The Knapsack Problem: Theory Overview

Varun Kumar

July 8, 2025

Contents

Listings

List of Algorithms

1 Introduction

The Knapsack Problem is a fundamental problem in combinatorial optimization. It models a situation where a set of items, each with a given weight and value, must be selected to include in a knapsack of limited capacity such that the total value is maximized without exceeding the weight constraint.

This problem arises in many real-world scenarios such as resource allocation, budgeting, cargo loading, and decision-making under constraints. There are two common variants:

- **Fractional Knapsack:** Items can be divided into smaller parts.
- **0/1 Knapsack:** Each item is either fully taken or not taken at all.

While the fractional version can be solved efficiently using a greedy strategy, the 0/1 version requires dynamic programming to guarantee an optimal solution.

2 Problem Statement

Given n items, each with a value v_i and weight w_i, and a knapsack with capacity W, the goal is to choose a subset of the items to maximize the total value without exceeding the weight capacity.

- Maximize: $\sum v_i x_i$
- Subject to: $\sum w_i x_i \leq W$
- Where $x_i \in \{0, 1\}$ for the 0/1 Knapsack

3 Fractional Knapsack (Greedy Approach)

- Items can be broken into fractions.
- Sort items based on the value-to-weight ratio $\frac{v_i}{w_i}$ in descending order.
- Pick the item with the highest ratio until the knapsack is full.
- Time Complexity: $\mathcal{O}(n \log n)$

Note: This method gives the optimal solution for the fractional variant only.

3.1 Problem Statement

You are given n items. Each item has:

- Value v_i
- Weight w_i

You are also given a knapsack with capacity W. You can take **fractions** of items.

Goal: Maximize total value such that the total weight does not exceed W.

3.2 Greedy Strategy

To solve the Fractional Knapsack problem, use the following greedy approach:

1. Compute the value-to-weight ratio $\frac{v_i}{w_i}$ for each item.
2. Sort items by this ratio in **descending** order.
3. Initialize `currentWeight = 0` and `totalValue = 0`.
4. For each item:
   - If the item fits fully ($w_i \leq$ `remaining capacity`), take the whole item.
   - Else, take the fraction of the item that fits.
5. Stop when the knapsack is full.

3.3 Worked Example

Given:

Item	**Value (v_i)**	**Weight (w_i)**
1	60	10
2	100	20
3	120	30

Knapsack capacity: $W = 50$

Step 1: Compute Value/Weight Ratios

Item	$\frac{v_i}{w_i}$
1	60 / 10 = 6
2	100 / 20 = 5
3	120 / 30 = 4

Order of selection based on ratio: **Item 1 → Item 2 → Item 3**

Step 2: Pick Items Greedily

- Take all of Item 1 (10kg): Value += 60, remaining capacity = 40
- Take all of Item 2 (20kg): Value += 100, remaining capacity = 20
- Take $\frac{2}{3}$ of Item 3 (20kg of 30kg):
  Value $+= 120 \times \frac{20}{30} = 80$

3.4 Final Answer

- Total value obtained $= 60 + 100 + 80 = 240$
- Total weight used $= 10 + 20 + 20 = 50$ (Knapsack is full)

Algorithm 1 Fractional Knapsack

Require: A list of n items with value v_i and weight w_i, capacity W
Ensure: Maximum total value without exceeding capacity

Compute $\frac{v_i}{w_i}$ for each item
Sort items by decreasing $\frac{v_i}{w_i}$
$totalValue \leftarrow 0$
$currentWeight \leftarrow 0$
for each item i in sorted order **do**
  if $currentWeight + w_i \leq W$ **then**
    Take the whole item
    $currentWeight \leftarrow currentWeight + w_i$
    $totalValue \leftarrow totalValue + v_i$
  else
    Take fraction $(W - currentWeight)/w_i$ of item i
    $totalValue \leftarrow totalValue + v_i \times \frac{W - currentWeight}{w_i}$
    break
  end if
end for
return $totalValue$

4 Fractional Knapsack Code Implementation

4.1 Python Implementation

```python
1 def fractional_knapsack(values, weights, capacity):
2     items = [(v, w, v/w) for v, w in zip(values, weights)]
3     items.sort(key=lambda x: x[2], reverse=True)
4
5     total_value = 0.0
6     for value, weight, ratio in items:
7         if capacity >= weight:
8             total_value += value
9             capacity -= weight
10        else:
11            total_value += ratio * capacity
12            break
13    return total_value
```

Listing 1: Fractional Knapsack in Python

4.2 Line-by-Line Breakdown

Line 1

```
def fractional_knapsack(values, weights, capacity):
```

- Defines a function with:
  - `values` – list of item values
  - `weights` – list of item weights
  - `capacity` – total capacity of the knapsack

Line 2

```
items = [(v, w, v/w) for v, w in zip(values, weights)]
```

- Creates a list of tuples `(value, weight, ratio)`.
- `zip(values, weights)` combines both lists.
- `v/w` computes the value-to-weight ratio.

Example:

```
values  = [60, 100, 120]
weights = [10, 20, 30]
# → items = [(60, 10, 6.0), (100, 20, 5.0), (120, 30, 4.0)]
```

Line 3

```
items.sort(key=lambda x: x[2], reverse=True)
```

- Sorts items in descending order by value-to-weight ratio.
- Highest "value per kg" items come first.

Line 4

```
total_value = 0.0
```

- Initializes the total value of items added to the knapsack.

Line 5–11 (Main loop)

```
for value, weight, ratio in items:
```

- Iterates over each sorted item.

  ```
  if capacity >= weight:
      total_value += value
      capacity -= weight
  ```

- If the full item fits, take it completely.
- Add its value, and subtract its weight from capacity.

  ```
  else:
      total_value += ratio * capacity
      break
  ```

- If the item doesn't fully fit:
  - Take a fraction that fits.
  - Add proportional value.
  - Break the loop as knapsack is full.

Line 12

```
return total_value
```

- Return the maximum value that can be carried in the knapsack.

Example Call

```
fractional_knapsack([60, 100, 120], [10, 20, 30], 50)
```

Output:

```
240.0
```

Why?

- Take all of item 1: 60 (10kg)
- Take all of item 2: 100 (20kg)
- Take 2/3 of item 3: $120 \times \frac{2}{3} = 80$

Total $= 60 + 100 + 80 = 240$

Example Input and Output

```
# Input
values  = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50

# Function call
result = fractional_knapsack(values, weights, capacity)

# Output
print(result)  # Output: 240.0
```

4.3 C++ Implementation

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Item {
    int value, weight;
};

bool compare(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(int W, vector<Item> &items) {
    sort(items.begin(), items.end(), compare);
    double totalValue = 0.0;

    for (Item &item : items) {
        if (W >= item.weight) {
            W -= item.weight;
            totalValue += item.value;
        } else {
            totalValue += item.value * ((double)W / item.weight
               );
            break;
        }
    }

    return totalValue;
}
```

Listing 2: Fractional Knapsack in C++

5 0/1 Knapsack (Dynamic Programming)

- Items cannot be broken; each item is either taken or not.
- Define a DP table: $dp[i][w]$ = maximum value for first i items with weight limit w.
- Recurrence relation:

$$dp[i][w] = \begin{cases} dp[i-1][w], & \text{if } w_i > w \\ \max(dp[i-1][w], dp[i-1][w-w_i] + v_i), & \text{otherwise} \end{cases}$$

- Time Complexity: $\mathcal{O}(nW)$

Note: This method guarantees the optimal solution for the 0/1 Knapsack.

5.1 0/1 Knapsack Problem (Dynamic Programming)

Problem Statement

Given n items with:

- Values: v_i
- Weights: w_i
- A knapsack of capacity W

Determine the maximum total value that can be obtained by selecting a subset of items such that:

- Each item is either fully included or excluded (no fractions)
- Total weight $\leq W$

Example

Given:

Item	**Value** (v_i)	**Weight** (w_i)
1	60	1
2	100	2
3	120	3

Knapsack Capacity: $W = 5$

Approach: Dynamic Programming

Let $dp[i][w]$ be the maximum value that can be obtained by considering the first i items with capacity w.

We use the recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \\ \max(dp[i-1][w], dp[i-1][w-w_i] + v_i) & \text{otherwise} \end{cases}$$

—

DP Table Construction

Let's build a table for $dp[0..3][0..5]$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

Explanation:

- Row i considers first i items
- Column w represents capacity
- Final answer is at $dp[3][5] = \boxed{220}$

Result

The maximum value we can carry in the knapsack is:

$$\boxed{220}$$

Items Selected

Using backtracking from the table:

- Item 3 is included (weight 3, value 120)
- Remaining capacity = 2
- Item 2 is included (weight 2, value 100)

Final Selection: Item 2 and Item 3
Total Weight: $2 + 3 = 5$, **Total Value:** $100 + 120 = \boxed{220}$

5.2 Step-by-Step Construction of DP Table

We define a DP table `dp[i][w]` where:

- i = number of items considered
- w = current knapsack capacity (from 0 to W)
- `dp[i][w]` stores the **maximum value** using first i items and capacity w

Initialization:

- For all w: `dp[0][w] = 0` (No items $\rightarrow$ 0 value)
- For all i: `dp[i][0] = 0` (0 capacity $\rightarrow$ 0 value)

Input:

Item i	Value v_i	Weight w_i
1	60	1
2	100	2
3	120	3

Capacity $W = 5$

Filling the table:
We use this recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \quad \text{(Item can't fit)} \\ \max(dp[i-1][w], dp[i-1][w - w_i] + v_i) & \text{otherwise} \end{cases}$$

—

Row 1 (Item 1: value=60, weight=1):

- $w = 0$: can't include $\to dp[1][0] = 0$
- $w = 1$: can include $\to dp[1][1] = \max(0, 0 + 60) = 60$
- $w = 2$: $dp[1][2] = \max(0, 0 + 60) = 60$
- $w = 3$: $dp[1][3] = \max(0, 0 + 60) = 60$
- $w = 4$: $dp[1][4] = \max(0, 0 + 60) = 60$
- $w = 5$: $dp[1][5] = \max(0, 0 + 60) = 60$

Row 2 (Item 2: value=100, weight=2):

- $w = 0, 1$: can't include $\to$ same as above
- $w = 2$: $dp[2][2] = \max(60, 0 + 100) = 100$
- $w = 3$: $dp[2][3] = \max(60, 60 + 100) = 160$
- $w = 4$: $dp[2][4] = \max(60, 60 + 100) = 160$
- $w = 5$: $dp[2][5] = \max(60, 60 + 100) = 160$

Row 3 (Item 3: value=120, weight=3):

- $w = 0, 1, 2$: can't include $\to$ same as above
- $w = 3$: $dp[3][3] = \max(160, 0 + 120) = 160$
- $w = 4$: $dp[3][4] = \max(160, 60 + 120) = 180$
- $w = 5$: $dp[3][5] = \max(160, 100 + 120) = \boxed{220}$

—

Final DP Table

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	**220**

Final Answer: Maximum value = $\boxed{220}$

6 Theoretical Logic Behind 0/1 Knapsack Implementation

The 0/1 Knapsack problem is a classic example of **Dynamic Programming**, where the optimal solution of a problem depends on the optimal solutions of its subproblems. The idea is to build a table that stores the maximum value for each subproblem defined by:

- The number of items considered so far (i)
- The remaining capacity of the knapsack (w)

Let $dp[i][w]$ represent the maximum value achievable by considering the first i items with a knapsack capacity w.

Recurrence Relation

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \quad \text{(Item doesn't fit)} \\ \max(dp[i-1][w], dp[i-1][w-w_i] + v_i) & \text{otherwise} \end{cases}$$

Where:

- v_i is the value of the i^{th} item
- w_i is the weight of the i^{th} item

Base Case Initialization

- $dp[0][w] = 0$ for all w (no items means no value)
- $dp[i][0] = 0$ for all i (zero capacity means no value)

Bottom-Up Construction

We iterate over all items and capacities to fill the DP table. For each item and capacity:

- If the item weight is more than the current capacity, we can't include it, so we inherit the value from above.
- Otherwise, we decide whether to include the item or not by taking the maximum of:
  - Excluding the item: $dp[i-1][w]$
  - Including the item: $dp[i-1][w-w_i] + v_i$

Final Answer

The final result is stored in $dp[n][W]$, where:

- n is the total number of items
- W is the maximum capacity of the knapsack

This value represents the **maximum total value** that can be achieved without exceeding the knapsack's capacity using the given items.

6.1 Python Implementation

```python
def knapsack(values, weights, capacity):
    n = len(values)
    # Initialize DP table with 0
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    # Fill the table
    for i in range(1, n + 1):
        for w in range(0, capacity + 1):
            if weights[i - 1] > w:
                dp[i][w] = dp[i - 1][w]  # Can't include item
            else:
                # Max of including or excluding the item
                dp[i][w] = max(dp[i - 1][w],
                               dp[i - 1][w - weights[i - 1]] + values[i - 1])

    return dp[n][capacity]
```

Listing 3: 0/1 Knapsack in Python

```python
values = [60, 100, 120]
weights = [1, 2, 3]
capacity = 5

result = knapsack(values, weights, capacity)
print(result)  # Output: 220
```

6.2 C++ Implementation

```cpp
#include <iostream>
#include <vector>
using namespace std;

int knapsack(vector<int>& values, vector<int>& weights, int
    capacity) {
    int n = values.size();
    vector<vector<int>> dp(n + 1,vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int w = 0; w <= capacity; ++w) {
            if (weights[i - 1] > w) {
                dp[i][w] = dp[i - 1][w]; // Cannot include item
            } else {
                dp[i][w] = max(
                    dp[i - 1][w],
                    dp[i-1][w - weights[i-1]] + values[i-1]
                );
            }
        }
    }
    return dp[n][capacity];
}

int main() {
    int n, capacity;
    cout << "Enter number of items: ";
    cin >> n;

    vector<int> values(n), weights(n);

    cout << "Enter values:\n";
    for (int i = 0; i < n; ++i) cin >> values[i];
    cout << "Enter weights:\n";
    for (int i = 0; i < n; ++i) cin >> weights[i];
    cout << "Enter knapsack capacity: ";
    cin >> capacity;
    int result = knapsack(values, weights, capacity);
    cout << "Maximum value: " << result << endl;

    return 0;
}
```

Listing 4: 0/1 Knapsack in C++

7 Key Points to Remember

1. Knapsack problems involve selecting items to maximize total value under a weight constraint.
2. 0/1 Knapsack: each item is either taken fully or not at all.
3. Fractional Knapsack: items can be broken and partially taken.
4. 0/1 Knapsack is solved using Dynamic Programming.
5. Fractional Knapsack is solved using Greedy strategy.
6. 0/1 Knapsack does not follow greedy choice property.
7. Fractional Knapsack follows both greedy choice and optimal substructure properties.
8. Time complexity of 0/1 Knapsack: $\mathcal{O}(nW)$, where n = items, W = capacity.
9. Time complexity of Fractional Knapsack: $\mathcal{O}(n \log n)$ (due to sorting).
10. In 0/1 Knapsack, a DP table is built based on item count and capacity.
11. Backtracking the DP table can give the list of selected items.
12. Fractional Knapsack always yields the optimal solution.
13. 0/1 Knapsack may require approximation if constraints are large.
14. Space optimization is possible using 1D arrays in 0/1 Knapsack.
15. Knapsack is a classic NP-complete problem (for 0/1 case).

8 Real-World Applications

- **Cargo Loading:** Selecting goods to load onto a truck/ship with weight limits.
- **Budget Allocation:** Choosing the best set of projects under a limited budget.
- **Resource Scheduling:** Assigning limited compute or memory to jobs for maximum value.
- **Investment Planning:** Selecting the best combination of assets under risk/capital constraints.
- **Time Management:** Choosing the most rewarding activities within limited time.
- **Memory Management in OS:** Efficiently selecting data blocks to keep in memory.
- **Marketing Campaigns:** Allocating limited resources (ad budget, team) to most impactful actions.
- **Cloud Computing:** VM placement and workload optimization under physical constraints.
- **Logistics Optimization:** Selecting orders or shipments based on value-to-weight ratio.
- **Resource-constrained AI Agents:** Selecting actions under energy/-time restrictions.

9 Comparison

Feature	**Fractional Knapsack**	**0/1 Knapsack**
Problem Type	Optimization with fractions	Combinatorial subset selection
Item Division	Allowed (can take part of item)	Not Allowed (take whole or none)
Approach	Greedy	Dynamic Programming
Guarantees Optimality	Only for fractional case	Yes, for 0/1 selection
Time Complexity	$\mathcal{O}(n \log n)$	$\mathcal{O}(nW)$

10 Conclusion

The Knapsack problem illustrates a fundamental trade-off between resource usage and value optimization. It exists in two major forms:

- The **0/1 Knapsack**, which is discrete and requires Dynamic Programming due to its combinatorial nature.
- The **Fractional Knapsack**, which is continuous and optimally solvable using a Greedy approach.

These problems are widely applicable in fields like operations research, finance, computer science, and logistics. A deep understanding of Knapsack variants not only helps in mastering algorithmic techniques but also equips one to model and solve many real-world optimization problems effectively.