Dynamic Programming-3

Department of Computer Science, Tsinghua University

Outline

- Challenge I: what are the subproblems in DP?
  - Take one choice of the optimal solution away!
  - Maximum subarray (Ch4.1), Knapsack
- Challenge II: establish recurrences
  - Optimal substructure (Ch15.3)
  - Knapsack, Matrix Chain Multiplication (Ch15.2)
- Challenge III: implementations
  - Bottom-up vs. Top-down
  - Matrix Chain Multiplication, Knapsack

Recurrence of Knapsack

- Theorem (optimal substructure property): Define $z_{i,u}$ is an opt. profit for the subproblem of selecting from $a_1, \dots, a_i$ with a total weight $\leq u$. Then,
  - If a_i belongs to the opt. solution, then $z_{i,u} = z_{i-1,u-w_i} + v_i$
  - If a_i does not belong to the opt. solution, then $z_{i,u} = z_{i-1,u}$
    So we compare both cases, and take the larger one.
- Recursive solution:
- $z_{1,u} = \begin{cases} 0 & if\ u < w_1 \\ v_1 & otherwise \end{cases}$
- $z_{i,u} = \begin{cases} 0 & if\ u \leq 0 \\ z_{i-1,u} & if\ u < w_i \\ \max(z_{i-1,u}, z_{i-1,u-w_i} + v_i) & otherwise \end{cases}$

of Subproblems Knapsack

- Original Solution Space:
  - Use {0,1,1,…,0} to represent a subset of items
  - 2^n possible subsets
- # of subproblems in DP
  - Define $z_{i,u}$ is an opt. profit for the subproblem of selecting from $a_1, \dots, a_i$ with a total weight $\leq u$

Recurrence of MCM

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to computer the matrix chain $A_{i..j}$
- **Setup a recurrence for** $m[i,j]$, then the original problem: a cheapest way would thus be $m[1,n]$
- If the optimal solution of $A_{i..j}$ cuts at k
  - $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
- Try all possible k and choose the minimum
  - $m[i,j] =$
$$\begin{cases} 0 & if\ i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & if\ i < j \end{cases}$$

of Subproblems MCM

- Original Solution Space:
  - $P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$
- # of subproblems in DP
  - $m[i,j] = \begin{cases} 0 & if\ i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & if\ i < j \end{cases}$

Steps of DP

- Steps of dynamic programming
  - Step1: Characterize the structure of an optimal solution
  - Step2: Recursively define the value of an optimal solution
  - Step3: Compute the value of an optimal solution

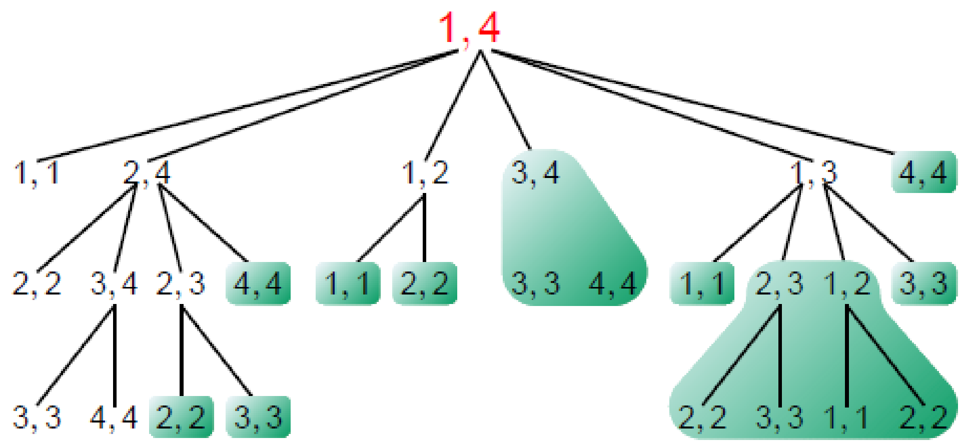
Step3: Recursive

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

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
//A_i has dimensions p_{i-1} × p_i
if i == j
     return 0
m[i, j] = ∞
for k = i to j - 1
     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]
          m[i, j] = q
return m[i, j]
```

Step3: Recursive

- However without memoization, RECURSIVE-MATRIX-CHAIN is still exponential in n.



RECURSIVE-MATRIX-CHAIN(p, 1, 4)

- $T(n) > \sum_{k=1}^{n-1}\big(T(k) + T(n-k)\big) = 2\sum_{k=1}^{n-1} T(k) \geq 2^{n-1}$

Overlapping Subproblems

- Overlapping subproblems property: We have relatively few distinct subproblems, a recursive algorithm may encounter each subproblem many times.

- Two solutions:
  - recursive algorithm with memoization: store the outputs of distinct subproblems
  - bottom-up algorithm if possible: instead of computing the solution recursively, we compute the optimal cost by using a tabular, bottom-up approach

Step3: Memoization

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length - 1
for i = 1 to n
    for j = i to n
        m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)
```

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

- MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. Each of $\theta(n^2)$ table entries is filled by just one call of LOOKUP-CHAIN, each call takes $O(n)$, excluding the time spent in computing other table entries.

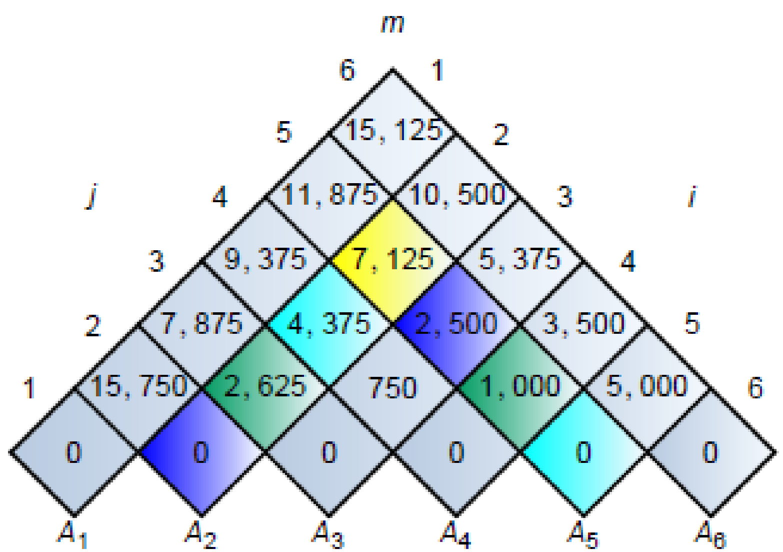
Step3: Bottom-up

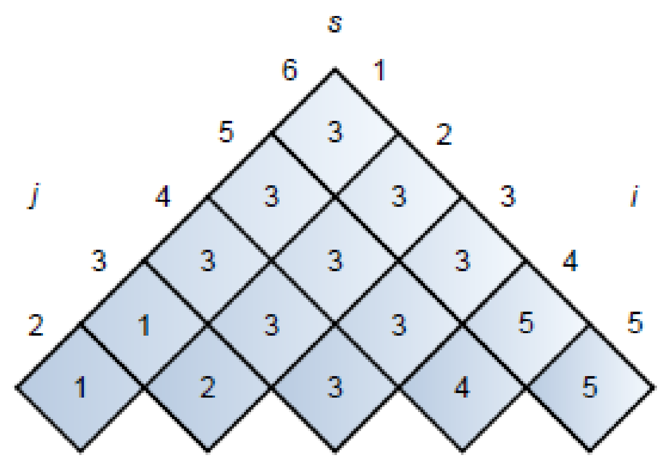
```
MATRIX-CHAIN-ORDER(p)  //A_i has dimensions p_{i-1} × p_i
n = p.length - 1
for i = 1 to n
    m[i, i] = 0
for l = 2 to n  //l is the length of subchains.
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

The number of subproblems is $\theta(n^2)$; each one has $O(n)$ choices; yielding a total running time of $O(n^3)$

Step3: an Example

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





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

Step 3: Knapsack

Example: Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$$z_{iu} = \begin{cases} 0 & if\ u \le 0 \\ z_{i-1,u} & if\ u < w_i \\ \max\left(z_{i-1,u}, z_{i-1,u-w_i} + v_i\right) & otherwise \end{cases}$$

capacity u

	i	0	1	2	3	4	5
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

Step 3:
Group Discussion

- Pros and Cons of bottom-up and top-down implementations.

Steps of DP

- Steps of dynamic programming
  - Step1: Characterize the structure of an optimal solution
  - Step2: Recursively define the value of an optimal solution
  - Step3: Compute the value of an optimal solution (bottom-up)
  - Step4: Construct an optimal solution from computed information

Step4: Constructing an optimal solution

PRINT-OPTIMAL-PARENS(s, i, j)

1 **if** $i == j$

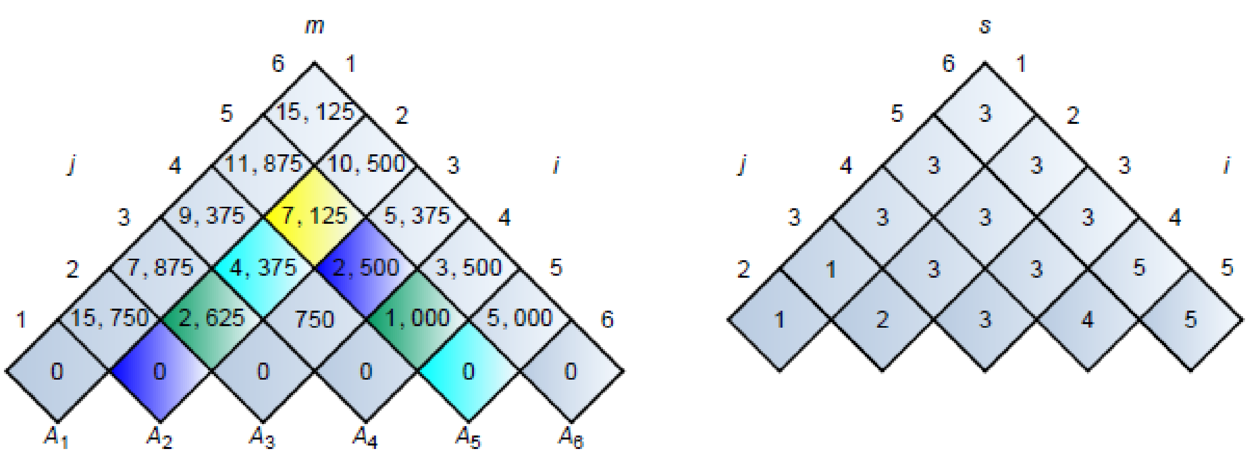
2 print "A"i

3 **else** print "("

PRINT-OPTIMAL-PARENS(s, i, $s[i, j]$)

PRINT-OPTIMAL-PARENS(s, $s[i, j] + 1$, j)

print ")"



Design Points of DP

- When should we look for dynamic-programming solution to a problem?
  - Optimal Substructure
  - Easy to construct subproblems from the solution perspective.
- Steps of dynamic programming
  - Step1: Characterize the structure of an optimal solution
  - Step2: Recursively define the value of an optimal solution
  - Step3: Compute the value of an optimal solution (bottom-up)
  - Step4: Construct an optimal solution from computed information
- Two factors decide the running time of a bottom-up implementation:
  - the number of subproblems
  - the number of choices for each subproblem

Comparison with D&C

	Divide & Conquer	Dynamic Programming
“Recursive” nature	Yes	Yes
Combine solutions to subproblems	Yes	Yes
Partition subproblems	Even partitions	Making one choice at a time
Overlapping subproblems	No	Yes
Primarily for optimization	No	Yes
Optimal substructure		Yes (to develop a recurrence)
Preprocessing	Sometimes	
Top-down vs. Bottom-up	Top-down (but…)	Bottom-up (but…)
Characteristic running time	Recurrence of running time	The space of subproblems

In-class Exercise

- T/F
  - Both D&C algorithms and DP algorithms view a problem as a collection of subproblems.
  - We normally determine the running time of both D&C algorithms and DP algorithms by solving a recurrence.