Koneru Lakshmaiah Education Foundation
(Deemed to be University estd. u/s. 3 of the UGC Act, 1956)
Off-Campus: Bachupally-Gandimaisamma Road, Bowrampet, Hyderabad, Telangana - 500 043.
Phone No: 7815926816, www.klh.edu.in

Department of Computer Science and Engineering

2025-2026
Odd Semester

DESIGN AND ANALYSIS ALGORITHMS
(24CS2203)

ALM – PROJECT BASED LEARNING

Greedy Knapsack in Disaster Relief

SAI SRI KRUTHI. S **2420090003**

VIZZAPU AKSHAYA **2420090087**

COURSE INSTRUCTOR

Dr. J Sirisha Devi
Professor
Department of Computer Science and Engineering

Greedy Knapsack in Disaster Relief

The Greedy Knapsack algorithm is a key optimization tool in disaster relief, helping organizations pack the most crucial resources into limited transport capacities for rapid and effective emergency response. Below is a comprehensive case study illustrating the theoretical approach, practical application, and tangible examples in real disaster scenarios.

Overview of the Greedy Knapsack Algorithm

The Greedy Knapsack approach is used to solve variation of the classical knapsack problem, particularly the Fractional Knapsack problem. In this setting, supplies/items can be divided (e.g., boxes of medicine, bags of rice) and the goal is to maximize the sum of a "value" metric—such as utility, lives saved, or need met—subject to weight or volume constraints. The greedy solution is to choose items based on the highest value-to-weight (or volume) ratio, filling the knapsack until the limit is reached.

Disaster Relief Context

During a disaster, relief organizations must rapidly allocate and transport resources (such as food, water, medical aid, and shelter materials) to affected areas. Transport vehicles, like helicopters or trucks, have strict capacity limits due to load, space, or travel constraints**.**

Resource selection often occurs under pressure:

- There is incomplete information about future needs.
- Supplies vary in weight and criticality.
- The wrong allocations delay lifesaving aid.

Case Study: Greedy Knapsack for Medical Supply Allocation

Scenario Description:

After a major earthquake, a relief agency has a single helicopter with a payload capacity of 15 kg. There is a warehouse with the following available supplies:

Item	**Weight (kg)**	**Urgency Value (utility points)**
Antibiotics Kit	5	10
Painkillers Pack	3	6
Bandages Bundle	4	7

Item	Weight (kg)	Urgency Value (utility points)
IV Fluids	6	8
Water Purifier Tabs	2	4

The objective: maximize total "urgency value" delivered to a remote field clinic.

Application of the Greedy Algorithm

Steps:

1. **Calculate Value-to-Weight Ratios:**

Item	Value/Weight ratio
Antibiotics Kit	2.0
Painkillers Pack	2.0
Bandages Bundle	1.75
IV Fluids	1.33
Water Purifier Tabs	2.0

2. Sort supplies by highest ratio.
3. Select items in order, starting from highest ratio, until reaching the 15 kg capacity:

- Select Antibiotics Kit (5 kg), Painkillers Pack (3 kg), Water Purifier Tabs (2 kg), Bandages Bundle (4 kg): Total weight = 14 kg. Capacity left = 1 kg.
- Cannot add IV Fluids (6 kg) without exceeding limit.

4. If fractional items allowed, fill 1 kg out of IV Fluids (1.33 value/kg), adding about 1.33 value.

Results

- Total value achieved: 10 (Antibiotics) + 6 (Painkillers) + 4 (Water Tabs) + 7 (Bandages) + 1.33 (partial IVs) = 28.33 utility points.

ALGORITHM / PSEUDO CODE

Algorithm: Greedy Knapsack in Disaster Relief

Input:

- A set of items, each with a weight and urgency value.
- Maximum payload capacity of the helicopter.

Output:

- Set of selected items (including fractions if necessary).
- Maximum urgency value achievable.

Steps:

1. Calculate Ratios
    - For each item, compute the value-to-weight ratio = (urgency value ÷ weight).
2. Sort Items
    - Arrange all items in descending order of ratio (highest ratio first).
3. Initialize
    - Set total urgency value = 0.
    - Set remaining capacity = maximum payload.
    - Create an empty list for selected items.
4. Select Items
    - Traverse the sorted items one by one:
        - If the item's weight is less than or equal to remaining capacity → take the whole item.
        - If the item's weight is greater than remaining capacity → take only the fraction that fits, then stop.
5. Update Totals
    - Add the value (full or fractional) to total urgency.
    - Reduce remaining capacity accordingly.
6. Return Results
    - Output the chosen items and the total urgency value.

Pseudo Code for Greedy Knapsack in Disaster Relief

Algorithm Greedy Knapsack Disaster Relief (Items, Capacity):

Input:

Items = { (weight[i], value[i]) for i = 1 to n } // Each item has weight and urgency value

Capacity = C // Maximum payload (e.g., 15 kg)

Output:

SelectedItems, MaximumUrgencyValue // Final chosen items and total urgency score

Begin

// Step 1: Compute value-to-weight ratio for each item

For each item i in Items do

ratio[i] ← value[i] / weight[i] // Urgency per kg of weight

// Step 2: Sort items by ratio in descending order

Sort Items by ratio[i] in descending order

// Step 3: Initialize tracking variables

totalValue ← 0 // To store total urgency achieved

remainingCapacity ← C // Current available payload space

SelectedItems ← ∅ // Set to record chosen items

// Step 4: Select items based on greedy strategy

For each item i in sorted Items do

If weight[i] ≤ remainingCapacity then

// Item can fully fit in remaining capacity

totalValue ← totalValue + value[i]

remainingCapacity ← remainingCapacity - weight[i]

Add item i (fully) to SelectedItems

Else

// Item is too heavy, take fractional part

```
        fraction ← remainingCapacity / weight[i]
        totalValue ← totalValue + (value[i] * fraction)
        Add fraction of item i to SelectedItems
        remainingCapacity ← 0                    // Knapsack is full
        Break                                    // Stop selection

    // Step 5: Return results
    Return SelectedItems, totalValue

End
```

TIME COMPLEXITY

Algorithm Greedy Knapsack Disaster Relief(Items, Capacity):

Input:

Items = {(weight[i], value[i]) for i = 1 to n } // O(1)

Capacity = C // O(1)

Output:

SelectedItems, MaximumUrgencyValue // O(1)

Begin

For each item i in Items do // O(n)

ratio[i] ← value[i] / weight[i] // O(1)

Sort Items by ratio[i] in descending order // O(n log n)

totalValue ← 0 // O(1)

remainingCapacity ← C // O(1)

SelectedItems ← ∅ // O(1)

For each item i in sorted Items do // O(n)

If weight[i] ≤ remainingCapacity then // O(1)

totalValue ← totalValue + value[i] // O(1)

remainingCapacity ← remainingCapacity - weight[i] // O(1)

Add item i to SelectedItems // O(1)

Else

fraction ← remainingCapacity / weight[i] // O(1)

totalValue ← totalValue + (value[i] * fraction) // O(1)

Add fraction of item i to SelectedItems // O(1)

remainingCapacity ← 0 // O(1)

Break // O(1)

Return SelectedItems, totalValue // O(1)

End

Explanation:

1. **Input Reading**
   Reading inputs or assigning initial values is constant time → O(1).

2. **Ratio Calculation**

   Loop runs for **n items**.
   - Each ratio calculation is constant time → **O(1)**.
   - Total = **O(n × 1) = O(n)**.

3. **Sorting**
   - Sorting dominates the algorithm.
   - Any efficient sort (Merge Sort, Quick Sort, Timsort) requires **O(n log n)** in the average/worst case.

4. **Initialization**
   - Just simple assignments → O(1) each.

5. **Selection Loop**
   - The loop runs at most **n times** (one per item).
   - Inside loop: all operations are constant → **O(1)**.
   - Total = **O(n × 1) = O(n)**.

6. **Return Statement**
   - Just outputs results → O(1).

SPACE COMPLEXITY

Components of Space Complexity

The total space can be divided into the following parts:

1. Fixed Part (Constant Space):
    - Memory required for algorithm constants, variables, and instructions.
    - Does not depend on input size.
    - Example: variables like n (number of items), capacity, and loop counters.→ Space: **O(1)**
2. Input Storage:
    - Memory needed to store weights and values of n items.
    - Each item has:
        - Weight
        - Value
        - Ratio (value/weight, computed and stored)
    - Total per item: constant space.
    - For n items: **O(n)**
3. Auxiliary Data Structures:
    - Temporary storage during sorting (e.g., arrays, sorting algorithm stack).
    - Sorting requires O(n log n) extra space if merge sort is used, or O(1) if heap sort is used.
    - Typically: **O(n)**
4. Output Storage:
    - To store the selected set of items for relief distribution.
    - In the worst case, all n items are selected.
    - Space: **O(n)**

3. Total Space Complexity

Adding all components:

$S(n)=O(1)+O(n)+O(n)+O(n)=O(n)$

$S(n) = O(1) + O(n) + O(n) + O(n) = O(n)$

Therefore, the overall space complexity is linear: O(n).

Git-Hub Repository Link:

https://github.com/Vizzapuakshaya/2420090087_CASESTUDY.git

Design and Analysis of Algorithms (24CS2203)

Greedy Knapsack in Disaster Relief

S. Sai Sri Kruthi: 2420090003

Vizzapu Akshaya: 2420090087

Course Instructor

Dr. J Sirisha Devi

Professor

Department of Computer Science and Engineering

Case study - statement

- **Problem Statement**:

- During disasters such as earthquakes, floods, or hurricanes, relief teams must quickly transport vital supplies like food, water, and medicines.

- Transportation capacity is limited vehicles such as helicopters or trucks can only carry a certain weight.

- Selecting which items to send is critical, as the wrong choices can delay lifesaving aid.

- The goal is to maximize the total urgency value of items loaded within the available capacity.

Algorithm /Pseudo code

Steps:

Input:

List of items, each with a *weight* and *urgency value*.

Maximum payload capacity (e.g., helicopter's limit).

Compute Ratios:

For each item, calculate value-to-weight ratio = value ÷ weight.

Sort Items:

Arrange all items in descending order of ratio (highest first).

Select Items:

Add items one by one:

If the whole item fits → take it fully.

If it doesn't → take the fraction that fits and stop.

Output:

List of selected items (full or fractional).

Maximum total urgency value achieved.

Algorithm /Pseudo code

```
Algorithm Greedy_Knapsack_DisasterRelief(Items, C):
    For each item i:
        ratio[i] = value[i] / weight[i]
    Sort items by ratio descending
    totalValue = 0
    remainingCapacity = C
    For each item i in Items:
        if weight[i] ≤ remainingCapacity:
            take fully
        else:
            take fraction
            break
    return totalValue, SelectedItems
```

Time Complexity

- **Input Initialization:**
- Defining or reading the list of items → **O(1)** (constant number of items in case study)
- For dynamic input (n items), it would be **O(n)**

- **Ratio Calculation:**
- Each item's value-to-weight ratio is computed once → **O(n)**

- **Sorting Step:**
- Items are sorted in descending order of ratio → **O(n log n)**
- This is the **dominant term** in total complexity

- **Selection Loop:**
- Traverses each item once → **O(n)**
- Each iteration does constant-time operations (comparisons and arithmetic)

- **Final Output Display:**
- Printing selected items and totals → **O(n)**

Overall Time Complexity:

$$T(n) = O(n) + O(n \log n) + O(n) = O(n \log n)$$

Space Complexity

- **Fixed Part (Constant Space):**
- Variables like total value, remaining capacity, and loop counters.
- Occupy constant memory → **O(1)**

- **Input Storage:**
- Each item stores name, weight, value, and ratio → constant per item.
- For n items → **O(n)**

- **Sorting Overhead:**
- Sorting uses additional temporary space depending on the algorithm.
- Java's Arrays.sort (Timsort) requires **O(n)** auxiliary space.

- **Output Storage:**
- Storing and printing selected items → up to n entries → **O(n)**

Overall Space Complexity

S(n) = O(1) + O(n) + O(n) + O(n) = O(n)