CS307 Spring 2010 Midterm 2 Solution and Grading Criteria.

Grading acronyms:
ABA - Answer by Accident
AIOBE - Array Index out of Bounds Exception may occur
BOD - Benefit of the Doubt. Not certain code works, but, can't prove otherwise
ECF - Error carried forward.
Gacky or Gack - Code very hard to understand even though it works or solution is not elegant. (Generally no points off for this.)
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.

1. Answer as shown or -2 unless question allows partial credit.
No points off for differences in spacing and capitalization. If quotes included, okay. On big O okay if missing O( )

A. 14

B. A stack overflow error will occur. (Or a runtime error or exception or words to that effect.) OR an infinite loop occurs.

C. BBA

D. 36

E. O(N^2)

F. O(N)

G. O(N^2)

H. O(N^2)

I. O(N)

J.O(N^3)

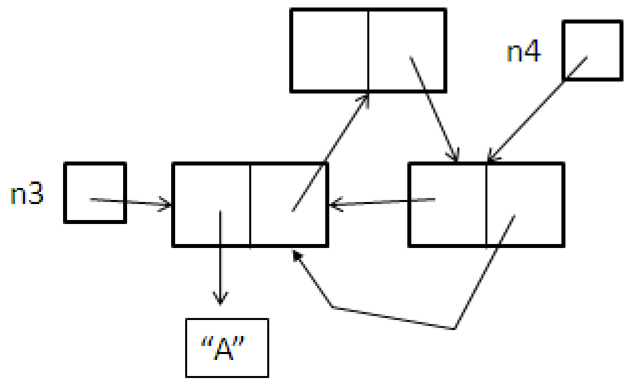
K. 40 seconds

L. 54 seconds

M. 2000

N. 2

O. Picture as shown below:



2. Comments. This question evaluated the ability to use iterators. Students did fairly well on the question. Probably the easiest one of the three coding questions. Since it is a set once a match is found for a particular element the search for that item can stop, but points were not taken off if the search kept going.

Common problems:

- not creating iterators correctly
- not using iterators correctly.
- using methods not allowed such as contains or the ISet remove with writing them yourself.
- treating sets like arrays
- not obtaining the inner loop iterator each time as necessary.

Suggested Solution:

```
public void removeAll(ISet<E> other) {
      Iterator<E> otherIt = other.iterator();
      while(otherIt.hasNext()) {
            E temp = otherIt.next();
            boolean found = false;
            Iterator<E> thisIt = this.iterator();
            while(!found && thisIt.hasNext()) {
                  found = temp.equals(thisIt.next());
            }
            if(found)
                  thisIt.remove();
      }
}
```

This can be cleaned up to use a for each loop:

```
public void removeAll(ISet<E> other) {
      for(E elementFromOther : other)
            boolean found = false;
            Iterator<E> thisIt = this.iterator();
            while(!found && thisIt.hasNext()) {
                  found = elementFromOther.equals(thisIt.next());
            }
            if(found)
                  thisIt.remove();
      }
}
```

General Grading Criteria: 25 points

obtain Iterator for this set: 2
loop through elements with iterator: attempt 2, correct: 4
temp var for current object from iterator if necessary: 3
obtain Iterator for other set: 2
loop through other: 3
check equality of items. 3 (-2 if == instead of .equals)
if item found removed via iterator. attempt: 2, correct: 4

3. Comments: A good linked list question. There were some special cases to be dealt with and some intricacies in dealing with data from two nodes. But, it was not necessary to alter the list, just move through it. There were many different, correct solutions.

Common problems:

- not handling the case when the list is empty correctly
- not moving through the list
- destroying the list in the process
- comparing nodes instead of the data in the node
- not using the compareTo method or not using it correctly
- trying to use an iterator
- not dealing with the previous node correctly
- off by one errors on the last node

```
public boolean isStrictlyDecreasing() {
      boolean good = true;
      // only have to check if more than 1 element
      if(first != null && first.getNext() != null) {
            E previousData = first.getData();
            Node<E> temp = first.getNext();
            while(good && temp != null) {
                  good = temp.getData().compareTo(previousData) < 0;
                  previousData = temp.getData();
                  temp = temp.getNext();
            }
      }
      return good;
}
```

Suggested Solution

General Grading Criteria: 25 points

handling case when list empty: 2
handling case with 1 element 1
use look ahead, trailer, or temp data variable as necessary: 3
loop through list: attempt: 2, correct: 2
compare data in list correctly: 4 points
move through list: attempt 2, correct 8
return value: 1

4. Comments. A classic recursive backtracking problem. The golf background was just an abstraction. The question could have been take a group of values and what is the minimum number of values needed to add up to some target with the ability to reuse each value as many times as necessary. The question could have been phrased as minimum number of coins to make change given an unlimited number of coins.

I thought I did a good job of pointing out the base cases in the examples. 0 distance needs 0 hits, negative distance is not possible.

If not at the base case the choices are what club to use now. There is an alternate solution where we use a club once or not at all. There was a nice alternate solution that stored all possibilities in an ArrayList and then found the min of that list.

Common problems:
- by the far the most common and serious problem was early return. In other words, having a loop, but returning the first value found:

```
for(int i = 0; i < clubs.length; i++)
    return 1 + minHits(distance - clubs[i], clubs);
```

if that value is returned you never try the other clubs FOR THE CURRENT DISTANCE.

- not tracking the current results, comparing them, and finding the min.
- not dealing with impossible cases correctly.
- lots of solutions that were way off base
- not adding one for each hit taken

Suggested Solution

```
public static int minHits(int distance, int[] clubs) {
       // base case, 0 hits required to hit ball 0 distance
       if(distance == 0)
              return 0;
       // if distance is negative can't do it
       else if(distance < 0)
              return -1;
       else{
              int best = distance + 1; // best has to be better than this if possible
              // try all the clubs
              for(int i = 0; i < clubs.length; i++) {
                     int current = 1 + minHits(distance - clubs[i], clubs);
                     if(current != 0 && current < best)
                            best = current;

              }
              if(best == distance + 1)
                     // never found an answer!
                     best = -1;
              return best;
       }
}
```

alternate solution with no loop:

```
public static int helper(int distance, int[] clubs, int index) {
       if(distance == 0)
              return 0;
       else if(distance < 0 || index == clubs.length)
              return -1;
       else {
              // try using current club once
              int hitsWith = helper(distance - clubs[index], clubs, index);
              // was it possible???
              if(hitsWith != - 1)
                     hitsWith++;
              // don't use current club
              int hitsWithout = helper(distance, clubs, index + 1);
              // if neither way possible report it no solution
              if(hitsWith == -1 && hitsWithout == -1)
                     return -1;
              // if only one way possible use it
              else if(hitsWith == -1 || hitsWithout == -1)
                     return Math.max(hitsWith, hitsWithout);
              // if both ways possible, pick min
              else
                     return Math.min(hitsWith, hitsWithout);
       }
}
```

General Grading criteria: 20 points

base case, distance 0: 2 points
base case, distance < 0: 2 points,
track best possible: 2 points,
loop through clubs (no early return): 4 points
try current choice (club): 2 points
recursive call with new distance: 4 points

store result and compare to best possible: 2 points
return -1 if not possible: 3 points
return for recursive case: 2 points

early return: -8