Question 4 on Quiz 3 asked the following:

Explain how the program (repeated below) can be improved and derive/justify the complexity of the new program.

```python
for i in range(n):
    for j in range(n):
        if !(A[i][j] == A[j][i]):
            return False
return True
```

Part 1 (actual correction) was worth 4 points, and part 2 (justification and/or derivation) was worth 6 points. We received many versions of incorrect responses to this. This note lists them so that you can learn from these mistakes. Ask yourselves why these answers were incorrect and why they lost points. If you see your answer in the list below, please come to office hours and make sure you understand.

Please also note that a number of wrong answers are because the students assumed incorrectly, that the original program was detecting cycles, not testing whether the graph was un/directed. Unfortunately, a NoDJ policy in this case (giving you full points because you have already been penalized for the mistake) would be blatantly unfair to other students.

Compendium of answers

1. Since the adjacency matrix is $n \times n$ and mirrored across the diagonal, the above algorithm is checking the same values twice resulting in $O(n^2)$ time complexity. The algorithm can be improved to $O(n)$ time complexity by only considering one of the diagonal halves. [4+0 pts]

2. for i in range(n // 2):
   for j in range(n // 2):
       if !(A[i][j] == A[j][i]):
           return False
return True
# range(n // 2) compares the value to the adjacent value > n // 2.
# No need to check above n // 2 because it is already checked.
# BigO = n log(n)
This only tests (and that too redundantly) edges among the first half of the nodes. [0+0 pts]

3. Can combine the two for loops into one: for (i in range(n)) & (j in range(n)):
   and can simplify the
   ```python
   if !(A[i][j] == A[j][i]): into
   if A[i][j] != -A[j][i]:
   ```
which would result in the complexity being $O(n)$

4. I would iterate through all the adjacency array instead of through range of n and then just check to ensure that $A[i][j]$ and $A[j][i]$ exist, ideally this would make the new algorithms run time $O(n)$ instead. Additionally you would need to have a visited array to make sure that you aren’t missing any of the values at the end of the for loop. [0+0 pts]

5. The program can be improved by turning the adjacency matrix into an adjacency list. This would decrease the run time from $O(n^2)$ to $O(m + n)$. [0 pts]

6. The program could be implemented only using one for loop in range n. This would improve the complexity to $O(n)$. You only need to visit each element once and compare to its counter part (element with reverse indices). [0+0 pts]

7. You could rewrite the section of code as $O(n)$ by making i an integer before the loop and the while loop for j. Check the condition and then add another if statement to check if j has been through every index for i before i reached the value of n. Such as:

```python
int i = 0;
While j < n:
    if !(A[i][j] == A[j][i]):
        return False
    if (j == (n-1) and i != n ):
        j = 0
        i ++
        j ++
return True
```

8. The program can be improved through the method of coloring and depth first search. When we visit a node, we can color it an intermediate color, such as gray. Then, if somewhere along our path of discovery we come across a node that is gray, we know that we have come across a cycle and can return false at that point. This program runs in $O(m + n)$ time, improving from the previous implementation. [0 pts]

9. You only need to loop through n nodes once with two variables in order to detect if it is undirected or not. Looping through twice is unnecessary. [0 pts]

```python
for I, J in range(n):
        return False
return True
```

10. This algorithm could be changed to check the edges to see if there is an immediate cycle as in (let node = i, if node j == i, then there is a cycle). [0 pts]
11. The if statement can be simplified by undoing the not (!) and switching the return statements (Return true would be return false, vice versa. This will keep the flow of the algorithm while making it more comprehensible. [0 pts]

12. The program can be improved by using the Kruskal’s algorithm because it considers and adds edges unless it would produce a cycle. The time complexity of this algorithm is \((E \log E)\) time, with \(E\) as the edges of the graph. [0 pts]

13. If we chose a node to start at and did a breadth first search, we could then see if the nodes adjacent to the original also had a path back to its parent, and if so we would return false. Otherwise, we would return true. This new algorithm would be order \(O(n)\). [0 pts]

14. The keyword in question 2 is adjacency matrix. If we were to use an adjacency list instead, we could reduce the running time. [2 pts]

In the Cormen et al. book it states on pages 590 and 591: “If \(G\) is an undirected graph, the sum of the lengths of all the adjacency lists is \(2|E|\) since if \((u, v)\) is an undirected edge, then \(u\) appears in \(v\)'s adjacency list and vice versa” \((E\) is the set of edges). I’m not sure how to formally prove that the sum of the lengths of the adjacency lists being \(2|E|\) always means it is undirected, but I am going to assume that the statement is always true.

In question 3 it states “For a graph with \(n\) nodes and \(m\) edges.” Therefore, I am going to assume that we know how many edges are in the graph before we run the algorithm. The new algorithm will have \(O(n)\) worst case running time, assuming that it takes constant time to calculate the sum of the adjacency lists.

```python
def sumAdjacencyLists(A, n, m):
    totalSum = 0
    for i in range(n):
        totalSum += len(A[i])
    if (totalSum == (2 * m)):
        return true
    return false
```

Sanjay’s comments to this student: You did not account for the cost of building the adjacency list in the first place. In particular, since \(A\) is the adjacency matrix, \(len(A[i])\) is the length of the \(i\)-th row, which is always \(n\) (it counts zeroes and ones). What you really wanted was the number of non-zero entries in \(A[i]\) and since you don’t know where exactly they are, you will end up inspecting all the elements.

But I like your thought process.

15. This program has the smallest possible time complexity, as you need to check not only the head-end but the tail-end of the edge to know if it the graph is undirected. There are no
data structures more suited to checking than an array (as it is constant time), and the test must check both tail and head, so this is the smallest required amount of work. [0 pts]

16. The program can be improved if it accounted for visited nodes. [0 pts]

17. If the second loop only goes through edges $m$ instead of all nodes, the complexity will be $O(nm)$ and won’t check extra nodes. [0 pts]

18. As we have to increment over every single node at least once, the problem we find with ourselves is having to repeat some nodes which give us the $O(m+n)$ which we typically see. The reason that it is $O(m+n)$ is that we have to identify all edges (this is seen in the dictionary example in the notes. One way we can cut down on the complexity of the program is to not take into account repeated connections/edges i.e., visited nodes. By cutting out visited nodes, we can decrease the complexity as repeated nodes no longer occur. This can reduce the complexity seemingly down to a $O(\log n)$ complexity, with a possible $n$ multiplied based on how non-repeated nodes. [0 pts]

19. If you change the outer for loop to a while you have

```python
while(I < n):
    for j in range n:
```

can help cut down the time complexity to $n \log(n)$. [0 pts]

20. The program can be improved by not checking each connection twice. For example, you’ll check if $A[1][2] == A[2][1]$ and later on in the loop you’ll check $A[2][1] == A[1][2]$. If we can cut this out, we avoid repeating work. The new complexity would be $O(\log n)$. [2 pts]

21. You can use an outer while loop instead of a for loop. This should bring the complexity towards $O(n + m)$ which is much better. [0 pts]

22. The program could be implemented to do a breadth first search of the graph instead that checks if a node refers to it’s parent when the algorithm loops through the children of a given node, returning false if it doesn’t then. The new worst case time complexity would be $O(n + m)$.

23. instead of using 2 for loops use one and instead of j use n-i, which will improve the time complexity to $O(n)$. [0 pts]