CS314 Spring 2017 Exam 1 Solution and Grading Criteria.

Grading acronyms:
AIOBE - Array Index out of Bounds Exception may occur
BOD - Benefit of the Doubt. Not certain code works, but, can't prove otherwise
Gacky or Gack - Code very hard to understand even though it works. (Solution is not elegant.)
LE - Logic error in code.
NAP - No answer provided. No answer given on test
NN - Not necessary. Code is unneeded. Generally no points off
NPE - Null Pointer Exception may occur
OBOE - Off by one error. Calculation is off by one.
RTQ - Read the question. Violated restrictions or made incorrect assumption.

1. Answer as shown or -1 unless question allows partial credit.
No points off for minor differences in spacing, capitalization, commas, and braces.

A. O(N^3)

B. O(N)

C. between 5 and 15 acceptable. Actual number is logN (on average where does the min occur? What did it replace? Where did that occur? So on average how may "new mins" do we find if the data is in random order? What is $\log_2 1000$ in CS314? Turns out the actual base is e, not 2.

D. O(N^2)

E. O(NlogN) // base 2 okay

F. 6 seconds // code is O(N)

G. 45 seconds // code is now O(N^2)

H. Syntax error or Compiler error (can't add non Strings to the list)

I. [CS, M, GOV, ECE, ECE] // any quotes = -1

J. 54 seconds

K. 6,000

L. 12 seconds

M. {A=3, B=1, C=2, D=0, I=3} No brackets okay, order matters

N. 1. invalid (KidViewable an interface) 2. invalid (Live not a Descendant of Musical) (.5 points each)

O. 1. valid 2. valid (.5 points each)

P. swan15

Q. ben10 60

R. bnb12 6

S. true false

T. 6 (list not required, but legal types are Cartoon, Animated, Recoreded, KidViewable, PerformArt, Object)

2. Comments. The simple generic list problem. The real difficulty was in comparing objects next to each other and ensuring the bounds of the loop were correct.

Common problems:

- Index out of bounds errors or checking all but the last element
- Using con.length or size() instead of size
- Assuming that val1 and val2 could not be the same and checking against that
- Only checking for val1, val2 and not the other way around (val1, val2 OR val2, val1)
- off by one problems
- - starting at 0 and going up to size (out of bounds)
- - starting at 1 and going up to size-1 and checking to both sides of val1

```
public boolean nextTo(E val1, E val2) {

     for (int i = 1; i < size; i++) {
          int prev = i - 1;
          if (con[prev].equals(val1) && con[i].equals(val2)) {
               // found val1, val2
               return true;
          }
          else if (con[prev].equals(val2) && con[i].equals(val1)) {
               // found val2, val1
               return true;
          }
     }
     // never found val1 next to val2
     return false;
}
```

16 points , Criteria:

- loop through size elements, not con.length, 4 points
- handle previous or next element correctly, no array index out of bounds exception, 3 points
- check for val1, val2 OR val2, val1, 3 points
- use .equals, not ==, 3 points
- efficiency: return true as soon as pair found next to each other, 3 points

Usage errors:
using disallowed methods

3. Comments: A question involving multiple Generic Lists. This method is in the GenericList class so we have access to private fields of any GenericLists in scope such as other and result. Very similar to the Fall 2016 question, but in a way easier, because we stop at the end of the list with the fewest elements.

Common problems:

- Not adjusting the size instance variable of the resulting GenericList (most common issue)
- Not adding to the correct position in the resulting list (spacing out elements in the resulting GenericList's container). In other words, you needed to keep track of the index to add to the array of the resulting list. The easiest way to do this was to use the resulting list's size variable or re-implement the add method
- Not implementing the add method if calling it

```
public GenericList<E> getMatchingElements(GenericList<E> other) {

      int minSize = size < other.size ? this.size : other.size;

      // create result large enough to hold all pairs with a little extra capacity
      GenericList<E> result = new GenericList<E>(minSize + 10);

      for (int i = 0; i < minSize; i++) {
            E o1 = this.con[i];
            E o2 = other.con[i];
            if (o1.equals(o2)) {
                  // matching so add to result
                  result.con[result.size] = o1;
                  result.size++;
            }
      }
      return result;
}
```

16 points, Criteria:

- create result with enough space (or implement resize method) , 1 point
- determine min size, 3 points
- loop through minSize, 3 points
- check if elements match using .equals, 3 points
- add one instance of element to resulting GenericList at **correct** spot, 3 points
- update size of resulting GenericList correctly, 3 points

Other deductions:

- using add method without implementing (-6), don't lose for size
- O(N^2) -4
- accessing list variable as if it is an array, other[index] instead of other.container[index], -5
- get() OR size() or Math.min() - 1
- destroys or damages either or both original lists

4. Comments: The efficient solution does NOT create a transpose of the calling object. Instead, check square first. Then each row must match the corresponding column. In other words, row 0 matches column 0, row 1 matches column 1 and so forth. As soon as one mismatch occurs, return false.

Common problems:

- Not checking if square, or check in wrong place
- Not returning as soon as possible
- Creating a transpose matrix

Suggested Solution:

```
public boolean isSymmetric() {
      if (myCells.length != myCells[0].length) {
            return false; // not square
      }

      // each row must match the corresponding column
      for (int r = 0; r < myCells.length; r++) {
            for (int c = 0; c < myCells.length; c++) {
                  if (myCells[r][c] != myCells[c][r]) {
                        return false;
                  }
            }
      }

      // every row matched the corresponding column
      return true;
}
```

16 points, Criteria:

- check not square correctly and return false if not, 3 points
- nested loop to check cells in Matrix, 4 points
- correctly check each element in current row is equal to current column, 4 points
- return false as soon as answer know, 4 points
- return correct answer if true, 1 point

Others:
Creating transpose and verifying equal to calling object. This is $O(N^2)$ and uses N^2 space. Much less efficient than suggested solution: -6

5. Comments: A very difficult problem due to all the processing that had to take place. If I had it to do over again I would not have included the rank requirement.

Common problems:

- Treating the names variable as an array. It's a list.
- Not dealing with differences in case. (Convert to upper or lower case)
- Not checking ranks first. The question stated there were many more names than ranks. Therefore we should ensure the rank requirement is met first before going through all the names
- Only checking the second requirement if the first is met
- Confusion on require rank. Recall lower numbers are 'better' with rankings
- Not stopping when answer is known.
- Hard coding values (such as 200) instead of using the parameters

Suggested Solution:

```
public ArrayList<String> getSubstringNames(int requiredRank, int requiredNames) {
    ArrayList<String> result = new ArrayList<>();
    for (int i = 0; i < names.size(); i++) {
        NameRecord nr = names.get(i);
        String currentName = nr.getName();
        if (hasRequiredRank(nr, requiredRank) // short circuit, doesn't do second part if false
                && isPresentInRequiredNames(currentName, requiredNames)) {
            result.add(nr.getName());
        }
    }
    return result;
}

    private boolean isPresentInRequiredNames(String name, int requiredNames) {
         int count = 0;
        // would be better if we could create list of all names as lower case
        name = name.toLowerCase(); // do once for current name
        int i = 0;
        while (i < names.size() && count < requiredNames) {
            NameRecord nr = names.get(i);
            String otherName = nr.getName().toLowerCase();
            if (otherName.length() > name.length() && otherName.contains(name)) {
                count ++;
            }
            i++;
        }
        return count == requiredNames; // we stopped when == or no more names
    }

    private boolean hasRequiredRank(NameRecord nr, int requiredRank) {
        for (int i = 0; i < nr.numDecadesRanked(); i++) {
            int rank = nr.getRank(i);
            if (rank != 0 && rank <= requiredRank) {
                return true; // found one. stop.
            }
        }
        return false;
    }
```

16 points, Criteria:

- loop through all NameRecords in names, 2 points

- check ranks before number of names current name is a substring, 2 points
- correctly check minimum rank requirement met, 5 points (not handling zeros, -1)
- correctly check name is substring in minimum required number of names, 6 points
    - -2 if don't handle case correctly
- correctly add to result, 1 point

Other:

- treating names as an array, -3
- hard coding values instead of using parameters, - 4
- creating substrings instead of using contains or indexOf, - 3 (very inefficient in terms of time and space)
- adding multiple instances of the same name, - 3(This came up when students **nested** one check inside of another. That is very slow and can result in multiple instances of the same name being added.

6. Comments: Hardest question on the exam. A very interesting abstraction and you had to read the question closely to understand that the value we were setting is guaranteed to initially be a non zero. So we know it is in the array. Added difficulty if changing from a non-zero to zero because now we have to remove that value from the array of SMEntry's

Common problems:

- not updating numberOfNonZeros if we set one to zero
- not shifting correctly if changed to zero.
- N^2 solutions. (Clearly just O(N))
- Not stopping as soon as we find the correct value in the array

Suggested Solution:
```
public void setNonZeroValue(int row, int col, int val) {

      // according to precondition, element specified by row, col MUST
      // be in the array. Don't need to check index < numNonZeros based
      // on precondition
      int index = 0;
      while (nonZeros[index].row != row || nonZeros[index].col != col) {
            index++;
      }

      if (val != 0) {
            // simple
            nonZeros[index].val = val;
      } else {
            // must shift and update instance variables
            numNonZeros--;
            for (int i = index; i < numNonZeros; i++) {
                  nonZeros[i] = nonZeros[i + 1];
            }
            // prevent memory leak
            nonZeros[numNonZeros] = null;
      }
}
```

16 points, Criteria:

- loop to find entry, 3 points
- correctly check if current entry is target entry, 3 points
- efficiency, stopping when we find entry, 3 points

- update val if still non zeror, 1 point — shift if necessary and correctly, 3 points
- update numNonZeros if necessary, 2 points — null out old ref if necessary, 1 point

For questions N - T, consider the following classes. You may detach this sheet from the test.

```
public abstract class PerformArt {

      private String name;

      public PerformArt(String n) { name = n; }

      public abstract int time();

      public String toString() {return name + time(); }

      public String getName() { return name; }
}

public class Live extends PerformArt {

      private int tx;

      public Live(int t, String n) {
            super (n);
            tx = t;
      }

      public int time() { return tx * 2; }
}

public class Recorded extends PerformArt {

      public Recorded(String s) { super(s); }

      public int time() { return 60; }
}

public interface KidViewable {
      public int minAge();
}

public class Dance extends Live {
      public Dance(int t, String n) {  super(t, n); }

      public Dance(String n) { super(30, n); }

      public int time() { return 15; }
}

public class Animated extends Recorded implements KidViewable {

      public Animated(String s) { super(s); }

      public String toString() { return getName(); }
```

```
     public int minAge() { return 10; }
}
public class Musical extends Live implements KidViewable {

     private int acts;

     public Musical(int t, String n, int a) {
          super(t, n);
          acts = a;
     }

     public int time() { return minAge() * 2; }

     public int minAge() { return acts * 2; }
}

public class Cartoon extends Animated {

     public Cartoon(String s) {
          super(s + "EX!");
     }
}
```