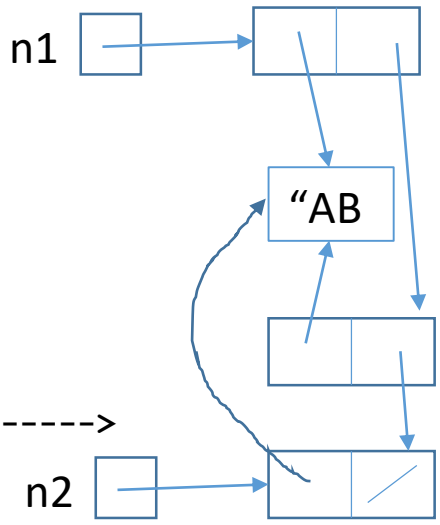
CS314 Fall 2018 Exam 2 Solution and Grading Criteria.

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.)
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. Runtime Error (due to stack overflow error)



B. 9

C. 329

D. 22

E. 20 seconds (It's O(N))

F. Unknown OR varies OR words to that effect

G. 40 seconds (It's O(N^3))

H. 84 seconds (It's O($N^2 \log_2 N$))

I. ---------->

J. -7

K. There is not output (Stack already empty)

L. 90 seconds

M. 400 seconds (All values same leads to worst case (N^2) quicksort)

N. 4

O. ---------->

P. CSAKL

Q. SKLAC

R. O(N^2)

S. O(NlogN) base 2 okay

```
      5
     /  \
    6    12
     \
      0
       \
        3
```

T. So clients can send ArrayLists to methods with parameters of type Iterable. (for-each loop only, not enough)

2. Comments. It was necessary to create new nodes otherwise the calling list would be altered. T

```
public LinkedList314<E> getWithoutTarget(E tgt) {
    LinkedList314<E> result = new LinkedList314<>();
    Node<E> tempInThis = first;
    Node<E> tempInResult = null;
    while (tempInThis != null) {
        E val = tempInThis.data;
        if (!val.equals(tgt)) {
            // add to the result
            if (tempInResult == null) {
                // first element
                result.first = new Node<>(val);
                tempInResult = result.first;
            } else {
                // not first element
                tempInResult.next = new Node<>(val);
                tempInResult = tempInResult.next;
            }
        }
        tempInThis = tempInThis.next;
    }
    return result;
}
```

20 points, Criteria:

- create result, 1 point
- temp variables for nodes in this and other, 1 point (do not create new nodes initially)
- loop until temp in this is null, 3 points
- get data correctly from this list, 1 point
- compare data to target correctly, 2 points (must use equals)
- if data in this equals target ignore, 2 points
- if data not equal to target add correctly to result 4 points
- handle special case for first value in result correctly, 3 points
- move temp in this correctly, 3 points

Other deductions:

Not O(N) time -5	assuming size field, -5
NPE likely or always, - 3	not creating new nodes, -7 (or altering this list)
assuming iterator, -7	calling add method without implementing, -9

3. Comments: Even when creating a new queue answers had problems by assuming front would return null if the q was empty, assuming temp = q would restore the original queue (recall, q is a copy of a reference), calling dequeue or front on empty queues, or not checking strictly decreasing correctly.

Another common problem was starting the restore when the answer was found, but there were still elements left in the queue. To restore the original queue had to be entirely emptied out. Many of the answers that didn't create a temporary queue used the front value as the flag, but this can lead to logic errors if there are copies of that value in the middle of the queue.

Finally, some answers only compare pairs of elements so [A, B, C, D, E, F] A to B, C to D, E to F, when it is required to compare A to B, B to C, C to D and so forth.

This question turned out a lot harder than I expected.

```
public static boolean isStrictlyDecreasing(Queue314<Integer> q) {
    q.enqueue(0); // use as a flag
    int prev = q.dequeue();
    boolean descending = true;
    while (prev != 0) {
        int current = q.dequeue();
        if (descending) {
            descending = prev > current;
            // once we set to false, we don't want to check again
        }
        q.enqueue(prev);
        prev = current;
    }
    return descending;
}
```

20 points, Criteria:

- use flag value of some kind to tell when we have gone through all elements, 4 points
- loop until flag seen again, 4 points
- compare current value to previous value store result (not strictly descending) correctly, 5 points
- restore queue correctly, 2 points for attempt, 4 more points for correct
- return result, 1 point

Other deductions:

- Not O(N), -5
- not checking strictly decreasing, current > prev instead of current >= prev
- updating flag boolean each time in loop, so result only depends on last checked, -3
- Any other data structures, -6
- off by one, always calling dequeue or front when queue becomes empty, -2

4. Comments: Not a difficult question, but a lot of code to write. The exam contained a tree question until I wrote out my solution to this question and realized the exam was long enough.

Some answers tried to calculate average GPA as the went through this initial data. While it is possible to do so, many answers didn't handle this correctly. (For example simply averaging the current GPA with the single new value.)

Also some answers set the initial max to 0.0 which leads to a logic error if we have a data set where all GPAs for all classes are 0.0, F. Not likely, but possible.

```
public static String highestGPA(Map<String, Map<String, Double>> m) {
    // I'll store the number of students that took the class
    // in the first element and the total grade points
    // in the second element.
    Map<String, double[]> classes = new HashMap<>();
    for (String student : m.keySet()) {
        Map<String, Double> studentsGrades = m.get(student);
        for (String aClass : studentsGrades.keySet()) {
            double[] grades = classes.get(aClass);
            if (grades == null) {
                // First time we have seen this class.
                grades = new double[] {0.0, 0.0};
                classes.put(aClass, grades);
            }
            grades[0]++;
            grades[1] += studentsGrades.get(aClass);
        }
    }
    // Now find highest average GPA.
    double max = -1.0;
    String result = "";
    for (String aClass : classes.keySet()) {
        double[] data = classes.get(aClass);
        double aveGPA = data[1] / data[0];
        if (aveGPA > max) {
            max = aveGPA;
            result = aClass;
        }
    }
    return result;
}
```

20 points, Criteria:

- create temporary map, 2 points (must be HashMap, efficiency)
- loop through map of students, 2 points
- for each student loop through classes, 1 point
- for each grade student has, correctly update existing data for class, 4 points
- if first time class seen correctly create new array and add to map, 4 points
- loop through temp map correctly, 1 point
- correctly access class data and calculate average GPA for class, 2 points
- correctly compare and track max GPA and class with max, 4 points
- return result, 1 point

Others:

- disallowed classes, -5 incorrect iterator use, -2 new array for every class even if seen before, - 2

5. Comments: The recursive backtracking question. This one follows the classic pattern. The only real intricacy was realizing a double loop was needed to go through the choices, one for the TA's available times and one for the sections.

Suggested Solution:

```
public static boolean canSchedule(String[][] tas, String[][] sections) {
    return help(tas, sections, 0);
}
private static boolean help(String[][] tas, String[][] sections,
                                                        int index) {
    if (index == tas.length) {
        return true;
    } else if (index < tas.length) {
        // recursive case, choices are this TA's times
        int numTimes = tas[index].length;
        int nextIndex = index + 1;
        for (int i = 1; i < numTimes; i++) {
            String currentTime = tas[index][i];
            // see if there is an open time that matches this time
            for (int section = 0; section < sections.length; section++) {
                if (currentTime.equals(sections[section][0])
                        && sections[section][1] == null) {
                    // this time matches and its open, try it
                    sections[section][1] = tas[index][0];
                    if (help(tas, sections, nextIndex)) {
                        return true;
                    }
                    // didn't work out, undo choice
                    sections[section][1] = null;
                }
            }
        }
        return false;
    }
}
```

20 points, Criteria:

- creating correct helper and calling, 1 point
- handling base case correctly, 2 points (many answers didn't guard against the possibility that not all sections were covered, but the TA index was equal to the number of TAs and so trying to do the recursive step could lead to an AIOBE or stack overflow error
- 1 TA handled per method call (as required per the instructions). The single TA is the "one step" for the method, 3 points
- loop though choices (TA's times and sections) correctly, 3 points
- only go forward if a section is open, not just time matches, otherwise it's too brute force. 3 points
- call recursive method correctly and use result to stop if solution found, 4 points
- undo section assignment so we know it's open in the future if no solution found, 2 points
- return false after trying all choices and so solution found, 2 points

Other:

- early return, -7 not using result of call correctly, -7 not using .equals, 2 points
- trying option with not assigning TA to any section. (this was necessary in the fall 2017 question, but here the number of TAs equals the number of sections so it isn't an option to not assign a TA to a section, and therefore a waste of time to try it) - 2