# Optimized Bin Packing for Cargo Loading: A Computational Approach Using Permutation and Parallelization

Team 84

## Abstract

This report presents a comprehensive approach to solving a bin packing problem for cargo loading, specifically designed for Unit Load Devices (ULDs) used in logistics and aviation industries. The algorithm integrates advanced heuristics, such as height-based sorting and volume optimization, to prioritize and pack packages with minimal void spaces while adhering to constraints like weight and cost. The methodology employs permutation-based ULD configuration testing and leverages multiprocessing to expedite solution search, making it scalable for large datasets. The results demonstrate the effectiveness of the model in balancing packing efficiency, cost minimization, and adherence to physical constraints, providing a robust framework for real-world cargo management systems.

## Introduction

The bin-packing problem is a classical optimization challenge that involves fitting a set of objects of various sizes and weights into a finite number of bins or containers while minimizing cost. This problem has numerous real-world applications, including cargo loading, logistics, warehouse management, and resource allocation.

This report focuses on solving a 3D bin-packing problem, where both the dimensions and weight constraints of the bins (containers) must be considered. In addition, the objects to be packed are categorized into priority and economy packages, requiring the algorithm to prioritize high-value packages while efficiently utilizing the remaining space for lower-priority items.

## Problem Understanding

FedEx seeks a solution to optimize the packing of Unit Load Devices (ULDs) for air shipments. ULDs are standardized containers with specific dimensions and weight limits, used to transport packages. The goal is to assign packages to ULDs while ensuring the following:

### Primary Requirements

- The total weight of all packages in a ULD does not exceed its weight limit.
- Priority packages (higher-value shipments) must be shipped without fail.
- All assigned packages must fit within the dimensions of their designated ULDs.
- Packages should not overlap spatially inside the ULDs.

### Cost Optimization

- Minimize the total cost, which includes:
  - Cost of leaving economy packages behind (if necessary due to space or weight constraints).
  - Cost of distributing priority packages across multiple ULDs, where each ULD carrying a priority package incurs an additional cost ($K$ per ULD).

### Constraints and Assumptions

- All packages and ULDs are cuboidal in shape.
- Packages can only be aligned along the axes of the ULD (no inclined orientations).
- Priority packages should be grouped into as few ULDs as possible for faster delivery.

### Inputs

- List of ULDs with dimensions and weight limits.
- List of packages with dimensions, weights, types, and delay costs.
- Cost parameter ($K$) for each ULD carrying a priority package.

### Expected Outputs

- Assignment of each package to a ULD or `NONE` if it cannot be loaded.
- Coordinates and orientation of the packages inside the ULD.
- Total cost incurred by the solution.

## Tried Approaches

### Linear Programming

*Sets and Indices*

$$\begin{aligned} P &: \text{Set of all packages,} && \text{indexed by } p. \\ U &: \text{Set of all ULDs,} && \text{indexed by } u. \\ O_p &: \text{Set of orientations for package } p, && \text{indexed by } o. \end{aligned}$$

*Parameters*

$$\begin{aligned} \text{weight}_p &: \text{Weight of package } p. \\ \text{length}_{p,o}, \text{width}_{p,o}, \text{height}_{p,o} &: \text{Dimensions of package.} \\ \text{length}_u, \text{width}_u, \text{height}_u &: \text{Dimensions of ULD } u. \\ \text{weight\_limit}_u &: \text{Weight limit of ULD } u. \\ \text{cost}_p &: \text{Cost of delay for package } p. \\ K &: \text{ULD carrying Priority packages.} \\ M &: \text{A sufficiently large constant} \end{aligned}$$

*Decision Variables*

**Assignment Variables:**

$$\begin{aligned} y_{puo} &\in \{0,1\} : 1 \text{ if package is assigned to } u \text{ in orientation } o; \\ z_p &\in \{0,1\} : 1 \text{ if package } p \text{ is not assigned to any ULD;} \\ s_u &\in \{0,1\} : 1 \text{ if ULD } u \text{ carries any Priority packages.} \end{aligned}$$

**Position Variables:**

$$X_{puo}, Y_{puo}, Z_{puo} \geq 0 : \text{Reference corner coordinates .}$$

**Non-overlapping Variables:**

$$\delta_{pquo_x}, \delta_{pquo_y}, \delta_{pquo_z} \in \{0,1\} : \text{Non overlapping along axis.}$$

*Constraints*

**Assignment Constraints:**

$$\sum_{u \in U} \sum_{o \in O_p} y_{puo} + z_p = 1, \quad \forall p \in P. \tag{1}$$

$$z_p = 0, \quad \text{if } p \text{ is a Priority package.} \tag{2}$$

**ULD Weight Limits:**

$$\sum_{p \in P} \sum_{o \in O_p} \text{weight}_p \cdot y_{puo} \leq \text{weight\_limit}_u, \quad \forall u \in U. \tag{3}$$

**Package Fit within ULD Dimensions:**

$$X_{puo} + \text{length}_{p,o} \leq \text{length}_u + M \cdot (1 - y_{puo}), \tag{4}$$

$$Y_{puo} + \text{width}_{p,o} \leq \text{width}_u + M \cdot (1 - y_{puo}), \tag{5}$$

$$Z_{puo} + \text{height}_{p,o} \leq \text{height}_u + M \cdot (1 - y_{puo}), \quad \forall p, u, o. \tag{6}$$

**Non-overlapping Constraints:**

$$\delta_{pquo_x} + \delta_{pquo_y} + \delta_{pquo_z} \geq y_{puo} + y_{quo} - 1, \forall p, q \text{ with } p < q. \tag{7}$$

**ULD Carrying Priority Packages:**

$$s_u \geq y_{puo}, \quad \forall p \in \text{Priority Packages}, \forall o \in O_p. \tag{8}$$

**Position Variables Bounds:**

$$X_{puo}, Y_{puo}, Z_{puo} \geq 0. \tag{9}$$

*Objective Function*

Minimize the total cost:

$$\text{Minimize} \sum_{p \in P} z_p \cdot \text{cost}_p + K \cdot \sum_{u \in U} s_u. \tag{10}$$

*Variable Overload*

The linear programming model failed when scaled to 20,000,000 variables for the given problem. The primary reason for this failure lies in the excessive number of variables created due to the numerous possible orientations for each package. The combination of package dimensions and orientations resulted in a large number of variables, which significantly increased the computational complexity. As a result, the program took an impractical amount of time to solve and failed to reach an optimal solution within a reasonable time frame. The model's inefficiency stems from the challenge of handling the high-dimensional space created by the various orientations and the complexity involved in determining the optimal packing arrangement.

### Reinforcement Learning

Bin packing problems can be handled through supervise learning perspectives. Building on existing deep reinforcement learning algorithms and self-attention based encoding, satisfactory scores could be achieved.

**Action Space**

$$\pi^S : \text{Sequence Policy to select which package to pick next} \tag{11}$$

$$\pi^U : \text{ULD Policy to chose which ULD to send the package to} \tag{12}$$

$$\pi^P : \text{Placement Policy, pick orientation and location for package} \tag{13}$$

**Decomposing Observation Space** The observation space for each ULD can be decomposed into a grid of dimension L*B with each cell representing maximum height at that point. Using this observation space along with the action space policies, we train a deep learning model with penalties for exceeding weight limits, volume constraints or leaving priority packages.

*Limitation faced*

While models trained on single bins using Sequence and Placement policy perform reasonably well, introducing multiple ULDs with ULD Policy $\pi^S$ and the constraint of not missing any priority package produce results with low levels of accuracy.

## Greedy Algorithm

Being a NP-complete problem, relying on standard brute force or even dynamic programming to optimize the solution won't be efficient or practical for large-scale instances. These methods would take an exponential amount of time as the problem size grows, making them infeasible for real-world applications. While Dynamic Programming might provide an exact solution, its computational complexity limits its scalability, and brute force methods would only work for trivial cases, failing to deliver optimal results within a reasonable timeframe as the number of packages and ULDs increases.

Additionally, attempting to solve the problem using Linear Programming has proven to be ineffective, unless certain constraints are relaxed. Relaxing constraints, however, often requires making unrealistic assumptions about the problem. This compromises the accuracy of the solution, as the assumptions may not hold true in real-life scenarios. For example, assuming that package weights follow a normal distribution might not align with actual package weight distributions, leading to suboptimal results.

Instead of resorting to such assumptions, a more logical and heuristic approach could be employed to tackle the problem. By focusing on the physical properties and inherent constraints of the ULDs and packages, we can develop a more practical solution. This involves considering how packages can be logically placed inside ULDs, leveraging sorting, spatial packing algorithms, and greedy techniques that prioritize optimal space utilization without making oversimplified assumptions. This approach ensures that the solution is not only more realistic but also more computationally feasible, allowing for better scalability and effectiveness in addressing the constraints.

Such a heuristic approach, while not guaranteeing an exact optimal solution, can provide near-optimal results in a reasonable amount of time, making it a far more practical choice for large, complex problems where traditional methods are insufficient.The problem is to be applied on real world cases thus we can also expect some normality in the values.

## Greedy Approach

The pseudocode outlines a packing optimization process for loading packages into ULDs (Unit Load Devices) in a way that minimizes cost while adhering to various constraints. The process is divided into several functions, each with specific tasks that contribute to finding the best packing configuration.

- **ReadInput(file path)**:
  - This function is responsible for loading and parsing the necessary input data from a specified file path.
  - It processes ULD container specifications, such as dimensions and weight limits.
  - It also parses the list of packages, distinguishing between priority and economy packages.
  - Additionally, it computes rankings for the economy packages based on their cost-to-volume ratio, which helps in deciding how to prioritize and pack the economy packages.
- **GreedyPacking(l, r)**:
  - This function implements a greedy packing approach to find the most cost-effective packing configuration.
  - It takes two parameters, `l` and `r`, representing the range of packages to be considered for packing.
  - It initializes the minimum cost as infinity to track the best cost found during the packing process.
  - The function iterates through different subsets of economy packages within the range `l` to `r`.
  - For each subset, it applies the `SimilarHeightSorting` method to group packages with similar heights, optimizing the use of vertical space in the ULD.
- **Permutations of ULDs**:
  - For each subset of economy packages, the algorithm explores all permutations of ULD arrangements to find the most efficient packing configuration.
  - It initializes loaders (devices used to load packages into ULDs) for each permutation.
  - The function attempts to pack the packages using different orientations and records the optimal packing result.
  - Weight constraints are checked for each ULD permutation to ensure that no ULD exceeds its weight limit.
  - If a packing configuration results in a lower cost than the current minimum cost, the configuration is updated.
- **Optimization and Task Distribution**:
  - Once the greedy packing process is completed for an initial range of packages, the `Main` function distributes tasks across multiple threads to speed up the optimization process.

**Algorithm 1** Greedy Packing Optimization for ULD Containers

```
Require: File path with input data, processing constraints
Ensure: Optimal assignment of packages to ULDs minimizing cost
 1: function READINPUT(file_path)
 2:     Parse ULD container specifications
 3:     Parse Priority and Economy packages
 4:     Compute economy package rankings by cost-to-volume ratio
 5: end function
 6: function GREEDYPACKING($l$, $r$)
 7:     Initialize minimum cost $min\_cost \leftarrow \infty$
 8:     $best\_loaders \leftarrow \emptyset$
 9:     for each check in $l$ to $r$ do
10:         Subset economy packages up to $check$
11:         Sort packages using SIMILARHEIGHTSORTING
12:         for each permutation of ULDs do
13:             Initialize loaders for each ULD permutation
14:             for each ULD in the permutation do
15:                 Attempt packing using different orientations
16:                 Record optimal packing result
17:             end for
18:             Perform weight constraint validation
19:             Push packages to adjust coordinates
20:             Compute total cost for configuration
21:             if current cost $<$ $min\_cost$ then
22:                 Update $min\_cost$, $best\_loaders$
23:             end if
24:         end for
25:     end for
26:     return $min\_cost$, $best\_loaders$
27: end function
28: function MAIN
29:     Load input data using READINPUT
30:     Generate unique ULD permutations
31:     Execute GREEDYPACKING for initial range
32:     Distribute tasks across threads for optimization
33:     Merge results from all threads
34:     Pushing the packages packed after fixing orientation of ULDs.
35:     Write final output to file
36: end function
```

  - Each thread explores different subsets of the package list or different ULD configurations.
  - After completing parallel execution, the results from all threads are merged to determine the best packing configuration.
  - Adjustments to package coordinates may be necessary to ensure optimal placement in the ULDs.

- **Final Packing Adjustments**:
  - In the final step, the packed packages are adjusted to ensure that their orientation and positioning within the ULDs are correct.
  - The function ensures that the final configuration satisfies all constraints, such as weight and size limits.
  - The result is an optimal packing configuration that minimizes total cost while maximizing space utilization and adhering to all constraints.

## Heuristics

The following heuristics are used for optimizing the packing process:

1. **Minimum Height Sorting Heuristic**
   The minimum height sorting heuristic aims to identify a height dimension that results in the maximum number of packages having similar heights. The fundamental idea behind this approach is that when packages share similar height characteristics, they can be stacked or arranged more efficiently, making better use of available vertical space. By grouping packages with comparable heights, the packing process becomes more uniform, allowing for higher density packing. This heuristic involves analyzing the height of each package and determining which height value accommodates the greatest number of packages, reducing wasted space and ensuring that the vertical stacking within the container is optimized.
2. **Sorting Packages in Increasing Order of Height**
   Once the optimal height for sorting is selected using the minimum height sorting heuristic, the next step is to sort the packages in increasing order of height. This sorting strategy ensures that smaller packages occupy the available space first, with larger packages filling the remaining gaps. By starting with smaller items, we can make better use of the vertical space at the bottom of the container, allowing larger packages to rest on top without creating inefficient empty spaces. This approach helps to minimize priority splits, which could arise when special or large packages need to be packed first. It leads to a more efficient arrangement where the stacking sequence is optimized, maximizing the space utilization within the ULD.
3. **Cost and Volume Optimization**
   Cost and volume optimization focuses on strategically choosing packages based on their cost and the volume they occupy in the container. The process is divided into two main tasks: cost optimization and volume optimization. First, the most valuable packages are selected based on their cost, ensuring that the most expensive or high-priority items are packed first. Then, the remaining space is filled based on the volume optimization criteria, where packages are selected for their ability to fit efficiently into the remaining available space. This dual approach ensures that we prioritize both the cost of the items and the overall volume efficiency of the packing process, resulting in an optimal packing configuration that reduces costs while maximizing the use of space.

   a. **Cost Optimization**
      In cost optimization, the goal is to prioritize the most valuable packages, ensuring that they are packed first to maximize the overall value of the load. To achieve this, packages are sorted in decreasing order of cost and increasing order of volume. This ensures that high-value items are given precedence, while their size or volume is considered to minimize wasted space. By focusing on packing the highest-value items first, we can reduce the overall cost of the packing process and ensure that the most important or profitable items are securely placed within the container. This method helps optimize the value-to-space ratio and ensures that the packing configuration is as cost-effective as possible.

   b. **Volume Optimization**
      Volume optimization comes into play after the cost optimization step, focusing on efficiently filling the remaining space with the leftover packages. The packages that were not prioritized for cost reasons are sorted based on their height and volume. These packages are then arranged to maximize space utilization, filling in gaps that might otherwise be wasted. The goal is to ensure that the remaining volume within the container is used as effectively as possible, preventing any underutilized space. By carefully considering both the height and the volume of these remaining packages, we can pack them in such a way that maximizes the overall efficiency of the container, reducing empty gaps and making full use of the available storage capacity.

4. **ULD Permutations**
   ULD permutations involve evaluating different potential packing configurations by testing various permutations of how the packages are arranged within the ULD. This step aims to further reduce priority splits and ensure that the maximum number of packages can be packed efficiently. By exploring different permutations, the packing algorithm identifies the arrangement that best utilizes the available space while meeting the packing constraints. This process helps optimize space usage, minimizes the occurrence of priority splits, and ensures that the most efficient packing solution is chosen. ULD permutations are especially valuable for complex packing scenarios where various factors, such as package size and priority, need to be balanced for optimal space and cost utilization.

## Visualisation of the Output

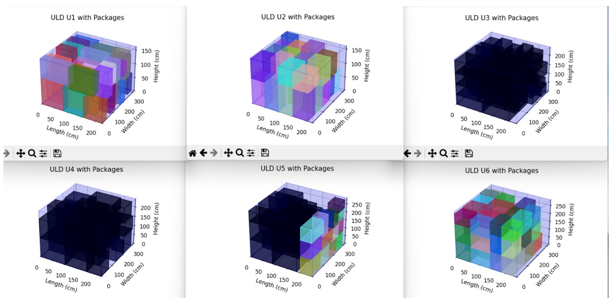

**Fig. 1.** Packing Visual of diffrent ULDs

**The above figure represents a Three dimensional view of the packages being fit inside the ULDs. The black packages are the priority packages and the others are economy packages.**

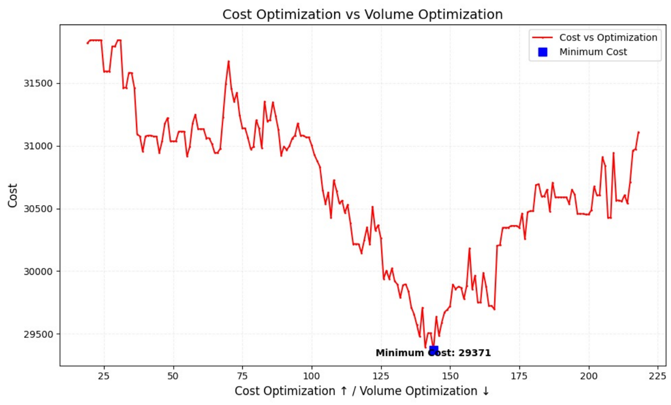

**Fig. 2.** Graph to visualise cost VS Volume optimisation

**The above figure represents a volume and cost optimisation curve. It is used to see the min cost (best optimised cost as per the algorithm) varying with the volume. The point demonstrating the minimum cost has been marked.**

## Time Complexity Analysis

The time complexity of the algorithm is influenced by several components, including the number of packages, ULDs, and permutations explored. We will break down the time complexity for each function in the algorithm:

### 1. ReadInput Function

The input reading function involves parsing ULD specifications and package data. Let $n$ be the number of packages and $m$ be the number of ULDs. The time complexity for this function is $O(n+m)$, where both the number of packages and ULDs contribute linearly to the time complexity.

### 2. GreedyPacking Function

The main complexity arises in the 'GreedyPacking' function. We analyze it step by step:

- **Sorting Packages**: The packages are sorted based on their height using 'SimilarHeightSorting'. Sorting takes $O(n \log n)$, where $n$ is the number of economy packages.
- **Permuting ULDs**: The algorithm explores permutations of $m$ ULDs. The number of permutations of $m$ ULDs is $O(m!)$, which is factorial in nature.
- **Orientations and Packing Configurations**: For each permutation of ULDs, different orientations (let's say $k$ orientations) are considered. This leads to an additional factor of $O(k \cdot m)$ for the number of possible packing configurations considered per permutation.
- **Iteration Over Package Subsets**: The algorithm iterates over subsets of economy packages within the range from $l$ to $r$. The number of subsets explored is $O(r-l)$. For each subset, the above operations are performed.
- **Packing with rectpack**: The worst case for using rectpack.pack() function on loader is $N^2$, where is N is the number of packages tried, which is approximately equal to $(area_U LD/area_P KG)^2$ which is roughly equal to 1000, in our case. Let us call this to be '$\alpha` for analysis purpose.$

Therefore, the time complexity of the 'GreedyPacking' function is dominated by:

$$O\left((r-l)\cdot(n\log n + m!\cdot k\cdot m\cdot\alpha)\right)$$

where:

- $r-l$ is the number of package subsets considered,
- $n \log n$ is the complexity for sorting the packages,
- $m!$ accounts for the number of permutations of $m$ ULDs,
- $k \cdot m$ accounts for the number of possible orientations per ULD (multiplied by $m$ because there are $m$ ULDs).

### 3. Main Function

The 'Main' function coordinates the overall process and distributes tasks across multiple threads. Assuming there are $t$ threads, the work is distributed evenly among them, which reduces the complexity of the 'GreedyPacking' function. The overall time complexity of the 'Main' function becomes:

$$O\left(\frac{(n\log n + m!\cdot k\cdot m\cdot\alpha)\cdot(r-l)}{t}\right)$$

However, since the time complexity of 'GreedyPacking' is the dominant factor, the overall time complexity of the entire algorithm remains primarily influenced by the 'GreedyPacking' function.

### Overall Time Complexity

Considering all the factors above, the total time complexity of the algorithm is:

$$O\left(\frac{(r-l)\cdot(n\log n + m!\cdot k\cdot m\cdot\alpha)}{t}\right)$$

where:

- $n$ is the number of packages,
- $m$ is the number of ULDs,

- $k$ is the number of orientations considered per ULD,
- $r - l$ is the number of subsets of economy packages considered,
- $t$ is the number of threads used for parallel processing.

.

## Example Computation

### Input Data

The following input data is provided:

- **ULDs (Unit Load Devices):**
  a. ULD-1: $100\,\text{cm} \times 80\,\text{cm} \times 80\,\text{cm}$, Weight Limit: 250 kg
  b. ULD-2: $100\,\text{cm} \times 80\,\text{cm} \times 80\,\text{cm}$, Weight Limit: 250 kg
- **Packages:**
  a. P-1: $70\,\text{cm} \times 40\,\text{cm} \times 50\,\text{cm}$, Wt: 100 kg, Priority
  b. P-2: $70\,\text{cm} \times 40\,\text{cm} \times 50\,\text{cm}$, Wt: 100 kg, Priority
  c. P-3: $70\,\text{cm} \times 40\,\text{cm} \times 50\,\text{cm}$, Wt: 150 kg, Economy, Cost of Delay: 20
  d. P-4: $70\,\text{cm} \times 40\,\text{cm} \times 50\,\text{cm}$, Wt: 150 kg, Economy, Cost of Delay: 30
- Cost per ULD carrying a Priority package: $K = 40$

### Step 1: Packing Results

- **Packed in ULD-1:**
  a. P-1 at coordinates $(0, 0, 0)$ to $(40, 50, 70)$
  b. P-2 at coordinates $(40, 0, 0)$ to $(80, 50, 70)$
- **Packed in ULD-2:**
  a. P-4 at coordinates $(0, 0, 0)$ to $(40, 50, 70)$
- **Unpacked:**
  a. P-3: Not packed due to weight or space constraints.

### Step 2: Void Space Calculations

- **For ULD-1:**

$$\text{Total Volume: } 100 \times 80 \times 80 = 640,000\,\text{cm}^3$$

$$\text{Packed Volume: } 2 \times (70 \times 40 \times 50) = 280,000\,\text{cm}^3$$

$$\text{Void Space: } 640,000 - 280,000 = 360,000\,\text{cm}^3$$

- **For ULD-2:**

$$\text{Total Volume: } 100 \times 80 \times 80 = 640,000\,\text{cm}^3$$

$$\text{Packed Volume: } 70 \times 40 \times 50 = 140,000\,\text{cm}^3$$

$$\text{Void Space: } 640,000 - 140,000 = 500,000\,\text{cm}^3$$

### Step 3: Cost Optimization

- Priority ULD cost:

$$1 \times K = 1 \times 40 = 40$$

- Economy package delays (P-3 unpacked):

$$\text{Cost of delay: } 20$$

- Total Cost:

$$\text{Cost} = 40 + 20 = 60$$

### Step 4: Final Results

- **Packed Volume:**

$$280,000\,\text{cm}^3 + 140,000\,\text{cm}^3 = 420,000\,\text{cm}^3$$

- **Void Space:**

$$360,000\,\text{cm}^3 + 500,000\,\text{cm}^3 = 860,000\,\text{cm}^3$$

- **Unpacked Volume:**

$$70 \times 40 \times 50 = 140,000\,\text{cm}^3$$

- **Total Cost:**

$$\text{Cost} = 60$$

### Visualization

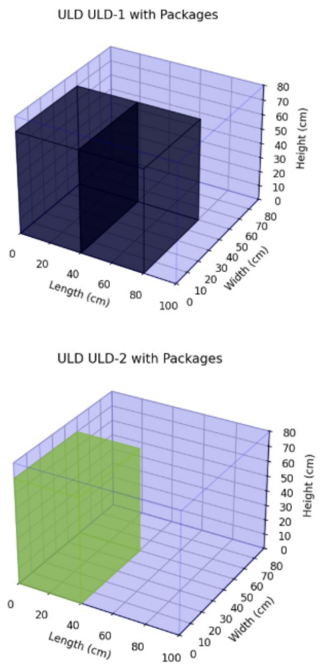

**Fig. 3.** Visualization of packed ULDs showing the placement of P-1, P-2, and P-4.

## How the Script Processes the Input

The script executes the following key steps:

### *Step 1: Parse the Input*

The function `read_input(file_path)` reads the input file and extracts:

- The constant $K$
- A list of ULD containers, each with its dimensions and weight limit
- Lists of Priority and Economy packages with their attributes

For this example:

- Two ULDs: `ULD-1`, `ULD-2`
- Two Priority packages: `P-1`, `P-2`
- Two Economy packages: `P-3`, `P-4`

*Step 2: Initialize the Loader*

A `PackageLoader` object is created for managing the ULDs and packages:

- Priority packages are sorted by height and volume (smallest first).
- Economy packages are sorted similarly but are processed after Priority packages.

*Step 3: Load Priority Packages*

The method `load_priority_packages()` is invoked: The script starts with `ULD-1`:

- It adds `P-1` (100 kg).
- Remaining capacity in `ULD-1`: $250 - 100 = 150$ kg.
- Next it adds `P-2` (100 kg).
- Remaining capacity in `ULD-1`: $150 - 100 = 50$ kg.
- Next it try to add remaining economy packages in `ULD-1` but it can't be added due to dimensional constraints. `P-2` (100 kg).

*Step 4: Load Economy Packages*

The method `load_economy_packages()` attempts to fit the Economy packages:

- `P-4` is added to `ULD-1`, but fails to do so because of dimensional constraints.
- `P-4` is added to `ULD-2`, because of higher `cost to volume` ratio:
  - Weight of `P-4`: 150 kg.
  - Remaining capacity in `ULD-2`: $150 - 150 = 0$ kg.

*Step 5: Compute Costs*

The cost is computed based on:

- **Priority Package Costs:** Only one ULD contain Priority packages, so the cost from Priority Package Packing is:

$$\text{Cost} = K \times (\text{No. of ULDs with Priority Packages}) = 40 \times 1 = 40$$

- **Economy Package Costs:** One Economy package is delayed, so an additional cost of 20.

## Output

- **Total Cost:** $40 + 20 = 60$
- **Allocation:**
  - `ULD-1: P-1, P-2`
  - `ULD-2: P-4`
- **Coordinates:**
  - `P-1: ULD-1,0,0,0,40,50,70`
  - `P-2: ULD-1,40,0,0,80,50,70`
  - `P-3: NONE,-1,-1,-1,-1,-1,-1`
  - `P-4: ULD-2,0,0,0,40,50,70`

## Summary of Execution

The script efficiently loads Priority packages first, ensures all packages fit within the constraints, and computes the minimal cost based on the ULDs carrying Priority packages.

## Problems Faced

### Code Implementation Challenges

During the development of the code, several issues were encountered:

- **Array Size Limitations:** The packing algorithm required handling a 3D representation of ULD containers. To efficiently manage these, the `numpy` library was used to overcome native Python limitations in handling large arrays and optimize operations.
- **Threading and Performance Bottlenecks:** To handle large input datasets efficiently, the use of multiprocessing was implemented. However, ensuring thread safety and balancing workload across threads proved challenging.
- **Sorting and Filtering Issues:** The use of multiple sorting mechanisms (e.g., for height and volume optimization) required careful debugging to ensure accuracy and efficiency.

### Optimization and Fixes

Improving the algorithm's performance required implementing several optimizations:

- **Weight Constraints:** Adjusting packages to ensure compliance with ULD weight limits required redesigning the logic and re-packing packages dynamically.
- **Package Orientation Adjustments:** Handling the different orientations of ULDs while maintaining correct volume and weight constraints introduced additional complexity.
- **Eliminating Redundant Permutations:** To enhance efficiency, similar permutations of ULD configurations were detected and removed, which significantly reduced computational overhead.

## Limitations

The greedy algorithm described in the pseudocode has a few limitations when applied to the problem:

a. **Suboptimal Global Solution:**
The algorithm focuses on a greedy approach by iteratively selecting subsets of economy packages and exploring ULD permutations. While this can yield locally optimized results for each iteration, it does not guarantee a globally optimal solution. There may exist better overall packing configurations that are overlooked due to the local optimization approach.

b. **Factorial Growth in ULD Permutations:**
The algorithm explores all permutations of ULDs for each subset of packages. For $m$ ULDs, this results in $m!$ permutations, which grows factorially as the number of ULDs increases. This can make the algorithm computationally infeasible for large numbers of ULDs.

c. **Sequential Package Selection:**
The economy packages are selected in subsets ranging from $l$ to $r$. This rigid approach may lead to inefficient utilization of the ULD space, as some subsets may not provide the most compact packing configurations.

d. **Scalability Challenges:**
With increasing numbers of packages and ULDs, the algorithm becomes computationally expensive. The

complexity of sorting packages, iterating over subsets, and testing permutations grows rapidly, making the algorithm less practical for larger datasets.

e. **Weight and Dimension Validation Overhead:** The algorithm performs weight and dimension validations after attempting to pack the packages. This reactive approach can lead to wasted computational effort if a configuration fails the validation checks, as there is no proactive mechanism to eliminate invalid configurations early in the process.

f. **Dependency on Parameter Tuning:** The range of subsets ($l$ to $r$) and other heuristic parameters heavily influence the algorithm's performance. Poorly chosen parameters can result in suboptimal packing solutions or excessive computational time.

g. **Stability of Packages in Stacking:** The algorithm does not account for how packages are stacked on top of each other within a ULD. This could result in unstable configurations where packages might topple during transit, especially if heavier or unevenly distributed packages are placed above lighter ones. Ensuring the stability of stacks is critical for real-world applications but is overlooked in this approach.

h. **Risk of Damage to Fragile Packages:** The algorithm does not consider the relative weights of packages when stacking. As a result, heavier packages might be placed on top of lighter or fragile packages, potentially causing them to break or sustain damage during transit. This oversight could lead to significant losses in scenarios involving fragile items.

## Conclusion

The greedy algorithm discussed above provides a practical and computationally feasible approach to addressing the ULD optimization problem. By leveraging heuristics such as height-based sorting, volume optimization, and permutation testing, the algorithm is able to generate solutions that are often close to optimal for small to medium-sized datasets. The incorporation of multiprocessing further enhances its scalability, making it suitable for real-world applications where quick and effective solutions are necessary. However, due to its inherently local optimization nature, the algorithm does not guarantee a globally optimal solution.

The limitations highlighted—such as the factorial growth in ULD permutations, inadequate handling of stacking stability, and insufficient consideration of weight distribution—illustrate areas where the algorithm could be improved. These challenges stem from the greedy algorithm's focus on locally optimizing specific configurations rather than evaluating the global solution space comprehensively.

To achieve a truly optimal solution, the problem can be formulated as a Linear Programming (LP) or Mixed-Integer Linear Programming (MILP) model. Such models can explicitly define constraints like weight limits, package dimensions, and priority grouping, and optimize the objective function (e.g., minimizing total cost) across the entire solution space. An LP-based approach provides several advantages:

- **Global Optimization:** LP guarantees a globally optimal solution, ensuring that all constraints are met while minimizing the cost function.
- **Flexibility:** Additional constraints, such as stacking stability and fragility, can be easily incorporated into the LP formulation.
- **Precision:** LP ensures that the best possible configuration is found for the given inputs, eliminating inefficiencies caused by heuristic approximations.

Despite its advantages, LP has computational limitations, especially for large datasets involving hundreds of packages and multiple ULDs. The problem's complexity grows exponentially as the number of variables and constraints increases, which may require significant computational resources. To balance efficiency and accuracy, a hybrid approach can be employed: the greedy algorithm can generate a good initial solution, which can then be refined using LP techniques to achieve near-optimal results.

In conclusion, while the greedy algorithm offers a practical solution for the ULD packing problem, the use of Linear Programming can address its limitations and guarantee optimality. Combining these approaches could provide a robust framework for solving both small-scale and large-scale instances of this challenging optimization problem.