CS314 Fall 2018 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.
No points off for minor differences in spacing, capitalization, commas, and braces.

A. $4N^3 + 6N + 4$, +/- 1 on each coefficient

B. 24 seconds (method is $O(N^3)$)

C. $O(N^2)$

D. O(NlogN) (base 2 okay)

E. 80 seconds

F. 25 seconds

G. 1.76 seconds or 44 / 25 seconds

H. Option B. Option A uses more space than is necessary because every GenericList object has its own copy of DEFAULT_CAP when we only need one copy, the static (class) constant. (Or words to that effect)

I. 18 seconds (method is $O(N^2)$ if the lists are the same size.

J. {2=IT, 3=BB, 5=Q} (quotes -1, no brackets okay)

K. 40 3 40

L. imp 5

M. COMPILE ERROR (Can't invoke refill method on variable with declared (static) type WritingImplement)

N. red 100

O. 30

2. Comments. A relatively simple array based list question. Not significant issues. Some students didn't null out the new empty spots. Even though the list doesn't contain nulls, elements in the array that are not currenlty part of the list can and should be null to prevent memory leaks.

```
public int removeFromEnd(E tgt) {
    int index = size - 1;
    int numRemoved = 0;
    while (index >= 0 && con[index].equals(tgt)) {
          // remove element at index
          con[index] = null;
          numRemoved++;
          index--;
     }
     size -= numRemoved;
     return numRemoved;
}
```

14 points, Criteria:

- start from back of list, 2 points
- start at size, not con.length, 1 point
- stop as first non-equal value, 2 points
- correct loop bounds (for or while),3 points (lose if AIOBE possible, not checking >= 0)
- null out removed elements, 2 points
- check equals correctly, 1 points
- alter size correctly, 2 points
- track and return number of elements removed, 1 point

Other deductions:

Error if list empty, always checking 'last' element, -3
nulling out elements not at end of list, -4
null out elements all the way to con.length in call cases, -2
doesn't check index in bounds first, -2 e.g. while (!con[index].equals(tgt) && index >= 0) causes NPE in some cases
Not O(1) space, -4
Not O(N) time, -5
Disallowed methods:

3. Comments: Biggest issue was not checking all pairs of rows. For example if row 0 is 6 4 2 and row 1 is 3 2 1 then row 1 is not an integer multiple of row 0, BUT row 0 is an integer multiple (2) of row 1 and the method shall return true. Other common problems: not stopping the check of a row as soon as one pair of elements does not meet the multiple of the first pair of elements, not accounting for integer division. 5 / 2 = 2, but 2 * 2 != 5. 5 is not an integer multiple of 2. Not checkign all pairs have the same multiple. Not returning true as soon as a pair of rows is found. Returning false on the first mismatch and not checking other rows.

```
public boolean hasIntegerMutlipleRow() {

	for (int r1 = 0; r1 < cells.length; r1++) {
		for (int r2 = r1 + 1; r2 < cells.length; r2++) {
			if (isMultiple(cells[r1], cells[r2]) ||
					isMultiple(cells[r2], cells[r1]) {
				return true;
			}
		}
	}
	return false; // never found a match
}

// Return true if all elements in r2 are integer multiples of
// corresponding elements in r1 and it is the same multiple.
private static boolean isMultiple(int[] r1, int[] r2) {
	int magicMultiple = r2[0] / r1[0];
	if (magicMultiple == 0)
		return false; // 2 / 5 == 0
	for (int i = 0; i < r1.length; i++) {
		if (magicMultiple * r1[i] != r2[i]) {
			// not the right multiple or not an integer multiple
			return false;
		}
	}
	// Never returned false. We're good!
	return true;
}
```

17 points, Criteria:

- check all rows pairwise AND not comparing a row to itself, 6 points (partial credit possible, -3 if compare rows to themselves, -3 if don't check all pairs, -5 if no third loop, only checking adjacent rows)
- When checking two rows:
  - correctly checking corresponding values have the same multiple, 4 points (partial credit possible)
  - stop checking elements in current rows as soon as two elements don't meet magic multiple (or have remainder), 2 points
- return true as soon as a pair of rows found that meet criteria, 2 points
- correctly access rows and elements in arrays, 2 points
- return false if all rows checked and criteria not met, 1 point

Others:

- worse than O(1) space, -4
- alter matrix, -6
- int div issue 4 / 2 = 2, 5 / 2 = 2, 2 * 2 = 4, 2 * 2 != 5, -2
- too early of a return, commonly on first mismatch return false for method, -5

4. Comments: The exam did not state the keys in the Maps were the names, but I thought that would be obvious. We took off for going through the map as opposed to the array list of names. Many students did this, so the score adjustments will help even this out. Other than that students did well on the question.

```
public ArrayList<String> getSteadyNames(ArrayList<String> names, int limit) {
      ArrayList<String> result = new ArrayList<String>();
      for (String name : names) {
            NameRecord nr = myRecs.get(name);
            if (nr != null) {
                  // name is in the Names object, is it steady?
                  boolean steady = true;
                  int prev = fixRank(nr.getRank(0));
                  int i = 1;
                  while (steady && i < numDecade) {
                        // get difference of current and previous rank
                        int current = fixRank(nr.get(i));
                        int diff = Math.abs(current - prev);
                        steady = diff <= limit;
                        prev = current;
                        i++;
                  }
                  if (steady) {
                        result.add(name);
                  }
            }
      }
      return result;
}

private static int fixRank(int rank) {  return (rank == 0) ? 1001 : rank;  }
```

17 points, Criteria:

- loop through ArrayList, not Map elements, 3 points
- correctly check if current name is present in Map via the get method, 3 points
- loop through ranks correctly, 1 point
- correctly check difference between adjacent ranks, 2 points (doesn't include converting 0 to 1001)
- stop comparing adjacent ranks as soon as limit exceeded, 2 points
- correctly call methods on NameRecord objects, 1 point
- change 0 ranks to 1001, 2 points
- correctly add name to result only if in steady name, 2 points
- return result, 1 point

Other:

- altering Map of NameRecords -5
- altering ArrayList names, -5
- adds all steady names in Map as opposed to the steady names in the Map that are also in the ArrayList<String> parameter called names.
- general logic errors, -3

5. Comments: Not an efficient implementation of a map, but a very interesting question. Students tended to do well. One common problem was resizing before it is necessary. If the key is already present and we are simply altering the associated value, there is no need to resize now, even if the container is full. (What if we remove next, which would free up space. Put off the resize until absolutely necessary.) Some students flipped the indices on the 2d array. Recall, by convention, the first index is the row and the second is the column. Some students didn't return the old value if the key was initially present or null if the key was not initially present.

```
public Object put(Object key, Object value) {
     for (int i = 0; i < size; i++) {
          if (kvPairs[0][i].equals(key)) {
               // key already present, replace current value.
               Object old = kvPairs[1][i];
               kvPairs[1][i] = value;
               return old; // And we're done here.
          }
     }

     // Never found key, this is a new key-value pair.
     // Do we need more capacity?
     if (size == kvPairs[0].length) {
          resizeArray(size + 10);
     }
     kvPairs[0][size] = key;
     kvPairs[1][size] = value;
     size++;
     return null; // There wasn't an old value.
}
```

17 points, Criteria:

- loop through current pairs correctly, up to size, not length.con, 2 points
- correctly check if key is already present using equals and correct row 0, 2 points
- if key found, replace old value with new value and return old value, 3 points
- return as soon as key found, 1 point
- if key not found, then check for resize and resize if necessary, 2 point (lose if resize before knowing if new pair)
- resize with some extra capacity, +2 spots or more (fixed or percentage), 1 point
- add new key-value pair in correct spot in array, 2 points
- increment size if new pair, 2 points
- return null if new key-value pair, 2 points

Other:

- Worse than O(N), -5
- AIOBE, -4
- too early of a return, -4
- compare keys to values, -4
- flipped indices [col][row] instead of [row][col], -3
- NPE possible, -3
- not returning anything, -3