DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Module: Dynamic Programming

Radhika Chapaneri

Sem V

- **<u>Longest Common Subsequence Problem</u>**: Given 2 sequences X, Y, find maximum-length common subsequence Z.
- Application
- Biologists need to measure similarity between DNA and thus determine how closely related an organism is to another.
- They do this by considering DNA as strings of letters A,C,G,T (adenine, guanine, cytosine or thymine) and then comparing similarities in the strings.
- Formally, they look at common subsequences in the strings. each element of set is a base:

Longest Common subsequence

- Compare DNA similarities
- S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
- One measure of similarity:
  - find the longest string S_3 containing bases that also appear (not necessarily *consecutively*) in S_1 and S_2
  - S_3 = GTCGTCGGAAGCCGGCCGAA

Example

- Example X = ABCBDAB, Y=BDCABA
- **A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous**
- Subsequences may be: ABA, BCA, BCB, BBA
  BCBA
  BDAB etc.

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

But the Longest Common Subsequences (LCS) are **BCBA** and **BDAB**.

- How to find LCS efficiently?

LCS: Setup for Dynamic Programming

- Let *c[i,j]* to be the length of LCS of sequences X_i and Y_j
- Then the length of LCS of X and Y will be *c[m,n]*
- *Recursive Formula for LCS is:*

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

LCS ALGORITHM

- LCS-LENGTH takes two sequences X =(x_1,x_2,…..x_m) and Y= (y_1,y_2,…y_m) as inputs.
- It stores the c[i, j] values in a table c[0….m,0…n], and it computes the entries in ***row-major*** order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.)
- The procedure also maintains the table b[1….m, 1…n] to help us construct an optimal solution.
- The procedure returns the b and c tables
- c[m, n] contains the length of an LCS of X and Y .

LCS ALGORITHM

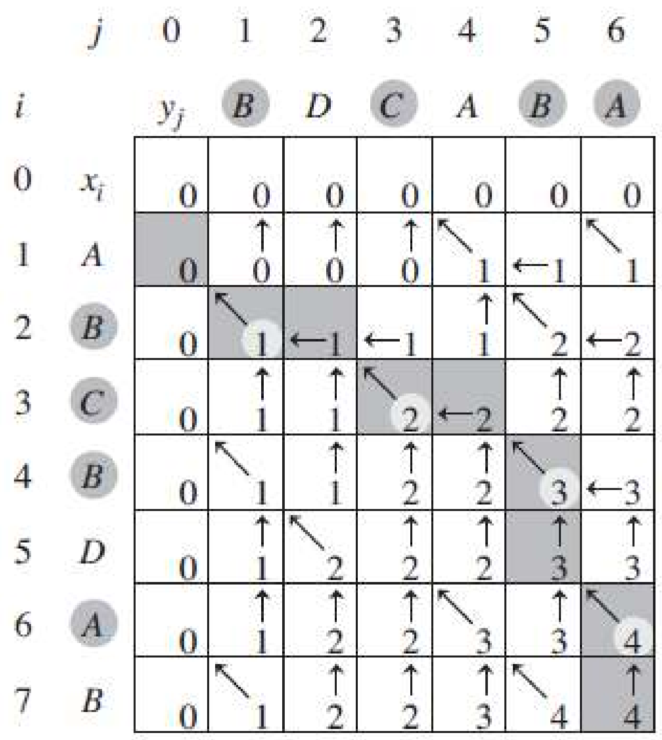
```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if $x_i$ == $y_j$
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif 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
```

LCS ALGORITHM

- The following recursive procedure prints out an LCS of X and Y in the proper, forward order.
- Initial call is PRINT-lCS(b,X,X.*length*,Y.*length*)

```
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)
```

LCS Problem



	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

Complexity

- The running time of LCS length procedure is O(mn) since each table entry takes O(1) time to compute.

Dynamic Programming

- Dynamic Programming is also used in optimization problems.
- Similar to divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems.
- Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are
- **overlapping sub-problems**
- **optimal substructure**.

- **Overlapping sub-problems**
- Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems.
- It is mainly used where the solution of one sub-problem is needed repeatedly.
- The computed solutions are stored in a table, so that these don't have to be re-computed.
- Hence, this technique is needed where overlapping sub-problem exists.
- For example, Fibonacci numbers have many overlapping sub-problems.

- **Optimal Sub-Structure**
- A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
- For example, the Shortest Path problem has the following optimal substructure property −
- If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.
- Example : All pair shortest path

All-Pairs Shortest Path Problem

Suppose we are given a directed graph G=(V,E,W)

The All-Pairs Shortest Path Problem asks to find the length of the shortest path between any pair of vertices in G.

Need of all pair shortest path

If the weight function is nonnegative for all edges, then we can use Dijkstra's single source shortest path algorithm for all vertices to solve the problem. But negative weight are not allowed in Dijkstra's algorithm.

For arbitrary weight functions, we can use the Bellman-Ford algorithm applied to all vertices. This yields an $O(n^4)$ algorithm for graphs with n vertices.

Floyd-Warshall

We will now investigate a dynamic programming solution that solved the problem in $O(n^3)$ time for a graph with n vertices. The graph can contain negative weight edges but not negative weight cycle.

This algorithm is known as the Floyd-Warshall algorithm,

Notations

We assume that the input is represented by a weight matrix $W = (w_{ij})_{i,j \text{ in } E}$ that is defined by

$w_{ij} = 0$	if $i=j$
$w_{ij} = w(i,j)$	if $i \neq j$ and (i,j) in E
$w_{ij} = \infty$	if $i \neq j$ and (i,j) not in E

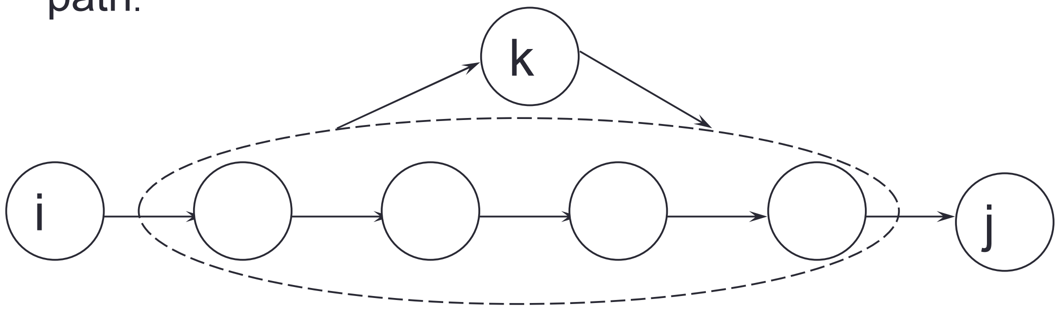
If the graph has n vertices, we return a distance matrix (d_{ij}), where d_{ij} the length of the path from i to j.

Intermediate Vertices

- we assume that V={1,2,…,n}, i.e., that the vertices of the graph are numbered from 1 to n.
- Given a path p=(v_1, v_2,…, v_m) in the graph, we will call the vertices v_k with index k in {2,…,m-1} the intermediate vertices of p.

Floyd Warshall Algorithm

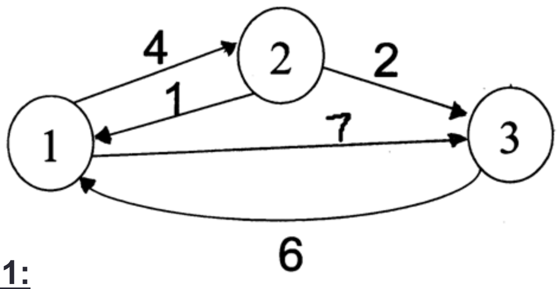
- Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2,…,k} only.
- There are two situations:
  1) k is an intermediate vertex on the shortest path.
  2) k is not an intermediate vertex on the shortest path.



Floyd Warshall Algorithm - Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \boxed{10} & 0 \end{bmatrix}$$

Consider Vertex 1:
D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & \boxed{6} \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:
D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

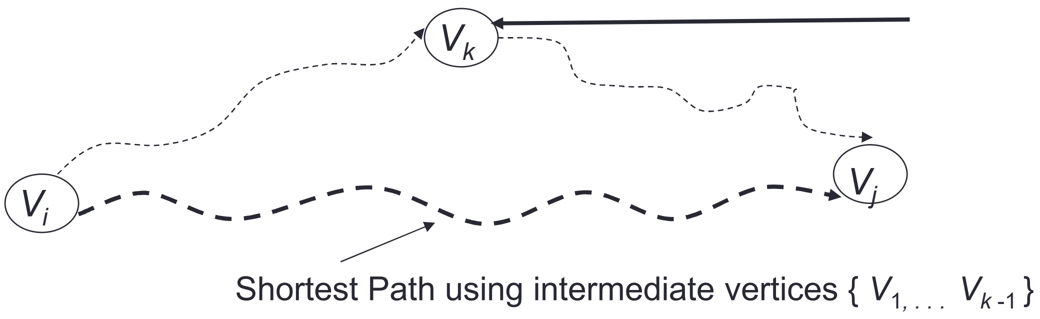
Consider Vertex 3:
Nothing changes.

- Let $d_{ij}^{(k)}$ denote the length of the shortest path from i to j such that all intermediate vertices are contained in the set {1,…,k}.
- Consider a shortest path p from i to j such that the intermediate vertices are from the set {1,…,k}.

- If the vertex k is not an intermediate vertex on p, then
- $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

If the vertex k is an intermediate vertex on p, then

$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$


V_k
V_i
V_j
Shortest Path using intermediate vertices { $V_{1,...}$ V_{k-1} }

Recursive Formulation

If we do not use intermediate nodes, i.e., when k=0, then

$d_{ij}^{(0)} = w_{ij}$

If k>0, then

$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$

The Floyd-Warshall Algorithm

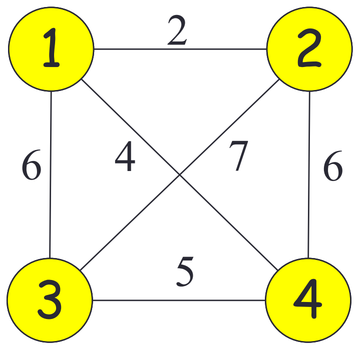
```
Floyd-Warshall(W)
n = # of rows of W;
$D^{(0)}$ = W;
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$;

  return $D^{(n)}$;
```

Time and Space Requirements

The running time is obviously $O(n^3)$.

Travelling Salesman Problem

- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.



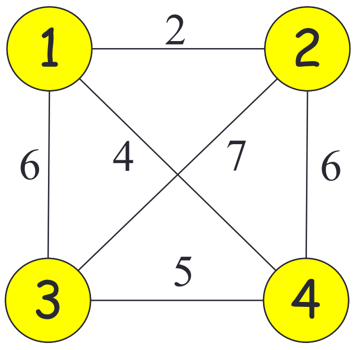
Brute Force approach

- 1) Consider city 1 as the starting and ending point.
  2) Generate all (n-1)! Permutations of cities.
  3) Calculate cost of every permutation and keep track of minimum cost permutation.
  4) Return the permutation with minimum cost.

Application of TSP

- The TSP arises as a subproblem in many transportation and logistics applications, for example the problem of arranging school bus routes to pick up the children in a school district
- Although transportation applications are the most natural setting for the TSP, the simplicity of the model has led to many interesting applications in other areas.
- A classic example is the scheduling of a machine to drill holes in a circuit board or other object. In this case the holes to be drilled are the cities, and the cost of travel is the time it takes to move the drill head from one hole to the next.
- The technology for drilling varies from one industry to another, but whenever the travel time of the drilling device is a significant portion of the overall manufacturing process then the TSP can play a role in reducing costs

Example (TSP)



(12341) = 18

(12431) = 19

(13241) = 23

(13421) = 19

(14231) = 23

(14321) = 18

Time Complexity: (n!)

Dynamic programming apparoach

Subproblem Formulation for TSP

Let the tour be a simple path that start and ends at vertex 1

$g(1, V-\{1\})$ length of the optimal TSP tour.

$$g(1,V-\{1\}=\min_{2\le k\le n}\left\{d_{ik}+g(k,V-\{1,k\})\right\}$$

Dynamic programming approach

Subproblem Formulation for TSP

Let the tour be a simple path that start and ends at vertex 1

$g(i, S)$ length of the shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1.

$g(1, V-\{1\})$ length of the optimal TSP tour.

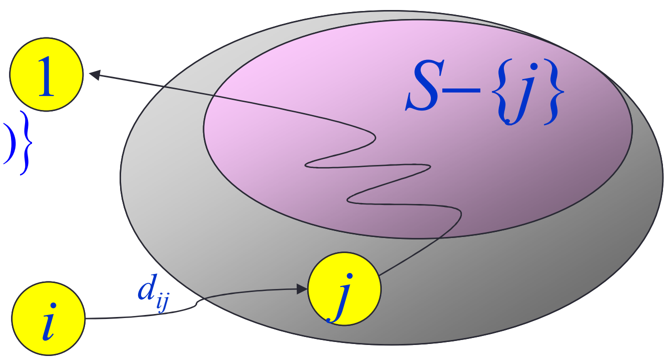
$$g(1,V-\{1\}=\min_{2\le k\le n}\left\{d_{ik}+g(k,V-\{1,k\})\right\}$$

Subproblem Formulation for TSP

Goal: $g(1,\ V-\{1\})$

$g(i,\ S)$ length of the shortest path from i to 1 visiting each city in S exactly once.

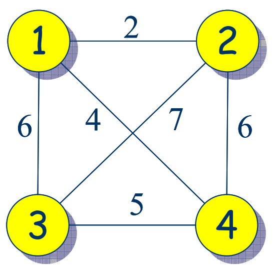
$$g(i,S)=\min_{j\in S}\left\{d_{ij}+g(j,S-\{j\})\right\}$$



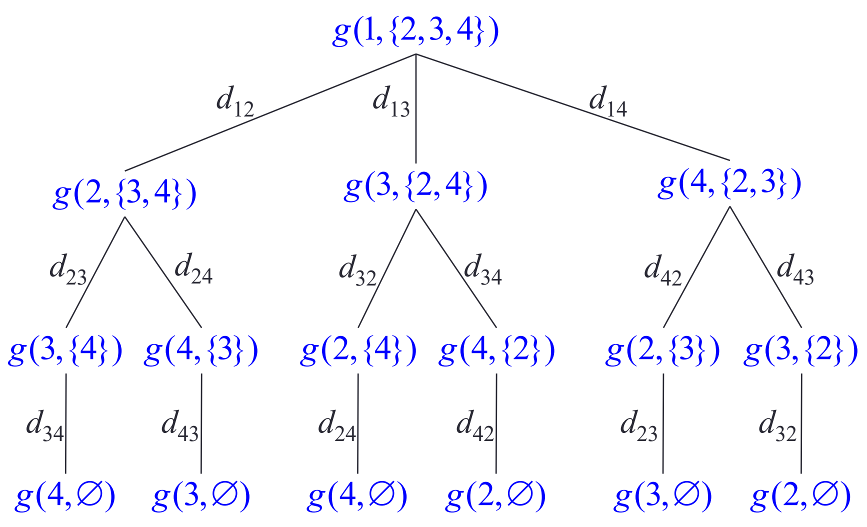
Example

Goal: $g(\underline{1},\ V-\{1\})$

$$g(i,S)=\min_{j\in S}\left\{d_{ij}+g(j,S-\{j\})\right\}$$



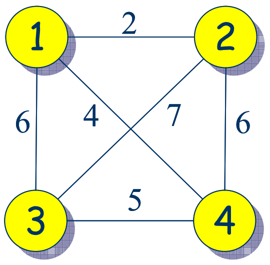
$$D=\left[d_{ij}\right]_{4\times 4}=\begin{bmatrix}\infty & 2 & 6 & 4\\ 2 & \infty & 7 & 6\\ 6 & 7 & \infty & 5\\ 4 & 6 & 5 & \infty\end{bmatrix}$$

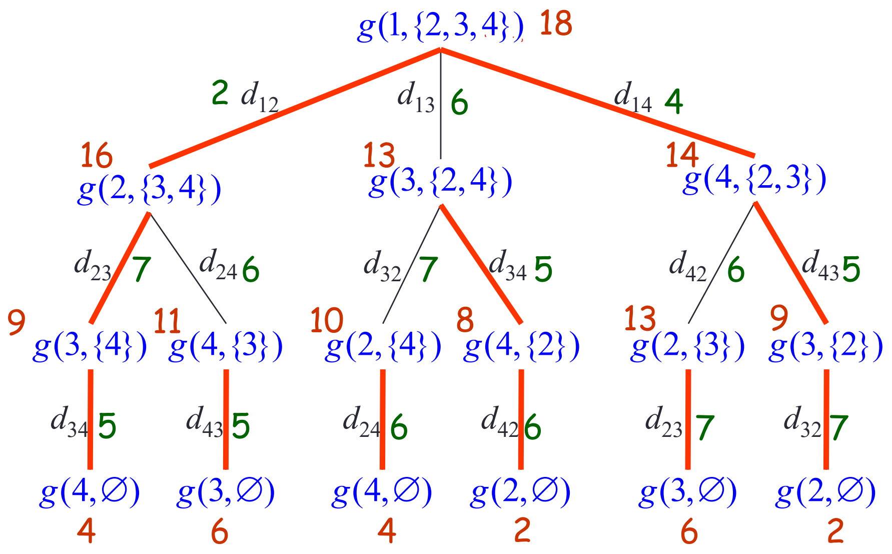


Example

Goal: $g(1, V - \{1\})$

$$g(i,S) = \min_{j \in S} \{d_{ij} + g(j, S - \{j\})\}$$





$$D = [d_{ij}]_{4\times4} = \begin{bmatrix} \infty & 2 & 6 & 4 \\ 2 & \infty & 7 & 6 \\ 6 & 7 & \infty & 5 \\ 4 & 6 & 5 & \infty \end{bmatrix}$$

Optimal tour will be 1-2-3-4-1 or 1-4-3-2-1

$$D = [d_{ij}]_{4\times4} = \begin{bmatrix} \infty & 2 & 6 & 4 \\ 2 & \infty & 7 & 6 \\ 6 & 7 & \infty & 5 \\ 4 & 6 & 5 & \infty \end{bmatrix}$$

$$g(1, V - \{1\} = \min_{2 \le k \le n} \{d_{ik} + g(k, V - \{1,k\})\}$$

- For problem solving refer notebook

Complexity

- The time complexity is O($n^2 * 2^n$).
- The time complexity is much less than O(n!), but still exponential.
- So this approach is also infeasible even for slightly higher number of vertices.

Matrix chain multiplication

Matrix: An $n \times m$ matrix $A = [a[i,j]]$ is a two-dimensional array

$$A = \begin{bmatrix} a[1,1] & a[1,2] & \cdots & a[1,m-1] & a[1,m] \\ a[2,1] & a[2,2] & \cdots & a[2,m-1] & a[2,m] \\ \vdots & \vdots & & \vdots & \vdots \\ a[n,1] & a[n,2] & \cdots & a[n,m-1] & a[n,m] \end{bmatrix},$$

which has n rows and m columns.

Example: The following is a 4×5 matrix:

$$\begin{bmatrix} 12 & 8 & 9 & 7 & 6 \\ 7 & 6 & 89 & 56 & 2 \\ 5 & 5 & 6 & 9 & 10 \\ 8 & 6 & 0 & -8 & -1 \end{bmatrix}.$$

Matrix chain multiplication

The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

for $1 \le i \le p$ and $1 \le j \le r$.

Example: If

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

then

$$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}.$$

Given a $p \times q$ matrix A and a $q \times r$ matrix B, the direct way of multiplying $C = AB$ is to compute each

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

for $1 \le i \le p$ and $1 \le j \le r$.

Complexity of Direct Matrix multiplication:

Note that C has pr entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

Matrix multiplication is **associative** , e.g.,

$$A_1A_2A_3 = (A_1A_2)A_3 = A_1(A_2A_3),$$

so parenthenization does not change result.

Given a $p \times q$ matrix A, a $q \times r$ matrix B and a $r \times s$ matrix C, then ABC can be computed in two ways $(AB)C$ and $A(BC)$:

The number of multiplications needed are:

$$\begin{aligned} mult[(AB)C] &= pqr + prs, \\ mult[A(BC)] &= qrs + pqs. \end{aligned}$$

When $p = 5$, $q = 4$, $r = 6$ and $s = 2$, then

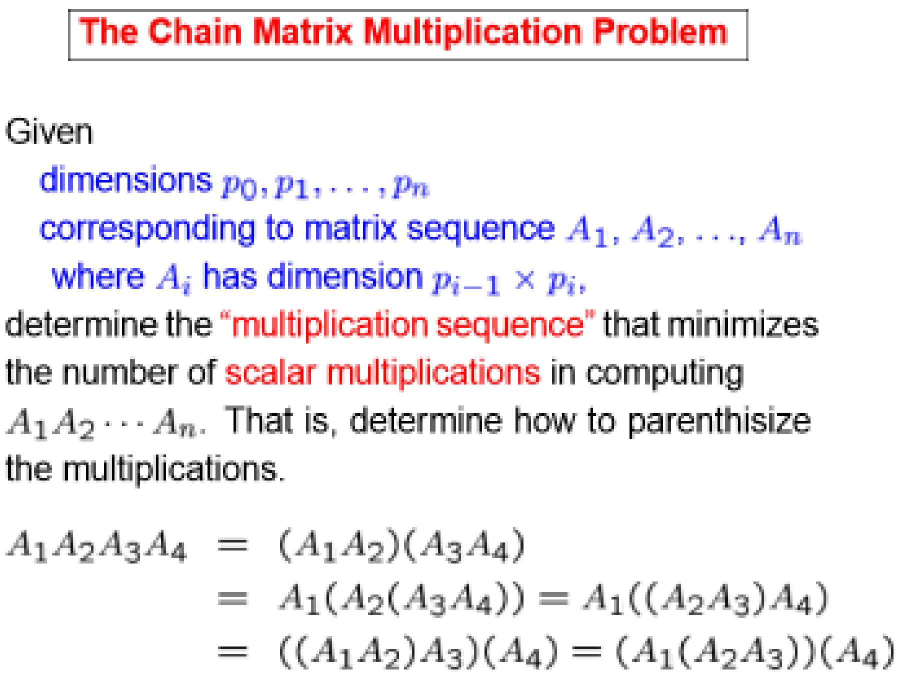
$$\begin{aligned} mult[(AB)C] &= 180, \\ mult[A(BC)] &= 88. \end{aligned}$$

A big difference!

Implication: The multiplication “sequence” (parenthesization) is important!!

- Our goal is to determine an order for multiplying matrices that has lowest cost

Our goal is to determine an order for multiplying matrices that has lowest cost



The Chain Matrix Multiplication Problem

Given
dimensions $p_0, p_1, \ldots, p_n$
corresponding to matrix sequence $A_1, A_2, \ldots, A_n$
where A_i has dimension $p_{i-1} \times p_i$,
determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1 A_2 \cdots A_n$. That is, determine how to parenthisize the multiplications.

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

Use dynamic programming

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution bottom-up
4. Construct an optimal solution from the computed information

Structure of an optimal solution If the outermost parenthesization is

$$((A_1 A_2 \cdots A_i)(A_{i+1} \cdots A_n))$$

then the optimal solution consists of solving A_{1i} and $A_{i+1,n}$ optimally and then combining the solutions.

- Recursive define the solution

If the final multiplication for A_{ij} is $A_{ij} = A_{ik}A_{k+1,j}$ then

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

We don't know k a priori, so we take the minimum

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

3. Compute the value of an optimal solution bottom-up
4. Construct an optimal solution from the computed information

Example for the Bottom-Up Computation

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

- Initialization
-

j

i

	1	2	3	4
1	0			
2		0		
3			0	
4				0

m [i, j]

j

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

s [i, j]

Stp 1: Computing $m[1,2]$ By definition

$$m[1,2] = \min_{1\le k<2}(m[1,k]+m[k+1,2]+p_0p_kp_2) = m[1,1]+m[2,2]+p_0p_1p_2 = 120.$$

j (columns), i (rows)

	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

m [i, j]

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

s [i, j]

Stp 2: Computing $m[2,3]$ By definition

$$m[2,3] = \min_{2\le k<3}(m[2,k]+m[k+1,3]+p_1p_kp_3) = m[2,2]+m[3,3]+p_1p_2p_3 = 48.$$

j (columns), i (rows)

	1	2	3	4
1	0	120		
2		0	48	
3			0	
4				0

m [i, j]

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

s [i, j]

Stp3: Computing $m[3,4]$ By definition

$$\begin{aligned} m[3,4] &= \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4) \\ &= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84. \end{aligned}$$

-

j

i

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

m [i, j]

j

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

s [i, j]

Stp4: Computing $m[1,3]$ By definition

$$\begin{aligned} m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\ &= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\ &= 88. \end{aligned}$$

-

j

i

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

m [i, j]

j

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

s [i, j]

Stp5: Computing $m[2,4]$ By definition

$$\begin{aligned} m[2,4] &= \min_{2\le k<4}(m[2,k]+m[k+1,4]+p_1p_kp_4) \\ &= \min\left\{\begin{array}{l} m[2,2]+m[3,4]+p_1p_2p_4 \\ m[2,3]+m[4,4]+p_1p_3p_4 \end{array}\right\} \\ &= 104. \end{aligned}$$

j

i

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

m [i, j]

j

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

s [i, j]

St6: Computing $m[1,4]$ By definition

$$\begin{aligned} m[1,4] &= \min_{1\le k<4}(m[1,k]+m[k+1,4]+p_0p_kp_4) \\ &= \min\left\{\begin{array}{l} m[1,1]+m[2,4]+p_0p_1p_4 \\ m[1,2]+m[3,4]+p_0p_2p_4 \\ m[1,3]+m[4,4]+p_0p_3p_4 \end{array}\right\} \\ &= 158. \end{aligned}$$

j

i

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

m [i, j]

j

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

s [i, j]

- Construct an optimal solution fro s[i,j] matrix

$$\begin{tikzcd}
 & s(1,4) \arrow[dl, dash] \arrow[dr, dash] & \\
s(1,3) & & s(4,4)
\end{tikzcd}$$

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

- Refer Notebook.

Algorithm

```
Matrix-Chain-Order(p)
n ← length[p] − 1
for i ← 1 to n
     do m[i,i] ← 0
for l ← 2 to n                    ▷ l is the chain length.
     do for i ← 1 to n − l + 1
            do j ← i + l − 1
               m[i,j] ← ∞
               for k ← i to j − 1
                   do q ← m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
                      if q < m[i,j]
                         then m[i,j] ← q
                              s[i,j] ← k
return m and s
```

Complexity = O(n^3)

```
PRINT-OPTIMAL-PARENS$(s, i, j)$
1  if $i == j$
2      print "$A$"$_i$
3  else print "("
4      PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
5      PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
6      print ")"
```