Unit - IV
Dynamic Programming

Contents

Introduction of Dynamic Programming

- Dynamic Programming is the most powerful design technique for solving **optimization problems**.
- Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.
- Dynamic Programming is used when the **subproblems are not independent**, e.g. when they share the same subproblems. **In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.**

Introduction of Dynamic Programming...

- Dynamic Programming solves **each subproblems just once and stores the result in a table** so that it can be repeatedly retrieved if needed again.
- Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to **obtain solutions** for bigger problems.
- Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the **"principle of optimality"**.

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

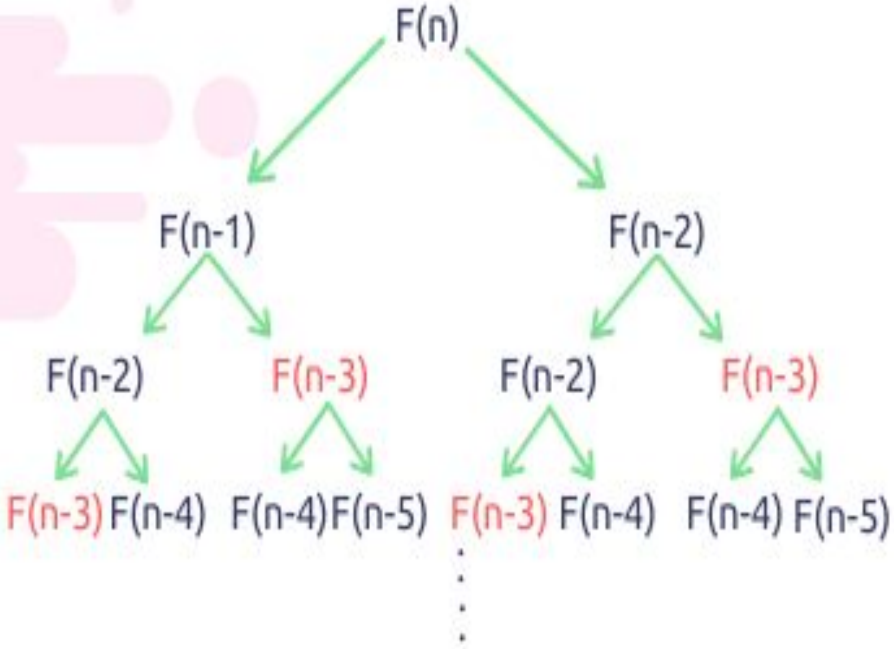
Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

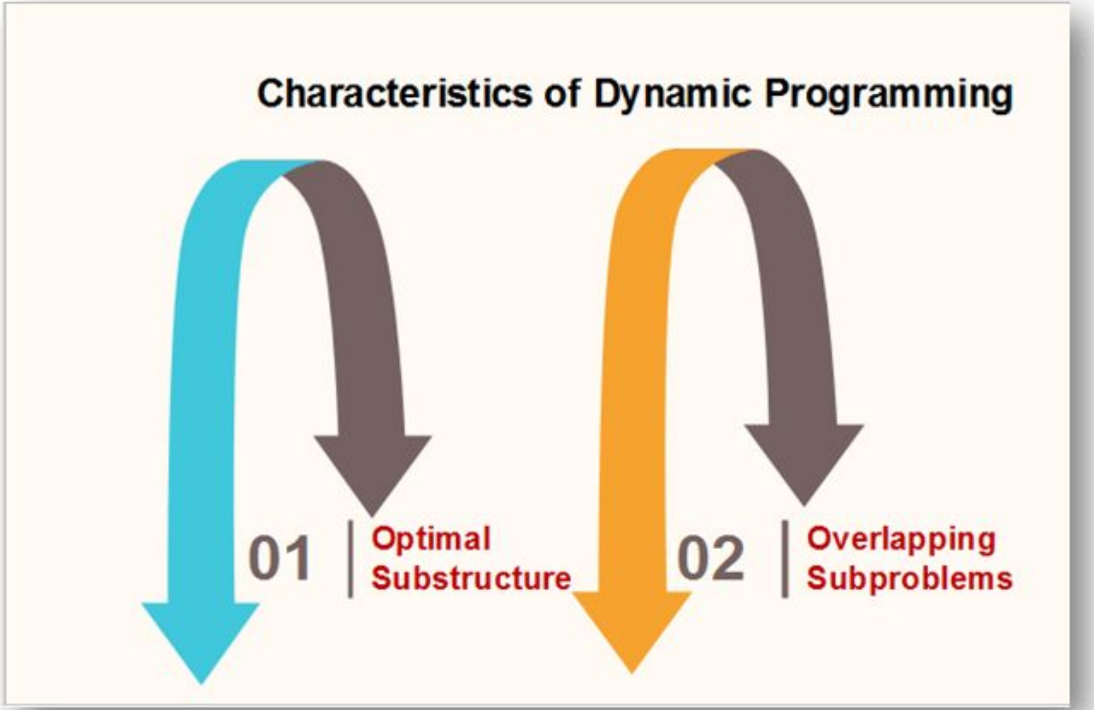
Dynamic Programming : Linear



Fibonacci Recursion and Dynamic Programming

Characteristics of Dynamic Programming

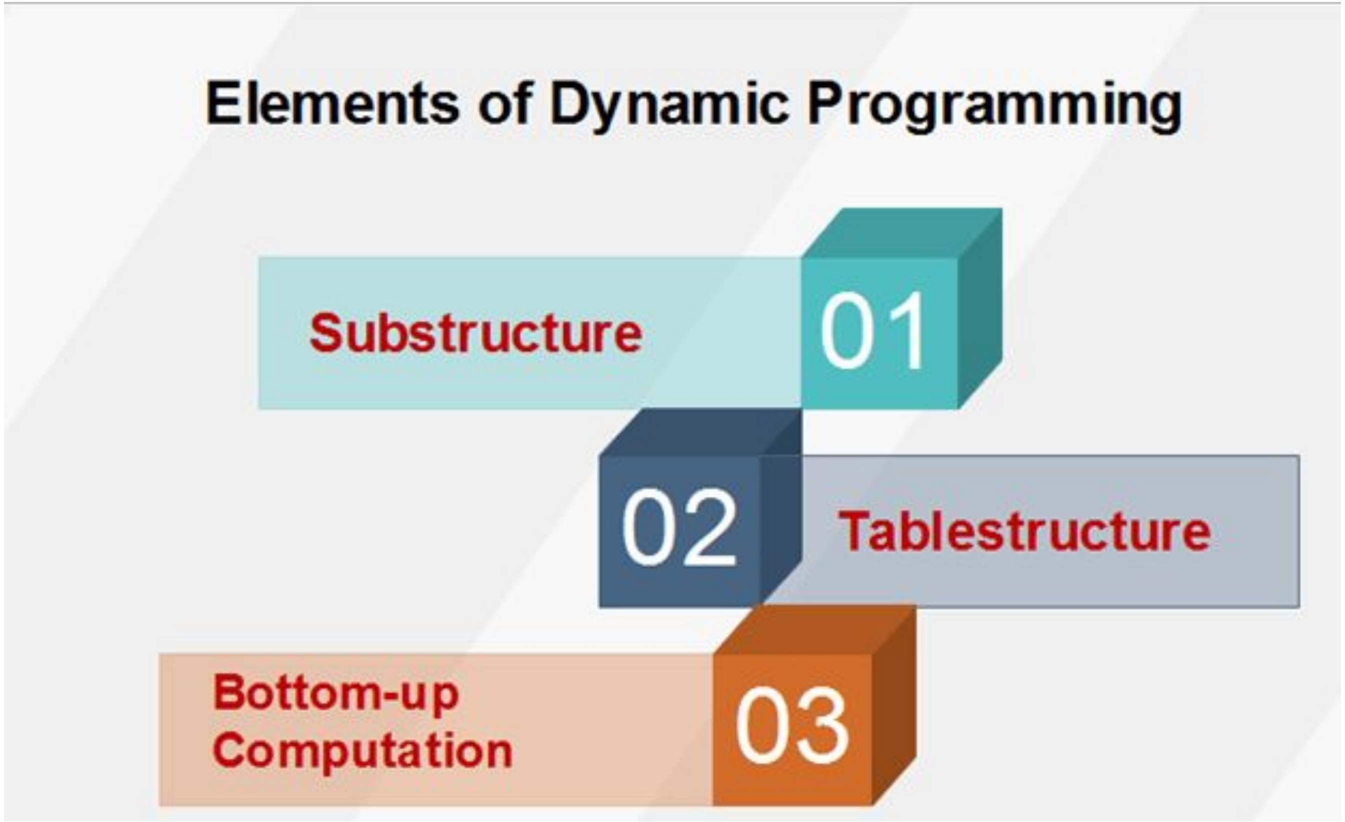
Dynamic Programming works when a problem has the following features:-



Characteristics of Dynamic Programming

1. **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
2. **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

Elements of Dynamic Programming



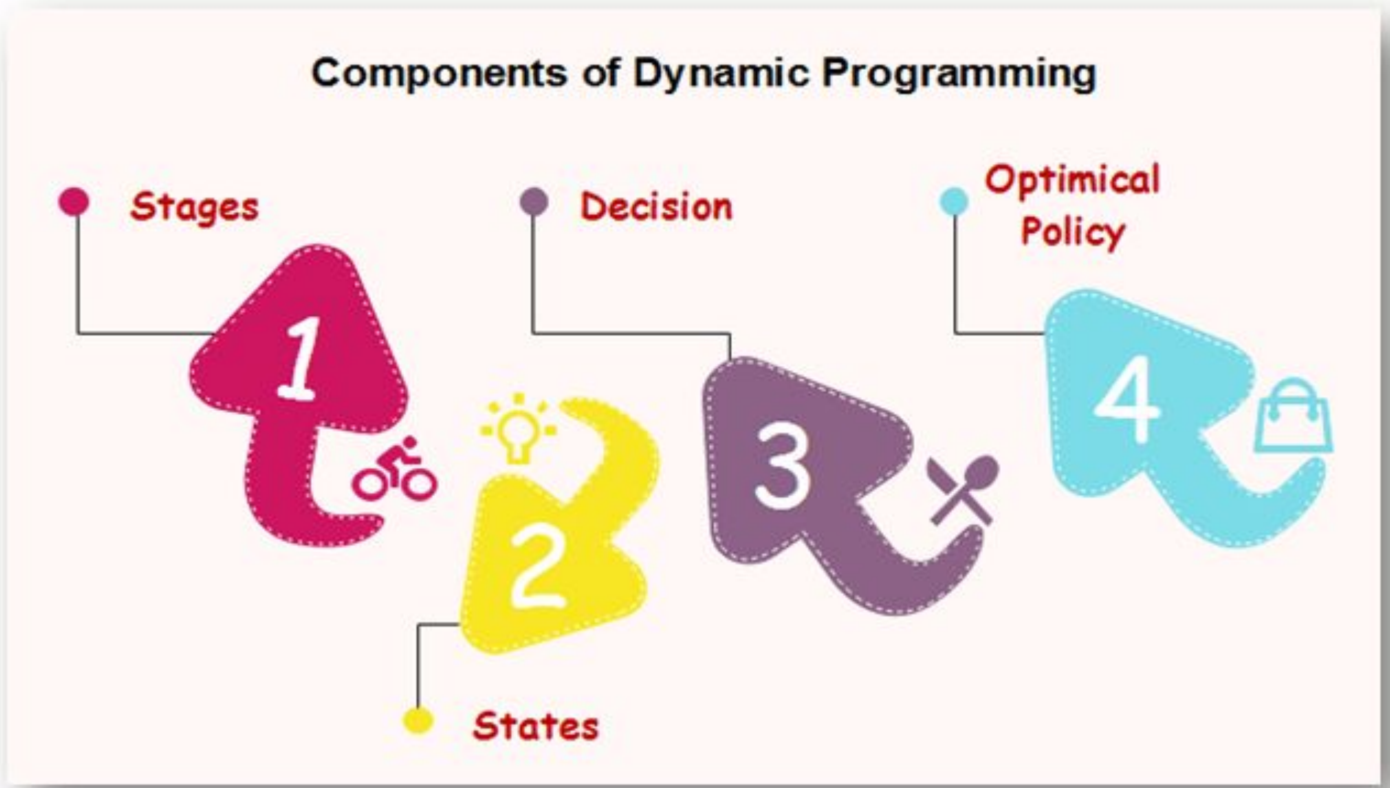
Elements of Dynamic Programming

Substructure: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.

Table Structure: After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and **we do not want to repeatedly solve the same problem** over and over again.

Bottom-up Computation: Using table, combine the solution of smaller subproblems **to solve larger subproblems** and eventually arrives at a solution to complete problem.

Components of Dynamic programming



1. **Stages:** The problem can be **divided into several subproblems, which are called stages.** A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. **The states for the shortest path problem was the node reached**.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The **decision taken at every stage should be optimal; this is called a stage decision**.
4. **Optimal policy:** It is a rule which **determines the decision at each stage; a policy is called an optimal policy if it is globally optimal**. This is known as **Bellman** principle of optimality.

- Given the current state, the optimal choices for each of the remaining states **does not depend** on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
- There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
- The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the **structure** of an optimal solution.
2. **Recursively** defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the **bottom up** (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. **All pair Shortest path problem**
4. Reliability design problem
5. **Longest common subsequence (LCS)**
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

Differentiate between

Divide & Conquer Method	**Dynamic Programming**
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: • Characterize the structure of optimal solutions. • Recursively defines the values of optimal solutions. • Compute the value of optimal solutions in a Bottom-up minimum. • Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.

Differentiate between ...

Divide & Conquer Method	Dynamic Programming
3. It does **more work** on subproblems and hence has more time consumption.	**3.** It solves subproblems **only once** and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are **independent** of each other.	**5.** In this subproblems are **interdependent**.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

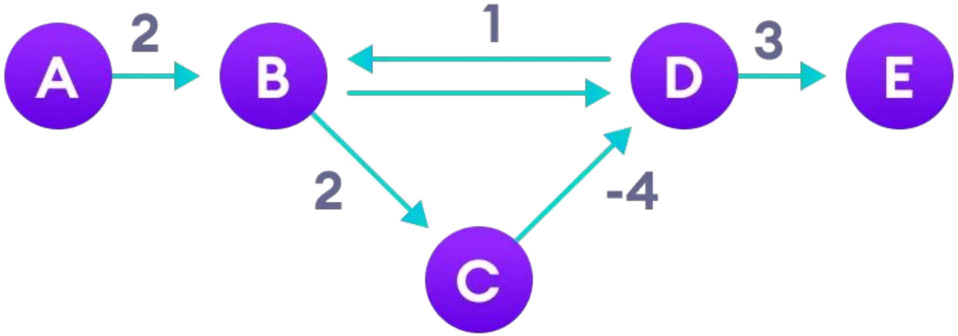
Bellman Ford's Algorithm

- Bellman Ford algorithm helps us find the **shortest** path from a **vertex to all other vertices of a weighted graph**.
- It is **similar to Dijkstra's algorithm** but it can work with graphs in which edges **can have negative weights**.
- It is **slower** than Dijkstra's Algorithm but more versatile, **as it capable of handling some of the negative weight edges**.
- Based on the **"Principle of Relaxation"** in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.
- **Memoization Table** is used to update the value of each vertex.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.

Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.



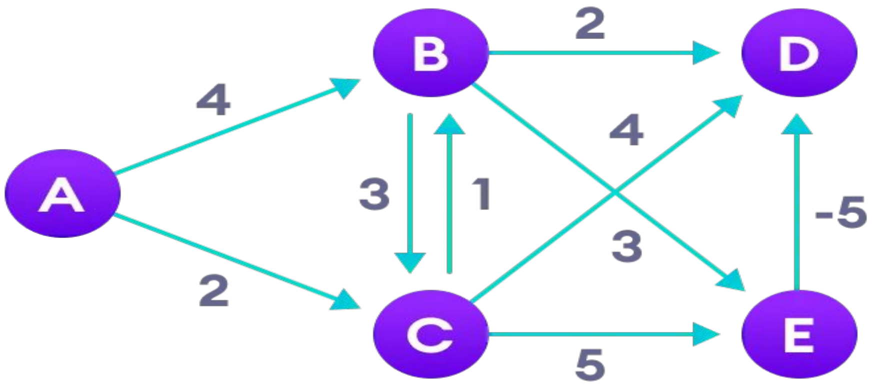
Negative weight cycles can give an incorrect result when trying to find out the shortest path

Bellman-ford Algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
  do for each edge (u, v)  E[G]
       do RELAX(u, v, w)
for each edge (u, v)  E[G]
  do if d[v] > d[u] + w(u, v)
       then return FALSE
return TRUE
```

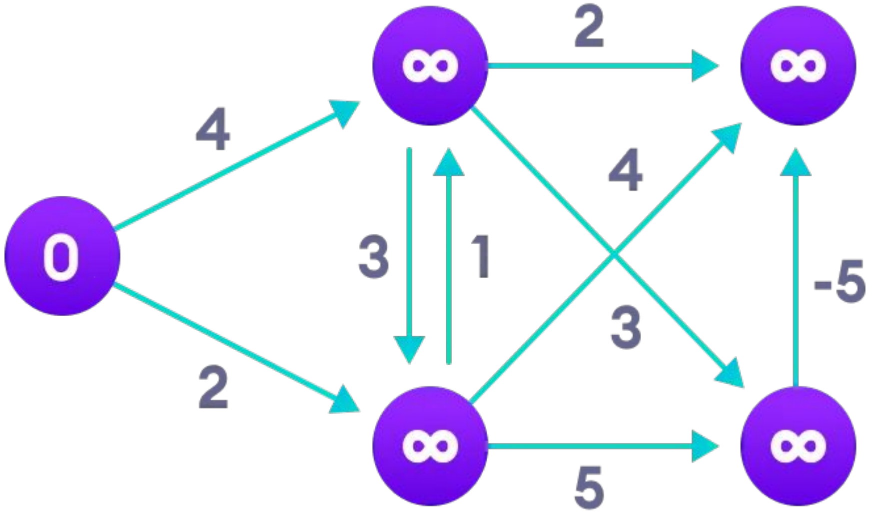
How Bellman Ford's algorithm works

- Bellman Ford algorithm works by **overestimating** the length of the path from the starting vertex to all other vertices. Then it **iteratively** relaxes those estimates by finding new paths that are **shorter than the previously overestimated paths.**
- By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

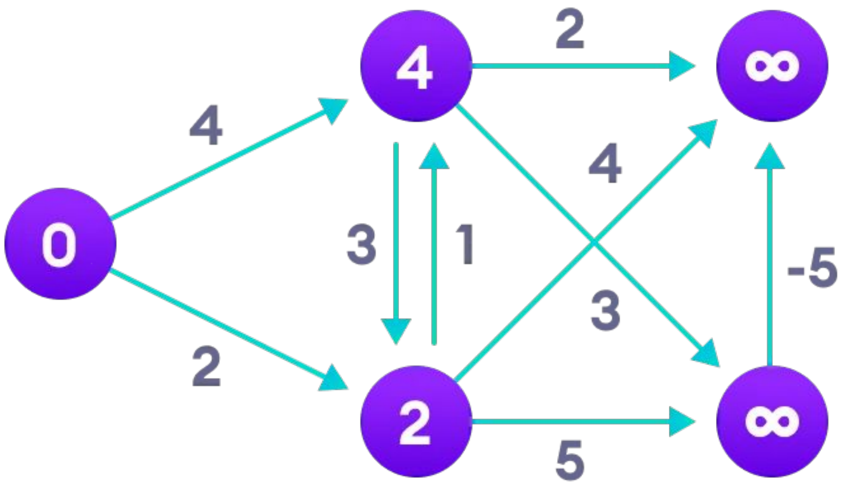


Step-1 for Bellman Ford's algorithm

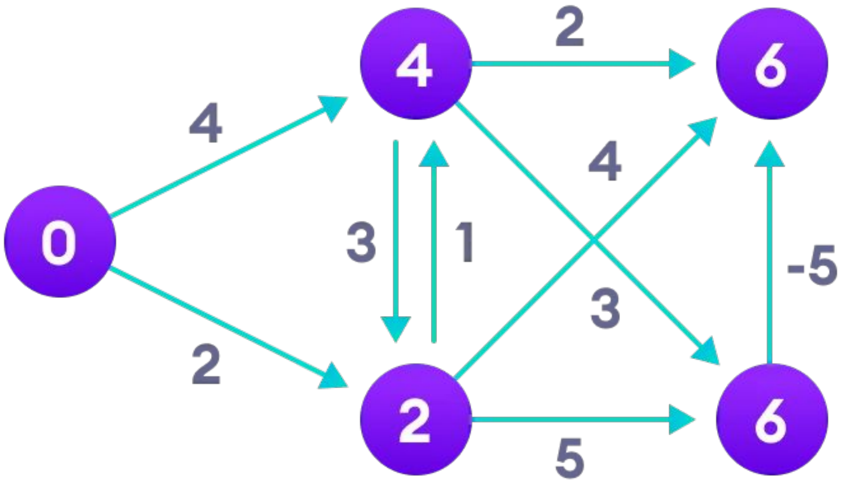
Step 2: Choose a starting vertex and assign infinity path values to all other vertices



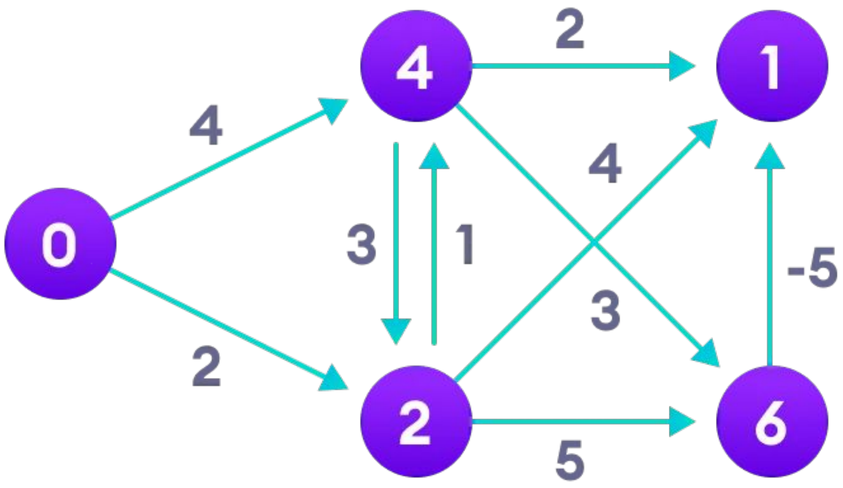
Step 3: Visit each edge and relax the path distances if they are inaccurate



Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step 5: Notice how the vertex at the top right corner had its path length adjusted



Memoization Table

Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

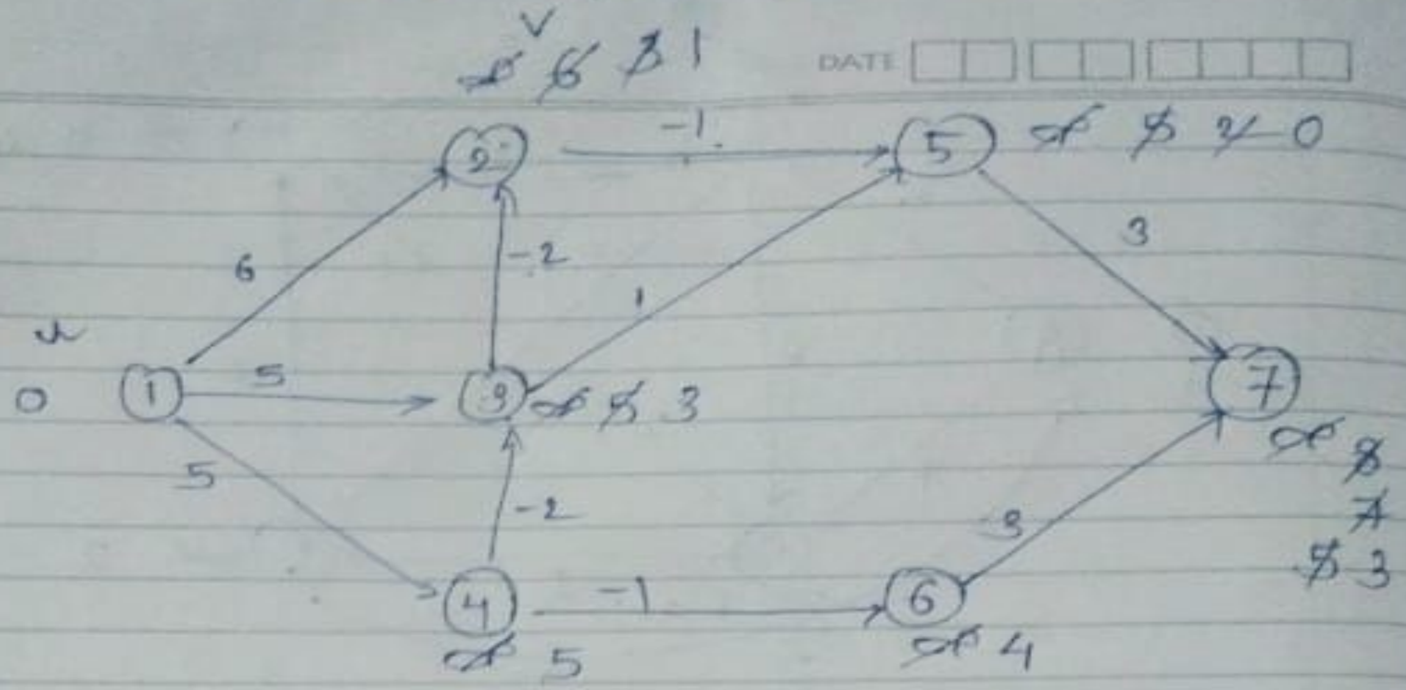
	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Bellman Ford Algoorithm

Ex (I)

DATE

1 = 0
2 = 1
3 = 3
4 = 5
5 = 0
6 = 4
7 = 3



(10) Edges - 1-2, 1-3, 1-4, 2-5, 3-2,
3-5, 4-3, 4-6, 5-7, 6-7

→ Relaxation

if $(d[u] + c(u, v) < d[v])$

$0 + 6 < \infty$

$d[v] = d[u] + c(u, v)$

$d[v] = 6$ ✓

I ✓

II ✓

III ✓

IV ✓ No change

V ✓

VI ✓

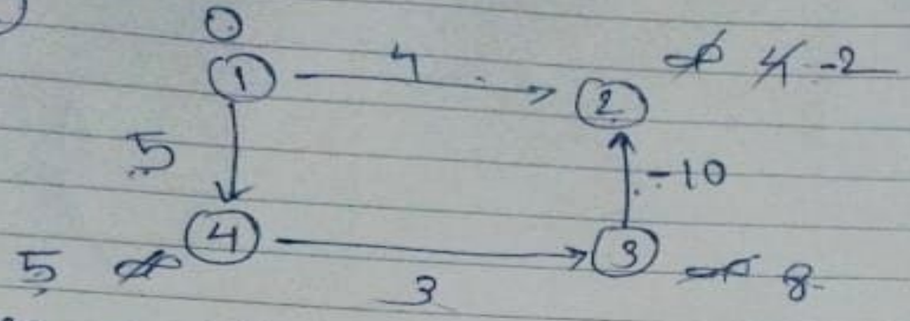
Iterations

$V - 1$

$7 - 1 = \underline{6 \text{ times}}$

Bellman ford Algorithm

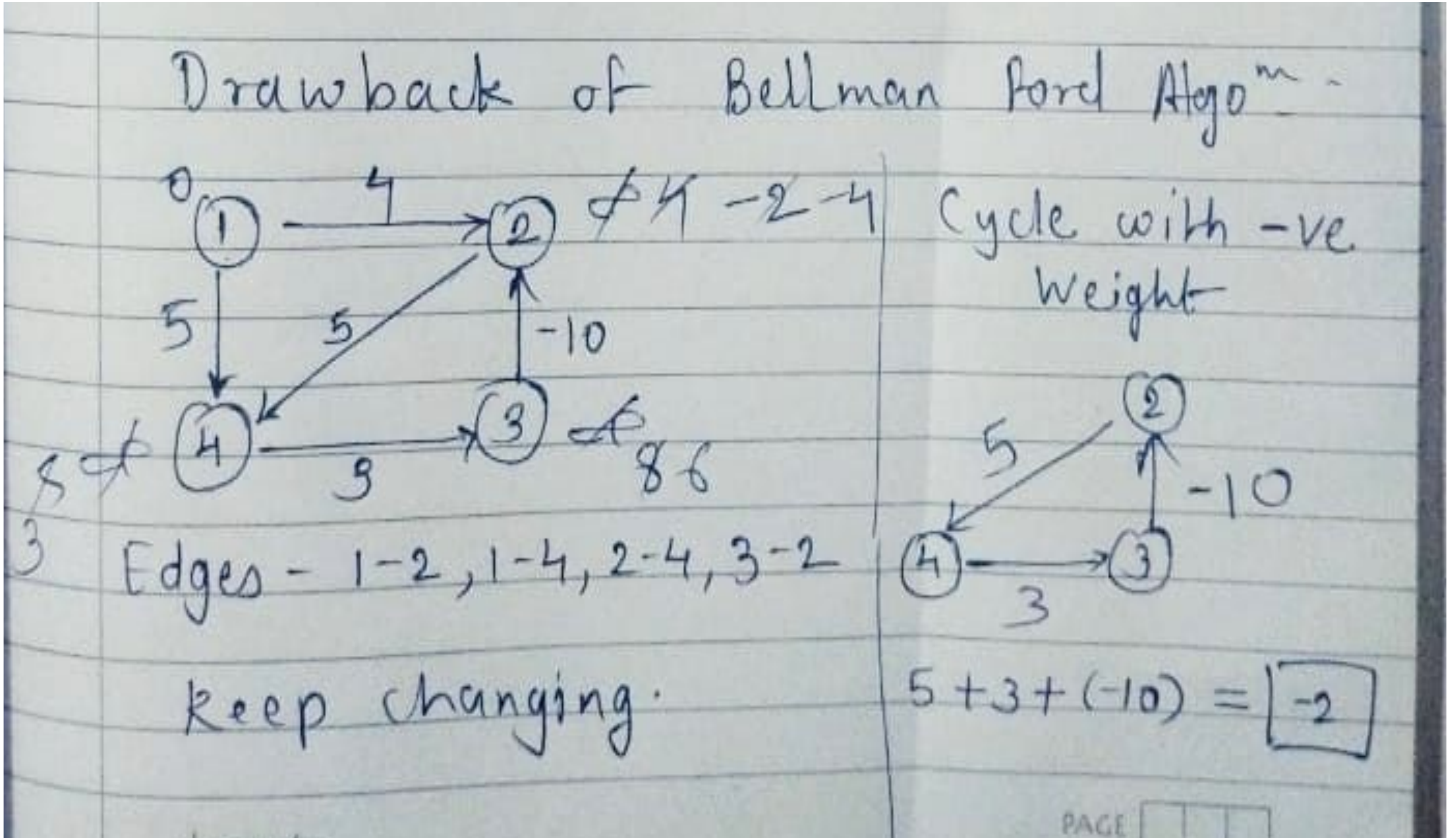
Ex (II)



Edges - 1-2, 1-4, 3-2, 4-3

Iterations - |V - 1| = 4 - 1 = 3 times

	1	2	3	4	
	0	∞	∞	∞	
Iteration I st	0	4	8	5	
Iteration II nd	0	2	8	5	
Iteration III rd	0	5	8	5	No change.



If there is negative weight cycle then graph cannot be solved.

Bellman Ford Complexity

Time Complexity

Best Case Complexity **O(E)**

Average Case Complexity **O(VE)**

Worst Case Complexity **O(VE)**

Space Complexity

The space complexity is **O(V)**.

Floyd-Warshall Algorithm

- Floyd-Warshall Algorithm is an algorithm **for finding the shortest path** between all the pairs of vertices in a weighted graph.
- This algorithm works for both the **directed and undirected weighted graphs.**
- But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
- A weighted graph is a graph in which each edge has a numerical value associated with it.
- Floyd-Warshall algorithm is also called as **Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.**
- This algorithm follows the **dynamic programming approach** to find the shortest paths.

Algorithm

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as **infinite** and distance to the **source itself as 0**.

Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following **|V|-1** times where |V| is the number of vertices in given graph.

…..a) Do following for each edge u-v

………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]

………………….dist[v] = dist[u] + weight of edge uv

Algorithm...

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If **dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"**

Note: The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

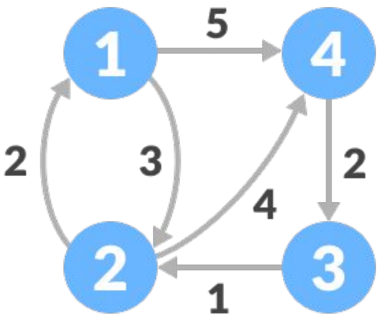
Time Complexity

1. The running time of Bellman-Ford is **O(VE)** , where **V** is the number of vertices and **E** is the number of edges in the graph.
2. On a complete graph of n vertices, there are around **n^2** edges, for a total running time of **n^3** .

How Floyd-Warshall Algorithm Works?

1. Create a matrix A0 of dimension n*n where **n is the number of vertices. The row and the column are indexed as i and j** respectively. i and j are the vertices of the graph.

. Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is **no path from ith vertex to jth vertex, the cell is left as infinity.**



$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

Fill each cell with the distance between ith and jth vertex

2. Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

- Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).
- That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].
- In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex k

For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4

3. Similarly, A2 is created using A3. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 4

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

```
n = no of vertices

A = matrix of dimension n*n

for k = 1 to n

    for i = 1 to n

        for j = 1 to n

            Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])

return A
```

Floyd Warshall Algorithm Complexity

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

Longest Common Subsequence

- The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.
- If **S1 and S2** are the two given sequences then, **Z is the common subsequence** of S1 and S2 if Z is a subsequence of both S1 and S2.
- Furthermore, **Z must be a strictly increasing sequence of the indices of both S1 and S2.**
- **In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.**

If

S1 = {B, C, D, A, A, C, D}

Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. **not strictly increasing sequence**).

Let us understand LCS with an example.

If

S1 = {B, C, D, A, A, C, D}

S2 = {A, C, D, B, A, C}

Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...

Among these subsequences, **{C, D, A, C} is the longest common subsequence.** We are going to find this longest common subsequence **using dynamic programming.**

Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Using Dynamic Programming to find the LCS

X	A	C	A	D	B

Longest Common Subsequence first sequence

Y	C	B	D	A

Longest Common Subsequence first sequence

Step 1:

Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

Step 2: Fill each cell of the table using the following logic.

Step 3: If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.

Step 4 : Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

		C	B	D	A
	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1
C	0				
A	0				
D	0				
B	0				

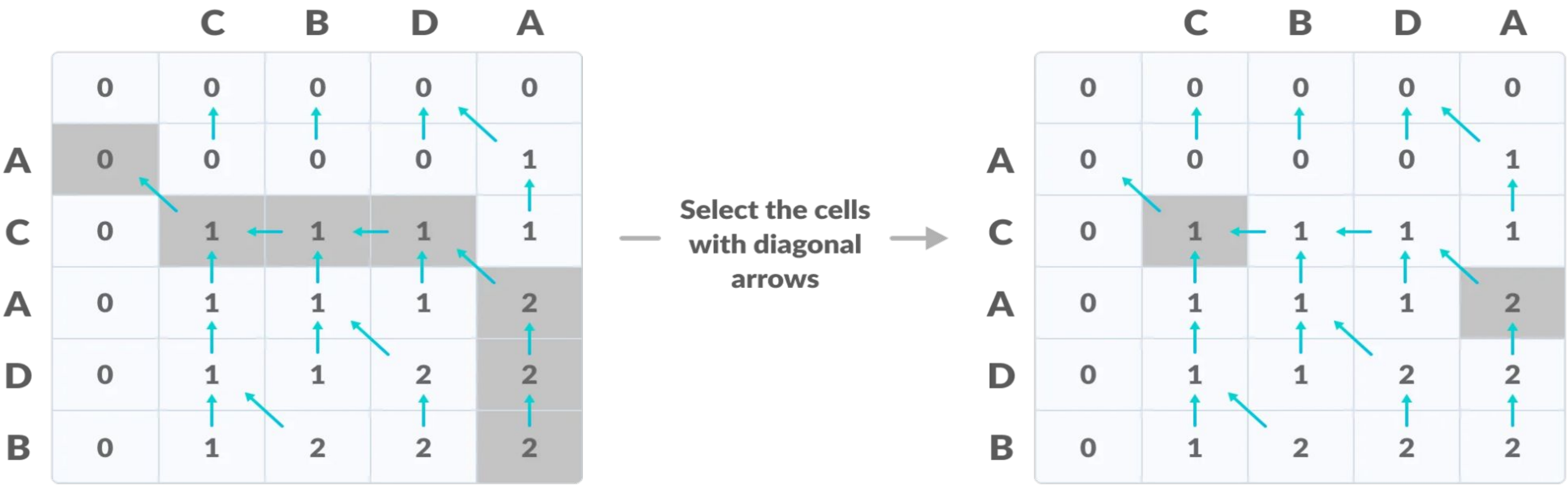
Step 5: Step 2 is repeated until the table is filled.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

Step 6: The value in the last row and the last column is the length of the longest common subsequence.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

Step 7: In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Step 8: LCS -

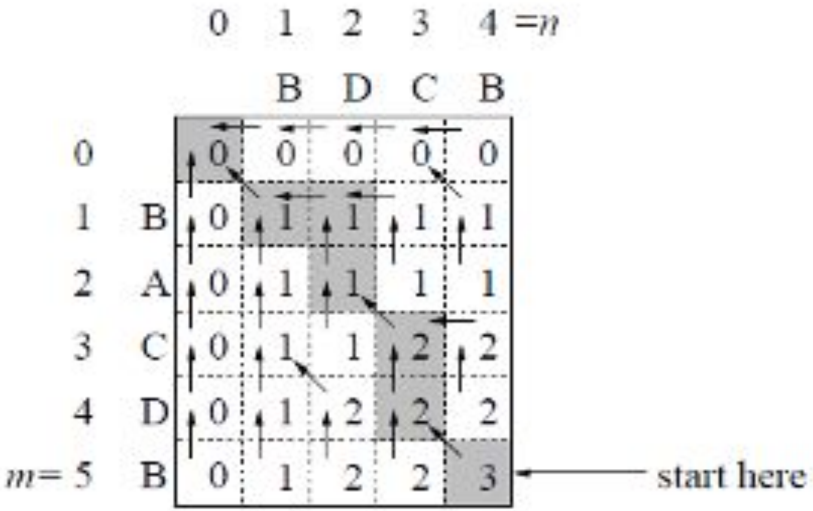
Example 2

		0	1	2	3	4 =n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m= 5	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BCB



Example 3

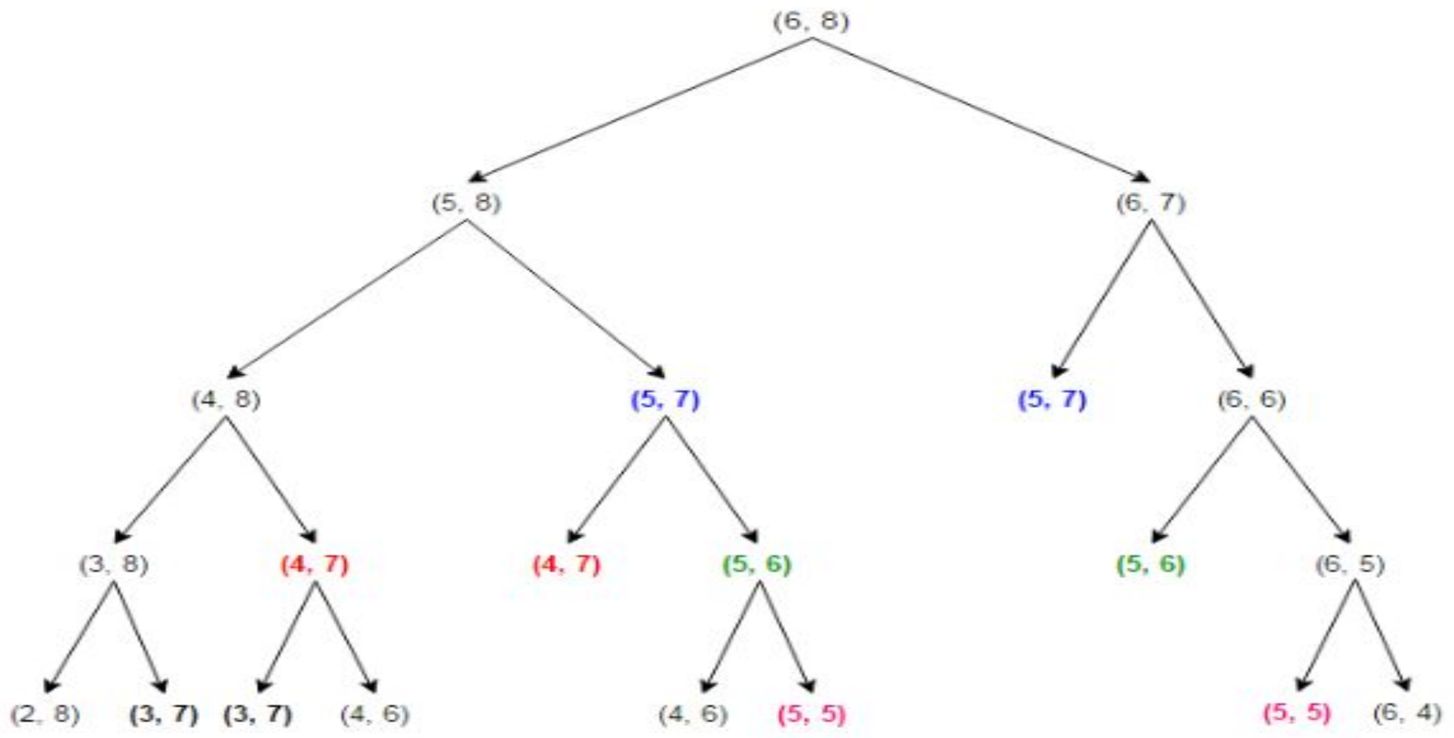
	Y		a	s	w	v
X		0	1	2	3	4
	0	0	0	0	0	0
a	1	0	↖ **1**	← 1	← 1	← 1
r	2	0	↑ 1	↑ 1	↑ 1	↑ 1
s	3	0	↑ 1	↖ **2**	← 2	← 2
w	4	0	↑ 1	↑ 2	↖ **3**	← 3
q	5	0	↑ 1	↑ 2	↑ 3	← 3
v	6	0	↑ 1	↑ 2	↑ 3	↖ **4**

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

- The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.
- In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.
- So, the time taken by a **dynamic approach** is the time taken to fill the table (ie. **O(mn)**). Whereas, the **recursion algorithm** has the complexity of **2^max(m, n).**

The LCS problem exhibits overlapping subproblems. A problem is said to have overlapping subproblems if the recursive algorithm for the problem solves the same subproblem repeatedly rather than generating new subproblems.

Let's consider the recursion tree for two sequences of length 6 and 8 whose LCS is 0.



Longest Common Subsequence Applications

1. In compressing genome resequencing data
2. To authenticate users within their mobile phone through in-air signatures

Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to **multiply these matrices together.** The problem is not actually to perform the multiplications, but merely **to decide in which order to perform the multiplications.**

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

- **Problem:** Given a series of n arrays (of appropriate sizes) to multiply: $A1 \times A2 \times \cdots \times An$
- Determine where **to place parentheses** to minimize the number of multiplications.
- Multiplying an **$i \times j$ array with a $j \times k$** array takes $i \times j \times k$ array
- Matrix multiplication is **associative**, so all placements give same result

Number of Multiplications

- Multiplying an $i \times j$ and a $j \times k$ matrix requires ijk multiplications

- Each element of the product requires j multiplications, and there are ik elements

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

$$2 \times 3 \longleftrightarrow 3 \times 2$$

$$C = A \times B = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} & a_{11} \times b_{12} + a_{12} \times b_{22} + a_{13} \times b_{32} \\ a_{21} \times b_{12} + a_{22} \times b_{21} + a_{23} \times b_{31} & a_{21} \times b_{12} + a_{22} \times b_{22} + a_{23} \times b_{32} \end{bmatrix}$$

Condition

1. **Column** of first matrix should be equal to **Rows** of second matrix i. e., **3**
2. Result will be in **2* 2 Matrix**
3. No. o Multiplications **2*3*2 = 12**
4. **Dimensions will be D0,D1,D2,D3 (D1=D2=3 so consider one dimension.So, Dimension will be D0,D1,D3**

$$A_1 \times A_2 \times A_3$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$A_1 \times A_2 \times A_3 \times \cdots \times A_{10}$

$(A_1 \times A_2) \times A_3$

$\underbrace{2 \quad 3 \quad 3 \quad 4} \quad \underbrace{4 \quad 2}$

$2 \times 3 \times 4 = 24 \qquad 0$

$\underbrace{2 \quad 4} \quad \underbrace{4 \quad 2}$

$24 + 0 + 2 \times 4 \times 2 = 40$

$A_1 \times (A_2 \times A_3)$

$2 \quad 3 \qquad 3 \quad 4 \quad 4 \quad 2$

$0 \qquad 3 \times 4 \times 2 = 24$

$\underbrace{2 \quad 3} \quad \underbrace{3 \quad 2}$

$0 + 24 + 2 \times 3 \times 2 = 36$

$C[1,3]$

$$A_1 \times A_2 \times A_3$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$(A_1 \times A_2) \times A_3$$

$$\underbrace{2 \quad 3 \quad 3 \quad 4}_{C[1,2]=24} \quad \underbrace{4 \quad 2}_{C[3,3]=0}$$

$$\underbrace{2 \quad 4 \quad 4 \quad 2}$$

$$d_0 \quad d_2 \quad d_3$$

$$C[\overset{i}{1}, \overset{k}{2}] + C[\overset{k+1}{3}, \overset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{k}{d_2} \times \underset{d_j}{d_3} = 40$$

$$A_1 \times (A_2 \times A_3)$$

$$\begin{array}{cc} C[1,1] & 3 \quad 4 \quad 4 \quad 2 \\ 0 & C[2,3]=24 \end{array}$$

$$\underbrace{2 \quad 3 \quad 3 \quad 2}$$

$$C[\overset{i}{1}, \overset{k}{1}] + C[\overset{k+1}{2}, \overset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{k}{d_1} \times \underset{d_j}{d_3} = 36$$

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$A_1 \times A_2 \times A_3 \times A_4$

$d_0 \ d_1 \quad d_1 \ d_2 \quad d_2 \ d_3 \quad d_3 \ d_4$

$n = 4 - 1$

1. $A_1(A_2(A_3A_4))$
2. $A_1((A_2A_3)A_4)$
3. $(A_1A_2)(A_3A_4)$
4. $(A_1(A_2A_3))A_4$
5. $((A_1A_2)A_3)A_4$

$$\frac{{}^{2n}C_n}{n+1}$$

$$\frac{{}^{2(n-1)}C_{(n-1)}}{n}$$

$n=4$

$$\frac{{}^{2\times3}C_3}{4} = \frac{{}^6C_3}{4} = \frac{\frac{\cancel{6}\times5\times\cancel{4}}{\cancel{3\times2}\times1}}{\cancel{4}}$$

$n=5$ 14

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$\underset{d_0 \quad d_1}{A_1} \times \underset{d_1 \quad d_2}{A_2} \times \underset{d_2 \quad d_3}{A_3} \times \underset{d_3 \quad d_4}{A_4}$$

$$C[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & C[1,1] + C[2,4] + d_0 \times d_1 \times d_4, \\ k=2 & C[1,2] + C[3,4] + d_0 \times d_2 \times d_4, \\ k=3 & C[1,3] + C[4,4] + d_0 \times d_3 \times d_4 \end{cases}$$

$A_1 (A_2 A_3 A_4)$

$(A_1 A_2)(A_3 A_4)$

$(A_1 A_2 A_3) A_4$

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$A_1 \times A_2 \times A_3 \times A_4$$

$$\underset{d_0}{3} \quad \underset{d_1}{\underline{2 \quad 2}} \quad \underset{d_2}{\underline{4 \quad 4}} \quad \underset{d_3}{\underline{2 \quad 2}} \quad \underset{d_4}{5}$$

$$C[\overset{i}{1},\overset{j}{2}] = \min_{1 \le k < 2} \; k=1 \left\{ \begin{array}{l} C[1,1] + C[2,2] + d_0 \times d_1 \times d_2 \\ 0 + 0 + 3 \times 2 \times 4 \end{array} \right\}$$

$$C[\overset{i}{2},\overset{j}{3}] = \min_{2 \le k < 3} \; k=2 \left\{ \begin{array}{l} C[2,2] + C[3,3] + d_1 \times d_2 \times d_3 \\ 0 + 0 + 2 \times 4 \times 2 = 16 \end{array} \right\}$$

$$C[\overset{i}{3},\overset{j}{4}] = \min_{3 \le k < 4} \; k=3 \left\{ \begin{array}{l} C[3,3] + C[4,4] + d_2 \times d_3 \times d_4 \\ 0 + 0 + 4 \times 2 \times 5 = 40 \end{array} \right\}$$

$j \longrightarrow$

$i \downarrow$

C	1	2	3	4
1	0	24		
2		0	16	
3			0	40
4				0

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

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$A_1 \times A_2 \times A_3 \times A_4$$

$$\underset{d_0}{3} \quad \underset{d_1}{\underline{2 \qquad 2}} \quad \underset{d_2}{\underline{4 \qquad 4}} \quad \underset{d_3}{\underline{2 \qquad 2}} \quad \underset{d_4}{\underline{5}}$$

$$C[\overset{i}{1},\overset{j}{3}] = \min_{1 \le k < 3} \begin{cases} k=1: & C[1,1] + C[2,3] + d_0 \times d_1 \times d_3 = 28 \\ & 0 + 16 + 3 \times 2 \times 2 \\ k=2: & C[1,2] + C[3,3] + d_0 \times d_2 \times d_3 = 48 \\ & 24 + 0 + 3 \times 4 \times 2 \end{cases}$$

$$C[2,4] = \min_{2 \le k < 4} \begin{cases} k=2: & C[2,2] + C[3,4] + d_1 \times d_2 \times d_4 = 80 \\ & 0 + 40 + 2 \times 4 \times 5 \\ k=3: & C[2,3] + C[4,4] + d_1 \times d_3 \times d_4 = 36 \\ & 16 + 0 + 2 \times 2 \times 5 \end{cases}$$

$j \longrightarrow$

$i \downarrow$

C	1	2	3	4
1	0	24	28	
2		0	16	36
3			0	40
4				0

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

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$A_1 \times A_2 \times A_3 \times A_4$$

$$\underset{d_0}{3} \quad \underset{d_1}{\underline{2 \quad 2}} \quad \underset{d_2}{\underline{4 \quad 4}} \quad \underset{d_3}{\underline{2 \quad 2}} \quad \underset{d_4}{\underline{5}}$$

$$C[1,4] = \min_{1 \le k < 4} \begin{cases} k=1: & \underset{0}{C[1,1]} + C[2,4] + d_0 \times d_1 \times d_4 = 86 \\ & 36 + 3 \times 2 \times 5 \\ k=2: & C[1,2] + C[3,4] + d_0 \times d_2 \times d_4 = 124 \\ & 24 + 40 + 3 \times 4 \times 5 \\ k=3: & C[1,3] + C[4,4] + d_0 \times d_3 \times d_4 = 58 \\ & 28 + 0 + 3 \times 2 \times 5 \end{cases}$$

C ($i \downarrow$, $j \rightarrow$)

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

K

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

$$C[i,j] = \min_{i \le k < j} \left\{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \right\}$$

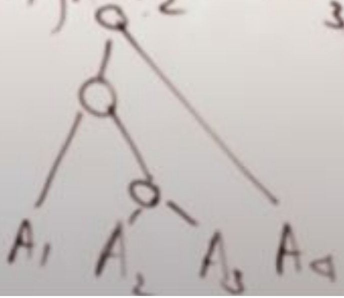
$$A_1 \times A_2 \times A_3 \times A_4$$

$$\underset{d_0}{3} \quad \underset{d_1}{\underline{2 \quad 2}} \quad \underset{d_2}{\underline{4 \quad 4}} \quad \underset{d_3}{\underline{2 \quad 2}} \quad \underset{d_4}{\underline{5}}$$

1 ↓ 4

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

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



j →

i ↓

C

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

K

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

Number of Parenthesizations

Example: Given the matrices A1,A2,A3,A4

Assume the dimensions of A1=d0×d1, etc

Below are the five possible parenthesizations of these arrays, along with the number of multiplications:

1. (A1A2)(A3A4):d0d1d2+d2d3d4+d0d2d4
2. ((A1A2)A3)A4:d0d1d2+d0d2d3+d0d3d4
3. (A1(A2A3))A4:d1d2d3+d0d1d3+d0d3d4
4. A1((A2A3)A4):d1d2d3+d1d3d4+d0d1d4
5. A1(A2(A3A4)):d2d3d4+d1d2d4+d0d1d4

Number of Parenthesizations

- The number of parenthesizations is at least **$T(n) \geq T(n-1)+T(n-1)$**
- Since the number with the first element removed is $T(n-1)$, which is also the number with the last removed
- Thus the number of parenthesizations is **$\Omega(2n)$**
- The number is actually **$T(n)=\sum_{k=1 \text{ to } n-1} T(k)T(n-k)$** which is related to the **Catalan numbers**.
- This is because the original product can be split into 2 subproducts in k places. Each split is to be parenthesized optimally.
- This recurrence is related to the Catalan numbers.

Characterizing the Optimal Parenthesization

- An optimal parenthesization of A1…An must break the product into two expressions, each of which is parenthesized or is a single array
- Assume the break occurs at position k
- In the optimal solution, the solution to the product A1…Ak must be optimal

Otherwise, we could improve A1…An by improving A1…Ak

But the solution to A1…An is known to be optimal

This is a contradiction

Thus the solution to A1…An is known to be optimal

Principle of Optimality

- This problem exhibits the Principle of Optimality:
  - The optimal solution to product A1…An contains the optimal solution to two subproducts
- Thus we can use Dynamic Programming
  - Consider a recursive solution
  - Then improve it's performance with memoization or by rewriting bottom up

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose

A is a 10 × **30** matrix,

B is a **30** × **5** matrix, and

C is a **5** × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = **4500** operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = **27000** operations.

Clearly the **first parenthesization requires less number of operations.**

Recursive Solution

- Let $M[i,j]$ represent the number of multiplications required for matrix product $A_i \times \cdots \times A_j$

  For $1 \le i \le j < n$

- $M[i,i]=0$ since no product is required
- The optimal solution of $A_i \times A_j$ must break at some point, k, with $i \le k < j$
- Thus, $M[i,j]=M[i,k]+M[k+1,j]+d_{i-1} d_k d_j$
- Thus, $M[i,j]=\begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j}\{M[i,k]+M[k+1,j]+d_{i-1} d_k d_j\} & \text{if } i<j \end{cases}$

Efficient Computation

- A way to calculate this bottom up
- Which values does M[i,j] depend on?
  - Consider a n×n matrix of values M[i,j]
  - Diagonal is 0
  - $1 \leq i \leq j \leq n$ is the upper right triangle
  - Consider some element M[i,j]
  - Where are M[i,k] (for $i \leq k \leq j$):
  - Where are M[k+1,j] (for $i \leq k \leq j$):
  - This tells us the order in which to build the table: By diagonals
    - By diagonal
    - Moving up and right from the diagonal that goes top-left to bottom-right

Algorithm for Location of Minimum Value

```
Bottom Up Algorithm to Calculate Minimum Number of Multiplications
n -- Number of arrays
d -- array of dimensions of arrays 1 .. n
P -- 2D array of locations of M[i, j]
    minimum_multiplication(n: integer; d: array(0..n))
        M: array(1..n, 1..n) := (others => 0);
        for diagonal in 1 .. n-1 loop
            for i in 1 .. n-diagonal loop
                j := i + diagonal;
                min_value := integer'last;
                for k in i .. j-1 loop
                    current := M[i, k] + M[k+1, j] + d(i-1)*d(k)*d(j)
                    if current < min_value then
                        min_value := current
                        mink := k
                    end if
                end loop
                M[i, j] := mi_nvalue
                P(i, j) := mink
            end loop
        end loop
        printbest(P, i, j):
            if i=j then
                print "A"_i
            else
                print '('
                printbest(P, i, M[i, j])
                printbest(P, M[i, j] + 1, j)
               print ')'
```

Algorithm for Minimum Value

Bottom Up Algorithm to Calculate Minimum Number of Multiplications

```
n -- Number of arrays
d -- array of dimensions of arrays 1 .. n
    minimum_multiplication(n: integer; d: array(0..n))
        M: array(1..n, 1..n) := (others => 0);

        for diagonal in 1 .. n-1 loop
            for i in 1 .. n - diagonal loop
                j := i + diagonal;
                min := integer'last;
                for k in i .. j-1 loop
              current := M[i, k] + M[k+1, j] +d(i-1)*d(k)*d(j)
                    if current < min then
                        min := current
                    end if
                end loop
                M[i, j] := min
            end loop
        end loop
```

Thank You