BYJU'S EXAM PREP

Vision 2023

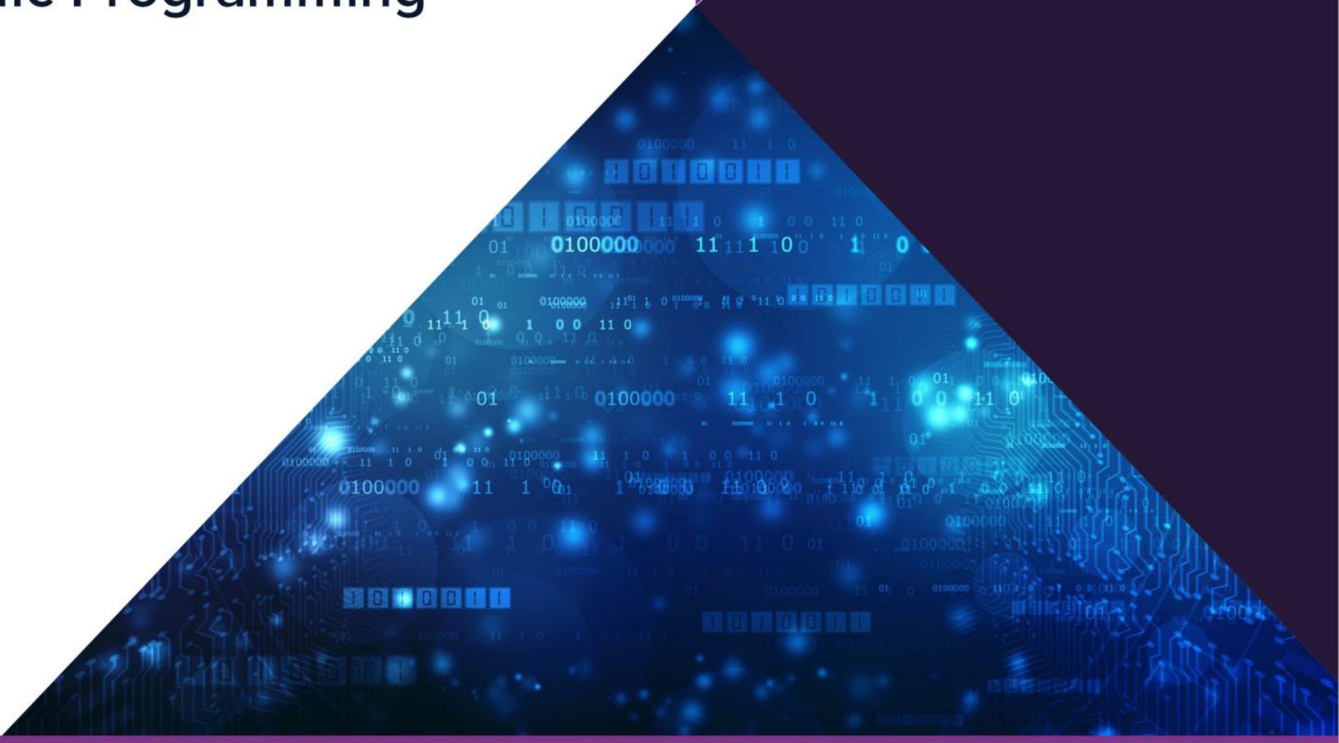
A Course for GATE & PSUs

Computer Science Engineering

Algorithm

CHAPTER 5

Dynamic Programming



CHAPTER 5 ALGORITHM

DYNAMIC PROGRAMMING

Dynamic Programming is the most powerful design technique for solving optimization 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.
- Dynamic Programming solves each subproblem 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**".
  - Assume N unbiased coins, their values A1, A2, … , AN, and sum is S. Find the minimum number of coins the sum of which is S.
  - Dynamic programming solution is: find a state for which an optimal solution gets and find the optimal solution for the next state by using previous one.

1. Characteristics of Dynamic Programming:

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

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

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

2. Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

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

2.**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.

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

3. Dynamic Programming Algorithm

It can be broken into four steps:

1.Characterize the structure of an optimal solution.

2.Recursively define 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.

4. Tabulation vs Memoization

There are two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving DP problem:

1.**Tabulation:** Bottom Up

2.**Memoization:** Top Down

	Tabulation	**Memorization**
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memorized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memorized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memorized version. The table is filled on demand.

5. APPLICATIONS OF DYNAMIC PROGRAMMING

0/1 knapsack problem-

In this item cannot be broken which means the thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

0/1 Knapsack Problem

Given weights and values of n items, select items to put items in a Knapsack of capacity W to maximise the total value in the Knapsack. You cannot break an item, either pick the complete item, or don’t pick it (0 to 1 property).

$$0/1\mathrm{KS}(M,\ N) = \begin{cases} 0; & \text{if } M = 0 \text{ or } N = 0 \\ 0/1\mathrm{KS}(M, N-1); & \text{if } w[n] > M \\ \max\begin{Bmatrix} 0/1\mathrm{KS}(M - W[n],\ N-1) + P[n] \\ 0/1\mathrm{KS}(M,\ N-1) \end{Bmatrix}; & \text{otherwise} \end{cases}$$

Time complexity = O(MN): If M value is very large then it behaves like an exponential and NP-complete problem.

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then ***S′* = *S* - {*i*}** is an optimal solution for ***W* - *w_i*** dollars and the value to the solution ***S*** is ***V_i*** plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if $w_i \le w$ then
      if $v_i$ + c[i-1, w-$w_i$] then
        c[i, w] = $v_i$ + c[i-1, w-$w_i$]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

Example-2

A thief enters a house for robbing it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

Item	**Weight (kg)**	**Value ($)**
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0th row and 0th column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

T (i , j) = max { T (i-1 , j) , value_i + T(i-1 , j – weight_i) }

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

T(1,1) = max { T(0,1) , 3 + T(0,-1) }

T(1,1) = T(0,1) { Ignore T(0,-1) }

T(1,1) = 0

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) }

T(1,2) = max { T(0,2) , 3 + T(0,0) }

T(1,2) = max {0 , 3+0}

T(1,2) = 3

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }

T(1,3) = max { T(0,3) , 3 + T(0,1) }

T(1,3) = max {0 , 3+0}

T(1,3) = 3

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(\text{value})_i = (\text{value})_1 = 3$
- $(\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1)   { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2)   { Ignore T(1,-1) }

T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

Finding T(2,5)-

We have,

- i = 2
- j = 5
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

6. Longest common subsequence (LCS)-

A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences X = (x_1 x_2....x_m) and Y = (y_1 y_2 y_n) and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is sequence that appears in the same relative order, but not necessarlly contigueous.

Recurrence Relation:

$$\mathrm{LCS}(i,\ j)=\begin{cases}0; & \text{if } i=0\,(\text{or})\,j=0\\ 1+\mathrm{LCS}(i-1,j-1); & \text{if } x[i-1]=y[j-1]\\ \max\left[\mathrm{LCS}(i-1,\ j),\mathrm{LCS}(i,\ j-1)\right]; & \text{if } x[i]\neq y[j]\end{cases}$$

Time complexity : by Brute force = O(2m), by Dynamic programming = O(m × n).

Space complexity = O(mn); where m and n are length of two given sequences.

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
   return
if B[i, j] = 'D'
   Print-LCS(B, X, i-1, j-1)
   Print($x_i$)
else if B[i, j] = 'U'
   Print-LCS(B, X, i-1, j)
else
   Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

Analysis

To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is *O(m, n)*, where ***m*** and ***n*** are the length of two strings.

Example-

Consider two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then x + 10y = ___.

The LCS is of length 4. There are 3 LCS of length 4 "qprr", "pqrr" and qpqr

A subsequence is a sequence that can be derived from another sequence by selecting zero or more elements from it, without changing the order of the remaining elements. Sub

		A	1	2	3	4	5
		0	q	p	q	r	r
B	0	0	0	0	0	0	0
1	p	0	0	1	1	1	1
2	q	0	1	1	2	2	2
3	p	0	1	2	2	2	2
4	r	0	1	2	2	3	3
5	q	0	1	2	3	3	3
6	r	0	1	2	3	4	4
7	p	0	1	2	3	4	4

sequence need not be contiguous. Since the length of given strings A = “qpqrr” and B = “pqprqrp” are very small, we don’t need to build a 5×7 matrix and solve it using dynamic programming. Rather we can solve it manually just by brute force. We will first check whether there exist a subsequence of length 5 since min_length(A,B) = 5.
Since there is no subsequence , we will now check for length 4. “qprr”, “pqrr” and “qpqr” are common in both strings.
X = 4 and Y = 3
X + 10Y = 34

Example-2

A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of lengths m and n respectively, with indexes of X and Y starting from 0.
We wish to find the length of the longest common sub-sequence(LCS) of X[m] and Y[n] as l(m,n), where an incomplete recursive definition for the function l(i,j) to compute the length of The LCS of X[m] and Y[n] is given below:

```
l(i,j) = 0, if either i=0 or j=0
       = expr1, if i,j > 0 and X[i-1] = Y[j-1]
       = expr2, if i,j > 0 and X[i-1] != Y[j-1]
```

Solution-

The last characters of two strings match.
The length of lcs is length of lcs of X[0..i-1] and Y[0..j-1]

The last characters don't match.

The length of lcs is max of following two lcs values

a) LCS of X[0..i-1] and Y[0..j]

b) LCS of X[0..i] and Y[0..j-1]

7. Floyd- Warshall's : All pair Shortest path problem-

For finding shortest paths between all paris of vertices in a weighted graph with positive or negative edge weights (but with no negative cycles).

→ $A^k(I, j)$ = the min cost required to go from i to j by considering all intermediate vertices are numbered not greater than k.

$A^0(I, j) = C(I, j)$

$A^k(I, j) = \min\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}$

Time complexity – $O(n^3)$ with help = $O(n^2 \log n)$.

→ Warshall's algorithm will not for negative edge cycle.

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
            $A^k[i, j] = \min(A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$
return A
```

```
// Floyd-Warshall Algorithm in C

#include <stdio.h>

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
```

```
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
             {2, 0, INF, 4},
             {INF, 1, 0, INF},
             {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

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

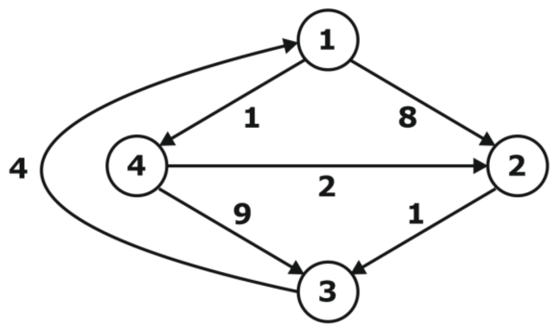
Advantages-

Floyd Warshall Algorithm has the following main advantages-

- It is extremely simple.
- It is easy to implement.

Example-

Consider the following directed weighted graph-



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

Solution-

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array} \qquad D_2 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array} \qquad D_4 = \begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

8. **Sum of Subset problem**

Find if there is a subject of the given set, sum of whose elements is equal to given sum.

Recurrence Relation:

$$\mathrm{SoS}(M, N, S) = \begin{cases} \text{return}(S); & M = 0 \\ \text{return}(-1); & N = 0 \\ \mathrm{SoS}(M, N-1, S); & \text{if } w[N] > M \\ \text{Min}\begin{cases} \mathrm{SoS}(M - w[N], N-1, S \cup w[N]) \\ \mathrm{SoS}(M, N-1, S) \end{cases}; & \text{otherwise} \end{cases}$$

Time complexity by Brute force = O(MN)

the following is the recursive formula for the isSubsetSum() problem.

isSubsetSum(set, n, sum)

= isSubsetSum(set, n-1, sum) ||

isSubsetSum(set, n-1, sum-set[n-1])

Base Cases:

isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0

isSubsetSum(set, n, sum) = true, if sum == 0

Example 1 :

The subset-sum problem is defined as follows: Given a set SS of n positive integers and a positive integer WW, determine whether there is a subset of SS whose elements sum to WW. An algorithm QQ solves this problem in O(nW)O(nW) time. Which of the following statements is false?

A. QQ solves the subset-sum problem in polynomial time when the input is encoded in unary

B. QQ solves the subset-sum problem in polynomial time when the input is encoded in binary

C. The subset sum problem belongs to the class NP

D. The subset sum problem is NP-hard

Solution-

The Subject problem is NP-Complete - there is reduction proof but I don't remember (Can see the below link). So, (**C**) and (**D**) are true as an NPC problem is in NP as well as NPH.

A. Input is encoded in unary. So, length of input is equal to the value of the input. So, complexity =O(nW)=O(nW) where both nn and WW are linear multiples of the length of the inputs. So, the complexity is polynomial in terms of the input length. So, (**A**) is true.

B. Input is encoded in binary. So, the length of W will be lgWlgW. (for W=1024W=1024, input length will be just 1010). So, now WW is exponential in terms of the input length of WW and O(nW)O(nW) also becomes exponential in terms of the input lengths. So, QQ is not a polynomial time algorithm. So, (**B**) is false.

Correct Answer: B

9. <u>Matrix Chain Multiplication-</u>

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

Matrix chain Multiplication

Given a sequence of matrices, find the most efficient order to multiply these matirices together in order to minimise the number of multiplications.

$$\text{MCM} = \begin{cases} 0 & \text{if } i = j \\ \min \left\{ \begin{matrix} \text{mcm}(i, K) + \text{mcm}(K+1, j) + Pi \times Pj \times Pk \\ \text{where } i \le K < j \text{ or} \\ i \le K \le j-1 \end{matrix} \right\} & \end{cases}$$

Number of ways to multiply matrix :

$$T(n) = \sum_{i=1}^{n-1} T(i)\,T(n-i)$$

Time complexity : Without dynamic programming = $O(n^n)$, with dynamic programming = $O(n^3)$.

Space complexity: without dynamic programming = $O(n)$, with dynamic programming = $O(n^2)$.

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) = ...

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.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Input: p[] = {40, 20, 30, 10, 30}

Output: 26000

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parentheses in following way

(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is 10*20*30

Algorithm

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n     // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + $p_{i-1}$ $p_k$ $p_j$
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

Example-1

We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.

Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})
On the basis of sequence, we make a formula

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.
We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. m (1,2) = m_1 x m_2

   = 4 x 10 x 10 x 3

   = 4 x 10 x 3 = 120

2. m (2, 3) = m_2 x m_3

   = 10 x 3 x 3 x 12

   = 10 x 3 x 12 = 360

3. m (3, 4) = m_3 x m_4

   = 3 x 12 x 12 x 20

   = 3 x 12 x 20 = 720

4. m (4,5) = m_4 x m_5

   = 12 x 20 x 20 x 7

   = 12 x 20 x 7 = 1680

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

M [1, 3] = M_1 M_2 M_3

1. There are two cases by which we can solve this multiplication: (M_1 x M_2) + M_3, M_1+ (M_2x M_3)
2. After solving both cases we choose the case in which minimum output is there.

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M_1 x M_2) + M_3 this combination is chosen for the output making.

M [2, 4] = M_2 M_3 M_4

1. There are two cases by which we can solve this multiplication: (M_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M_2+(M_3 x M_4) this combination is chosen for the output making.

M [3, 5] = M_3 M_4 M_5

1. There are two cases by which we can solve this multiplication: (M_3 x M_4) + M_5, M_3+ (M_4xM_5)
2. After solving both cases we choose the case in which minimum output is there.

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M_3 x M_4) + M_5this combination is chosen for the output making.

Now Product of 4 matrices:

M [1, 4] = M_1 M_2 M_3 M_4

There are three cases by which we can solve this multiplication:

1. (M_1 x M_2 x M_3) M_4
2. M_1 x(M_2 x M_3 x M_4)
3. (M_1 xM_2) x (M_3 x M_4)

After solving these cases we choose the case in which minimum output is there

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M_1 xM_2) x (M_3 x M_4) combination is taken out in output making,

M [2, 5] = M_2 M_3 M_4 M_5

There are three cases by which we can solve this multiplication:

1. (M_2 x M_3 x M_4)x M_5
2. M_2 x(M_3 x M_4 x M_5)
3. (M_2 x M_3)x (M_4 x M_5)

After solving these cases we choose the case in which minimum output is there

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M_2 x(M_3 x M_4 xM_5)combination is taken out in output making.

Now Product of 5 matrices:

M [1, 5] = M_1 M_2 M_3 M_4 M_5

There are five cases by which we can solve this multiplication:

1. (M_1 x M_2 xM_3 x M_4)x M_5
2. M_1 x(M_2 xM_3 x M_4 xM_5)
3. (M_1 x M_2 xM_3)x M_4 xM_5
4. M_1 x M_2x(M_3 x M_4 xM_5)

After solving these cases we choose the case in which minimum output is there

M [1, 5] = 1344

As comparing the output of different cases then '**1344**' is minimum output, so we insert 1344 in the table and M_1 x M_2 x(M_3 x M_4 x M_5)combination is taken out in output making.

Final Output is:

Step 3: Computing Optimal Costs: let us assume that matrix A_i has dimension p_{i-1}x p_i for i=1, 2, 3....n. The input is a sequence (p_0,p_1,......p_n) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

The algorithm first computes m [i, j] ← 0 for i=1, 2, 3.....n, the minimum costs for the chain of length 1.

Example:

Let A1, A2, A3, and A4 be four matrices of dimensions 1 x 2, 2 x 1, 1 x 4, and 4 x 1, respectively. The minimum number of scalar multiplications required to find the product A1A2A3A4 using the basic matrix multiplication method is

Answer: Let's apply the Bottom up approach

Make a table and solve the optimal part of the problems.

This table is for the number of multiplications needed.

A(i,j)= minimum multiplication needed for multiplying Ai and Aj,

A(1,1) = 0	A(2,2) = 0	A(3,3) = 0	A(4,4) = 0
A(1,2) =2	A(2,3)=8, split at 2	A(3,4)=4, split at 3	
A(1,3) = 6, split at 2	A(2,4)=6, split at 2		
A(1,4)=7, split at 2			

calculate the minimum value of A(1,3):

={A(1,1)+(2,3)+P0*P1*P3 , (1,2)+(3,3)+P0*P2*P3}

={0+8+8,2+0+4}

=6

calculate the minimum value of A(2,4):

={A(2,2)+A(3,4)+P1*P2*P4 , A(2,3)+A(4,4)+P1*P3*P4}

={0+4+2,8+0+8}

=6

calculate the minimum value of A(1,4):

={A(1,1)+(2,4)+P0*P1*P4 , (1,2)+(3,4)+P0*P2*P4, (1,3)+(4,4)+P0*P3*P4}

={0+6+2,2+4+1,6+0+4}

=7 (ANSWER)

So bracketing of the matrix multiplication = ((A1A2)(A3A4))

GATE Example 1

Let A1, A2, A3, and A4 be four matrices of dimensions 10 x 5, 5 x 20, 20 x 10, and 10 x 5, respectively. The minimum number of scalar multiplications required to find the product A1A2A3A4 using the basic matrix multiplication method is

A. 1500

B. 2000

C. 500

D. 100

Answer: A

Explanation: We have many ways to do matrix chain multiplication because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result of the matrix chain multiplication obtained will remain the same. Here we have four matrices A1, A2, A3, and A4, we would have:

((A1A2)A3)A4 = ((A1(A2A3))A4) = (A1A2)(A3A4) = A1((A2A3)A4) = A1(A2(A3A4)).

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. Here, A1 is a 10 × 5 matrix, A2 is a 5 x 20 matrix, and A3 is a 20 x 10 matrix, and A4 is 10 x 5.

If we multiply two matrices A and B of order l x m and m x n respectively, then the number of scalar multiplications in the multiplication of A and B will be lxmxn.

Then,

The number of scalar multiplications required in the following sequence of matrices will be :

A1((A2A3)A4) = (5 x 20 x 10) + (5 x 10 x 5) + (10 x 5 x 5) = 1000 + 250 + 250 = 1500.

All other parenthesized options will require number of multiplications more than 1500.

GATE Example 2:

Let A1, A2, A3, and A4 be four matrices of dimensions 10 x 100, 100 x 20, 20 x 5, and 5 x 80, respectively. The minimum number of scalar multiplications required to find the product A1A2A3A4 using the basic matrix multiplication method is:

Answer:

Answer: Let’s apply the Bottom up approach

Make a table and solve the optimal part of the problems.

This table is for the number of multiplications needed.

A(i,j)= minimum multiplication needed for multiplying Ai and Aj.

calculate the minimum value of A(1,3):

={A(1,1)+(2,3)+P0*P1*P3 ,A(1,2)+A(3,3)+P0*P2*P3}

={0+10000+5000 , 20000+0+1000}

=21000

calculate the minimum value of A(2,4):

={A(2,2)+A(3,4)+P1*P2*P4 , A(2,3)+A(4,4)+P1*P3*P4}

=$\{0+8000+160000,10000+0+50000\}$

=50000

calculate the minimum value of A(1,4):

=$\{A(1,1)+(2,4)+P0*P1*P4\ ,\ (1,2)+(3,4)+P0*P2*P4,\ \ (1,3)+(4,4)+P0*P3*P4\}$

=$\{0+50000+80000,20000+8000+16000,15000+0+4000\}$

=19000(ANSWER)

A(1,1)=0	A(2,2)=0	A(3,3)=0	A(4,4)=0
A(1,2)=20000, split at	A(2,3)=10000, split at 2	A(3,4)=8000, split at 3	
A(1,3)=15000, split at 1	A(2,4)=50000, split at 3		
A(1,4)= 19000, split at 3			

So bracketing of the matrix multiplication = ((A1)(A2A3)(A4)) and minimum of number of multiplication = 19000

10. Travelling Salesman Problem-

The travelling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the travelling salesman needs to minimize the total length of the trip.

$$TSP(A,R) = \begin{cases} C(A,S)\,\text{if}\, R = \phi \\ \min\{(C[A,K] + TSP(K,R-K))\,\forall K \in R\} \end{cases}$$

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are ***(n - 1)!*** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S ∈ {1, 2, 3, … , n}*** that includes ***1***, and ***j ∈ S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠jC(S,j) =minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S ∈ {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   for all j ∈ S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i ∈ S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most 2n.n2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n.n2)

Time:

Time complexity: without dynamic programming = $O(n^n)$, with dynamic programming = $O(2^n.n^2)$

Space complexity: without dynamic programming = $O(n^2)$, with dynamic programming = $O(n^n)$.

→ It is one of the NP hard problem.

****