

1 Fun with Hash Functions (from 61BL SU2014 MT2)

Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw/disadvantage.

Note: A "valid" `hashCode()` means that: any two `Integers` that are `.equals()` to each other should also return the same hash code value.

Another note: the `Integer` class extends the `Number` class, a direct subclass of `Object`. The `Number` class' `hashCode()` method directly calls the `Object` class' `hashCode()` method.

(a)

```
public int hashCode() {  
    return -1;
```

```
}
```

(b)

```
public int hashCode() {  
    return intValue() * intValue();
```

```
}
```

(c)

```
public int hashCode() {  
    return super.hashCode();
```

```
}
```

2 HashMap Modification (from 61BL SU2010 MT2)

(a) When you modify a key that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

Always Sometimes Never

(b) When you modify a value that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

Always Sometimes Never

3 Analyzing Proj1 ArrayDeque's Runtime

Recall the `ArrayDeque` from proj1. Our implementation uses a circular array with two pointers denoting the front and back of the `ArrayDeque`. Starting with an initial size of 8, the array doubles in size when it reaches full capacity, and halves in size when its load factor is lower than 0.25. Resizing will reposition the elements to start from index 0 for ease of maintenance. Fill in this table with best, worst, and average case runtimes of the `ArrayDeque` methods (in Θ notation).

	Best Case	Worst Case	Average Case
<code>addFirst/Last</code>			
<code>removeFirst/Last</code>			
<code>get</code>			

4 Recursion and Dynamic Programming

Implement Fibonacci using memoization (memorizing previously solved problems).

```
public class Fibonacci {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

    // For your reference, fib(0) = 0, fib(1) = 1, fib(2) = 1, ...
    public int fib(int n) {

    }
}
```

5 Hashing a Tic-Tac-Toe Board (Bonus)

Given the provided (minimal) implementations below, write the `.hashCode()` and `.equals()` methods for the `Piece` and `Board` classes (we did `Piece.equals()` for you). Try to ensure that different board configurations have different hash codes.

```
public class Piece {
    private String type; // Will be either "X" or "O".

    /** Some other code... */

    public boolean equals(Object o) {
        Piece otherPiece = (Piece) o;
        return this.type.equals(otherPiece.type);
    }

    public int hashCode() {

    }
}

public class Board {
    public static final int SIZE = 3; // Tic Tac Toe Boards are always 3x3

    private Piece[][] pieces;
    private boolean isXTurn;

    /** Some other code... */

    public int hashCode() {

    }
}
```

Bite-sized Bonus: How do you implement the `.equals()` method for `Board`?

Bigger Bonus: Is it possible to perform a "perfect hash"? If we now wanted to have three different types of pieces, X's, O's and Triangles, does that change your answer?