REPORT : THE FERRY LOADING PROBLEM - SOLVED

In this assignment, we were asked to solve the Ferry Loading problem using a specific algorithm: The **backtracking** and **memorization** algorithm.

For this assignment, we explored two alternatives to memorize the different visited states:

1. **BigTable** : This alternative uses a 2D array of size (L+1) x (n+1), where L is the length of the ferry, and n is the number of cars to load.
2. **HashTable** : This alternative uses a hashtable.

Part 1 - BigTable Solution Validation

The solution implemented with a 2-D array passed all the OnlineJudge tests and gave the following runtime

#	Problem	Verdict	Language	Run Time	Submission Date
25803425	10261 Ferry Loading	Accepted	JAVA	0.410	2020-12-03 17:38:14
25803423	10261 Ferry Loading	Accepted	JAVA	0.300	2020-12-03 17:37:46
25803420	10261 Ferry Loading	Accepted	JAVA	0.340	2020-12-03 17:37:27
25803417	10261 Ferry Loading	Accepted	JAVA	0.310	2020-12-03 17:37:07
25803413	10261 Ferry Loading	Accepted	JAVA	0.370	2020-12-03 17:36:40

Part 2 - HashTable Solution Validation

The solution implemented with a hashtable passed all the OnlineJudge tests and gave a **faster** runtime

#	Problem	Verdict	Language	Run Time	Submission Date
25803451	10261 Ferry Loading	Accepted	JAVA	0.240	2020-12-03 17:42:32
25803450	10261 Ferry Loading	Accepted	JAVA	0.200	2020-12-03 17:42:19
25803448	10261 Ferry Loading	Accepted	JAVA	0.210	2020-12-03 17:42:02
25803443	10261 Ferry Loading	Accepted	JAVA	0.200	2020-12-03 17:41:41
25803441	10261 Ferry Loading	Accepted	JAVA	0.190	2020-12-03 17:41:14

Part 3 - Hashtable Implementation & Design

- For this assingment, I decided to use Java's built-in HashMap ADT.
- I initialized it with a size of 16[*] and a load factor of 0.75[*] (the default values as per Java 8 SE Documentation)
- My hashmap stored boolean values. `Entry` objects that store the states (k,s) were required to access the values.
- I overrode the Entry class hashcode() function to implement my hash function. The hash function took two integers k and s, and returned the result of the following operation:

```
key.hashCode() * 37 + value.hashCode()
```

[*] **I experimented with different values but the difference in runtime were not significant. According to the Java SE documentation, ideally, we don't want to initialize our hashtable with a size that's too high or a load factor that's too low.**

Challenges

- While designing my solution with a hashtable, I ran into a couple of runtime issues. The runtime of my initial solution exceeded the time limit. This was due to the fact that I was creating a new `Entry` object before to mark a given state `visited`. To solve that problem, I added an instance variable of type `Entry` named `ks` and updated it with the current state that was just visited before inserting it into the hashtable.