CSC2023 - Algorithms assignment 2:

Rope Cutting Problem

# Introduction

For this assignment I was asked to design and implement two algorithms that would each provide a solution to a variation of the Bin Packing Problem. The problem states that a list of orders for ropes is placed to a shop. The shop must then iterate through the orders and use either a first-fit or best-fit algorithm to select a suitable rope for each order and then cut that rope. If the shop cannot find a suitable rope it must import a rope from the factory into its stock.

# Pseudo Code Design

In order for the two algorithms to function correctly we need a method to check whether a large enough rope is available to be cut and a method to then cut that rope.

Initially checks are performed to see if the rope can be cut (it can only be cut if the length of the rope is greater than the length of rope required by the current order).

## Required functions

### canCut(index, cutLength)

Variables: index (index of the rope in the stock)  
 cutLength (The length of rope required by the current order)  
 stock (An array of ropes)  
 ropeLength (Length of rope at position ‘index’ in ‘stock’)

BEGIN:  
 if ropeLength at index >= cutLength  
 return true  
 else return false  
END

Now I need a function to cut the rope that I can call after it has been checked with canCut.

### cut(index, cutLength)

Variables: index (index of the rope in the stock)  
 cutLength (The length of rope required by the current order)  
 stock (An array of ropes)  
 ropeLength (Length of rope at position ‘index’ in ‘stock’)

BEGIN:  
 ropeLength := ropeLength – cutLength  
END

I will also need a function that will remove ropes that are small enough to be scrapped.

### shouldRemoveRope(index)

Variables: index (index of the rope in the stock)  
 stock (An array of ropes)  
 ropeLength (Length of rope at position ‘index’ in ‘stock’)  
 scrapLength (length at which ropes should be scrapped)

BEGIN:  
 if ropeLength <= scrapLength  
 remove rope from stock  
END

Now that I have functions to find out if the ropes can be cut and functions to cut the ropes, I can now implement the actual bin packing algorithms. I shall now implement the First-Fit algorithm.

First-Fit: Find the first suitable rope in stock and then cut it, if no rope is found then import a new rope from the factory to the stock and cut that rope.

To start I need a function that returns the index of the first suitable rope in the array.

## First-Fit

### findFirstFit(orderSize)

Variables: orderLength (The length of rope required by the order)  
 stock (An array of ropes)  
 stockSize (Number of ropes in stock (The size of ‘stock’))

BEGIN:   
 for i := 0 to stockSize - 1 do  
 if canCut(i, orderLength)  
 return i  
 else return -1  
END

Now I can implement the part of the algorithm that will manage finding, cutting and importing ropes.

### cutFirstFit()

Variables: orderLength (The length of rope required by the order)  
 stock (An array of lengths of rope)  
 stockSize (Number of ropes in stock (The size of ‘stock’))  
 orders (An array of rope orders)  
 ordersSize (Number of orders in ‘orders’ (The size of ‘orders’))

BEGIN:  
 for i := 0 to orderSize - 1 do  
 orderLength := orders[i]  
 index := findFirstFit(orderLength)  
 if index == -1  
 import rope from factory to stock  
 index := stockSize – 1  
 cut(index, orderLength)  
 shouldRemoveRope(index)  
END

## Best-Fit

Next I shall implement the Best-Fit algorithm.

Best-Fit: Search the stock for a rope whose length is as close to the order length as possible but still long enough to be cut. If no rope equal to or greater in length than the ordered rope exists then import a new rope to the stock and cut that rope.

### findBestFit(orderSize)

Variables: orderLength (The length of rope required by the order)  
 stock (An array of ropes)  
 stockSize (Number of ropes in stock (The size of ‘stock’))  
 bestLength (length of the best fitting rope)  
 bestIndex (Index of the best fitting rope in ‘stock’)  
 ropeLength(i) (length of rope at position i in ‘stock’)

BEGIN:   
 bestLength := integer greater than the maximum length of rope  
 bestIndex := -1  
 for i := 0 to stockSize - 1 do  
 if canCut(i, orderLength) AND ropelength(i) < bestLength  
 if ropeLength(i) == orderSize  
 return i  
 bestLength := ropeLength(i)  
 bestIndex := i

return bestIndex  
END

Now I can implement the part of the algorithm that will manage finding, cutting and importing ropes

### cutBestFit()

Variables: orderLength (The length of rope required by the order)  
 stock (An array of ropes)  
 stockSize (Number of ropes in stock (The size of ‘stock’))  
 orders (An array of rope orders)  
 ordersSize (Number of orders in ‘orders’ (The size of ‘orders’))

BEGIN:  
 for i := 0 to orderSize - 1 do  
 orderLength := orders[i]  
 index := findBestFit(orderLength)  
 if index == -1  
 import rope from factory to stock  
 index := stockSize – 1  
 cut(index, orderLength)  
 shouldRemoveRope(index)  
END

# Implementation

For this assignment I implemented my algorithms in Java.

In order to represent a rope I have created a Rope object, this class takes a single argument in its constructor; an integer “length” which represents the length of the rope. The length is stored as a private member variable and has getters and setters so I can retrieve and change the length.

I created a RopeCutter object that will represent the shop, this takes two parameters, two ArrayLists of integers that will become the factory and orders. This object contains the following:

* An ArrayList of Integers to represent the factory from which the shop imports rope, every time a new rope is imported the index used to create the rope is incremented by one.
* A second ArrayList of Integers to represent the orders being sent to the shop, I will be iterating through this ArrayList in order to simulate a set of orders.
* A third ArrayList of Rope objects to represent the stock of the factory.
* The methods required for the First-Fit and Best-Fit algorithms to process a set of orders.

To then test my ropeCutter I have implemented a RopeCutterMain class, this class generates two ArrayLists of random integers within the bounds laid out in the specification, it then tests the rope cutter class many times, more detail can be found in the class.

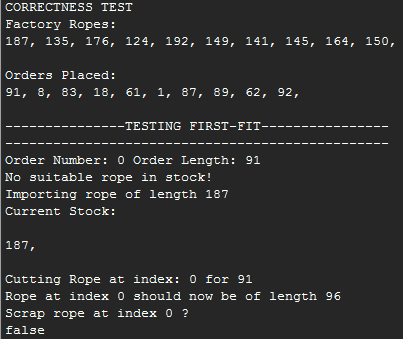
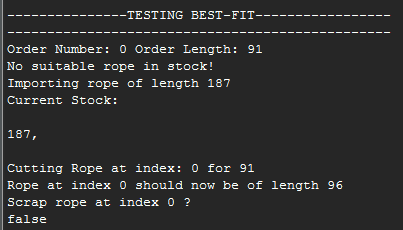
# Testing

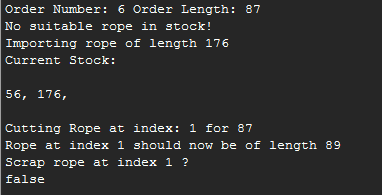
## Correctness Testing

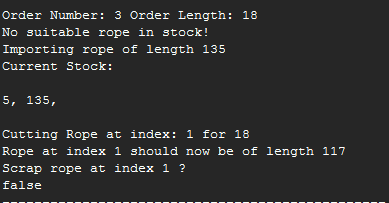
To check if my algorithms were operating as intended I created a runTest() method within the ropeCutterMain class. When this algorithm is called this will run the algorithms using the variable TEST\_ORDER\_NUMBER to determine how many orders to simulate and print each step of the algorithm to the console. Using this output I can read through each step of the algorithm and check if it is performing as I expect, I ran this test multiple times using an unseeded random generator to generate different sets of orders and factory ropes for my algorithms to process. The output for the test currently hard coded is included in “Example Output.txt”. To recreate this tests uncomment line in the ropeCutterMain class that should read “//runCorrectnessTest();” and make sure the seed for the random object in this method is 1000.

Using this output I checked for several things. The same test data was used in both tests.

Is the algorithm importing the correct rope from the factory?

The ropeCutter initially starts with no ropes in stock, and must import the first rope from the factory.

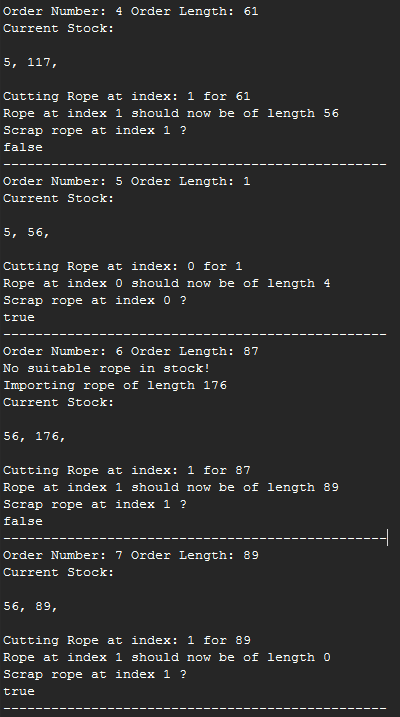
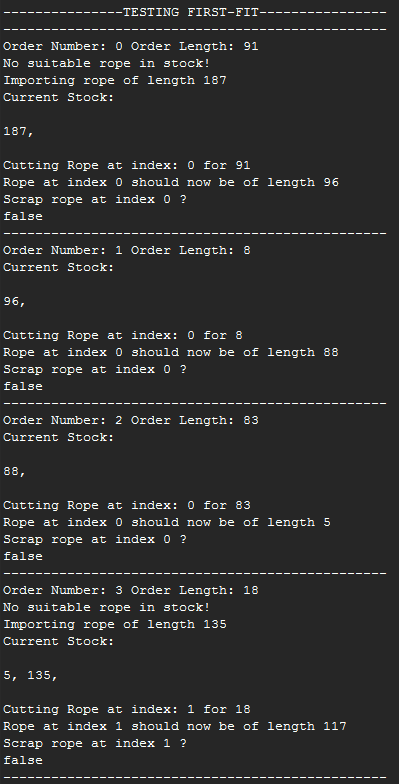
The tests show that it does correctly import the rope at the start of the algorithms as the rope of length 187 is being imported and is at the start of the factory. But I must also check if it imports correctly during the algorithm.



As shown above both algorithms are importing the correct ropes. The rope of length 135 being the second rope to be imported and the second rope in the factory. The rope of length 176 being the third rope to be imported in the factory.

I then checked the orders to see if they were being processed in the correct order, as shown by the tests above they are indeed being processed correctly, (Order Number is the index of that order and starts at 0).

Are the algorithms selecting the correct rope and cutting it? Also do they scrap ropes correctly?

The above console output shows the first 8 Iterations of the First Fit algorithm (more can be found in the text file, or generated if necessary), it is clearly shown that it is selecting suitable (suitable meaning the first rope of length large enough to meet the needs of the current order) ropes to be cut and is cutting them correctly. Orders 5 and 7 show that the ropes are being scrapped under the correct conditions.

I shall now check Best-Fit for the same conditions

# 

The above output shows the first 8 iterations of Best-Fit, again the algorithm is performing as expected, selecting, cutting and scrapping correctly.

## Performance testing

The specification outlined two methods of testing the performance of my algorithms, one based on how long they take to complete, the other on the number of rope coils supplied by the factory. I will test the algorithms using an array of inputs of different sizes to see how it performs for in increasing amount of orders. I will be using the same data input to test both algorithms.

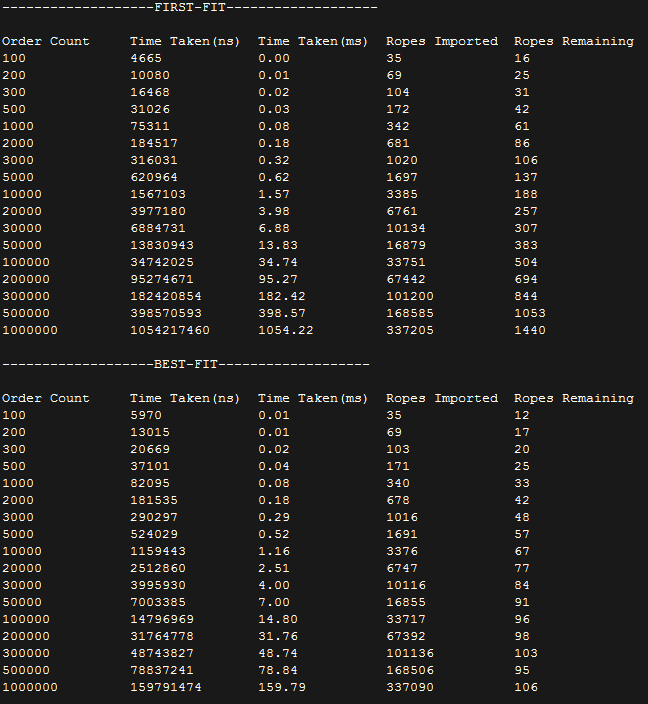
To get an accurate measurement of the performances of the algorithms for an increasing amount of orders I implemented a test that tests the algorithms ten times at each provided order size. I then stored the time it took each to complete, the amount of ropes imported and the ropes remaining at the end. I would then take an average for each order size and print the results in a table. However when I first ran this test I noticed the averaged values for the smaller order sizes were not giving me the results I expected, upon further analysis I discovered that the first few tests were taking much longer then the following tests and that it was through no fault of my own (possibly due to CPU or issues with the Java Virtual Machine).

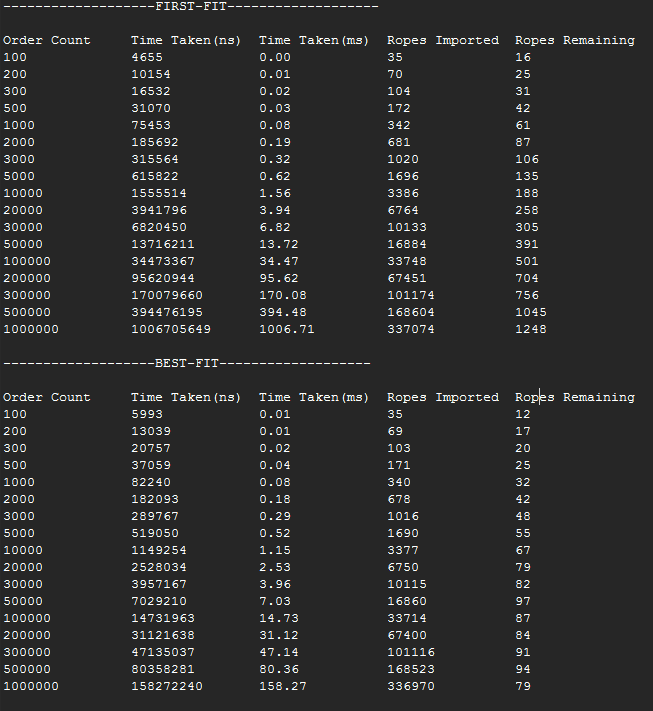
To counter this problem I implemented a way of testing the algorithm so that orders of smaller size were tested more than orders of larger size. I have done this by declaring how many times I would test an order of size one (the constant TEST\_COUNT in the code), I then decide how many times I will test each order size by doing TEST\_COUNT / the current order size.

For example if TEST\_COUNT = 10,000,000 (as it does in the source code) an order of size 10 will be tested 10,000,000 / 10 times.

Running the earlier order sizes so many times made the impact of the first few tests on the smaller order sizes negligible.

I first ran the tests without a seed for the random number generator to obtain an average of a set of truly random sets of orders, you will not be able to replicate these exact results but you can run a similar test by not providing a seed to the random object in the main method. The results were as follows:



I then tested the algorithms for a seeded random number generator (the seed I used was 1000). Seeding the generator should give a more accurate representation of time taken as it can fluctuate, the results were as follows:

The above tests show some trends in the performance of the algorithms, I will now go through my results and comment on my findings with support from a graphical representation.

# Results

The graph shows that at 100 orders, First Fit is taking 78% the time of Best Fit. As the order size increases the time difference between First Fit and Best Fit decreases until they become equal at around the order size of around 2000. For all orders greater than 2000 Best Fit becomes faster at processing. The difference between the First Fit and Best Fit times then starts to increase and at the order size of 1,000,000 First Fit takes 659% longer to process the orders than Best Fit.

The seeded data, supports the data from the non-seeded tests, it follows the same trend of First Fit being faster initially, then at around the 2000 mark the algorithms perform equally then after this mark the Best Fit algorithm is more efficient. At the 1,000,000 mark Best Fit is 636% faster than best fit.

The above graph shows the import counts the for both the algorithms (the ropes required from the factory in order for the algorithm to complete), you will notice that the results are so similar that you can’t see the results for First Fit, this is because the results for the two algorithms are nearly identical. For the first two results the algorithms imported exactly the same amount of rope on average, the maximum difference between the two occurred at 3000 and even then there was only a 1.5% difference between the two.

The seeded test shows the same results:

The above graph shows the average rope remaining in the stock. It is clear that the Best Fit algorithm performs much better at keeping the stock as small as possible. This holds true for all order sizes. The seeded test also supports this conclusion.

# Conclusion

Using my results I can now make some comments on the performance of the algorithms I have implemented

Which algorithm is more efficient?

As shown by my testing of each algorithms speed, it is apparent that the First Fit algorithm is more efficient for processing order sets of smaller sizes (approximately <2000). For order sets of greater magnitude the Best-Fit algorithm is clearly more efficient, more than 600% more efficient when the order size is 1 million. This difference in efficiency will only increase with greater order sizes.

However I believe that both the algorithms I implemented have a complexity of O(n). My results for the time taken would support this as generally increasing the input by a factor of 10 increases the time taken by a factor of 10.

If we use the amount of ropes supplied by the manufacturer to measure the performance of the algorithms then the algorithms are almost exactly equal in terms of efficiency. My results show the algorithms import roughly the same amount of rope for a given order size, though my Best Fit algorithm will tend to import slightly less.

Again if you base performance on the amount ropes imported then both my algorithms have complexity of O(n).

Why was Best-Fit more efficient for larger inputs?

The reason Best-Fit managed to outperform First-Fit lies in the way it cuts the ropes. Best fit looks for the smallest rope that can handle the current order, ropes that are selected to be cut using this method are more likely to become short enough to be scrapped. Scrapping more ropes keeps the stock small and thus the algorithm has to iterate through less ropes in order to find the best one to cut.

The First Fit algorithm out performs best fit for smaller inputs for a similar reason, since it just looks for the first suitable rope, it has to iterate through less of the stock. However this simplistic approach then backfires for larger inputs as the stock will become full of small ropes that are unlikely to be cut thus making the stock larger and causing more iterations.

Overall after the 2000 order mark, where Best Fit becomes faster, as both algorithms tend to an infinite number of orders the time difference between the two algorithms increases exponentially. Also both my algorithms import a near identical number of ropes with Best Fit tending to be slightly less.