Homework 2
CPSC 535: Advanced Algorithm

Group 1 Team members:

Vishal Vilas Shinde
(vishal1710@csu.fullerton.edu**)**

Yash Ashokbhai Savaliya
(yashhsavaliya27@csu.fullerton.edu)

Lemmie Stephen Carvalho
(lemmiecarvalho0910@csu.fullerton.edu)

Q1(15). Reverse a Doubly Linked List without creating any new node.

Since a doubly linked list allows traversal in both directions using next and prev pointers, an in-place reversal can be achieved by swapping these pointers for each node. The algorithm will consist of the following steps:

Step 1: Pointer Initialization
Start with a pointer current at the head of the doubly linked list. Also, maintain a pointer temp to assist in swapping.
Step 2: Iteration (Swapping Pointers)
For each node in the list:

- Swap its next and prev pointers.
- Move the current pointer to the previous node (which is now stored in prev after swapping).

Step 3: Updating Head
Once the traversal is complete, update the head of the linked list to the last node encountered. This approach ensures that we traverse the list only once, making the time complexity **O(n)**.

Pseudo Code:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

def reverseDoublyLinkedList(head):
    current = head
    temp = None

    # Traverse the list and swap next and prev for each node
    while current is not None:
        temp = current.prev
        current.prev = current.next
        current.next = temp
        current = current.prev  # Move to the next node (which was previously 'next')

    # After the loop, temp will be at the last node (new head)
    if temp is not None:
        head = temp.prev  # Set new head of the list

    return head
```

Time Complexity Analysis (O(n)):

- We traverse the doubly linked list **once**, swapping the next and prev pointers for each node.
- Since each node is visited exactly **once**, the total number of operations is proportional to **n** (where n is the number of nodes in the list).
- Hence, the time complexity is **O(n)**.

Space Complexity Analysis (O(1)):

- We **do not** use any extra data structures (like arrays, stacks, or recursion).
- The algorithm only uses a few **temporary pointer variables (current, temp)**, which consume constant space **O(1)**.
- Since the reversal is done **in place**, no additional memory is required.
- Thus, the space complexity is **O(1)**.

Q1.(30) FOUR-PERSON BRIDGE-CROSSING PUZZLE

Puzzle Statement

Four people need to cross a bridge at night, sharing one flashlight. Up to two can cross at a time at the pace of the slower person. Their times are 1, 2, 5, and 10 minutes. The flashlight must always be carried across

Goal: Get all four people from the left side of the bridge to the right side in 17 minutes or less.

Constraints:
1. Only two people can cross at a time.
2. They must walk at the pace of the slower person.
3. The flashlight must always be carried across (it cannot be thrown, etc.).
4. Someone has to bring the flashlight back to the remaining people each time until everyone has crossed.

Strategy:
1. Let the fastest individual, P1 (1 minute), handle most of the trips back with the flashlight to minimize wasted time.
2. Let the two slowest individuals, P3 (5 minutes) and P4 (10 minutes), cross together so that the 10-minute trip only occurs once.

Sequence of Crossings:

1. P1 and P2 cross to the right
2. P1 returns alone
3. P3 and P4 cross together
4. P2 returns alone
5. P1 and P2 cross again

The total time is 17 minutes.

ALGORITHM (PSEUDOCODE)

We know from puzzle theory and trials that the following crossing order leads to an optimal solution.

```
Algorithm puzzleBridgeCrossing( P1, P2, P3, P4 )
Sort the people by their times ascending, so: P1 < P2 < P3 < P4 (i.e., 1 < 2 < 5 < 10)

Steps:
  1. Cross( P1, P2 )
cost = max(time(P1), time(P2)) = 2
  2. Return( P1 )
cost = time(P1) = 1
  3. Cross( P3, P4 )
cost = max(time(P3), time(P4)) = 10
  4. Return( P2 )
cost = time(P2) = 2
  5. Cross( P1, P2 )
cost = max(time(P1), time(P2)) = 2

Total time = 2 + 1 + 10 + 2 + 2 = 17
```

Everyone is now across in 17 minutes.

Explanation of Algorithm Steps:

• We first send the two fastest individuals, P1 and P2, to the far side.
• Then P1 (the fastest) returns alone.
• Next, we send the two slowest individuals, P3 and P4, together.
• P2 returns to bring back the flashlight.
• Finally, P1 and P2 cross again.

By pairing P3 and P4 (the slowest two) on a single trip, we avoid paying the 10-minute cost twice. By letting P1 do most of the return trips, we minimize the time used to ferry the flashlight back and forth.

STEP-BY-STEP EXPLANATION WITH DIAGRAMS

Notation in Diagrams

• The left side of the bridge is labeled "Left Side"

• The right side is labeled "Right Side"

• The flashlight is shown in square brackets to indicate where it is after each step

• The people are shown along with their individual crossing times (in parentheses)

Initial State
Left Side: [P1(1), P2(2), P3(5), P4(10), Flashlight]
Bridge: --
Right Side: (empty) Time Elapsed: 0 minutes
Explanation: All four people start on the left side with the flashlight.
Step 1: P1 and P2 cross to the right
Action: P1(1) + P2(2) go -->, taking 2 minutes (dictated by P2's speed)
Left Side: [P3(5), P4(10)]
Bridge: --
Right Side: [P1(1), P2(2), Flashlight] Time Elapsed: 2 minutes
Explanation: P1 and P2 both leave the left side and reach the right side in 2 minutes.

Step 2: P1 returns with the flashlight
Action: P1(1) goes <--, taking 1 minute
Left Side: [P1(1), P3(5), P4(10), Flashlight]
Bridge: --
Right Side: [P2(2)] Time Elapsed: 3 minutes (2 + 1)
Explanation: P1 comes back alone to the left side with the flashlight. This takes only 1 minute since P1 is the fastest.

Step 3: P3 and P4 cross together
Action: P3(5) + P4(10) go -->, taking 10 minutes (dictated by P4's speed)
Left Side: [P1(1)]
Bridge: --
Right Side: [P2(2), P3(5), P4(10), Flashlight] Time Elapsed: 13 minutes (3 + 10)
Explanation: The two slowest individuals cross together. This uses 10 minutes in a single trip rather than having each do a slow crossing separately.

Step 4: P2 returns with the flashlight
Action: P2(2) goes <--, taking 2 minutes
Left Side: [P1(1), P2(2), Flashlight]
Bridge: ---
Right Side: [P3(5), P4(10)] Time Elapsed: 15 minutes (13 + 2)
Explanation: Now P2 returns to bring the flashlight back to P1 who is still on the left side. This takes 2 minutes.

Step 5: P1 and P2 cross again to the right
Action: P1(1) + P2(2) go -->, taking 2 minutes
Left Side: (empty)
Bridge: ---
Right Side: [P1(1), P2(2), P3(5), P4(10), Flashlight] Total Time Elapsed: 17 minutes (15 + 2)
Explanation: Finally, P1 and P2 cross together once more. This takes 2 minutes. Everyone is now on the right side at exactly 17 minutes total.

CONCLUSION:

The sequence described above meets the 17-minute requirement:
• P1 and P2 cross (2 minutes)
• P1 returns (1 minute)
• P3 and P4 cross (10 minutes)
• P2 returns (2 minutes)
• P1 and P2 cross again (2 minutes)

Total = 2 + 1 + 10 + 2 + 2 = 17 minutes

Q2.(15) Check for anagrams

Algorithm: Hashing Approach for Checking Anagrams

To determine if two words are anagrams, we can use a **hashing (frequency count) approach** as follows:

Steps:

1. Convert both words to lowercase (to ensure case insensitivity).
2. Check if lengths match—if not, they can't be anagrams.
3. Create a frequency dictionary manually for both words:
    - Traverse each word and count occurrences of each character using a dictionary.
4. Compare both dictionaries:
    - If they match, the words are anagrams.
    - Otherwise, they are not.

Pseudo Code:

```
def areAnagrams(word1, word2)

   # Convert words to lowercase to make comparison case insensitive

   word1, word2 = word1.lower(), word2.lower()


   # If lengths are different, they cannot be anagrams

   if len(word1) != len(word2):

      return False

   # Create frequency dictionaries manually

   freq_count1 = {}
```

```
freq_count2 = {}

# Count character frequencies for word1

for char in word1:

    if char in freq_count1:

        freq_count1[char] += 1

    else:

        freq_count1[char] = 1

# Count character frequencies for word2

for char in word2:

    if char in freq_count2:

        freq_count2[char] += 1

    else:

        freq_count2[char] = 1

# Compare both frequency dictionaries

return freq_count1 == freq_count2
```

Time Complexity: O(n)

- Counting characters in a word takes **O(n)**.
- Comparing two dictionaries takes **O(1)** (since at most **26** lowercase letters exist).
- Overall complexity: **O(n)**.

Space Complexity: O(1)

- Since the maximum number of unique letters is **26** (in the English alphabet), the hash table has a constant size.
- Thus, the space complexity is **O(1)**.