CS 201, Fall 2015 — Dec 4th — Exam 3 Name: Solution

Question 1. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the `list` parameter. You can assume that the call to `equals` is $O(1)$. Explain your answer briefly.

```
public static<E> int find(ArrayList<E> list, E elt) {
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).equals(elt)) {
      return i;
    }
  }
  return -1;
}
```

N times

list.get(i) is O(1) for ArrayList

$N \cdot O(1)$ is $O(N)$

Question 2. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the `list` parameter. You can assume that the call to `equals` is $O(1)$. Explain your answer briefly.

```
public static<E> int find(LinkedList<E> list, E elt) {
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).equals(elt)) {
      return i;
    }
  }
  return -1;
}
```

N times

list.get(i) is O(N) for LinkedList (must skip O(N) elements on average)

$N \cdot O(N)$ is $O(N^2)$

Question 3. [5 points] Complete the following generic method. It takes two values of type `E`, and a `Comparator` that can compare values of type `E`, and returns the smaller of the two values.

Hint: use the comparator's `compare` method to compare the two values.

```
public static<E> E min(E val1, E val2, Comparator<E> comp) {
    if ( comp.compare(val1, val2) < 0 ) {
        return val1;
    } else {
        return val2;
    }
}
```

Question 4. [5 points] Indicate whether a stack, or a queue would be best for implementing each of the following:

(a) A keyboard buffer: queue

(b) A text editor "undo" operation: stack

(c) Buffering events in a video game: queue

(d) Capturing the most recent 30 seconds of security video: queue

(e) Reversing the contents of a list: stack

Question 5. [5 points] For a certain card game, the value of each card in a Suit is multipied by its Suit's multiplier factor (an integer). Using the given Suit enum, declare and create a Map that allows the user to retrieve a Suit's multiplier factor, where SPADES = 8X, HEARTS = 4X, DIAMONDS = 2X and CLUBS = 1X. Be sure to show how the map is populated.

```
public enum Suit {
  SPADES,
  HEARTS,
  DIAMONDS,
  CLUBS
}
```

```
Map<Suit, Integer> m = new HashMap<Suit, Integer>();
m.put(Suit.SPADES, 8);
m.put(Suit.HEART, 4);
m.put(Suit.DIAMONDS, 2);
m.put(Suit.CLUBS, 1);
```

Question 6. [5 points] Given the following two enums, declare and create a Map that cross-references a Suit with its Color. Be sure to show how the map is populated.

```
public enum Suit {
  SPADES,
  HEARTS,
  DIAMONDS,
  CLUBS
}

public enum SuitColor {
  RED,
  BLACK
}
```

```
Map m = new HashMap<Suit, SuitColor>();
m.put(Suit.SPADES, SuitColor.BLACK);
m.put(Suit.HEARTS, SuitColor.RED);
m.put(Suit.DIAMONDS, SuitColor.RED);
m.put(Suit.CLUBS, SuitColor.BLACK);
```

Question 7. [10 points] Consider the following method to compute an iteration count to test whether a complex number is in the Mandelbrot set:

```
public int computeIterCount(Complex c){
  Complex z = new Complex(0, 0);
  int count = 0;

  while (z.getMagnitude() < 2 && count < MAX_COUNT){
    z = z.multiply(z).add(c);
    count++;
  }
  return count;
}
```

(a) Implement a recursive version of this computation:

```
// Recursive version
public int computeRecursiveIterCount(Complex c, Complex z, int count){
    if (count >= MAX_COUNT || z.getMagnitude() >= 2.0) {
        return count;
    }

    return computeRecursiveIterCount(c, z.multiply(z).add(c),
                                     count + 1);
}
```

Make sure you check a base case or base cases, that the recursive call(s) work towards a base case, and that the result of the recursive call or calls is extended to be a solution for the overall problem.

(b) Is it a good idea to implement this computation recursively? Briefly explain why or why not.

No, because very deep recursions can throw StackOverflowError.

Question 8. [5 points] Consider the following methods, which attempt to compute the n'th member of the Fibonacci sequence using memoization:

```
public static int fib(int n) {
  return fibMemo(n, new int[n+1]);
}

private static int fibMemo(int n, int[] memo) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    if (memo[n] == 0) {
      memo[n] = fib(n-2) + fib(n-1);
    }
    return memo[n];
  }
}
```

should call fibMemo

Breifly explain the error in this code, and how to fix it.

Calling fib rather than fibMemo creates a new (empty) memoization table. So, answers to previously encountered subproblems are not saved.

Solution is to change the recursive calls to

fibMemo(n-2, memo) + fibMemo(n-1, memo)

Question 9. [10 points] Given the constructor for the `CalculateTask` (which implements `Runnable`)

```
public CalculateTask(int[] arr, int start, int end)
```

where `start` and `end` specify the portion of array `arr` to process for each thread, complete the following fork/join code to process the array in parallel, using `numThreads` threads.

Hint: think about how you can divide up the elements of the array to split the work equally between tasks.

```
int chunk = arr.length / numThreads;

Thread[] threads = new Thread[nummThreads];
CalculateTask[] tasks = new CalculateTask[numThreads];

// create CalculateTasks and specify range to be processed
for (int i = 0; i < numThreads; i++) {
    tasks[i] = new CalculateTask(arr,
        i * chunk, (i+1) * chunk );

}

// create Threads, initialize from tasks array,
// and start the threads
for (int i = 0; i < numThreads; i++) {
    threads[i] = new Thread(tasks[i]);
    threads[i].start();
}

// wait for the threads to complete
try {
  for (int i = 0; i < numThreads; i++) {
      threads[i].join();


  }
} catch (InterruptedException e) {
  System.out.println("Error waiting for thread to complete: " + e);
}
```