CS314 Spring 2013 Midterm 2 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.)
GCE - Gross Conceptual Error. Did not answer the question asked or showed fundamental misunderstanding
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 bad 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. 9

B. 36

C. O(logN), base 4 okay

D. 30

E. -2 5 2 -2

F. Syntax error (Map is an interface)

G. 336

H. O(N)

I. O(N^2)

J. 21 seconds

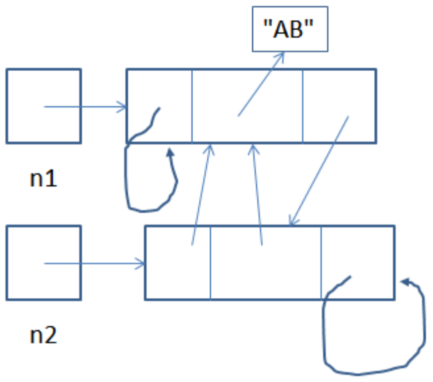
K. 32 seconds

L. 22

M. 1 1 2 2 -1

N. 11 9 7

O. see drawing:



P. w/o sort work = 10,000 * 500 = 5,000,000 ops (10,000,000 okay)

sort then search = 1000 * $\log_2$ 1000 + 10,000 * $\log_2$ 1000

= 10,000 + 100,000 = 110,000

110,000 << 5,000,000 so sort then search

Q. 45 seconds

R. 1.5 seconds

S. O(N^2)

T. O(N)

2. Comments. This question was all about dealing with various abstractions and data structures and types. Students did fairly well on the question.

Common problems:

- errors on bounds not checking that the last n decades were unranked
- using 0 or the constant listed instead of UNRANKED
- assuming NameRecord is Iterable (it isn't)
- assuming keys in map were NameRecords (they are Strings)

Suggested Solution:

```
public int countUnrankedNames(int n) {
     int count = 0;\
     final int START_DECADE = NUM_DECADES - n;
     for(String name : data.keySet()) {
          NameRecord current = data.get(name);
          int decadeIndex = START_DECADE;
          int allUnranked = true;
          while(decadeIndex < NUM_DECADES && allUnranked) {
               int rank = current.getRank(decadeIndex);
               allUnranked = rank == NameRecord.UNRANKED;
               decadeIndex++;
          }
          if(allUnranked)
               count++;
     }
     return count;
}
```

Alternate solution that is a little less efficient:

```
public int countUnrankedNames(int n) {
     int count = 0;
     final int START_DECADE = NUM_DECADES - n;
     for(String name : data.keySet()) {
          NameRecord current = data.get(name);
          int timesUnranked = 0;
          for(int i = START_DECADE; i < NUM_DECADES; i++)
               if(current.getRank(i) == NameRecord.UNRANKED)
                    timesUnranked++;
          if(timesUnranked == n)
               count++;
     }
     return count;
}
```

General Grading Criteria: 13 points

- iterator through keyset (for each or iterator), 2 points
- access NameRecord from data correctly with get method, 2 points
- loop through appropriate decades (bounds correct), 2 points
- access rank for decade correctly from NameRecord, 2 points
- check for ranked / unranked decades, 1 point
- use NameRecord.UNRANKED, not 0, 1 point
- correctly determine NameRecord not ranked in last n decades, 2 points
- return correct answer, 1 point

Assume NameRecord Iterable: -3

3. Comments: A fairly easy linked list question that required modification to the list.

Common problems:

- off by one errors, temp.getNext() != null usually leads to off by one errors without elaborate checks and results in a NPE for empty lists (first == null)
- destroying the list. Using first (instead of a temporary variable) destroys the list and does not successfully insert anything
- not moving the temporary node reference successfully
- using == instead of .equals
- stopping condition of temp.getData() == null. (The data has nothing to do with the structure of the list)
- using iterator or other disallowed classes

Suggested Solution:

```
public int insertAfter(E tgt, E insertVal) {
    int count = 0;
    Node<E> temp = first;
    while(temp != null) {
        if(tgt.equals(temp.getData())) {
            Node<E> newNode = new Node<E>(insertVal, temp.getNext());
            temp.setNext(newNode);
            temp = newNode; // skip past newNode (not strictly necessary)
            count++;
        }
        temp = temp.getNext();
    }
    return count;
}
```

General Grading Criteria: 12 points

- track number of nodes added and return correct number: 1 point
- temporary node declared: 1 point
- loop with correct stopping condition (or handle special cases): 2 points
- check data in current node using `.equals` tgt: 2 points (-1 if ==)
- if matches create new node with correct data and link: 2 points
- correctly link temp node to new node: 1
- move temp correctly. (okay if skip to when add): 3 points (just temp.getNext() -> -3)

Other common deductions:

- Using methods in LinkedList without writing: -3 per case
- Using arrays or other lists: -6
- Using iterator: -6
- checking temp.getData() != null instead of temp == null: -3
- Off By One Error / Null Pointer Exception on empty case: -1

  temp.getNext() != null is wrong unless handle special cases with ifs
- Destroy list (using first instead of a temp node): -6

4. Comments: A puzzle question because you were only allowed to use another Queue as your temporary container. Somewhat challenging because depending on how you studied, you may have never written code involving a queue before.

Common problems:

- taking no steps to preserve the queue
- forgetting that the parameter q is a copy of a reference, so the statement q = temp; has no effect on the original parameter.
- not returning -1 in the case where the target did not appear in the queue
- using disallowed classes
- infinite loop caused by only dequeue when front is equal to target (never get passed to other elements)

Suggested Solution:

```
public double getAvePosition(Queue<Object> q, Object target) {
      double positionSum = 0.0;
      int numPresent = 0;
      int position = 0;
      Queue<Object> tempQ = new Queue<Object>();
      while(!q.isEmpty()) {
            Object tempObj = q.dequeue();
            if(target.equals(tempObj)) {
                  numPresent++;
                  positionSum += position;
            }
            position++;
            tempQ.enqueue(tempObj);
      }
      // restore queue
      while(!temp.isEmpty())
            q.enqueue(temp.dequeue());
      return (numPresent == 0) ? -1 : positionSum / numPresent;
}
```

General Grading Criteria: 13 points

- track sum of positions: 1 point
- track number present: 1 point
- track index / position: 2 point
- loop through q until empty: 2 point (size() or iterator - 2)
- check if current element equals target and update values if so: 2 points (-1 ==)
- remove from original queue and store in temp queue: 1 point
- restore original queue: 2 points (q = temp -> -2)
- return -1 if none present: 1 point
- return average if 1 or more present: 1 point

Other common deductions:

- no attempt to store and restore original queue via temporary queue, -4
- infinite loop (only dequeue when equal) - 4
- disallowed classes / especially size -3
- infinite loop -4

5. Comments: I thought this was an easy problem. It is, in my mind, very much like the dice problem from the quiz. The 10 am lecture had a question during the review and I went over how to answer the dice question. (number of ways to roll a given value with n 6 sided dice). The correct solutions were much shorter in general than the incorrect solutions. (Usually it is the other way around.)

Common problems:

- by far tossing he biggest problem was looping through ALL the cards in the recursive case. If the base cases are correct this leads to counting far too many combinations of cards. (permutations of the combinations that add up to 15). The choices were not the cards themselves, but the values for the cards, 0 * value, value, 1 * value, 2 * value. (This was stated in the problem description)
- returning early without trying other combinations
- missing base cases
- not making the choice to NOT use a card

Suggested Solution:

```
private int helper(int[] data, int index, int tgt) {
    if(tgt == 0)
           // found a solution!
           return 1;
    else if(index == data.length)
           // no more cards to consider, assert tgt > 0
           return 0;
    else {
         // recursive case, choices are values times multipliers
         // multiplier of 0 means we are choosing to not use this card
        int result = 0;
        for(int multiplier = 0; multiplier <= 3; multiplier++) {
            int newTarget = tgt - data[index] * multiplier;
            if(newTarget >= 0)
                 // can't make a negative tgt, could handle 0 here
                 result += helper(data, index + 1, newTarget);
        }
        return result;
    }
}
```

General Grading criteria: 12 points

- base case success: 2 points
- base case failure, no more cards: 2 points
- recursive case, try all choices of multiples for a single card's value, including zero (not use card), 3
- making correct recursive call with index + 1 and new target value, 2
- correctly summing result, 2 (many answers were wrong due to misunderstanding of local variables
- return correct result in recursive case, 1 point

Other common deductions:

- too many permutations, looping through the cards instead of (or in addition to) the multipliers, -4
- returning early -6
- incorrect base cases leading to AIOBE, -4
- infinite recursion, stack overflow, -5