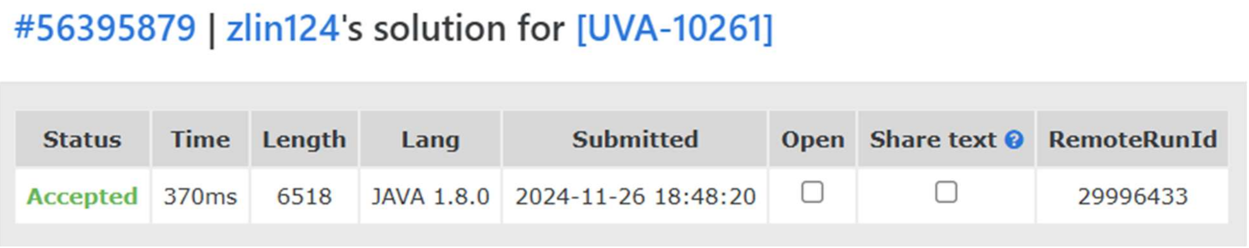
A. Status of Part 1.

The big table solution passed all the test cases below, and it passed the online virtual Judge too.

input1.txt input2.txt input3.txt input4.txt input5.txt



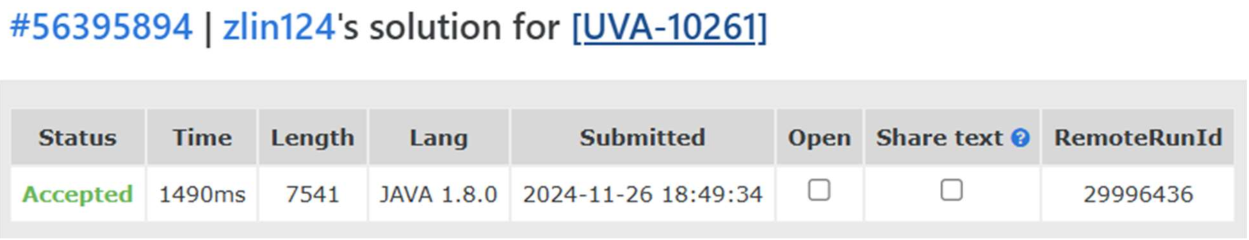
#56395879 | zlin124's solution for [UVA-10261]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	370ms	6518	JAVA 1.8.0	2024-11-26 18:48:20	□	□	29996433

B. Status of Part 2.

The hash table solution passed all the test cases below, and it passed the online virtual Judge too.

input1.txt input2.txt input3.txt input4.txt input5.txt



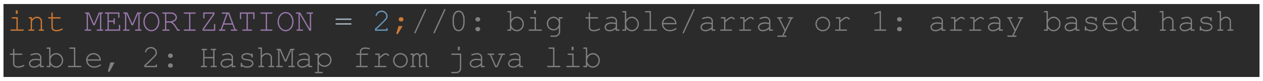
#56395894 | zlin124's solution for [UVA-10261]

Status	Time	Length	Lang	Submitted	Open	Share text	RemoteRunId
Accepted	1490ms	7541	JAVA 1.8.0	2024-11-26 18:49:34	□	□	29996436

C. Details on design of Part 2

Three types of memorization are implemented, the variable MEMORIZATION is used to switch

0: big table
1: array based 4 other hash functions and probing methods, from lab 7.
2: HashMap from Java JDK 1.8



```
int MEMORIZATION = 2;//0: big table/array or 1: array based hash
table, 2: HashMap from java lib
```

<u>1. Here are 4 array based probe hash functions. The minimal size of the size is 67 for all 5 inputx.txt test cases.</u>

//liner probe function
liner_probe(int k, int j)
{
return (k % SIZE +j) % SIZE;
}
	//Quadratic probe function
	private int q_probe(int k, int j) {
	return (k % SIZE + j*j) % SIZE;
	}
		//Double probe function
		private int d_probe(int k, int j) {
		return (k%SIZE + j * (7-(k % 7))) % SIZE;
		}
			//Quadratic Double probe function
			private int dq_probe(int k, int j) {
			return ((3*k + 1)%SIZE + j*j) % SIZE;
			}

When set initial capacity to SIZE = 256 * (L) *2, it can pass all 5 input text, but could not pass online judge due to Time Limit Exceed.

The size impact to performance is not significant.

Hash Function	Hash Table size	Probe Times	Runtime	
liner	256	3273	0.0289747	
Quadratic	256	692	0.0192863	
Double	256	1220	0.0189539	
Quadratic Double	256	621	0.0190746	

Hash Function	Hash Table size	Probe Times	Runtime	
liner	128	3273	0.020342	
Quadratic	128	692	0.0188186	
Double	128	1220	0.0196258	
Quadratic Double	128	621	0.0190746	

Hash Function	Hash Table size	Probe Times	Runtime	
liner	94	3273	0.0207875	
Quadratic	94	692	0.0190983	
Double	94	1220	0.0196976	
Quadratic Double	94	647	0.0202203	

Hash Function	Hash Table size	Probe Times	Runtime	
liner	64			couldn't find a position.
Quadratic	64			couldn't find a position.
Double	64			couldn't find a position.
Quadratic Double	64			couldn't find a position.

<u>2. With the HashMap, the original design is car number as key and state as value.</u>

```
HashMap<Integer, Integer> hashMap;
```

- No matter what `initialCapacity` and `loadFactor` are set, the run time will exceed 3000 ms.
- After redesign getRS which calculates the right lane free space using cache, the run time still more than 3000 ms.

```
int getRS(int k, int s) {
    rsCount++;
    if (k == 0)
        return L;
    int t = sub[k-1];
    int re= L*2-t-s;
    return re;
}
```

- Tried getOrDefault instead of get, still more than 3000 ms.
- Remove all the debug info to system err, no help either.
- Use hashMap.clear() instead of create new object, no help.

At last, the HashMap is used in another way: concatenate car number and state together as key, this finally improve the running time, passed online judge .

```
HashMap<String, Boolean> hashMap;
```

```
hashMap.put(k+"."+s,b);
```

```
Boolean re = hashMap.getOrDefault(k+"."+s,null);
```