

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 $(L+1) \times (n+1)$, 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
25810359	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-04 23:25:04
25810357	10261 Ferry Loading	Accepted	JAVA	0.260	2020-12-04 23:24:42
25810355	10261 Ferry Loading	Accepted	JAVA	0.240	2020-12-04 23:24:19
25810351	10261 Ferry Loading	Accepted	JAVA	0.230	2020-12-04 23:23:54
25810349	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-04 23:23:38
25810347	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-04 23:23:20
25810346	10261 Ferry Loading	Accepted	JAVA	0.260	2020-12-04 23:22:58
25810345	10261 Ferry Loading	Accepted	JAVA	0.230	2020-12-04 23:22:37
25810342	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-04 23:22:21
25810340	10261 Ferry Loading	Accepted	JAVA	0.240	2020-12-04 23:22:02

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
25810301	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-04 23:15:38
25810282	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-04 23:13:06
25810258	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-04 23:10:44
25810249	10261 Ferry Loading	Accepted	JAVA	0.160	2020-12-04 23:09:29
25810246	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-04 23:09:12
25810243	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-04 23:08:53
25810242	10261 Ferry Loading	Accepted	JAVA	0.170	2020-12-04 23:08:21
25810241	10261 Ferry Loading	Accepted	JAVA	0.160	2020-12-04 23:08:06
25810240	10261 Ferry Loading	Accepted	JAVA	0.170	2020-12-04 23:07:47
25810236	10261 Ferry Loading	Accepted	JAVA	0.130	2020-12-04 23:07:19

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.