Final Exam Practice Problems – Solutions

1. Two different versions; others are possible.

```
public static boolean hasDuplicate(int[] array)
{
    for (int i = 0; i < array.length; i++)
    {
        for (int j = i+1; j < array.length; j++)
        {
            if (array[i] == array[j]) {
                return true;
            }
        }

    }
    return false;
}

public static boolean hasDuplicate_Version2(int[] array)
{
    for (int i = 0; i < array.length; i++)
    {
        int count = 0;
        for (int j = 0; j < array.length; j++)
        {
            if (array[i] == array[j]) {
                count++;
            }
        }
        if (count >= 2) { // there will always be at least one match when i==j
            return true;
        }
    }
    return false;
}
```

- The big oh time is: **$O(n^2)$**. Any way you write this function (with the content we know in 142), it's going to be polynomial. [In CS241, you will learn strategies to reduce the run time to linear.]

2. If the array is sorted, any duplicate items will appear next to each other, so the problem is reduced to finding neighboring elements that match.

```
public static boolean hasMatchingNeighbor(int[] array)
{
    for (int i = 0; i < array.length - 1; i++)
    {
        if (array[i] == array[i+1]) {
            return true;
        }
    }
    return false;
}
```

- This function is linear time: **O(n)**-you only have to compare each index with its neighbors. In the worst case (where there are no duplicates), you go through the entire array once.

3.

```
public static int sumOdd(int n) {
    if (n == 1) {
        return 1;
    }
    else if (n % 2 == 0) { // even
        return sumOdd(n-1);
    }
    else {  // odd
        return n + sumOdd(n-2); // interestingly, also works with n-1
    }
}
```

4. [on separate lines:] 7 8 9 8 10 7 9

5. (Lots of ways to do this one)

```
public static String stars(int n) {
    if (n == 0) {
       return "";
    }
    else {
       return "*" + stars(n-1);
    }
}
```

6. (Note that the instance variables in the problem are private and so this solution would actually throw an error if we implemented it in Java*. The solution shown here is the one we'd use if the variables were public.

   *This is because, since topLeftX is private, if we try to call other.topLeftX, Java will complain. This is true of the other instance variables as well.)

   Algorithm: For one rectangle (call it "A") to be completely contained inside another one (call it "B"), we need four things to be true:

   * The x-coordinate of A's top left corner must be greater than the x-coord of B's top left corner
   * The x-coordinate of A's top right corner must be less than the x-coord of B's top right corner
   * The y-coordinate of A's top left corner must be greater than the y-coord of B's top left corner
   * The y-coordinate of A's bottom left corner must be less than the y-coord of B's bottom left corner

   So in our code, "this" Rectangle is "A" and the "other" rectangle is "B".

```
public boolean isInside(Rectangle other) {
  int topLeftXA = this.topLeftX;
  int topLeftXB = other.topLeftX;
  int topRightXA = this.topLeftX + width;
  int topRightXB = other.topLeftX + other.width;
  int topLeftYA = this.topLeftY;
```

```
    int topLeftYB = other.topLeftY;
    int bottomLeftYA = this.topLeftY + height;
    int bottomLeftYB = other.topLeftY + other.height;

    return (topLeftXA > topLeftXB) && (topRightXA < topRightXB) &&
           (topLeftYA > topLeftYB) && (bottomLeftYA < bottomLeftYB);
}
```

7.

Here's one way to do it: (other ways possible)

```
class Time {
    private int hour;
    private int minute;
    private boolean isAM;

    public String toString() {
        String minuteString = String.valueOf(minute); // convert minute to string
        if (minute < 10) {
            minuteString = "0" + minuteString;
        }
        String answer = hour + ":" + minuteString;
        if (isAM) {
            return answer + " AM";
        }
        else {
            return answer + " PM";
        }
    }

    public boolean isEarlierThan(Time otherTime) {
        if (isAM && !otherTime.isAM) {  // we are AM and they are PM
            return true;
        }
        else if (!isAM && otherTime.isAM) { // we are PM and they are AM
            return false;
        }
        else {
            // both are AM or both are PM
            if (hour < otherTime.hour) {
                return true;
            }
            else if (hour > otherTime.hour) {
                return false;
            }
            else {
                // hours match, check minutes
                if (minute < otherTime.minute) {
                    return true;
                }
                else if (minute > otherTime.minute) {
                    return false;
                }
                else {
                    // times are exactly the same, so return false
                    return false;
                }
            }
        }
```

```
        }
    }
```

8. You will notice that the amount of time the algorithm takes roughly quadruples when the input size doubles. Therefore this is (most likely) a quadratic-time algorithm [$O(n^2)$]. Doubling 80,000 elements in the array would be 160,000 elements, which would take roughly, therefore, 4 * 1451 seconds = 5804 seconds = roughly 96 minutes = a little more than an hour and a half. Since we are looking for how long it would take to process 150,000 elements (less than 160,000), we would say roughly an hour and a half (certainly somewhere in the vicinity of 1-2 hours because all these estimates are rough).

9. Notice how the code starts at index 1 for both row & column and also ends at the length of each dimension minus 1. This is because the description of the problem says to ignore any numbers in the array that don't have all four neighbors, which would be any number on the "border" of the array. If I had said you still need to check those, then you would have to start the loops at zero, go up to the length of each dimension, and use additional "if" checks to see if the appropriate neighbors exist.

```
public static boolean findNumberWithLargerNeighbors(int[][] array) {
    for (int row = 1; row < array.length - 1; row++) {
        for (int col = 1; col < array[row].length - 1; col++) {
            if (    array[row][col] < array[row+1][col] &&
                    array[row][col] < array[row-1][col] &&
                    array[row][col] < array[row][col+1] &&
                    array[row][col] < array[row][col-1])
                return true;
        }
    }
    return false; // no numbers found that match the "if"
}
```

10. All lines are legal except for lines 7, 9, and 10. Boat does not have a refuel method and so this line would throw an error!

    1-f's fuel is 3.
    2-g's fuel is 3.
    3-h's fuel is 3.
    4-f's fuel is 1. (We subtract both 1 and the cargoQuantity (which is also 1).)
    5-g's fuel is 1. (We subtract 1 and the cargoQuantity (which is 0) in super.aheadFull and then 1 more.)
    6-h's fuel is 1. (We subtract 1 and the cargoQuantity (which is 0) in super.aheadFull and then 1 more.)
    7-illegal; no comment
    8-g's fuel is 11.
    9-illegal; no comment
    10-illegal. Note that the line f.hail((Boat) g) would work, but, as written, this would not work.
    11-Fuel does not change, but output is "Sub2 sends a radio message to Boat1"
    12-Fuel does not change, but output is "Boat1 sends a radio message to Boat3"

11. You would make a single change: the class would become public abstract class Boat {...}. This is the only change needed to make Boat abstract! You wouldn't be able to instantiate a Boat object any more (new Boat() would cause Java to throw an error), but otherwise the class would remain the same.

    In order to implement the abstract function getSpeed() **in Boat**, you would simply add:
    public abstract int getSpeed();
    This function isn't actually implemented in Boat but would have to be implemented in all of the child classes.