Name: Inho choi **ID:** 1801787.

Problem 1: Custom Hash

Suppose we created the following object for an application:

```
struct BadMovie {
	int databaseID;
	string name;
	string director;
	int runtimeInSeconds;
	int rating;
};
```

We decided that a Hash Table is the most appropriate data structure for our purposes. However, in our experiments, hashing on the databaseID alone results in many collisions. So, we decided to write a custom hashing function that incorporates all of the member variables to determine the appropriate bucket. Complete such a hash function below, you may assume that the constant ARRAY_SIZE is defined.

```
int hash(const BadMovie &bm) {
    int hash_id = bm.databaseID;
   int hash_name = 0;
  for (int i=0; i < bm.name.size(); ++i)
         hash_name += (i+1) * bm.name[i];


  int hash_dir.= 0;
  for (int i=0 ; i < bm.director.size();++i)
        hash_dir += (i+1)*bm.director[i];
  int hash_rt = bm.runtimeIn Seconds;
  int hash_rate = bm.rating;
 return (hash_ID + hash_name+ hash_dir + hash_rt + hash_rate) % ARRAY_SIZE;
```

Name: Inho Choi **ID:** 1801787.

Problem 2: Hash Tables

Consider a **Closed Hash Table** with HASH_SIZE of 10:

```
class ClosedHashTable {
public:
        void insert(int key){
                int bucket = hashFunc(key);
                for (int tries = 0; tries<HASH_SIZE; tries++) {
                        if (hash_array[bucket] == EMPTY) {
                                hash_array[bucket] = key;
                                return;
                        }
                        bucket = (bucket + 1) % HASH_SIZE;
                }
        }
private:
        int hashFunc(int x) {
                return (x * 2) % HASH_SIZE;
        }
        int hash_array[HASH_SIZE];
};


ClosedHashTable ch;
ch.insert(7);
ch.insert(1);
ch.insert(23);
ch.insert(14);
ch.insert(19);
ch.insert(53);
ch.insert(37);
ch.insert(83);
```

0	83
1	
2	1
3	
4	7
5	37
6	23
7	53
8	14
9	19

a. Show result of the insert commands in the array to the right. Using the state of the hash table **after all** the inserts, what is the load factor and average number of checks for this hash table?

Load Factor = 0.8

Ave # checks $= \frac{1}{2}(1+\frac{1}{1-L}) = 3$

b. How much larger would we have to make the hash table if we wanted to have average number of checks to be roughly 1.10?

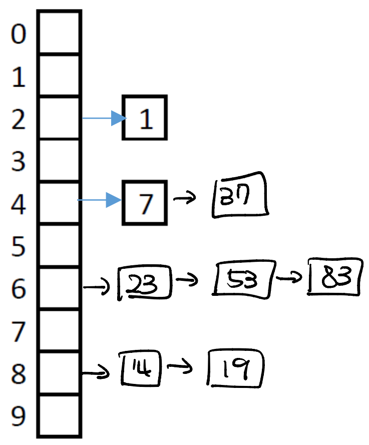
$1.10 = \frac{1}{2}(1+\frac{1}{1-L})$

$L = 0.1667$

arraysize of 48.

Name: Inho Choi **ID:** 1801787

c. Suppose we, instead, inserted those values into an **open hash table** using the same hash function. What would the state of the table look like? (This first two have been done for you.)



d. What is the average number of checks for this open hash table given its state after all the inserts?

We're storing 8 elements with 10 buckets, the load factor is given by $L = 8/10 = 0.8$
Average number of tries for an open hash table is given by $1 + \frac{L}{2} = 1 + 0.8/2 = 1.4$

e. Discuss the quality of this hash function.

The primary factor in how well a hash table performs relates to collisions.
Meaning the quality of a hash function is determined by how relates to collisions in the table.
This one in particular effectively halves the number of buckets in the entire table,
thus increasing collision. Because of this, this is not a particularly good hash function.