# **Truck Loading Problem**

## **Introduction**

The following is a report about comparing methods of solving the truck loading problem. This report compares the first fit and the next fit algorithmic approach to solving the truck loading problem. The two methods were implemented in the Java programming language. Below are sections illustrating the pseudo code of these implementations, details of the implementation itself, testing of the implementations and the results drawn from running the algorithms under various scenarios.

## **Pseudo Code**

### **First Fit**

Inputs: List of boxes, list of trucks.

Begin

For i = 0 to boxesi do:

For j = 0 to trucksj do:

If fits( boxesi , trucksj ) do:

Place( boxesi , trucksj )

goToNextBoxLoop

Fi

od

Add( trucks, newTruck )

Place( boxesi, newTruck )

Od

End

Above is a pseudo code implementation of the first fit algorithm, given a list of boxes and a list of trucks. The algorithm takes each box (0 to i) and loops through each truck (0 to j) checking if the box can fit in that truck. If the box can fit there, it is placed in the truck and the program continues to the next box in the loop. If the program loops through all the trucks and no place is found, a new truck is made and the box is placed there.

### **Next Fit**

Inputs: List of boxes, list of trucks

Begin

lastBoxPosition = 0

For i = 0 to boxesi do:

For j = lastBoxPosition to trucksj do:

If fits( boxesi , trucksj ) do:

Place( boxesi , trucksj )

lastBoxPosition = j

goToNextBoxLoop

fi

od

Add( trucks, newTruck )

Place( boxesi, newTruck )

Od

End

Above is a pseudo code implementation of the next fit algorithm. An integer variable will keep a track of the position in which the last box was placed, starting at 0 initially. A loop through each box occurs, which then checks if the box can fit in every truck from the truck the last box was placed in to the end of the truck list. This results in the algorithm never back tracking, as is a feature of the next fit algorithm. If the box fits in any of these trucks, the box is placed in that truck, the lastBoxPosition variable is updated and the program continues to the next box in the loop. If no truck with space for it is found, a new truck is made and the box is placed in there.

## **Implementation**

The two algorithms have been implemented in the Java programming language. The main code is based around a Truck and a Box class, to represent the trucks and the boxes which will be loaded into them. The box class is a simple class, storing the boxes height and width as well as providing get methods to access these fields for a given box. The truck class stores its basic information, such as truck height and width; providing get methods for these fields. It also stores a list of lists to represent the piles within the truck. For example, a truck with two piles will have two lists of boxes that are within the given piles, and these two lists will be stored in a list, representing the piles as a whole.

These are implemented using Java’s Array List to allow for quick indexing to elements within the pile. This is most effective when calculating the floor space taken up in a truck to see if there is room for a new pile, as it allows o(n) searching to find the bottom element of the pile. The current number of boxes is also tracked in the truck, so that if L is given (the maximum number of boxes per truck) then this can be adhered to. The truck also contains a method called displayContents(), which neatly prints out information about the truck and its piles, as well as information about the boxes within these piles.

The algorithms themselves are implemented as methods in a separate class called Algorithms. They give a Java implementation of the pseudo code described in the section above. They will alter the truck list given, adding new trucks when required. This makes it simple to see the number of trucks used, as it allows a simple size check of the truck list it was given. This class also contains a generateBoxes() method which, given a number, will return a list filled with the given amount of boxes. The boxes will have a randomly generated height and width, between 1 and the truck height / width. This allows for quick testing of various loads.

## **Testing**

The Java source code itself is tested using a JUnit class, containing various test methods. These methods test and prove that the code follows the constraints set out in the specification. That being that piles are stacked according to truck height, a box cannot top a pile if it is wider than the top box on the pile, most one box can be directly on top of another and that if a box per truck constraint is given, that it is adhered to.

Performance testing of the algorithms themselves is simple. The truck list given to each algorithm is modified, adding trucks when required, so checking the size of this list after running each algorithm tells us how many trucks were used. The algorithms are tested using Java’s System.currentTimeMillis() before and after the run. The time after the run minus the time before the run gives a total run time for the algorithm in milliseconds.

## **Results & Comments**

|  |  |  |
| --- | --- | --- |
| **100 Boxes** | | |
|  | **Trucks Used** | **Time Taken (milliseconds)** |
| **First Fit** | 32 | 7 |
| **Next Fit** | 51 | 1 |

|  |  |  |
| --- | --- | --- |
| **500 Boxes** | | |
|  | **Trucks Used** | **Time Taken (milliseconds)** |
| **First Fit** | 161 | 29 |
| **Next Fit** | 287 | 2 |

|  |  |  |
| --- | --- | --- |
| **1,000 Boxes** | | |
|  | **Trucks Used** | **Time Taken (milliseconds)** |
| **First Fit** | 301 | 46 |
| **Next Fit** | 565 | 2 |

|  |  |  |
| --- | --- | --- |
| **1,500 Boxes** | | |
|  | **Trucks Used** | **Time Taken (milliseconds)** |
| **First Fit** | 464 | 76 |
| **Next Fit** | 881 | 4 |

|  |  |  |
| --- | --- | --- |
| **2,000 Boxes** | | |
|  | **Trucks Used** | **Time Taken (milliseconds)** |
| **First Fit** | 592 | 103 |
| **Next Fit** | 1,146 | 4 |

Above are the results of running the two algorithms given the same number of boxes. Note, the boxes are generated with random sizes using the generateBoxes() function that has been implemented, so running tests again with the same number of boxes will, most likely, produce different results as different sizes boxes will be used, though the number of boxes will be consistent.

The above results highlight a few common trends. Firstly, the next fit algorithm is considerably quicker than its first fit counterpart. Across the above tests, next fit averages 2.6 milliseconds to pack the boxes into the trucks, whereas first fit takes a much longer 52.5 milliseconds on average to pack the boxes into the trucks.

The first fit algorithm is evidently dependant on the load size / the number of boxes given, due to its potentially exhaustive search per box, potentially resulting an exponential order. Next fit is able to maintain a consistent time as, at most, it will perform three comparisons per box: checking if it fits on the last box’s pile, if a new pile can be made in that truck and making a new truck and placing it there.

Though next fit gives a much quicker performance, it is evidently not the undisputed best of the two. Next fit uses considerably more trucks to pack the boxes. Across the above tests, next fit averages a use of 585 trucks, whereas first fit averages only 310.

Looking at these two main points it is clear that the two algorithms are situational dependant. The next fit algorithm will give a quick performance, but at the cost of a lot of wasted space, illustrated by how first fit is able to pack the boxes into far fewer trucks. First fit is evidently better at packing items into space more efficiently, but the time required can be up to exponential.

In conclusion, neither algorithm is perfect; both have their flaws. In a scenario where space is in abundance but time is short next fit should be used. However, if space is an issue, then the longer time of first fit may be acceptable in order to get achieve efficient space usage.