CS314 Fall 2019 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.
MCE - Major Conceptual Error. Answer is way off base, question not understood.
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. (Statements in parenthesis not required, only for explanation of answer to students.)
No points off for minor differences in spacing, capitalization, commas, and braces.

A. $4N^2 + 5N + 4$, +/- 1 on each coefficient (no syntax error)

B. O(N)

C. $O(N^2)$ (array creation is O(N))

D. $O(N^2)$

E. $O(N^2)$

F. 44 seconds

G. 16 seconds

H. 1024 or 1000

I. [Y, N, T, P, O] (minor differences in spacing and brackets okay)

J. 7 8

K. (A=-5, M=7, Z=5) (minor differences in spacing and brackets okay)

L. compile error

M. .0004 seconds (Code is $O((\log_2 N)^2)$

N. invalid invalid (both correct or -1)

O. L25

P. L50

Q. L15 0

R. 30 (House constructor calls add is sub class)

S. compile error

T. false

For questions N through T, refer to the following classes. You may detach this page if you wish.

```
public class Living {
      private int sp;

      public Living() { sp = 5; }

      public Living(int s) { sp = s;}

      public int get() { return sp; }

      public void add(int s) { sp += s; }

      public void inc() {  sp += 10; }

      public String toString() { return "L" + get(); }
}

public class House extends Living {
      public House (int x) { add(x); }

      public void add() { inc(); }

      public int space() { return 100; }
}

public class Split extends House {
      private int num;

      public Split() { super(20); }

      public Split(int n) {
            this();
            num = n;
      }

      public void add(int n) { num += n; }

      public int get() { return num; }
}

public class Apt extends Living {
      private int fs;

      public String toString() { return super.toString() + " " + fs; }

      public void set(int f) { fs = f; }

      public void add(int f, int s) {
            fs += f;
            add(s);
      }
}
```

2. Comments. Relatively straight forward list problem. Recall, we have access to all GenericLists' private instance vars. No constructor available with initial capacity.

```
public GenericList<E> scale (int factor, E tgt) {
    // Determine how much space we need. Avoid resizing
    int newSize =  scaleHelp(factor, tgt);
    GenericList<E> result = new GenericList<E>();
    // we know how big to make the con, add a bit of extra capacity
    result.con = (E[]) new Object[newSize + 10];
    for (int i = 0; i < size; i++) {
        if (!con[i].equals(tgt))
            // add factor copies of element to result
            for (int j = 0; j < factor; j++) {
                result.con[result.size] = con[i];
                result.size++;
            }
    }
    return result;
}

private int scaleHelp(int factor, E tgt) {
    int newSize = 0;
    for (int i = 0; i < size; i++)
        if (!con[i].equals(tgt))
            newSize += factor;
    return newSize;
}
```

22 points, Criteria:

- correctly create resulting GenericList, 1 point
- determine space needed 4 points, (-3 if resize multiple times) (factor * size + 10 OK)
- some extra capacity in new container 2 points,
- loop through elements correctly, 3 points (lose if con.length)
- correctly check current element not target element with equals method, 3 points
- correctly add factor references to new array, 3 points
- correctly set size for result, 3 points
- correctly set result con field to array, can do early or late, 2 point
- return result, 1 point

Other deductions:

- just use array length from default constructor, -6
- worse than O(factor * this.size), -5
- using disallowed methods: resize -6, add -6
- array of ints or String or other non-Object type, -4
- result[] instead of result.con[]
- ignores factor, -6
- uses result[] instead of result.con[];

Calling GenricList(int) constructor, -5

3. Comments: A very simple 2d array problem. Note, when checking precon you can assume, per the class comments, that the cells 2d arrays of each MathMatrix is rectangular. No need to check this in precon, but no points off if done.

```
public boolean diagonallyDominates(MathMatrix other) {
    if (!square() || !other.square() || cells.length != other.cells.length) {
        throw new IllegalArgumentException("Matrices not square or size");
    }

    for (int r = 0; r < cells.length; r++) {
        for (int c = 0; c <= r; c++) {
            if (cells[r][c] <= other.cells[r][c]) {
                return false;
            }
        }
    }
    return true;
}

private boolean square() {
    // recall, we know (assume) cells is rectangular
    return cells.length == cells[0].length;
}
```

14 points, Criteria:

- check both matrices square correctly, 1 point
- check dimensions' match, 1 point
- correctly throw exception if precon not met, no String necessary, 1 point
- loop through all rows, 1 point
- correctly loop through columns in row up to and including the main diagonal, 3 points
- correctly access cells from this and other and check this cell > other, 3 points
- return false at first cell in other that is <= this cell, 3 points
- return true correctly, 1 point

Others:

- using methods not written, -3
- making public method that exposes 2d array of MathMatrix, -3
- off by one errors, -2s
- loop all cells and have if in loop to see if we check, -3 (efficiency)

4. Comments: Not too hard of a question. Biggest issues were not stopping check on number of unranked decades when we reach the limit. At that point we know we want to remove, so stop checking. And lots of logic errors in removing from array list and then incrementing loop control variable. Recall removing shifts elements down and this may cause us to skip over a NameRecord that should be removed.

```
public int remove(int numUnranked) {
    int oldSize = records.size();
    // Go backwards through the list so removing doesn't
    // causes a logic error.
    for (int i = records.size() - 1; i >= 0; i--) {
        NameRecord r = records.get(i);
        if (unrankedEnough(r, numUnranked)) {
            records.remove(i);
        }
    }
    return oldSize - records.size();
}

// return true if the given NameRecord
// is unranked in limit or more decades
private static boolean unrankedEnough(NameRecord r, int limit) {
    int count = 0;
    for (int i = 0; i < NUM_DECADES; i++) {
        if (r.getRank(i) == 0) {
            count++;
            if (count == limit) {
                return true;
            }
        }
    }
    return false;
}
```

22 points, Criteria:

- loop through array list correctly, 2 points
- correctly avoid the logic error when removing an element and elements after shifted. Can do this by going backwards, or not incrementing loop control var in a while loop, or even (GACK) decrementing loop control var in a loop, 5 points
- access NameRecords from ArrayList correctly with get method, 2 points
- check NameRecord for number of unranked decades
    - correct loop bounds, 1 point
    - correctly call getRanks, 1 point
    - test if rank is 0, 1 point
    - cumulative sum variable, 1 point
    - return as soon as we know we need to remove, 3 points
- remove if condition met, 3 points
- return number removed correctly, 3 points

Other:

- using iterator, -7 not allowed per question
- create new ArrayList, disallowed per question, -6
- ConcurrentModification error due to for - each loop and removing, -6
- listVar[] instead of proper get, -5

OBOE - 2
create new ArrayList -6
infinite loop, -6
remove object not pos method -4

5. Comments: A tough question. First, the abstraction of the sparse list had to be understood. Then we had to find the non default elements that were not to be shifted and their positions in the list had to be updated. Then elements had to be shifted and unused spots nulled out. Finally, it was simple code, but the number of elements stored and the size of the list had to be updated.

Note, I saw some very good, clean single loop solutions that used an if stamentment to decide what to do with an element as in null it out OR shift it and update it's position and then null out the old spot.

```
 public void removeFirstN(int n) {
      // Find the first non-default element
      // with a position >= n.
      int index = 0;
      while (index < elementsStored && values[index].getPosition() < n) {
                  index++;
      }
      // Shift elements that remain to front and
      // update position in list.
      for (int i = index; i < elementsStored; i++) {
          ListElem<E> cur = values[i];
          int oldPos = cur.getPosition();
          cur.setPosition(oldPos - n);
          values[i - index] = cur;
      }
      int oldElementsStored = elementsStored;
      elementsStored -= index;
      // null out unused references
      for (int i = elementsStored; i < oldElementsStored; i++) {
          values[i] = null;
      }
      sizeOfList -= n;
}
```

22 points, Criteria:

- find first non-default element that is going to remain, 4 points
- shift non-default elements that will remain to front of list, 4 points
- update the position in list of the non-default elements that will remain, 4 points
- null out old spots in array, 4 points
- update number of elements explicitly stored correctly, 3 points
- update size of list correctly, 3 points

Other:

- Create new array, disallowed per question, -8
- Worse than O(N) where N is the number of non-default elements in list, -6
- loop on size of list or n (number of elements to remove), up to -6 if caused an inefficient solution and / or a possible NPE. Recall, number of elements to remove and / or size of list may be much, much larger than the number of non-default elements we are storing. Good chance we lose our efficiency gains if we loop based on either of those values.
- NPE possible, -4
- shifting elements more than 1 time, -3 (efficiency)
- nested loop for n or pos or elements stored, -6