BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

Febin.A.Vahab

ONLINE SESSION 9-PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
9	**Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem**	**T1: 5.3, 7.2**

Dynamic Programming

- Invented by a prominent U.S. mathematician, Richard Bellman
- The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.

Dynamic Programming

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, given problem’s solution can be related to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming

- A straightforward application of dynamic programming can be interpreted as a special variety of space-for-time trade-off.
- Optimisation of plain recursion.

Dynamic Programming-Example

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…
- F(n) = F(n-1) + F(n-2), F0 = 0 and F1 = 1

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

Dynamic Programming-Example

- Time complexity:
- $T(n) = T(n-1) + T(n-2)$
- which is exponential.

Dynamic Programming-Properties

Overlapping Sub-problems:

- Sub-problems needs to be solved again and again.
- In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use

Optimal Substructure:

- A problem can be solved by using the solutions of the sub problems

Dynamic Programming-Example

```
Algorithm DynamicFibonacci(n)
{
 f[0]=0,f[1]=1
 for(i =2;i<=n;i++)
      f[i]=f[i-1]+f[i-2]
return f[n];
}
```

Time complexity??

Dynamic Programming-Example

- A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male/female pair at the end of every month?

Dynamic Programming

- The circumstances and restrictions are not realistic.
- Still, this isn't THAT unrealistic a situation in the short term.

Dynamic Programming

- Similar to Fibonacci problem.
- To solve Fibonacci's problem, we'll let f(n) be the number of pairs during month n.
- By convention, f(0) = 0. f(1) = 1 for our new first pair.
- f(2) = 1 as well, as conception just occurred.
- The new pair is born at the end of month 2, so during month 3, f(3) = 2.
- Only the initial pair produces offspring in month 3, so f(4) = 3.

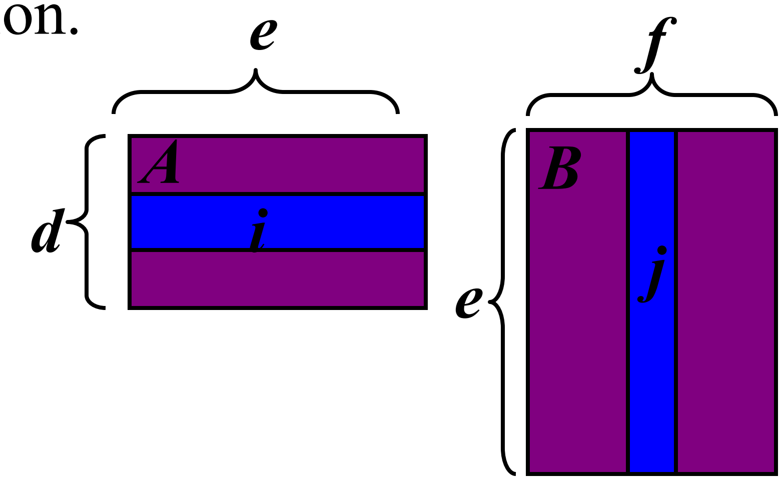
Dynamic Programming

- In month 4, the initial pair and the month 2 pair breed, so f(5) = 5. We can proceed this way, presenting the results in a table. At the end of a year, Fibonacci has 144 pairs of rabbits.

Month	0	1	2	3	4	5	6	7	8	9	10	11	12
Pairs	0	1	1	2	3	5	8	13	21	34	55	89	144

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A * B$
  - A is $d \times e$ and B is $e \times f$
  - $O(d \cdot e \cdot f)$ time



$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

$2 \times 3 \times 2 = 12$

A $\quad m \times n \quad 2 \times 3$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

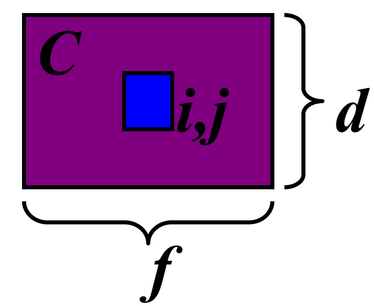
B $\quad n \times p \quad 3 \times 2$

$$\begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

$m \times p$

$m \times n \times p$

$$C = \begin{bmatrix} 7+18+33 & 8+20+36 \\ 28+45+66 & 32+50+72 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$



Matrix Multiplication

Matrix Chain-Products

- Matrix multiplication is associative
- $B \cdot (C \cdot D) = (B \cdot C) \cdot D$.
- Thus, we can parenthesize the expression for multiplication any way we wish and we will end up with the same answer.
- **Number of primitive (that is, scalar) multiplications in each parenthesization, however, might not be the same.**

Matrix Chain-Products

- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

BC = 3x5

5x5

3 x100x5 = 1500

\+

BC D = 3x5

= 3x5x5 = 75 = 1575

CD = 100x5

= 100x5x5 = 2500

BxCD = (3x100) * (100x5) = 3x5

= 3x100x5 = 1500

= 4000

Matrix Chain-Products

- **Matrix Chain-Product:**
- Suppose we are given a collection of n two-dimensional matrices for which we wish to compute the product
  - Compute $\mathbf{A=A_1*\ldots*A_n}$
  - A_i is $d_{i-1} \times d_i$ matrix, for i = 1 , 2, . . . , n .

    d_0 d_1 d_2 d_3 d_4
  - ie. Input A[]={10,20,30,40,50}
  - A1 is10x20 matrix,A2=20X30 matrix,A3 is 30X40 matrix and A4 is 40X50 matrix.
  - Problem: How to parenthesize?

Matrix Chain Product

- Input A[]={10,20,30,40,50}
- A1=10 X 20
- A2= 20 X 30
- A3=30 X 40
- A4= 40 X 50
- A_i is $d_{i-1} \times d_i$ Matrix
- $A_{i..j} = A_{i..k}$ X $A_{k+1..j}$
- $A_{1..4} = (A_{1..2})$ X $(A_{3..4})$

$A_{1..3} = A_1 \times A_2 \times A_3$

$A_{1..1} = A_1$

$A_{1..3} = A_{1..2} \times A_{3..3}$

$A_{1..4} = A_{1..2} \times A_{3..4}$

$A_{1..3} = A_{1..1} \times A_{2..3}$

$$A_{i..j} = \underbrace{A_{i..k} \times A_{k+1..j}}$$

$A \times B \times C$

$$\left.\begin{array}{l} A_i = \boxed{d_{i-1}} \times d_i \\ A_k = d_{k-1} \times d_k \end{array}\right\} \; d_{i-1} \times d_k$$

$$\left.\begin{array}{l} A_{k+1} = d_k \times d_{k+1} \\ A_j = d_{j-1} \times \boxed{d_j} \end{array}\right\} \; d_k \times d_j$$

$$\left.\begin{array}{l} d_{i-1} \times d_k \\ d_k \times d_j \end{array}\right\} \; d_{i-1} \times d_j$$

Matrix Chain-Products

innovate achieve lead

- The Matrix Chain-Products problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed.
- The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Matrix Chain-Products-Enumeration Approach

- **Matrix Chain-Product Algorithm:**
  - Try all possible ways to parenthesize $A=A_1*\ldots*A_n$
  - Calculate number of ops for each one
  - Pick the one that is best

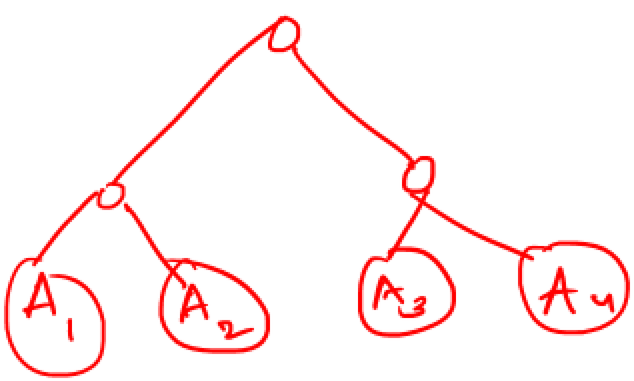
$$
\begin{aligned}
A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) \\
&= A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\
&= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4)
\end{aligned}
$$

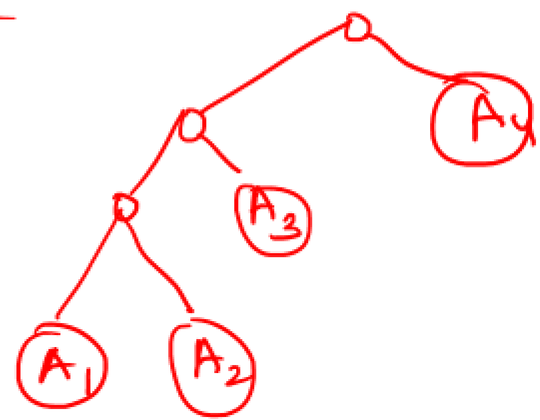
Matrix Chain-Products-Enumeration Approach

- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes.
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

$(A_1 \cdot A_2)(A_3 \cdot A_4)$

$((A_1 \cdot A_2) A_3) A_4$





Number of Binary trees with n nodes

- Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = $\boxed{(2n)!/(n+1)!*n!}$
- For n = 0,1, 2, 3, … values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …. So are numbers of Binary Search Trees.
- Total number of possible Binary Trees with n different keys (countBT(n)) = countBST(n) * n!

Matrix Chain-Products-Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Matrix Chain-Products-Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

Example-Dynamic Programming

$A_1 = 5 \times 4$

$A_2 = 4 \times 6$

$A_3 = 6 \times 2$

$A_4 = 2 \times 7$

$A_1 \; A_2 \; A_3 \; A_4$

$d_0 \; d_1 \; d_2 \; d_3 \; d_4$

$\{5, 4, 6, 2, 7\}$

$A_{1..4}$ ie $A_1 \times A_2 \times A_3 \times A_4$

m	1	2	3	4
1	0	120 $A_1 \times A_2$		
2		0		
3			0	
4				0

① $m[1,1]$ = number of x-tions needed to compute $A_{1..1}$ ie $A_1 = \underline{\underline{0}}$

III^ly $m[2,2] = 0$

$m[3,3] = 0$

$m[4,4] = 0$

S	1	2	3	4
1		1		
2				
3				
4				

II) Multiply 2 matrices

$m[1,2]$ = number of x-tions needed to compute $A_{1..2}$ ie $A_1 \times A_2$

$= 5 \times 4 \times 6 = 120$

Example-Dynamic Programming

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$
$A_4 = 2 \times 7$

$m[2,3] = A_{2..3}$ ie $A_2 \times A_3$
$= 4 \times 6 \times 2 = \underline{\underline{48}}$

$m[3,4] = A_{3..4}$ ie $A_3 \times A_4$
$= 6 \times 2 \times 7 = \underline{\underline{84}}$

m	1	2	3	4
1	0	120	✓	
2		0	48	
3			0	84
4				0

III <u>Multiply 3 matrices</u>

$\underline{m[1,3]} = A_{1..3}$ ie $A_1 \times A_2 \times A_3$

2 possibilities (i) $A_1 \cdot (A_2 \times A_3)$ ✓

(ii) $\underline{(A_1 \cdot A_2)} \cdot A_3$

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

(i) $\underset{5\times4}{A_1} \cdot (\underset{4\times2}{A_2 \cdot A_3})$

// $m[1,1] + m[2,3] + (5 \times 4 \times 2) = 0 + 48 + 40$
$= 88$ //

Example-Dynamic Programming

innovate achieve lead

$A_1 = 5\times4$
$A_2 = 4\times6$
$A_3 = 6\times2$
$A_4 = 2\times7$

(ii) $\overset{5\times6}{(A_1 \cdot A_2)} \cdot \overset{6\times2}{A_3}$

$= m[1,2] + m[3,3] + \text{cost of } (A_1 \cdot A_2) \times A_3$

$= 120 + 0 + 5\times6\times2$

$= 120 + 60 = \underline{180}$

$\underline{m[2,4]} = \underline{A_{2..4}} = \underline{A_2 \times A_3 \times A_4}$

$\underset{4\times6}{A_2} \cdot \underset{6\times7}{(A_3 \cdot A_4)}$

$m[2,2] + m[3,4] + \underline{4\times6\times7}$

$= 0 + 84 + 168 = \underline{252}$

$\underset{(4\times2)}{(A_2 \cdot A_3)} \; \underset{(2\times7)}{A_4}$

$= m[2,3] + m[4,4] + \underline{4\times2\times7}$

$= 48 + 0 + 56$

$= \underline{104}$

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

innovate achieve lead

Example-Dynamic Programming

IV

$d = \{5, 4, 6, 2, 7\}$ (d_0, d_1, d_2, d_3, d_4)

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$
$A_4 = 2 \times 7$

Multiply 4 matrices

$$m[1,4] = \min \begin{bmatrix} m[1,1] + m[2,4] + 5 \times 4 \times 7, \\ m[1,2] + m[3,4] + 5 \times 6 \times 7, \\ m[1,3] + m[4,4] + 5 \times 2 \times 7 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 104 + 140, \\ 120 + 84 + 210, \\ 88 + 0 + 70 \end{bmatrix} = \min \begin{bmatrix} 244, \\ 414, \\ 158 \end{bmatrix} = 158$$

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

$A_{1..4} = A_1 \cdot (A_{2..4})$
$= (A_{1..2})(A_{3..4})$
$= (A_{1..3}) \cdot A_4$

$$m[i,j] = [m[i,k] + m[k+1,j] + d_{i-1} \times d_k \times d_j]$$

Example-Dynamic Programming

$A_1 = 5 \times 4$

$A_2 = 4 \times 6$

$A_3 = 6 \times 2$

$A_4 = 2 \times 7$

$(A_1 \; A_2 \; A_3) \; A_4$

k value at $(1,4] = 3$

$(A_1 \; A_2 \; A_3) \; A_4$

k value at $\underline{[1,3]}$

$((A_1)\,(A_2\,A_3))\,(A_4)$

m

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s

	1	2	3	4
1			1	3
2				
3				
4				

Example-Dynamic Programming

Example-Dynamic Programming

Example-Dynamic Programming

Matrix Chain-Products-Dynamic Programming

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{1,n}$.

Matrix Chain-Products-Dynamic Programming

- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_1 * \ldots * A_i) * (A_{i+1} * \ldots * A_n)$.
  - Then the optimal solution $N_{1,n}$ is the sum of two optimal subproblems, $N_{1,i}$ and $N_{i+1,n}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

Matrix Chain-Products-Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_{i-1} \times d_i$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$\boxed{\mathbf{N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_{i-1} d_k d_j\}}}$$

$$N_{i,i} = 0$$

Matrix Chain Products-Dynamic Programming Algorithm


```
Matrix-Chain(p, 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] + p[i − 1] * p[k] * p[j];
                if (q < m[i, j])
                {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s; (Optimum in m[1, n])
}
```

Matrix Chain-Products-Dynamic Programming Algorithm

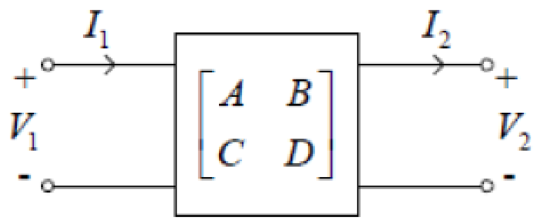
- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry.

Matrix Chain Products-Dynamic Programming Algorithm

- Since subproblems overlap, we don’t use recursion.
- Instead, we construct optimal subproblems “bottom-up.”
- $N_{i,i}$’s are easy, so start with them
- Then do problems of “length” 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

Matrix Chain Products-Applications

- **Perspective projections**, which is the foundation for 3D animation-Orthographic projection (sometimes orthogonal projection) is a means of representing three-dimensional objects in two dimensions.
- Minimum and Maximum values of an expression with * and +
- **Network analysis (electrical circuits)- Network analysis** is the process of finding the voltages across, and the currents through, every component in the network. For example, when two or more N-port networks are connected in cascade, the combined network is the product on the individual ABCD matrices
- Used extensively in NLP and Machine learning: Principal Component Analysis and Singular Value Decomposition



Read about "Word to Vectors—Natural Language Processing"

Matrix Chain Products-Applications

The ABCD Matrix is defined as:

$$\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} V_2 \\ I_2 \end{bmatrix}$$

For which:

$$A = \left.\frac{V_1}{V_2}\right|_{I_2=0}, \quad B = \left.\frac{V_1}{I_2}\right|_{V_2=0}$$

$$C = \left.\frac{I_1}{V_2}\right|_{I_2=0}, \quad D = \left.\frac{I_1}{I_2}\right|_{V_2=0}$$

"The usefulness of the ABCD matrix is that cascaded two port networks can be characterized by simply multiplying their ABCD matrices"

Matrix Chain Products-Applications

- Used extensively in NLP and Machine learning

Read about "Word to Vectors—Natural Language Processing"

- **sentence**=" Word Embeddings are Word converted into numbers "
- A *word* in this **sentence** may be "Embeddings" or "numbers " etc.
- A *dictionary* may be the list of all unique words in the **sentence.**
- So,a dictionary may look like ['Word', 'Embeddings', 'are', 'Converted', 'into', 'numbers']
- A ve*ctor* representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else.
- The vector representation of "numbers" in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

Matrix Chain Products-Applications

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =['He','She','lazy','boy','Neeraj','person']

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

	He	She	lazy	boy	Neeraj	person
D1	1	1	2	1	0	0
D2	0	0	1	0	1	1

Matrix Chain Products-Applications

Co-occurrence matrix.

Corpus = He is not lazy. He is intelligent. He is smart.

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

Principal Component Analysis and Singular Value Decomposition

THANK YOU!

BITS Pilani
Hyderabad Campus