ALGORITHMS AND DATA STRUCTURES II

Lecture 9
Dynamic Programming,
Matrix chain multiplication,
The knapsack problem.

Lecturer: K. Markov
markov@u-aizu.ac.jp

1/31

DYNAMIC PROGRAMMING

- Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.
- "Programming" here means "planning".

2/31

DYNAMIC PROGRAMMING

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances.
  - solve smaller instances once and record solutions in a table.
  - extract solution to the initial instance from that table.
- Different from divide-and-conquer which partitions the problem into **independent** sub-problems and solves them recursively.

3/31

DYNAMIC PROGRAMMING

- Recall definition of Fibonacci numbers:
  - $F(n) = F(n-1) + F(n-2)$
  - $F(0) = 0$
  - $F(1) = 1$
- Computing the n^{th} Fibonacci number recursively (top-down):

$F(n)$

$F(n-1)$ + $F(n-2)$

$F(n-2)$ + $F(n-3)$ $F(n-3)$ + $F(n-4)$

. . .

4/31

DYNAMIC PROGRAMMING

- Dynamic programming applications:
  - Computing a binomial coefficient.
  - Longest common subsequence.
  - **Warshall's** algorithm for transitive closure.
  - **Floyd's** algorithm for all-pairs shortest paths.
  - Constructing an optimal binary search tree.
  - Some instances of difficult discrete optimization problems:
    - - traveling salesman problem.
    - - knapsack problem.

5/31

MATRIX CHAIN MULTIPLICATION

- Given an $l \times m$ matrix A and an $m \times q$ matrix B, the product

$$\underset{(l \times q)}{C} = \underset{(l \times m)}{A} \times \underset{(m \times q)}{B}$$

is an $l \times q$ matrix, where

$$C[i,j] = \sum_{k=1}^{m} A[i,k] \times B[k,j]$$

and the number of multiplications used to compute C is

$$l \times m \times q$$

6/31

MATRIX CHAIN MULTIPLICATION

- Assume n matrices are to be multiplied together:

$$\overset{(r_1 \times r_2)}{M_1} \quad \overset{(r_2 \times r_3)}{M_2} \quad \overset{(r_3 \times r_4)}{M_3} \quad \ldots \quad \overset{(r_{n-1} \times r_n)}{M_{n-1}} \quad \overset{(r_n \times r_{n+1})}{M_n}$$

where each matrix M_i has r_i rows and r_{i+1} columns for $1 \leq i < n$.

7/31

MATRIX CHAIN MULTIPLICATION

- The product can be computed in many different **orders**!
- Example 1: Let's compute the product of four matrices.

$$\overset{(4\times2)}{A} \times \overset{(2\times3)}{B} \times \overset{(3\times1)}{C} \times \overset{(1\times2)}{D}$$

there are 5 different ways:

1. (((A×B)×C)×D)
2. (A×((B×C)×D))
3. ((A×B)×(C×D))
4. ((A×(B×C))×D)
5. (A×(B×(C×D)))

8/31

MATRIX CHAIN MULTIPLICATION

$$\overset{A}{\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}} \times \overset{B}{\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix}} \times \overset{C}{\begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix}} \times \overset{D}{\begin{pmatrix} d_{11} & b_{11} \end{pmatrix}}$$

- Way 1: 44 multiplications

(A×B)	((A×B)×C)	(((A×B)×C)×D)
4×2×3=24	4×3×1=12	4×1×2=8

- Way 5: 34 multiplications

(C×D)	(B×(C×D))	(A×(B×(C×D)))
3×1×2=6	2×3×2=12	4×2×2=16

9/31

MATRIX CHAIN MULTIPLICATION

- Example 2: Given the matrix chain A_1, A_2, A_3, A_4 and matrix sizes:

  size(A_1) = 30×1, size(A_2) = 1×40,
  size(A_3) = 40×10, size(A_4) = 10×25.

Order of multiplications	Number of scalar multiplications
$(A_1 \times (A_2 \times (A_3 \times A_4)))$	40*10*25+1*40*25+30*1*25 =1,750
$(A_1 \times ((A_2 \times A_3) \times A_4))$	1*40*10+1*10*25+30*1*25 = 1,400
$((A_1 \times A_2) \times (A_3 \times A_4))$	30*1*40+40*10*25+30*40*25 = 41,200
$((A_1 \times (A_2 \times A_3)) \times A_4)$	1*40*10+30*1*10+30*10*25 = 8,200
$(((A_1 \times A_2) \times A_3) \times A_4)$	30*1*40+30*40*10+30*10*25 = 20,700

10/31

MATRIX CHAIN MULTIPLICATION

- The **matrix chain product problem** is to find the order of multiplying the matrices that **minimizes** the total number of multiplications used.

- We will use dynamic programming approach to find a solution for the matrix chain product problem.

11/31

MATRIX CHAIN MULTIPLICATION

- The **matrix chain product** algorithm.

$$\overset{(r_1 \times r_2)}{M_1} \quad \overset{(r_2 \times r_3)}{M_2} \quad \overset{(r_3 \times r_4)}{M_3} \quad \ldots \quad \overset{(r_{n-1} \times r_n)}{M_{n-1}} \quad \overset{(r_n \times r_{n+1})}{M_n}$$

  - First: there is only one way to compute M_1M_2 which takes $r_1 \times r_2 \times r_3$ multiplications, M_2M_3 which takes $r_2 \times r_3 \times r_4$ multiplications,..., and $M_{n-1}M_n$ which takes $r_{n-1} \times r_n \times r_{n+1}$ multiplications.
  - Next: We store these costs in a **TABLE**.

12/31

MATRIX CHAIN MULTIPLICATION

- Next: We find the best way to multiply successive triples:

  $(M_1M_2M_3),(M_2M_3M_4),\ldots,(M_{n\text{-}2}M_{n\text{-}1}M_n)$.

  The minimum cost of $(M_1M_2M_3)$ is the smaller of the cost of

1. $((M_1M_2)M_3) \rightarrow$ the cost of (M_1M_2) plus $r_1 \times r_3 \times r_4$
2. $(M_1(M_2M_3)) \rightarrow$ the cost of (M_2M_3) plus $r_1 \times r_2 \times r_4$

13/31

MATRIX CHAIN MULTIPLICATION

- When finding the costs of $((M_1M_2)M_3)$ and $(M_1(M_2M_3))$, we do not re-compute the costs of (M_1M_2) and (M_2M_3) but simply find them from the TABLE.
- The minimum costs of $(M_1M_2M_3)$, $(M_2M_3M_4),\ldots,(M_{n\text{-}2}M_{n\text{-}1}M_n)$ are also kept in the TABLE.

14/31

MATRIX CHAIN MULTIPLICATION

- Minimum Costs TABLE.

	M_1	M_2	M_3	...	M_{n-1}	M_n
M_1		M_1M_2	$M_1M_2M_3$		$M_1\ldots M_{n-1}$	$M_1\ldots M_n$
M_2			M_2M_3		$M_2\ldots M_{n-1}$	$M_2\ldots M_n$
M_3					$M_3\ldots M_{n-1}$	$M_3\ldots M_n$
...						
M_{n-1}						$M_{n-1}M_n$
M_n						

15/31

MATRIX CHAIN MULTIPLICATION

- In general, we find the best way to compute $(M_iM_{i+1}\ldots M_{i+j})$ by finding the minimum cost of computing $(M_iM_{i+1}\ldots M_{k\text{-}1})(M_k\ldots M_{i+j})$ for $i < k < j$.
- The cost of $(M_iM_{i+1}\ldots M_{k\text{-}1})(M_k\ldots M_{i+j})$ is the sum of the cost of $(M_iM_{i+1}\ldots M_{k\text{-}1})$, the cost of $(M_k\ldots M_{i+j})$, and $r_i \times r_k \times r_{i+j+1}$.

16/31

MATRIX CHAIN MULTIPLICATION

- The cost of $(M_iM_{i+1}\ldots M_{k\text{-}1})$ and the cost of $(M_k\ldots M_{i+j})$ are found from the TABLE.
- The minimum cost of $(M_iM_{i+1}\ldots M_{i+j})$ is kept in the TABLE.

17/31

MATRIX CHAIN MULTIPLICATION

- The pseudo-code is:

```
def MATRIX-CHAIN-ORDER (r):
    // r- list of matrices dimensions.
    // m - table of costs, s – optimal cost index table.
    n = r.length - 1
    m = matrix (1…n,1…n), s = matrix (1…n-1, 2…n)
    for i = 1 to n: m[i,i] = 0
    for l = 2 to n:
        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] + r_{i-1} r_k r_j
                if q < m[i,j]: m[i,j] = q; s[i,j] = k
    return m, s
```

18/31

MATRIX CHAIN MULTIPLICATION

- The pseudo-code for printing the optimal parenthesization is:

```
def PRINT-OPTIMAL-PARENS (s,i,j):
// Given s, prints the optimal parethesization.
  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 ")"
```


MATRIX CHAIN MULTIPLICATION

- Example - Cost **TABLE** with parenthesization:

[ABC][DEF] -> [A[BC]][DEF] -> [A[BC]][[DE]F]

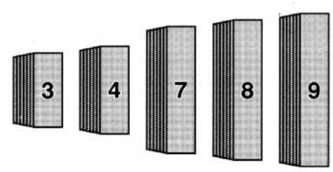
	B	C	D	E	F
A	24 [A] [B]	14 [A] [BC]	22 [ABC] [D]	26 [ABC] [DE]	36 [ABC] [DEF]
B		6 [B] [C]	10 [BC] [D]	14 [BC] [DE]	22 [BC] [DEF]
C			6 [C] [D]	10 [C] [DE]	19 [C] [DEF]
D				4 [D] [E]	10 [DE] [F]
E					12 [E] [F]

THE KNAPSACK PROBLEM

- A thief robbing a safe finds it filled with N types of items of varying size and value, but has only a small knapsack of capacity M to use to carry the goods.
- The knapsack problem is to find the combination of items which the thief should choose for his knapsack in order to **maximize the total value** of all the items it takes.

THE KNAPSACK PROBLEM

- For example, suppose that he has a knapsack of capacity 17 and the safe contains many items of the following sizes and values:



size	3	4	7	8	9
value	4	5	10	11	13
name	A	B	C	D	E

THE KNAPSACK PROBLEM

- The thief can take:
  - 5 A's that make total value of 20 (space of 2 unused)
  - 1 D and 1 E that make total value of 24 (space of 3 unused)
- But how to **maximize** the total value?

THE KNAPSACK PROBLEM

- There are many commercial situations in which a solution to the knapsack problem could be important, for example a shipping company wishing to find the **best** way to load a truck or cargo plane, etc.
- In the dynamic-programming solution to the knapsack problem, we calculate the **best** combination for all knapsacks of sizes up to M.

THE KNAPSACK PROBLEM

- In this case, ***cost*[*k*]** is the highest value that can be achieved with a knapsack of capacity ***k*** and ***best*[*k*]** is the last item that was added to achieve that maximum.
- First, we calculate the best we can do for all knapsack sizes when only items of type ***A*** are taken, then we calculate the best that we can do when only *A*'s and ***B***'s are taken, etc.

25/31

THE KNAPSACK PROBLEM

- Suppose an item ***j*** is chosen for the knapsack.
- Then the best value that could be achieved for the total would be:

  ***val*[*j*] + *cost*[*k* - *size*[*j*]].**

- If this value exceeds the best value that can be achieved without an item ***j***, then we update ***cost*[*k*]** and ***best*[*k*]**.

26/31

THE KNAPSACK PROBLEM

- Costs table for the previous example

k		3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1																
	cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2																
	cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3																
	cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4																
	cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5																
	cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

27/31

THE KNAPSACK PROBLEM

- The first pair of lines shows the best that can be done with only *A*'s.
- The second pair of lines shows the best that can be done with only *A*'s and *B*'s, etc.
- The highest value that can be achieved with a knapsack of size *17* is *24*.

28/31

THE KNAPSACK PROBLEM

- The pseudo-code to calculate the ***cost*[*k*]** and ***best*[*k*]**.

```
for j = 1 to n:
    for i = 1 to m:
        if  i >= size[j]:
            if cost[i] < cost[i-size[j]]+val[j]
                cost[i] = cost[i-size[j]]+val[j])
                best[i] = j
```

29/31

THE KNAPSACK PROBLEM

- The knapsack problem is easily solved for small size ***M***, but the computing time may become unacceptable when the ***M*** is sufficiently large.
- This method does NOT work if ***M*** and the sizes or values are real numbers instead of integers.

30/31

THAT'S ALL FOR TODAY!

31/31