CS314 Fall 2025 Exam 2 Solution and Commentary

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. Lack of Zen.)
LE - Logic Error in code.
MCE - Major Conceptual Error. Answer is way off base, question not understood based on answer provided.
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.
EFF - Efficiency. Order is worse than expected or unnecessary computations done.

1. Answer as shown or -2 unless question allows partial credit.
Text in parenthesis not required. It is simply grading guidance and / or a brief explanation for answer.

A. 22
B. 30
C. -3
D. O(logN)
E. 13
F. {0=3, 2=6, 5=5}
G. 4 seconds (HashMap ops with key are O(1))
H. 18 seconds (O(N^2))
I. 20 seconds (O(N) -> 1 + 2 + 4 + 8 + … + N/2 + N = 2N - 1)
J. B (Insertion sort best case O(N))
K. 22 seconds (O(NlogN), TreeMap key ops O(logN)
L. 3 (just like subtle question from lecture, does not empty the stack)
M. F
N. 19 9
O. 6 seconds (O(N))
P. A (logN to find row, + logN to find value in row. 2logN = O(logN))
Q. A (each half sorted before final merge)
R. 27 seconds (O(3^N))
S. 12
T. C
U. 0

```
         A     depth 0
       /   \
      B     C  depth 1
    /  \
   D    E     depth 2 (no leaves)
  / \  / \
  F  G H  I   depth 3
```

V. 18
W. 2
X. 4
Y. -8

Commentary on short answer questions where the student average was < 80%

C. A bit detailed. I think the best thing to do is draw the stack.

c(2634) -> 4 - c(263)

c(263) -> 3 + c(26)

c(26) -> 6 - c(2)

c(2) -> 2 - c(0)

c(0) -> 0

c(2) -> 2 - 0 = 2

c(26) -> 6 - 2 = 4

c(263) -> 3 + 4 = 7

c(2634) -> 4 - 7 = -3

D. We how many times do we divide N by 10 until we get to 0? $\log_{10}N$ times. -> O(logN)

E. Maps do not allow duplicate keys. We will get every value between 0 and 12 inclusive. Thus 13 keys. If interested here is a sorted list of the result of all values between 1 and 13 and the remainder of 7, 11, and 13.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12,
12, 12, 12, 12, 12]
```

G. Given the key HashMap operations are O(1). Therefore, the code is O(N) where N is the size of the list. As the size of the list doubles we expect the time to double.

I. We keep going over this so I am surprised at how many people missed this question. This dependent loop is the basis of why an array-based data structure such as a list or queue that increases capacity by doubling number of elements when necessary keeps the average case of adding at the end O(1). **That is why this is so important.**

Yes, it looks like an ONlogN) loop, but it is not.

It is a dependent loop. The number of times the inner boolean condition is true is the sum of the sequence:

1 + 2 + 4 + 8 + 16 + 32 + 64 + ... N/8 + N/4 + N/2 + N = 2N - 1

Yes, there are logN + 1 terms in the sequence but the first logN terms only add up to N - 1.

The sum of a finite geometric sequence where the common ratio (r) is a power of 2, such as $r = 2$, can be found using the general formula for the sum of a finite geometric series:

$$S_n = a\frac{(r^n - 1)}{(r - 1)}$$

where:

- S_n is the sum of the first n terms.
- a (or a_1) is the first term of the sequence.
- r is the common ratio.
- n is the number of terms.

Applying the Formula when $r = 2$

When the common ratio $r = 2$, the formula simplifies significantly:

$$S_n = a\frac{(2^n - 1)}{(2 - 1)}$$

$$S_n = a \times (2^n - 1)$$

Example

For the specific sequence of powers of 2 starting from 2^0 (i.e., $1 + 2 + 4 + \ldots + 2^{n-1}$), the first term $a = 1$, and the sum of the first n terms is:

$$S_n = 1 \times (2^n - 1)$$

$$S_n = 2^n - 1$$

For example, the sum of the first 4 terms ($1 + 2 + 4 + 8$) is:

$$S_4 = 2^4 - 1 = 16 - 1 = 15$$

Note, in the above explanation N is the number of terms not the limit. If N is the limit we have logN + 1 terms.

$2^{(\log_2 N + 1)} = 2N$

a = 1, r = 2, n = logN + 1-> Sum = 1 x ($2^{(\log 2N) + 1))}$ - 1) = 1 x (2N - 1) = 2N - 1

K. TreeMap operations are O(logN) where N is the number of keys in the Tree. So, the code is array.length * (log_2Number of keys + 0.5* log_2Number of keys) -> O(array.length * log_2Number of keys .

(1,000,000 * $log_2$1,000,000) / (2,000,000 * $log_2$4,000,000) = 10 sec / ?

(1 * 20) / (2 * log_{2}(1,000,000 * 4)) = 10 sec / ?

20 / (2 * (20 + 2) = 10 sec / ?

algebra

1 / 22 = 1 sec / ?

algebra

? = 22 sec

L. Very, very much like the subtle question we did as an Instapoll when covering stacks. Subtle yes, but we did an example in lecture.

After first for loop. top of stack -> [0, 1, 2, 3, 4, 5] <- bottom of stack

i	stack size	stack after pop	sum
0	6	top of stack -> [1, 2, 3, 4, 5]	0
1	5	top of stack -> [2, 3, 4, 5]	1
2	4	top of stack -> [3, 4, 5]	3
3	3	3 < 3 is false, loop terminates	4

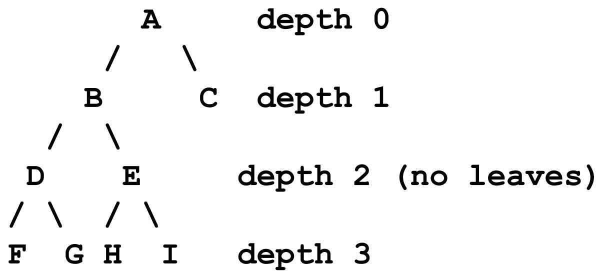
M. All of those data structure may be used. LL314, singly linked, treat the front of list as top of stack. java.util.LinkedList either end works. java.util.ArrayList. end of list is top of stack. Adding is average case O(1) because capacity goes up by a fixed percentage when resizing. (See 1.I). Native array, treat first element as botton, keep track of top, just like adding to end of array-based list. Increase capacity by a fixed percentage when resizing.

P. Use binary search on the first element of each row to find the row the element would have to be in if present. Then do binary search on that row. logN + logN = 2logN -> O(logN)

S. Note the check at the top. The base case is !it.hasNext(). In that case the code does NOT call next and we simply return 0. In the recursive case the two recursive calls will not result in a runtime error (NoSuchElementException) as the base case guards against this. Just a very interesting way to add up all the elements using an iterator.

T. Both are average case NlogN and given sorted data both are still NlogN. Quicksort on distinct (no repeats) data that is already sorted, picking the middle element leads to the best case.

U. Recall a full binary tree is one in which all internal nodes (nodes with children) have 2 children. No internal nodes with a single child. Maybe tough to imagine, but it is possible to have a full binary tree with no leaves at a depth of 2. (2 links from the root to the nodes.)



V. I hope you didn't draw this out. Complete tree means nodes filled in left to right, top to bottom. Consider each level and you can derive the answer with a little bit of math.

root 1 node, 2 children

level 1, 2 nodes both have 2 children (3 nodes total, 3 nodes with 2 children)

level 2, 4 nodes all have 2 children (7 nodes total, 7 nodes with 2 children)

level 3, 8 nodes all have 2 children (15 nodes total, 15 nodes with 2 children) The next level would have 16 nodes, so 31 nodes total so we are getting close to our limit.

level 4, 16 nodes. 31 nodes total. Only 6 more nodes to go on the next level so 3 of these 16 nodes must have 2 children. (31 nodes total, 18 nodes with 2 children)

level 5, 6 nodes, all are leaves as the level is not filled in (37 nodes total, 18 nodes with 2 children)

2. Comments: A very straight forward map question. average of 88%.

Biggest issues:

1. Repeatedly calling get. Calling get dos not create a new list. It gives you a reference to the internal list. We did this on the Evil Hangman assignment when splitting up words into word families. Repeatedly calling get hurts our efficiency if a TreeMap as each one is O(logN). Better to just do once.
2. Like most vowels from A1 or fair teams from A6, just track the max so far. (Granted that was the min so far and recursive.) I expect students who have completed a CS1 course to know how to find a min or max of a data structure without creating any new data structures by simply tracking the best so far. Some students created an unnecessary frequency map.
3. Trying to compare all keys pairwise. Not that common of an issue, but I don't understand why some folks thought this was the way to solve the problem.

It was correct to assume that if a team played a ranked team twice that counted as two ranked opponents.

```
    public static String playedMostRanked(
                    Map<String, ArrayList<String>> teams,
                    Set<String> ranked) {
        String bestTeam = "";
        int maxRanked = -1;
        for (String team : teams.keySet()) {
            // How ranked teams did this team play?
            int count = 0;
            for (String opponent : teams.get(team)) {
                if (ranked.contains(opponent)) {
                    count++;
                }
            }
            // Is this the best team so far?
            if (count > maxRanked) {
                maxRanked = count;
                bestTeam = team;
            }
        }
        return bestTeam;
    }
}
```

3. Comments: A good linked list question. To the surprise of no one I gave you a stripped-down linked list. Singly linked nodes, reference to the first node and nothing else. Evaluating the ability to move through a linked list and possibly add a node. I am really proud of this question. Just the right level of difficulty with a couple of nuances. 72% average on this question.

Various issues and suggestions.

1. I think it was easiest to deal with an empty list as a special case. Saw far too many solutions that checked if first was null and if so then said first.data = tgt. But first is null. That was just checked. null.data = tgt causes a null pointer exception. If adding a new node when must create that node. (And no ,a single node is not considered a data structure.) 38% of solutions did not handle this correctly
2. If not empty we needed to traverse the list checking how many times target occurred. If target occurred the max allowed frequency or more times we are not adding. So as soon as we know that stop computing and return false. This could vastly improve the best-case T(N).) 70% of solutions did not handle this correctly
3. If the frequency of the target is less than the max allowed we want to end up on the last node of the structure if it isn't empty. This can be achieved by using two Node references, one that trails behind the lead refence OR using look ahead. Both solutions are presented below. I was very glad a student asked about this technique during the review session. I believe we mentioned in in lecture when talking about making the singly linked list iterator remove method O(1), but it was nice to go over the specifics in the review session. 18% of solutions did not handle this correctly.
4. Other common problems were not using the equals method on data correctly and not adding a node correctly in the non-empty case.

```
// trailer solution
public boolean addIfFrequencyLessThan(E tgt, int freq) {
        // special case, empty, simple
        if (first == null) {
            first = new Node<>(tgt);
            return true;
        }
        // general case, traverse to count number present
        // Need to hang on to the last node, so look ahead
        // or trailer.
        Node<E> scout = first;
        Node<E> trailer = first;
        int count = 0;
        while (scout != null && count < freq) {
            if (scout.data.equals(tgt)) {
                count++;
            }
            trailer = scout;
            scout = scout.next;
        }
        if (count < freq) {
            trailer.next = new Node<>(tgt);
        }
        return count < freq;
    }
```

```
//=* trailer with return in loop. (most common correct way of stopping
when answer known. */

public boolean addIfFrequencyLessThan(E tgt, int freq) {
        // special case, empty, simple
        if (first == null) {
            first = new Node<>(tgt);
            return true;
        }
        // general case, traverse to count number present
        // Need to hang on to the last node, so look ahead
        // or trailer.
        Node<E> scout = first;
        Node<E> trailer = first;
        int count = 0;
        while (scout != null) {
            if (scout.data.equals(tgt)) {
                count++;
                if (count == freq) { // will never be > because returning
                    return false;
                }
            }
            trailer = scout;
            scout = scout.next;
        }
        trailer.next = new Node<>(tgt);
        return true;
    }
```

```
// Look ahead solution
public boolean addIfFrequencyLessThan(E tgt, int freq) {
        // special case, empty, simple
        if (first == null) {
            first = new Node<>(tgt);
            return true;
        }
        // general case, traverse to count number present
        // Need to hang on to the last node, so look ahead
        // or trailer.
        Node<E> temp = first;
        int count = temp.data.equals(tgt) ? 1 : 0; // checks first node
        while (temp.next != null && count < freq) {
            temp = tmep.next;
            if (scout.data.equals(tgt)) {
                count++;
            }

        }
        if (count < freq) {
            temp.next = new Node<>(tgt);
        }
        return count < freq;
    }
```

4. Comments: I was disappointed by the results on this question. The average was 59%. There were many solutions that had lots of code but had many, many mistakes and received no credit.

I though this problem matched up fairly well with the basic recursive backtracking algorithm but with a couple of twists similar to what we saw on assignments 6 and 7. One TA even commented that this was very, very similar to anagrams except we stop when we find one anagram. I don't disagree.

Common Issues:

1. The base case. The simplest base case is the size of the loop is equal to the goal or the goal is 0 if decrement. This is not a trivial base case though. (Details) Unless special cases handled in the choice of words loop for the recursive case we need to check if the last letter in the last word matches the first letter in the first word. And simply return if that is true or not. Many solutions just returned true with out checking. Others returned true if the condition was met. If not, we should return false, but these solutions would end up looping through the words again. Another order of magnitude of unnecessary work. Sometimes this was caught on the next recursive call with size of loop > goal, but still significant unnecessary work.
2. In the recursive case the choices are the words in the dictionary. We want to try them all.
3. The first word is a special case. Many solutions did not handle this correctly. Sometimes accessing an element of the loop outside the loop when it would be empty on the initial call. Some, trying a nested loop, the outer loop for the first word and the inner loop for the second word. But what about subsequent recursive calls? We only want to add one.
4. When adding 2nd and subsequent words we only want to do so if the condition is met. Meaning the first letter of the word we are adding matches the last letter of currently the last word. Some solutions postponed checking this until loop size equaled goal. This leads to a lot of unnecessary computation. Like putting a digit in a Sudoku board that repeats a digit in the row or column, but going ahead anyway.
5. If the current work words then we make a recursive call. If that recursive call succeeds we are done. We found a loop and return true. And we want to return true NOW to avoid unnecessary work. Many students called canForm but ignored the return value. Many student simply returned the result of canForm, true or false. But if it is false we are giving up too soon. We may have many more words to try.
6. If the recursive call to canForm returns false we want to remove the last word of the loop. Many solutions called remove that does a linear search. But this is unnecessary work. We know we added the word to the end of the list so that is the one we should remove. Hopefully this is how you did anagrams as well.
7. Details matter. Being fluent in a programming language means knowing the correct way to handle things. Yes, this is a data structures class and we try to focus on data structures, but we are programming in Java and I expect students are fluent in Java. The number of solutions that had errors on the details of the variables involved was disappointing. Perhaps this was because students did not have adequate time to read and understand the problem and devise a solution. Common issues we saw that were penalized:
    a. Treating ArrayList objects like arrays. using loop[i] to try and access elements instead of loop.get(i). Using dictionary.length instead of dictionary.size
    b. Not knowing how to compare chars. Chars are primitives. They are value variables. We compare value variables with the equality relational operatior, ==. Given char a and char b, a.equals(b) does not compile. The compiler does not try to autobox a.

c. Using substring. This method was disallowed. To compare individual characters in a String use charAt. substring creates unnecessary String objects. A waste of time (when the garbage collector runs to clean up the String objects) and space.

```
private static boolean canForm(ArrayList<String> dictionary,
                ArrayList<String> loop, int goal) {
    if (loop.size() == goal) {
        String first = loop.get(0);
        String last = loop.get(goal - 1);
        return charsMatch(last, first);
    }
    for (String nextWord : dictionary) {
        if (loop.size() == 0) {
            // loop empty, can always add
            // GACKY repeated code. Alternative is a
            // complex boolean expression.
            loop.add(nextWord);
            if (canForm(dictionary, loop, goal)) {
                return true;
            }
            loop.remove(loop.size() - 1);
        } else if (!loop.contains(nextWord)) {
            // Check the chars
            String lastWord = loop.get(loop.size() - 1);
            if (charsMatch(lastWord, nextWord)) {
                loop.add(nextWord);
                if (canForm(dictionary, loop, goal)) {
                    return true;
                }
                loop.remove(loop.size() - 1);
            }
        }
    }
    return false;
}

private static boolean charsMatch(String s1, String s2) {
    return s1.charAt(s1.length() - 1) == s2.charAt(0);
}
```

```java
// alternate without repeated code
private static boolean canForm2(ArrayList<String> dictionary,
                ArrayList<String> loop, int goal) {
    if (loop.size() == goal) {
        String first = loop.get(0);
        String last = loop.get(goal - 1);
        return charsMatch(last, first);
    }
    for (String nextWord : dictionary) {
        if (loop.size() == 0 || (!loop.contains(nextWord)
                    && charsMatch(loop.get(loop.size() - 1), nextWord))) {
            loop.add(nextWord);
            if (canForm2(dictionary, loop, goal)) {
                return true;
            }
            loop.remove(loop.size() - 1);
        }
    }
    return false;
}
```