0
m[i, j] ← ∞
for k ← i to j − 1
    do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
              + RECURSIVE-MATRIX-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
       if q < m[i, j]
          then m[i, j] ← q
return m[i, j]
```

Retain: Second hallmark of applicability of Dynamic Prog.
→ overlapping subproblems in recursive (top-down) algorithm

MCM Problem: alternative to recursive algorithm

Step 3: Compute value of an optimal solution bottom-up

- Dimensions of A_i are $p_{i-1}p_i$, for $i = 1, \ldots, n$
- Input: $\langle p_0, p_1, p_2, \ldots, p_n \rangle$ of length $n+1$
  Example: $\langle 30, 35, 15, 5, 10, 20, 25 \rangle$ is the sequence of 6 matrices of dimensions 30×35 (A_1), 35×15 (A_2), 15×5 (A_3), $5 \times 10 (A_4)$, 10×20 (A_5), 20×25 (A_6)
- Use auxiliary table $m[1 \ldots n, 1 \ldots n]$ for storing $m[i,j]$ (i.e., costs). Example: $m[6,6]$
- Use auxiliary table $s[1 \ldots n, 1 \ldots n]$ to record index k that achieved optimal cost in computing $m[i,j]$
  for constructing optimal solution, in Step 4
  Example: $s[6,6]$

MCM Problem: bottom-up algorithm

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              m[i, j] ← ∞
              for k ← i to j − 1
                  do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                     if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

MCM Problem: bottom-up algorithm (II)

Fills up m by solving the parenthesization problem on chains of increasing length:

- all chains of length $l = 1$, $m[i, i]$,
- all chains of length $l = 2$, $m[i, i+1]$,
- all chains of length $l = 3$, $m[i, i+2]$,
- all chains of length $\ldots$,
- all chains of length $l = n$, $m[i, i+n-1]$.

MCM Problem: bottom-up algorithm (III)

Lines 2-3: $m[i, i] \leftarrow 0$, chains of length 1

Line 4: l all chain lengths from 2 to n

Loop 4-12: 1. $l = 2$, we compute $m[i, i+1]$ for $1 \leq i \leq (n-1)$, that is, for $n = 6$, $m[1,2]$, $m[2,3]$, $m[3,4]$, $m[4,5]$, $m[5,6]$ ('second' diagonal)

2. $l = 3$, we compute $m[i, i+2]$ for $1 \leq i \leq (n-2)$, that is, $m[1,3]$, $m[2,4]$, $m[3,5]$, $m[4,6]$
3. $l = 4$, we compute $m[i, i+3]$ for $1 \leq i \leq (n-3)$, that is, $m[1,4]$, $m[2,5]$, $m[3,6]$
4. $l = 5$, we compute $m[i, i+4]$ for $1 \leq i \leq (n-4)$, that is, $m[1,5]$, $m[2,6]$
5. $l = 6$, we compute $m[i, i+5]$ for $1 \leq i \leq (n-5)$, that is, $m[1,6]$

Line 9: computes $m[i, j]$ using

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

For instance, for $m[2,5]$, $l = 4$, $2 \leq k < 5$ we compute the

$$\text{minimum of: } \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 \\ m[2,3] + m[4,5] + p_1p_3p_5 \\ m[2,4] + m[5,5] + p_1p_4p_5 \end{cases}$$

$m[2,2]$, $m[5,5]$ (=0) were computed when $l = 1$

$m[2,3]$, $m[4,5]$ were computed for $l = 2$

$m[2,4]$, $m[3,5]$ were computed for $l = 3$ So, for each $m[i, j]$, we need $m[i, k], m[k+1, j]$ that were computed at previous step.

Line 12: Keeps track of k corresponding to minimum in $s[i, j]$

For example, for $s[2,5] = 3$ (useful for reconstructing solution)

This means: optimal parenthesization of $A_2A_3A_4A_5$ is $(A_2A_3A_4)A_5$

Time: Loops nested three deep $(l, i, k) \rightarrow O(n^3)$ (actually $\Theta(n^3)$)

Space: $\Theta(n^2)$ to store $m[i, j]$ and $s[i, j]$

Conclusion: `Matrix-Chain-Order` is much more efficient than exponential-time brute-force solution (i.e., enumerating all possible FPs, computing their value, choosing the best one.)

MCM: Constructing optimal solution

`Matrix-Chain-Order` determines value of optimal solution but not does not directly show multiplication order

Step 4: constructs optimal solution using $s[1 \ldots n, 1 \ldots n]$

Each $s[i,j]$ records k for optimal parenthesization of $A_i A_{i+1} \ldots A_j$
$\rightarrow k$ splits $A_i A_k$ into $A_{k+1} \ldots A_j$
MCM is optimal for $A_{1 \ldots s[1,n]} A_{s[1,n]+1 \ldots n}$
Earlier matrix multiplications can be computed recursively

Optimal solution is constructed with `Matrix-Chain-Multiply`

MCM: Constructing optimal solution

First, call `Matrix-Chain-Multiply`$(A, s, 1, n)$

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
if j > i
   then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
        Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
        return MATRIX-MULTIPLY(X, Y)
   else return A_i
```


Constructing optimal solution: Example

`Matrix-Chain-Multiply`$(A, s, 1, 6)$

$i = 1, j = 6$
$X \rightarrow (A, s, 1, s[1,6] = 3)$
$i = 1, j = 3$
$X \rightarrow (A, s, 1, s[1,3] = 1)$ returns A_1
$Y \rightarrow (A, s, s[1,3] + 1 = 2, 3)$
$i = 2, j = 3$
$X \rightarrow (A, s, 2, s[2,3] = 2)$ returns A_2
$Y \rightarrow (A, s, s[2,3] + 1 = 3, 3)$ returns A_3
returns $A_2 A_3$
returns $A_1(A_2 A_3)$
$Y \rightarrow (A, s, s[1,6] + 1 = 4, 6)$
$i = 4, j = 6$
$X \rightarrow (A, s, 1, 4, s[4,6] = 5)$
$i = 4, j = 5$
$X \rightarrow (A, s, 1, 4, s[4,5] = 4)$ returns A_4
$Y \rightarrow (A, s, 1, s[4,5] + 1 = 5, 5)$ returns A_5
returns $A_4 A_5$
$Y \rightarrow (A, s, 1, s[4,6] + 1 = 6, 6)$ returns A_6
returns $(A_4 A_5) A_6$
returns $(A_1(A_2 A_3))((A_4 A5) A_6)$

Elements of Dynamic Programming

Applicability (hallmarks)

1. Optimal substructure within optimal solution
2. Overlapping subproblems in recursive (top-down) algorithm

Optimal substructure

- a problem exhibits optimal substructure when optimal solution contains within it optimal solutions to subproblems
- Strategy: assume there is a better solution to a subproblem and show this assumption contradicts optimality of the solution to original problem
- Choice of subproblem remains an art and determines performance of the algorithm

Overlapping subproblems

Space of subproblems must be small

- Better reuse few subproblems then generate and solve more subproblems
- Typically, total number of distinct subproblems polynomial in size of input

Summary

MCM problem solved either by :

- a bottom-up dynamic programming algorithm
- a top-down (recursive) memorized algorithm

in $O(n^3)$, where n is the number of matrices in chain