Knapsack Problem

The *Knapsack Problem* is a decision and optimization problem where you must select items with given weights and values to maximize the total value without exceeding the weight capacity of the knapsack. Also called a classic optimization problem.

Why Do We Need the Knapsack Problem?

- Optimal resource allocation
- Budget distribution
- Load balancing
- Inventory selection
- Cryptography
- Data compression

Example : Imagine a traveler with a 15 kg bag who must choose items of different weights and values to maximize total value without exceeding the weight limit.

Cargo loading: selecting packages to maximize value within weight limit of a truck

- **Items:** electronics (high value, moderate weight), books (low value, high weight), clothing (moderate value, low weight)

Knapsack: truck with 1000 kg capacity

Knapsack Problem

Types of Knapsack Problems:

➢ 0/1 Knapsack

- Each item can be taken either 0 or 1 time
- No fractions allowed
- Solved using Dynamic Programming
- Example : A traveler has space for only one laptop. They must choose between two laptops—either take one (1) or leave it (0); they cannot take half of a laptop.

➢ Fractional Knapsack

- Items can be divided into fractions
- You can take a portion of an item
- Solved using the Greedy Algorithm (value/weight ratio)
- Example : A gold merchant fills a bag with gold dust. They can take any fraction of the gold, such as 0.5 kg or 0.2 kg, based on the value per gram.

➢ Unbounded Knapsack

- Each item can be taken multiple times
- Useful when items have unlimited supply
- Solved using Dynamic Programming
- Example : A shopkeeper can take unlimited packets of rice, sugar, or flour to fill storage, as supplies are restocked continuously.

Knapsack Problem

Knapsack Problem – Solving Methods

1. Brute-Force Approach

- Examines all possible combinations of items
- Ensures the optimal solution
- Highly inefficient for large inputs (Time complexity = 2^n)

2. Recursive Approach

- Solves the problem using a top-down divide-and-conquer method
- Uses recursion to explore include/exclude decisions
- May lead to overlapping subproblems

3. Dynamic Programming

- Efficient optimization of the recursive approach
- Uses memoization or tabulation
- Avoids repeated calculations
- Best suited for 0/1 and Unbounded Knapsack

Knapsack Problem

Knapsack Problem – Solving Methods

4. Greedy Algorithm

- Selects items based on value/weight ratio
- Works only for Fractional Knapsack (gives optimal result)
- Not suitable for 0/1 Knapsack

5. Genetic Algorithm

- Uses principles of evolution and natural selection
- Generates near-optimal solutions for large and complex problem sizes
- Useful when exact DP solutions become too slow

6. Branch-and-Bound Technique

- Systematically explores different combinations
- Uses upper/lower bounds to prune suboptimal branches
- Efficient for finding exact solutions in constrained search spaces

Knapsack Problem

Dynamic programming is a technique for solving optimization problems such as the knapsack problem. It entails breaking the problem down into smaller subproblems and recording the solutions in a table. Larger subproblems can be solved using the stored solutions of smaller subproblems, resulting in a more efficient method than brute force or recursive approaches.

Why Dynamic Programming?

- Subproblems repeat → DP avoids recomputation
- DP builds a bottom-up table(*The DP table has rows = items + 1 and columns = capacity + 1*)
- Guarantees optimal solution
- Efficient for medium-size input

DP Table Formula: For each item i and weight w, there are two case:

Case 1: Item can fit (w[i] ≤ w), then We have two choices:

- **Include the item**: Value = v[i] + dp[i-1][w - w[i]]
- **Exclude the item**: Value = dp[i-1][w]
- Select max, i.e., dp[i][w] = max(v[i] + dp[i-1][w - w[i]], dp[i-1][w])

Case 2: Item cannot fit (w[i] > w), then We have only one choice, i.e., dp[i][w] = dp[i-1][w]

Knapsack Problem

Dynamic programming Example:

- Number of items: 4
- Weights: 1 3 4 5
- Values: 1 4 5 7
- Capacity: 7

Item	Weight	Value
1	1	1
2	3	4
3	4	5
4	5	7

Build a DP table with:

- Rows = Items + 1 (0 to 4)
- Columns = Capacity + 1 (0 to 7)

dp[i][w] = Maximum value achievable using first i items with capacity w.

DP Table:

Capacity → Items↓	0	1	2	3	4	5	6	7
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Item	Weight	Value
1	1	1

Explanation:

Row 1: Only Item 1 (weight = 1, value = 1)

Capacity →	0	1	2	3	4	5	6	7
Items↓								
1 Item	0	1	1	1	1	1	1	1

For capacity:

- 0 → 0 (w=0 → dp[1][0]=0)
- 1 → we can take item → 1
- 2 to 7 → item still fits → value remains 1

DP Table:

Capacity →	0	1	2	3	4	5	6	7
Items↓								
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Explanation:

Row 2: Items 1 & 2 available (Item2: weight = 3, value = 4)

Capacity →0 1 2 3 4 5 6 7

Items↓

2 Item 0 1 1 4 5 5 5 5

We check each capacity:

Capacity 1 & 2: Item 2 (weight 3) does not fit→ copy value from above row i.e., 0 1 1

Capacity 3: Item 2 fits → choose max:

- Not taking item2 → 1
- Taking item2 → 4, put 4

Capacity 4: Try both:

- Without item2 → 1
- With item2 → value 4 + dp[1][1] = 4 + 1 = 5

Item	Weight	Value
1	1	1
2	3	4

DP Table:

Capacity →	0	1	2	3	4	5	6	7
Items↓								
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Explanation:

Row 3: Items 1,2,3 available (Item3 weight=4, value=5)

Capacity →0 1 2 3 4 5 6 7

Items↓

Item 3 0 1 1 4 5 6 6 9

We check each capacity:

Capacity 1, 2 & 3: Item 2 (weight 3) already checked→ copy value from above row i.e., 0 1 1 4

Capacity 4: Try and select max:

- Without item3 → 5
- With item3 → value 5 + dp[2][0] = 5 + 0 = 5, put 5

Capacity 5: Try select max:

- Without item3 → 5
- With item3 → 5 + dp[2][1] = 5 + 1 = 6, put 6, also same for Capacity 6

Capacity 7: Try select max:

- Without item3 → 5
- With item3 → 5 + dp[2][3] = 5 + 4 = 9, put 9

Item	Weight	Value
1	1	1
2	3	4
3	4	5

DP Table:

Capacity →	0	1	2	3	4	5	6	7
Items↓								
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Item	Weight	Value
1	1	1
2	3	4
3	4	5
4	5	7

Explanation:

Row 4: Items 1–4 available (Item4 weight=5, value=7)

Capacity →0 1 2 3 4 5 6 7

Items↓

Item 4 0 1 1 4 5 7 8 9

We check each capacity:

Capacity 1, 2, 3 & 4: item4 doesn't fit at w≤4 → copy dp[3][w] → 0 1 1 4 5

i.e., 0 1 1 4 5

Capacity 5: Try select max:

- Without item4 → 6
- With item4 → 7+ dp[3][0] = 7 + 0 = 7, put 7

Capacity 6: Try select max:

- Without item4 → 6
- With item4 → 7 + dp[3][1] = 7 + 1 = 8, put 8

Capacity 6: Try select max:

- Without item4 → 9
- With item4 → 7 + dp[3][2] = 7 + 1 = 8, put 9

DP Table:

Capacity →	0	1	2	3	4	5	6	7
Items↓								
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Explanation:

- So, the optimal value with all 4 items and capacity 7 is 9.
- Backtracking to find selected items.
- So selected items are Item 2 and Item 3:
- Total weight = 3 + 4 = 7
- Total value = 4 + 5 = 9

Item	Weight	Value
1	1	1
2	3	4
3	4	5
4	5	7

DP Table:

Capacity → Items↓	0	1	2	3	4	5	6	7
0 items	0	0	0	0	0	0	0	0
Item 1	0	1	1	1	1	1	1	1
Item 2	0	1	1	4	5	5	5	5
Item 3	0	1	1	4	5	6	6	9
Item 4	0	1	1	4	5	7	8	9

Knapsack Problem

Q55. Write a C program to solve the 0/1 Knapsack Problem using Dynamic Programming. The program should print the DP table and the selected items.