Part 7

Dynamic Programming Algorithm Description

To solve the knapsack problem using dynamic programming, I define a 2D table dp[i][w], where i represents the number of experiments considered and w is the current weight capacity. Each cell dp[i][w] stores the maximum rating achievable using the first i experiments without exceeding weight w.

The algorithm works as follows:

1. **Initialization**:
   Set dp[0][w] = 0 for all w from 0 to 700. This represents the case where no experiments are selected.
2. **Iteration**:
   For each experiment i from 1 to n, and for each weight w from 0 to 700:
   - If the experiment's weight is more than w, we cannot include it:
     dp[i][w] = dp[i-1][w]
   - Otherwise, we choose the better of:
   - Excluding the experiment: dp[i-1][w]
   - Including it: dp[i-1][w - weight[i]] + rating[i]
3. **Result**:
   The final answer is stored in dp[n][700], which gives the maximum rating achievable. We then backtrack through the table to find the subset of experiments that were selected.

Part 8:

Dynamic Programming Implementation Report

I implemented the dynamic programming solution in Java using a 2D array dp[i][w] to store the maximum rating for each weight capacity. Each experiment is represented by a class with fields for name, weight, and rating. After filling the DP table, I used backtracking to identify the selected experiments.

Java Code Summary:

- **Class**: Experiment stores name, weight, and rating.
- **DP Table**: dp[i][w] is a 2D array storing max ratings.

- **Backtracking**: After filling the table, we trace back to find which experiments were selected.

Output:

 Maximum Rating Achieved: 43

- **Selected Experiments**:
    - Micrometeorites (170 kg, 9)
    - Cosmic Rays (80 kg, 7)
    - Mice Tumors (65 kg, 8)
    - Relativity (104 kg, 8)
    - Cloud Patterns (36 kg, 5)
    - Seed Viability (7 kg, 4)
    - Yeast Fermentation (27 kg, 4)
- **Total Weight Used**: 489 kg

Code Flexibility:

This implementation can be adapted to accept arbitrary rating functions by modifying the Experiment class to include a method such as getAdjustedRating(), which computes the rating dynamically based on user-defined logic.

Part 9:

Walkthrough and Debugging Strategy

The DP table is initialized with zeros. For each experiment, we iterate through all weight capacities from 0 to 700. If the experiment fits, we update the table with the maximum of including or excluding it.

Example:

- After processing "Cloud Patterns" (36 kg, 5 rating), all weights ≥ 36 have a value of 5.
- After adding "Micrometeorites" (170 kg, 9 rating), weights ≥ 170 are updated to reflect the best combination of "Cloud Patterns" and "Micrometeorites".

Backtracking:

Starting from dp[12][700], we trace back:

- If dp[i][w] != dp[i-1][w], experiment i was included.
- We subtract its weight from w and continue.

Memory Considerations:

The current implementation uses a 2D array of size 12 × 701, which is approximately 8,400 integers (~33 KB). This is efficient for small datasets. For larger datasets, we can optimize to a 1D array:

int[] dp = new int[MAX_WEIGHT + 1];

Debugging Strategy:

1. Method 1: Print the DP Table
2. Print the values during execution to see how ratings are calculated. This helps verify that the table is filled correctly.
3. Method 2: Trace Backtracking
4. Print each experiment as it is selected during backtracking. This confirms that the final subset matches the expected rating.
5. Method 3: Use Smaller Test Cases
6. Run the algorithm with a smaller weight limit (e.g., 100 kg) and fewer experiments to manually verify the output.
7. Method 4: Graphical Visualization (Optional)
8. Export the DP table to a CSV file and visualize it using Excel or Python. A heatmap or line chart can show how ratings increase with weight.