Project 2 Report

I pledge that this test/assignment has been completed in compliance with the Graduate Honor Code and that I have neither given nor received any unauthorized aid on this test/assignment.

Name: B Ankit Signed: B Ankit

1. Optimization Problem and Description

In today's e-commerce-driven world, warehouse packing efficiency is crucial for minimizing shipping costs and ensuring timely delivery of goods. The problem focuses on optimizing the placement of different-sized packages into a minimum number of containers, considering container volume limits.

1.1. Input Parameters

- Let $\boldsymbol{P} = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}, \dots, \boldsymbol{p_n}\}$ be the set of all packages, where each package $\boldsymbol{p_i}$ is defined by its dimensions $(\boldsymbol{l_i}, \boldsymbol{w_i}, \boldsymbol{h_i})$ and volume $\boldsymbol{v_i} = \boldsymbol{l_i} * \boldsymbol{w_i} * \boldsymbol{h_i}$, for $\boldsymbol{i} = \mathbf{1,2,\dots,} \boldsymbol{n}$.
- Let $\boldsymbol{C} = \{\boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{c_3}, \dots, \boldsymbol{c_m}\}$ be the set of containers available for packing, where each container $\boldsymbol{c_j}$ has a volume capacity $\boldsymbol{V_{cj}}$, for $\boldsymbol{j} = \mathbf{1,2,\dots,} \boldsymbol{m}$.

1.2. Objective Function

The goal is to minimize the number of containers used m while maximizing the volume utilization of each container. This can be broken down into two distinct objective functions:

1.2.1. Minimization of Containers Used

- Minimize $\boldsymbol{m}$, the number of containers used to pack all packages in $\boldsymbol{P}$.

1.2.2. Maximization of Volume Utilization

- Maximize the sum of the ratios of used volume to the total volume of each container, which can be formally expressed as: $\max \sum_{j=0}^{m} \frac{\sum_{i \in S_j} v_i}{V_{cj}}$

  where $\boldsymbol{S_j}$ is the set of packages allocated to container $\boldsymbol{j}$, and $\sum_{i \in S_j} \boldsymbol{v_i}$ represents the total volume of packages in container $\boldsymbol{j}$.

1.3. Expected Output

An array **C** where each element is a sub-array listing the volumes of packages allocated to each container.

1.3.1. Sample Input

Packages and their Volumes are represented as **P = [60, 40, 30, 20, 10].** This array represents the volumes of the packages to be allocated, where each element corresponds to the volume of a package $\boldsymbol{P_i}$ *for i = 0 to 4*

1.3.2. Sample Output

Container Allocations: **C = [[60, 20, 10], [40, 30]]**, where the 0th index here represents the allocation of packages $[\boldsymbol{P_0}, \boldsymbol{P_3}, \boldsymbol{P_4}]$, and the 1st index represents the allocation of packages $[\boldsymbol{P_1}, \boldsymbol{P_2}]$.

1.4. Challenge Complexity:

The Efficient Warehouse Packing problem is an NP-hard problem, with its focus on minimizing the number of containers while maximizing volume utilization(considering 3D dimensions of length, width, height), is a complex and challenging optimization scenario, especially for large-sized instances. As the problem size increases, the search space expands factorially, making the computation of an exact solution impractically time-consuming for large datasets. This factorial complexity underscores the significant challenge in achieving global optimality in real-world applications of warehouse packing.

2. Algorithm Implementation and Description

This section delineates the implementation details of two distinct greedy algorithms formulated to address the Efficient Warehouse Packing problem. Each algorithm offers a unique strategy to optimize the packing process by prioritizing different aspects of the packages and containers.

2.1 Greedy Largest Package First Algorithm(*LPF*)

```
function LPF_Packing(Packages P, Containers C)
    Sort P in descending order based on volume
    Initialize an empty array of Containers C
    for each package p in P
        container_found = False
        for each container c in C
            if p fits in c without exceeding its capacity
                Add p to c
                container_found = True
                break
        if not container_found
            Create a new container c_new with enough space for p
            Add p to c_new
            Append c_new to C
    return C
```

In the LPF algorithm, we commence with an empty container array *C* and a package array *P* sorted in descending order by volume. The goal is to minimize the number of containers while maximizing the utilization of their volume capacities.

For each iteration, we select the largest unallocated package p_i from *P*. We then identify the container c_j from *C* that has the optimal space to accommodate p_i. The optimal fit is determined by the container that, after the addition of p_i, will have the least remaining volume, thus maximizing the space utilization.

If p_i cannot fit in any existing c_j, we allocate a new container c_{new}, thereby incrementing our container count. This new container becomes the current optimal fit for p_i. The process iterates until all packages in *P* are allocated into containers in *C*.

This greedy strategy aims for a local optimal at each step by choosing the best immediate container allocation for the largest package, which indirectly contributes to the global goal of minimizing the number of containers used. Though this heuristic does not guarantee a globally optimal solution, it provides a practical approach to solving large-scale instances of the warehouse packing problem, where exact solutions are computationally infeasible due to the problem's NP-hard nature.

2.2 Best Fit Decreasing Volume Utilization Algorithm (*BFD*)

```
function BFD_Packing(Packages P, Containers C)
    Sort P in descending order based on volume
    Initialize an empty array of Containers C
    for each package p in P
        best_fit = null
        min_leftover_space = Infinity
        for each container c in C
            leftover_space = c.capacity - (c.used_space + p.volume)
            if leftover_space >= 0 && leftover_space < min_leftover_space
                best_fit = c
                min_leftover_space = leftover_space
        if best_fit is not null
            Add p to best_fit
        else
            Create a new container c_new with enough space for p
            Add p to c_new
            Append c_new to C
    return C
```

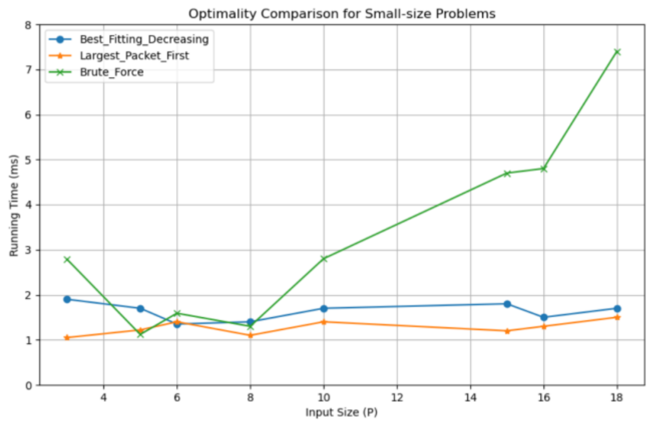
This Volume Utilization algorithm enhances the packing process by intricately selecting the placement of each package. We initiate by organizing the packages P in descending order by their volume. The core principle guiding this algorithm is to allocate a package not just to any container with sufficient space but to the one that, post-placement, will leave the least volume unused. This choice maximizes space utilization by ensuring that packages are placed in a way that leaves the smallest possible gaps, aiming for an optimal fit.

Each package p_i is considered in turn, scanning through the existing array of containers C. We calculate the volume remaining for each container c_j if p_i were to be placed inside. The package is allocated to the container that results in the minimum remaining volume that can still accommodate the package without surpassing the container's volume limit. Should no such container exist, a new container is provisioned to accommodate p_i, and the algorithm proceeds.

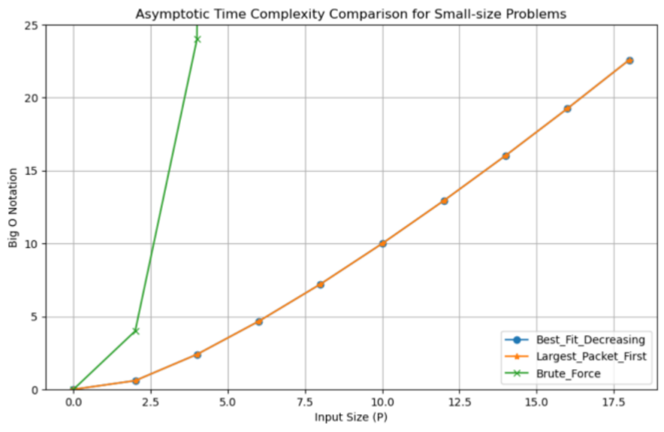
The BFD approach iteratively refines the packing configuration, leading to potentially greater volume efficiency compared to the LPF algorithm. While this may involve more computational steps per package due to the need to evaluate fits across all containers, the trade-off can lead to a reduction in the total number of containers used, which is valuable in scenarios where container space is at a premium.

3. Comparative Performance Analysis

3.1. Comparative performance analysis result for small-size problem



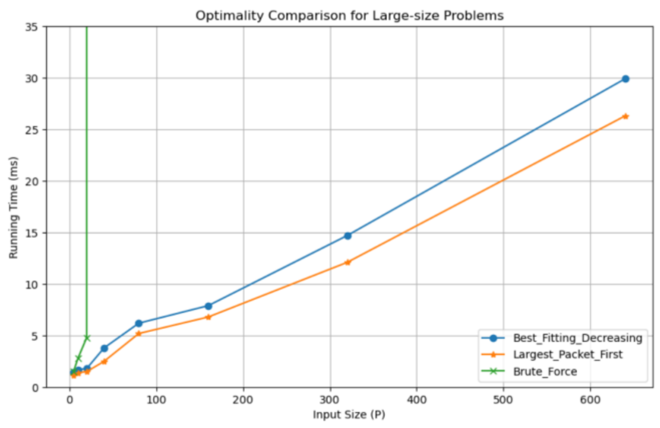
3.2. Asymptotic complexity analysis result for small-size problem



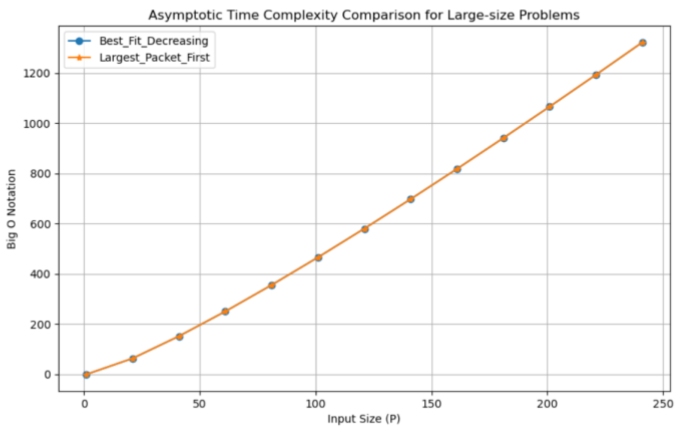
In the small-size problem Comparative analysis, from the first graph we can see that for very small values of input array, there are points where the brute force approach is better than both the LPF and BFD greedy algorithms but as the input size increases slightly, the running time(measure of optimality) increases rapidly whereas the LPF algorithm is slightly better in terms of running time than the BFD greedy algorithm in terms of both space utilisation as well running time. While the brute-force algorithm guarantees an optimal solution at some points, it does so at the cost of significantly higher computational resources, as shown by its prohibitive running time.

For the asymptotic complexity analysis, the Brute-Force is O(n!) which is way more higher than both the Greedy algorithms which have a similar Asymptotic Complexity of n(log(n)). So, we can see as we compare for input sizes of the array, the Brute force rises exponentially due to the factorial part of n, and the other two greedy algorithms overlap as they have same time complexity.

3.3. Comparative performance analysis result for small-size problem



3.4. Asymptotic complexity analysis result for large-size problem



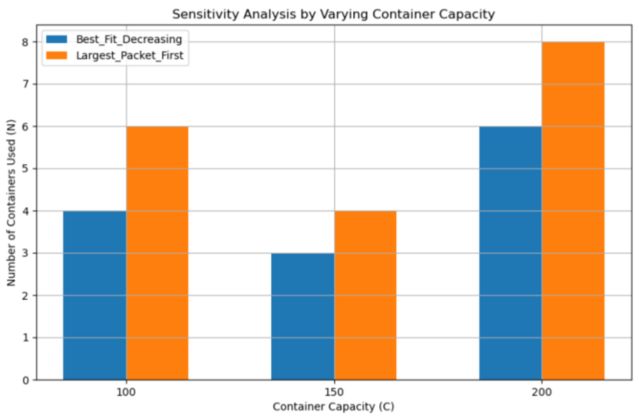
As we increase the input array size, the Brute force algorithm goes beyond the capacity of my laptop and the highest I could record was 22480 milliseconds for the input size array of 40 numbers. The other two Greedy Algorithm(LPF and BFD) were able to solve the problems with fairly low running times and were polynomial in in nature.

For optimality, both greedy algorithms exhibit what appears to be a linear relationship between the input size and the running time, consistent with an $O(n\log n)$ complexity, which is typical for algorithms where sorting is the most significant operation. The better algorithm here is the LPF greedy approach as we can see from the graph it takes lower time for both the large and smaller input arrays.

For the Asymptotic Complexity Comparison, both the Greedy algorithms like the smaller size problems have similar time complexity so both the graphs overlap at all the points. Thus, we can derive that the best optimal solution for the Effective Warehouse packaging problem is the LPF(Largest Package First) Greedy Approach that can be adopted as it is better in both comparisons for the smaller as well as the larger input sizes.

4. Sensitivity Analysis

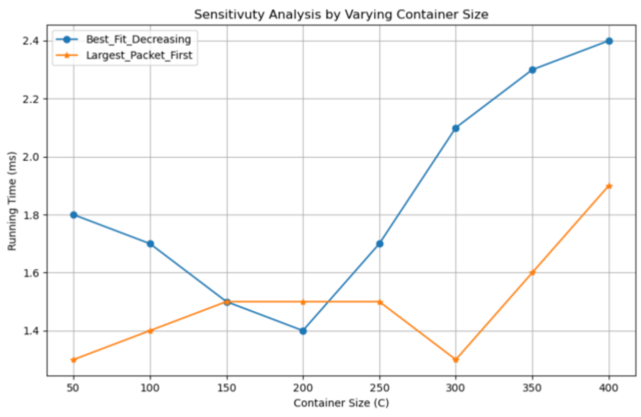
4.1. Varying the container size and tracking the number of containers used



In this analysis, we investigate how varying container capacities influence the efficiency of the Best Fit Decreasing and Largest Packet First algorithms. The bar graph illustrates the number of containers utilized by each algorithm at three different container capacities: 100, 150, and 200 units.
It is observed that as the container capacity increases from 100 to 200 units, both algorithms tend to use fewer containers. This trend is expected because larger containers can accommodate more packages, reducing the total number of containers required. Notably, the Best Fit Decreasing algorithm consistently outperforms the Largest Packet First approach across all tested capacities. This performance difference highlights BFD's superior space optimization, particularly at a capacity of 150 units, where it uses fewer containers than LPF.

The data suggests that container capacity is a significant factor in determining packing efficiency and that the choice of algorithm can be influenced by the expected range of container sizes. In practice, when container size can be controlled, BFD's ability to minimize wasted space could translate into cost savings by reducing the number of containers required for shipping or storage.

4.2. Varying the container capacities and tracking running time



The second graph presents a sensitivity analysis focused on the packing time of the Best Fit Decreasing and Largest Packet First algorithms against varying container sizes. The line graph elucidates a noteworthy pattern where running time escalates with the increase in container capacity.
The Largest Packet First algorithm exhibits a relatively steady increase in running time as container size grows, reflecting the added complexity of filling larger spaces.

In contrast, the Best Fit Decreasing algorithm shows more fluctuation, with a marked peak at a container size of 250 units. This peak may indicate a threshold where the BFD algorithm's strategy of finding the best fit becomes more time-consuming due to the increased number of potential placement choices in larger containers.
Interestingly, at a size of 300 units, the LPF's running time sharply decreases, suggesting a point of optimization where the container size aligns more harmoniously with package sizes, allowing for quicker decisions. This insight can be crucial for operations and space utilization, particularly in high-throughput logistics environments.

The analysis underscores the importance of understanding how the characteristics of containers interact with algorithmic efficiency. The findings support that while BFD may offer better space utilization, it does so with varying impacts on running time, which should be considered alongside space efficiency when selecting an algorithm for large-scale packing operations.