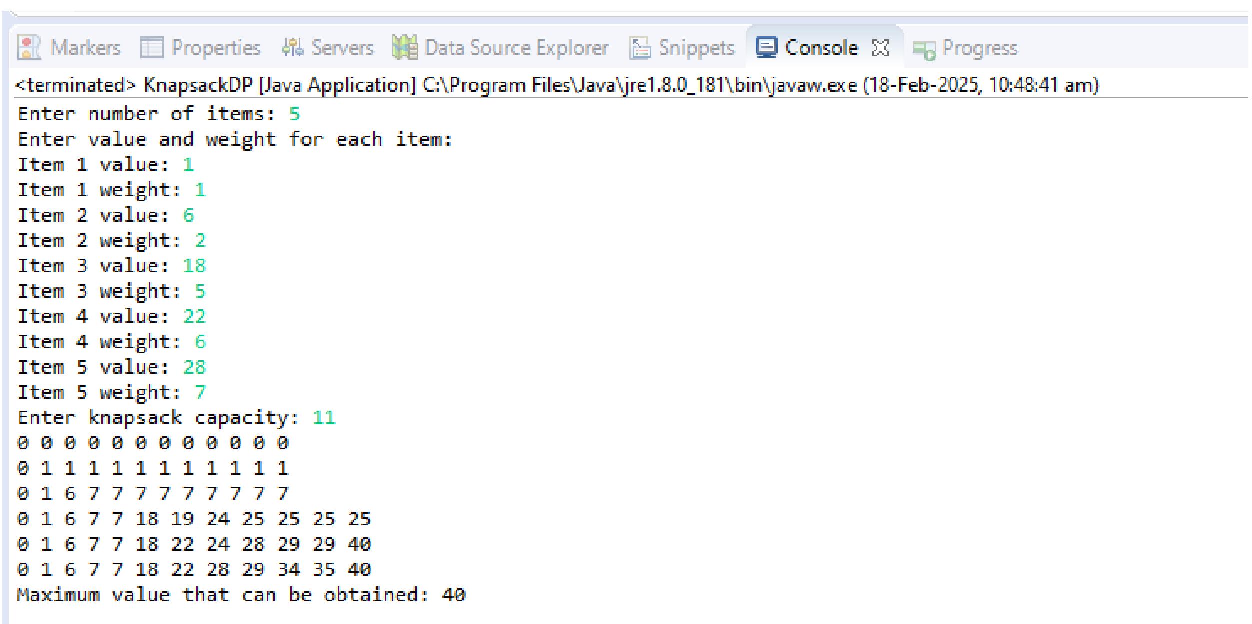
UNIT-4
Dynamic programming

Program no-14 -Write a program to implement Knapsack (0/1) using Dynamic Programming.

```
import java.util.Scanner;

public class KnapsackDP {
   // Function to solve 0/1 Knapsack Problem using Dynamic Programming
   public static int knapsack(int capacity, int weights[], int values[], int n) {
      int dp[][] = new int[n + 1][capacity + 1];

      // Build table dp[][] in bottom-up manner
      for (int i = 0; i <= n; i++) {
         for (int w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
               dp[i][w] = 0;
               System.out.print(dp[i][w]+"  ");
            } else if (weights[i - 1] <= w) {
               dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i-1]], dp[i - 1][w]);
               System.out.print(dp[i][w]+"  ");
            } else {
               dp[i][w] = dp[i - 1][w];
               System.out.print(dp[i][w]+"  ");
            }
         }

         System.out.println();
      }

      return dp[n][capacity];
   }

   public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);

      // Taking user input
      System.out.print("Enter number of items: ");
      int n = sc.nextInt();

      int values[] = new int[n];
      int weights[] = new int[n];

      System.out.println("Enter value and weight for each item:");
```

```
        for (int i = 0; i < n; i++) {
            System.out.print("Item " + (i + 1) + " value: ");
            values[i] = sc.nextInt();
            System.out.print("Item " + (i + 1) + " weight: ");
            weights[i] = sc.nextInt();
        }

        System.out.print("Enter knapsack capacity: ");
        int capacity = sc.nextInt();

        // Solving knapsack problem
        int maxProfit = knapsack(capacity, weights, values, n);

        System.out.println("Maximum value that can be obtained: " + maxProfit);

        sc.close();
    }
}
```

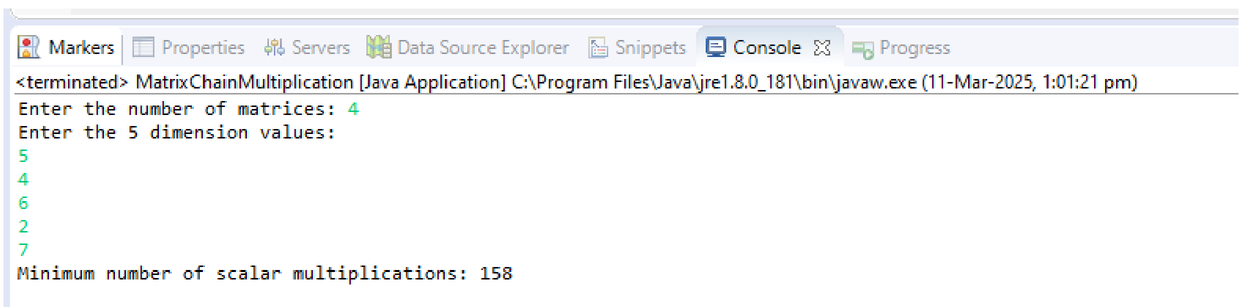



```
<terminated> KnapsackDP [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (18-Feb-2025, 10:48:41 am)
Enter number of items: 5
Enter value and weight for each item:
Item 1 value: 1
Item 1 weight: 1
Item 2 value: 6
Item 2 weight: 2
Item 3 value: 18
Item 3 weight: 5
Item 4 value: 22
Item 4 weight: 6
Item 5 value: 28
Item 5 weight: 7
Enter knapsack capacity: 11
0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1
0 1 6 7 7 7 7 7 7 7 7 7
0 1 6 7 7 18 19 24 25 25 25 25
0 1 6 7 7 18 22 24 28 29 29 40
0 1 6 7 7 18 22 28 29 34 35 40
Maximum value that can be obtained: 40
```

Program no-15 Write a program to implement Matrix chain multiplication using Dynamic Programming.

```
import java.util.Scanner;

public class MatrixChainMultiplication {

  public static int matrixChainOrder(int[] d)
{
    int n = d.length - 1; // Number of matrices
    int[][] m = new int[n + 1][n + 1]; // DP table

    // Fill DP table with base cases (single matrix cost is 0)

    for (int i = 1; i <= n; i++) {
      m[i][i] = 0;
    }

    // Compute the minimum multiplication cost

    for (int len = 2; len <= n; len++) { // Length of chain
      for (int i = 1; i <= n - len + 1; i++) {
        int j = i + len - 1; // End index
        m[i][j] = Integer.MAX_VALUE; // Initialize to large value

        for (int k = i; k < j; k++) {
          int cost = m[i][k] + m[k + 1][j] + d[i - 1] * d[k] * d[j];
          m[i][j] = Math.min(m[i][j], cost);
        }
      }
    }
    return m[1][n]; // Minimum cost for multiplying the full chain
  }

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Take input for number of matrices
    System.out.print("Enter the number of matrices: ");
    int n = scanner.nextInt();

    int[] d = new int[n + 1]; // Array to store dimensions

    // Take input for dimensions d[0], d[1], ..., d[n]
    System.out.println("Enter the " + (n + 1) + " dimension values:");
    for (int i = 0; i <= n; i++) {
```

```
            d[i] = scanner.nextInt();
        }

        // Compute minimum scalar multiplications
        int minMultiplications = matrixChainOrder(d);
        System.out.println("Minimum number of scalar multiplications: " + minMultiplications);


    }
}
```



Explanation:

int[] d = {5, 4, 6, 2, 7};

A1 → (5×4)

A2 → (4×6)

A3 → (6×2)

A4 → (2×7)

Step 1: Base Case Initialization

For single matrices,

m[i][i]=0

i \ j	1	2	3	4
1	0	∞	∞	∞
2	-	0	∞	∞
3	-	-	0	∞
4	-	-	-	0

Step 2: Compute Costs for Length 2 Chains (`m[i][i+1]`)

Compute m[1][2] (A1 × A2)
$m[1][2]=d[0]\times d[1]\times d[2]=5\times 4\times 6=120$

Compute m[2][3] (A2 × A3)
$m[2][3]=d[1]\times d[2]\times d[3]=4\times 6\times 2=48$

Compute m[3][4] (A3 × A4)
$m[3][4]=d[2]\times d[3]\times d[4]=6\times 2\times 7=84$

i \ j	1	2	3	4
1	0	120	∞	∞
2	-	0	48	∞
3	-	-	0	84
4	-	-	-	0

Step 3: Compute m[1][3] (A1 × A2 × A3)

We check different ways to **split**:

Split at k = 1 → (A1 × A2) × A3
$m[1][3]=m[1][1]+m[2][3]+(d[0]\times d[1]\times d[3])$

$=0+48+(5\times 4\times 2)=0+48+40=88$

Split at k = 2 → A1 × (A2 × A3)
$m[1][3]=m[1][2]+m[3][3]+(d[0]\times d[2]\times d[3])$

$=120+0+(5\times 6\times 2)=120+0+60=180$

Optimal Choice: m[1][3] = 88 (Minimum of 88 and 180)

Step 4: Compute m[2][4] (A2 × A3 × A4)

We check different splits:

Split at k = 2 → (A2 × A3) × A4
$m[2][4]=m[2][2]+m[3][4]+(d[1]\times d[2]\times d[4])$

$=0+84+(4\times 6\times 7)=0+84+168=252$

Split at k = 3 → A2 × (A3 × A4)
m[2][4]=m[2][3]+m[4][4]+(d[1]×d[3]×d[4])

= 104=48+0+(4×2×7)=48+0+56=104

Optimal Choice: m[2][4] = 104 (Minimum of 252 and 104)

i \ j	1	2	3	4
1	0	120	88	∞
2	-	0	48	104
3	-	-	0	84
4	-	-	-	0

Step 5: Compute m[1][4] (A1 × A2 × A3 × A4)

Split at k = 1 → (A1 × A2) × (A3 × A4)
m[1][4]=m[1][1]+m[2][4]+(d[0]×d[1]×d[4])

=0+104+(5×4×7)=0+104+140=244

Split at k = 2 → A1 × (A2 × A3 × A4)
m[1][4]=m[1][2]+m[3][4]+(d[0]×d[2]×d[4])

=120+84+(5×6×7)=120+84+210=414

Split at k = 3 → (A1 × A2 × A3) × A4
m[1][4]=m[1][3]+m[4][4]+(d[0]×d[3]×d[4])

= 158=88+0+(5×2×7)=88+0+70=158

Optimal Choice: m[1][4] = 158

Minimum number of multiplications: 158

Program no-16 Write a program to implement all pair shortest paths (Floyd Warshall) using Dynamic Programming.

```
import java.util.Arrays;

public class FloydWarshall {
    final static int INF = 99999; // A high value to represent infinity

    public static void floydWarshall(int graph[][]) {
```

```
    int V = graph.length;
    int dist[][] = new int[V][V];

    // Copy input graph into dist matrix
    for (int i = 0; i < V; i++)
        dist[i] = Arrays.copyOf(graph[i], V);

    // Floyd-Warshall Algorithm
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    // Print the shortest distances
    printSolution(dist);
}

public static void printSolution(int dist[][]) {
    int V = dist.length;
    System.out.println("Shortest distance matrix:");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j] + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int graph[][] = {
        {0, 8, INF, 1},
        {INF, 0, 1, INF},
        {4, INF, 0, INF},
        {INF, 2, 9, 0},

    };

    floydWarshall(graph);
```

```
    }
}
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Progress
<terminated> FloydWarshall [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (03-Mar-2025, 4:01:32 pm)
Shortest distance matrix:
0 3 4 1
5 0 1 6
4 7 0 5
7 2 3 0
```

Working:

`dist[i][k] != INF && dist[k][j] != INF`

- We are checking whether there **exists a valid path** from `i → k` **and** from `k → j`.
- If either `dist[i][k]` or `dist[k][j]` is `INF` (i.e., no direct or indirect path exists), then `dist[i][j]` should **not be updated** using `k`.

- **`dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j])`**

- This follows the fundamental logic of Floyd-Warshall:
  - **Either keep the current shortest distance** (`dist[i][j]`),
  - **Or update it if going through `k` gives a shorter path** (`dist[i][k] + dist[k][j]`)