Dynamic Programming II

CSci 4041: Algorithms and Data Structures

Instructor: Amir Asiaee

March 26, 2015

Dynamic Programming vs. Divide and Conquer

	Properties	Purpose	What we see
Dynamic Prog.	• Shared subprob. • Optimal substructure • Unknown cut point	Optimization	Max/Min(f1(sol.(1)), ..., fk(sol(k)))
Divide & Conquer	• Disjoint subprob. • Known cut point	General purpose	Comb(conq(1), ..., conq(k))

- Examples:
  - Merge sort: Merge(sorted(subproblem 1), sorted(subproblem 2))
  - Rod cutting: Max(p_1 + solution(subproblem 1), ..., p_n + solution(subproblem n))
- Extra work in DP: reconstruct the solution from the optimal path

Elements of Dynamic Programming

- Optimal Substructure
  - Optimal solution to the problem contains within it optimal solutions to subproblems
- Overlapping subproblems
  - Number of distinct subproblems is polynomial in the input size
  - Simple recursion revisits the same problem repeatedly
- Main idea: solve each subproblem once.
- Implementing the idea:
  - Memoization (Top Down)
    - When encounter a subproblem solve it and store the answer.
    - Not always all subproblems being solved.
  - Dynamic programming (Bottom up)
    - Start from smallest subproblem (base of recursion).

Review: Matrix Multiplication

MATRIX-MULTIPLY(A, B)

if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let C be a new $A.rows \times B.columns$ matrix
    for $i = 1$ **to** $A.rows$
        for $j = 1$ **to** $B.columns$
            $c_{ij} = 0$
            for $k = 1$ **to** $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
    return C

- Assume A is $p \times q$, B is $q \times r$
- Number of columns of A = number of rows of B
- C is $p \times r$, total pqr scalar multiplications

Matrix Chain Multiplication

- Matrix multiplication is associative: $(A_1A_2)A_3 = A_1(A_2A_3)$
- Example: $A_1A_2A_3A_4$ can be parenthesized in 5 ways

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Matrix Chain Multiplication: Problem Formulation

- Fully parenthesize the product: $A_1 A_2 \cdots A_n$
  - Each A_i has dimensions $p_{i-1} \times p_i$
  - Find parenthesization with minimum number of scalar multiplications
- Different parenthesizations have different complexity
- Consider $A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3)$
  - A_1 is 10×100, A_2 is 100×5, A_3 is 5×50
  - $((A_1 A_2) A_3)$ takes a total of $5000 + 2500 = 7500$ scalar multiplications
  - $(A_1 (A_2 A_3))$ takes a total of $25,000 + 50,000 = 75,000$ scalar multiplications

Total Number of Parenthesizations

- $P(n)$ denote the total number of parenthesis

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- Similar recurrence as Catalan numbers, which grows as $\Omega(4^n/n^{3/2})$
- Can show that the recurrence grows as $\Omega(2^n)$

Dynamic Programming Steps

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

Step 1: The Structure of an Optimal Parenthesization

- Product splits to two subproduct in each level:

$$(\underbrace{(\overbrace{A_1}\overbrace{(A_2A_3)})}\underbrace{A_4}) \tag{1}$$

Step 1: The Structure of an Optimal Parenthesization

- Product splits to two subproduct in each level:

$$\underbrace{(\overbrace{A_1}\ \overbrace{(A_2A_3))}}\ \underbrace{A_4}) \tag{1}$$

- If a number of scalar multiplication in a subproduct is not optimum the the whole parenthesization is not optimum.

$$(\underbrace{(A_1(A_2A_3))}_{\text{opt. for } A_1A_2A_3}\ A_4) \tag{2}$$

  - If the answer is optimum $(A_1(A_2A_3))$ should also be optimum, e.g. better than $((A_1A_2)A_3)$

Step 1: Formalizing Optimal Substructure

- For any chain $A_i, \ldots A_j$ that the optimum cut occurs at $i \leq k < j$ then followings subproblems also must be solved optimally:
  - $A_i, \ldots A_k$
  - $A_{k+1}, \ldots A_j$

Step 2: A Recursive Solution

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - We are after $m[1, n]$

Step 2: A Recursive Solution

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - We are after $m[1, n]$
- Recursively compute $m[i, j]$
  - Meaningless: $j \leq i$
  - Base (trivial) case: $m[i, i] = 0$, no multiplication involved for $A_{i..i} = A_i$

Step 2: A Recursive Solution

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - We are after $m[1, n]$
- Recursively compute $m[i, j]$
  - Meaningless: $j \leq i$
  - Base (trivial) case: $m[i, i] = 0$, no multiplication involved for $A_{i..i} = A_i$
  - Assume we know the cut point $i \leq k < j$ of the chain $A_i, \ldots A_j$
  - Assume we have solved all subproblems.

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \tag{3}$$

Step 2: A Recursive Solution

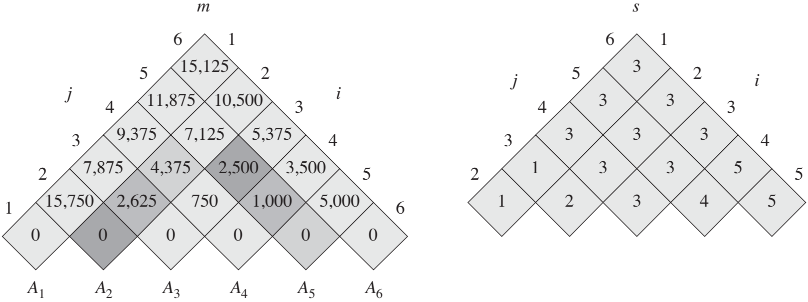
- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - We are after $m[1,n]$
- Recursively compute $m[i,j]$
  - Meaningless: $j \le i$
  - Base (trivial) case: $m[i,i] = 0$, no multiplication involved for $A_{i..i} = A_i$
  - Assume we know the cut point $i \le k < j$ of the chain $A_i, \ldots A_j$
  - Assume we have solved all subproblems.

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \tag{3}$$

- Putting it all together:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\left(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\right) & \text{if } i < j \end{cases} \tag{4}$$

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



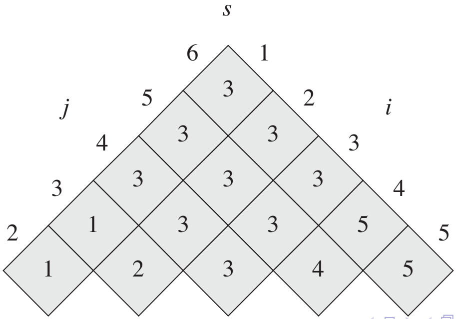
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \right) & \text{if } i < j \end{cases} \tag{5}$$

Step 3: Computing the Optimal Costs

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n              // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Step 4: Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
      print "A"_i
else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```



LCS: Longest Common Subsequent

- Give two sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$
  - $\langle B, C, A \rangle$ is a subsequent
  - $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are LCS

LCS: Longest Common Subsequent

- Give two sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$
  - $\langle B, C, A \rangle$ is a subsequent
  - $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are LCS
- Given any two sequence $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ find the LCS
- Define $X_i = \langle x_1, \ldots, x_i \rangle$

Step 1: Optimal Substructure

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 2: Recursive Solution

- $c[i, j]$ is the length of LCS for X_i and Y_j.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \,. \end{cases}$$

Step 3: Bottom up Value Computation

LCS-LENGTH(X, Y)

```
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Top down Solution Construction

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4