07. Dynamic Programming for Matrix Chain Multiplication

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Big Idea: 2D Table

- *Recall:* dynamic programming
  - problem has recursive structure
  - overlapping subproblems
  - use table to store solutions, avoid duplicated effort
  - top-down or bottom-up
- so far: **1D table** – has one index
- now: **2D table** – has *two* indices

Recap: Dynamic Programming Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
    4.1 **top-down** alternative: add table base case (**memoization**)
    4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

Rod Cutting Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

rod cutting value problem
input: an array of non-negative prices $P = \langle p_1, \ldots, p_n \rangle$
output: the maximum total price that can be achieved by cutting an n-inch rod into pieces

rod cutting problem
input: an array of non-negative prices $P = \langle p_1, \ldots, p_n \rangle$
output: the list of cut-lengths of maximum total price for an n-inch rod

Recap: Rod Cutting Step 4.b

```
function CUT-ROD-BU(P[1..n])
    Create array R[0..n]
    R[0] = 0
    for j from 1 to n do
        q = −∞
        for i from 1 to j do
            q = max(q, P[i] + R[j − i])
        end for
        R[j] = q
    end for
    return R[n]
end function
```

Moving from Values to Solutions

- **value** version of rod cutting: output is a number (total price)
- **solution** version: output is a list of cuts
- Example: for input $n = 11$, output might be $\langle 4, 4, 2, 1 \rangle$
- Naïve approach: $R[i]$ stores a list of cuts, instead of just a number

Rod Cutting Step 5 – First Draft

```
function CUT-ROD-SOLUTION(P[1..n])
    Create array R[0..n]
    R[0] = ⟨⟩                                              ▷ empty sequence
    for j from 1 to n do
        q = ⟨⟩
        for i from 1 to j do
            cut-i = R[j − i] ∪ ⟨j⟩                         ▷ copy of R[j − i] with j appended
            if TOTAL-PRICE(P, cut-i) > TOTAL-PRICE(P, q) then
                q = cut-i
            end if
        end for
        R[j] = q
    end for
    return R[n]
end function
```

Rod Cutting Step 5 – First Draft

```
function TOTAL-PRICE(P[1..n], cuts)
    x = 0
    for j in cuts do
        x = x + P[j]
    end for
    return x
end function
```

Analysis

- worst-case length of an $R[j]$ is $\Theta(n)$
    - (all 1-cuts)
- so TOTAL-PRICE takes $\Theta(n)$ time
- creating each cut-i takes $\Theta(n)$ time
- CUT-ROD-SOLUTION takes $\Theta(n^3)$ time
- **space** is an issue: CUT-ROD-SOLUTION takes $\Theta(n^2)$ space

Backtracking

- algorithm computes optimal value, and **logs (records) how it made each decision**
- after all optimal values have been computed, follow a “trail” to create solution object
- trail ends at the optimal solution
- each log entry says how to go one step backwards
- follow them until we get to the start (a base case)
- traverses solution in backwards order; reverse it if order matters
- backtracking is usually only $\Theta(n)$ time, and $\Theta(n)$ space overhead

Rod Cutting Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

- bottom-up algo. makes optimal choices with

$$q = \max(q, P[i] + R[j-i])$$

  step
- i.e. it chooses how many inches to cut right now
- **log** these choices in another array
- recall $R[j]$ = maximum total price starting from j inches
- define $S[j]$ = size of the first optimal cut starting from j inches
- need to update pseudocode to
  - create S
  - update S inside the loops
  - at the end, backtrack S to compute a list of lengths

Rod Cutting Step 5 – Pseudocode

```
function CUT-ROD-SOLUTION(P[1..n])
    Create arrays R[0..n] and S[0..n]
    R[0] = 0
    for j from 1 to n do
        q = −∞
        for i from 1 to j do
            if  q < (P[i] + R[j − i])  then
                q = P[i] + R[j − i]
                S[j] = i
            end if
        end for
        R[j] = q
    end for
    return CUT-ROD-BACKTRACK(S, n)
end function
```

Rod Cutting Step 5 – Pseudocode

```
function CUT-ROD-BACKTRACK(S[0..n], n)
    cuts = ⟨⟩                                  ▷ empty sequence
    j = n                                      ▷ remaining rod
    while j > 0 do
        cuts.append(S[j])
        j = j − S[j]
    end while
    cuts.reverse()                             ▷ put in forward order
    return cuts
end function
```

Analysis

- CUT-ROD-SOLUTION solves the *rod cutting problem*
  - it returns a list of cut-lengths, not a price
- analysis is actually straightforward
- time efficiency:
  - nested **for** loops: $\Theta(n^2)$
  - backtracking: **while** loop iterates at most n times $\Rightarrow \Theta(n)$ time
  - reverse soln: $\Theta(n)$
  - total $\Theta(n^2 + n + n) = \Theta(n^2)$ time
- space efficiency: R and S take $\Theta(n + n) = \Theta(n)$ space
- (same as the step-4 algorithms)

Matrix Multiplication

for matrices A_1, A_2 :

$$A_1 A_2$$

Recall:

$$\begin{bmatrix} 5 & 12 & 5 \\ 16 & 9 & 4 \end{bmatrix} \times \begin{bmatrix} 19 & 2 \\ 9 & 5 \\ 8 & 11 \end{bmatrix} = \begin{bmatrix} 5 \times 19 + 12 \times 9 + 5 \times 8 & 125 \\ 417 & 121 \end{bmatrix}$$

Matrix Multiplication Algorithms

Recall:

- Naïve algorithm: three nested loops, $O(n^3)$
- Strassen's algorithm: divide-and-conquer, $\approx O(n^{2.8074})$
- Those analyses assumed A_1, A_2 are both square $n \times n$ matrices
- Now: matrix sizes may differ
- **Compatible:** A_1 and A_2 are compatible when $A_1.columns = A_2.rows$

Naïve Matrix Multiplication Algorithm

```
function MATRIX-MULTIPLY(A, B)
    C = new A.rows × B.columns matrix
    for i from 1 to A.rows do
        for j from 1 to B.columns do
            c_ij = 0
            for k from 1 to A.columns do
                c_ij = c_ij + a_ik · b_kj
            end for
        end for
    end for
    return C
end function
```

Analysis: $\Theta(A.rows \times A.columns \times B.columns)$

Matrix Chain Multiplication

Given n compatible matrices $A_1, A_2, \ldots, A_n$, compute

$$A_1 A_2 \ldots A_n$$

- Recall: matrix multiplication is **associative**
- May parenthesize $A_1 A_2 \ldots A_n$ in any order
- Q: which order is most efficient?

Equivalent Parenthesizations

$$
\begin{aligned}
A_1A_2A_3A_4 &= A_1(A_2(A_3A_4)) \\
&= A_1((A_2A_3)A_4) \\
&= (A_1A_2)(A_3A_4) \\
&= (A_1(A_2A_3))A_4 \\
&= ((A_1A_2)A_3)A_4
\end{aligned}
$$

Total runtime depends on the dimensions of $A_1 \ldots A_4$.

Example: Different Runtimes

Given three matrices A_1, A_2, A_3 with dimensions

matrix	rows	columns
A_1	10	100
A_2	100	5
A_3	5	50

- $((A_1 A_2) A_3)$ costs $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5,000 + 2,500 = 7,500$ multiply operations
- $(A_1 (A_2 A_3))$ costs $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25,000 + 50,000 = 75,000$ mult. operations
- first is order of magnitude faster

Matrix Chain Multiplication Problem

matrix chain multiplication problem
input: a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns
output: a parenthesization of $A_1 A_2 \ldots A_n$ that minimizes scalar multiplications

matrix chain multiplication value problem
input: a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns
output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \ldots A_n$

Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)
   4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

Matrix Chain Multiplication Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.

matrix chain multiplication value problem
input: a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns
output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \ldots A_n$

- **solution:** parenthesized expression e.g. $(A_1(A_2A_3))(A_4A_5)$
- **value:** number of multiplications e.g. $75,000$
- goal: **value**

Matrix Chain Multiplication Step 2

2. Derive a **recurrence** for an optimal value.

- define $r_{i,j}$ = minimum number of multiplies for $A_i A_{i+1} \ldots A_j$
- (note: **two** indices)
- solution to whole problem is $r_{1,n}$
- base case: A_i by itself; so when $i = j$, $r_{i,j} = 0$
- general case:
  - **think** divide-and-conquer; define $r_{i,j}$ in terms of $r_{<i,<j}$
  - make the problem **one piece** smaller
  - given $A_i A_{i+1} \ldots A_j$, split w/ parenthesis at index k :

$$A_i A_{i+1} \ldots A_j = (A_i A_{i+1} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j)$$

  - try every option and keep the optimal one

$$r_{i,j} = \min_{i \leq k \leq j} r_{i,k} + r_{k+1,j} + p_{i-1} p_k p_j$$

Matrix Chain Multiplication Step 3

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
function MATRIX-CHAIN-VALUE-DC(p[0..n])
    return MC-DC(p, 0, n)
end function
function MC-DC(p[0..n], i, j)
    if i == j then
        return 0
    end if
    q = ∞
    for k from i to j − 1 do
        q = min(q, MC-DC(p, i, k) + MC-DC(p, k + 1, j) + p[i − 1] × p[k] × p[j])
    end for
    return q
end function
```

Sidebar: Analysis of MATRIX-CHAIN-VALUE-DC

- MC-DC-REC calls itself $O(n)$ times in general case
- like CUT-ROD-DC
- exponential time
- again, dynamic programming will circumvent all this recursion

Matrix Chain Multiplication Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)

- Recall **memoization:** use a hash dictionary to make a "memo" of pre-calculated solutions
- create hash table T
- use pair (i, j) as key in table T, storing $r_{i,j}$

Matrix Chain Multiplication Step 4.a

```
function MATRIX-CHAIN-VALUE-MEMOIZED(p[0..n])
    T = HashTable()
    return MC-M(T, p, 1, n)
end function
function MC-M(T, p[0..n], i, j)
    if T.contains((i, j)) then
        return T.get((i, j))
    end if
    if i == j then
        q = 0
    else
        q = ∞
        for k from i to j − 1 do
            q = min(q, MC-M(p, i, k) + MC-M(p, k + 1, j) + p[i − 1] × p[k] × p[j])
        end for
    end if
    T.set((i, j), q)
    return q
end function
```

Memoized Algorithm Analysis

- T contains $\Theta(n^2)$ pairs (i, j)
- each entry is inserted exactly once
- in the general case, MC-M takes $\Theta(n)$ expected time
- $\Rightarrow$ MATRIX-CHAIN-VALUE-MEMOIZED takes $\Theta(n^3)$ expected time

Matrix Chain Multiplication Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)
   4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion

- create 2D array m where $m[i][j] = r_{i,j}$
- **bottom-up:** write an explicit **for** loop that computes and stores every general case
- need to order loops so we never use an uninitialized element
- $\therefore$ initialize chain length $1(\text{base case}), 2, \ldots, n$

Matrix Chain Multiplication Step 4.b

```
function MATRIX-CHAIN-BU(p[0..n])
    Create array m[1..n][1..n]
    for i from 1 to n do
        m[i][i] = 0                                  ▷ base case, length=1
    end for
    for ℓ from 2 to n do                             ▷ ℓ = general-case length
        for i from 1 to (n − ℓ + 1) do
            j = i + ℓ − 1
            q = ∞
            for k from i to j − 1 do
                q = min(q, m[i][k] + m[k + 1][j] + p[i − 1] × p[k] × p[j])
            end for
            m[i][j] = q
        end for
    end for
    return m[1][n]
end function
```

Matrix Chain Multiplication Analysis

- MATRIX-CHAIN-BU is clearly $\Theta(n^3)$ time
- top-down memoized algorithm: $\Theta(n^3)$ expected time
- bottom-up algorithm: $\Theta(n^3)$ time with faster constant factors

Matrix Chain Multiplication Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

matrix chain multiplication value problem
input: a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of $n > 0$ compatible matrices, and sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of integers, where matrix A_i has p_{i-1} rows and p_i columns
output: the minimum number of scalar multiplies necessary to multiply $A_1 A_2 \ldots A_n$

matrix chain multiplication problem
input: (same)
output: a parenthesization of $A_1 A_2 \ldots A_n$ that minimizes scalar multiplications

Matrix Chain Multiplication Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

- **idea:** for each (i, j), record which k defines the minimum $m[i][j]$
- happens inside the inner-most k loop
- define
$$s[i][j] = \text{ the index } k \text{ that minimizes } r_{i,k} + r_{k+1,j} + p_{i-1}p_k p_j$$
- rewrite $\min(q, \ldots)$ statement as an **if** so we can update $s[i][j]$

Matrix Chain Multiplication Step 5

```
function MATRIX-CHAIN-SOLUTION(p[0..n])
    Create arrays m[1..n][1..n] and s[1..n][1..n]
    for i from 1 to n do
        m[i][i] = 0                                    ▷ base case, length=1
    end for
    for ℓ from 2 to n do                               ▷ ℓ = general-case length
        for i from 1 to (n − ℓ + 1) do
            j = i + ℓ − 1
            q = ∞
            for k from i to j − 1 do
                q′ = m[i][k] + m[k + 1][j] + p[i − 1] × p[k] × p[j]
                if q′ < q then
                    q = q′
                    s[i][j] = k
                end if
            end for
            m[i][j] = q
        end for
    end for
    return MC-BTRACK(s, 1, n)
end function
```

Matrix Chain Multiplication Step 5

```
function MC-PARENS(s[1..n][1..n], i, j)
    if i == j then                                        ▷ single matrix
        return "A_i"
    end if
    k = s[i][j]
    return "(" + MC-PARENS(s, i, k) + ")(" + MC-PARENS(s, k, j) + ")"
end function
```