TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 1: Linked Lists II

Session Overview

In this session, students will deepen their ability to work with linked lists, solving problems involving traversal, node manipulation, and cycle detection. These exercises will cover techniques common to many linked lists problems and enhance their understanding of this crucial data structure.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

Problem 1: Selective DNA Deletion

As a biologist, you are working on editing a long strand of DNA represented as a linked list of nucleotides. Each nucleotide in the sequence is represented as a node in the linked list, where each node contains a character ('A', 'T', 'C', 'G') representing the nucleotide.

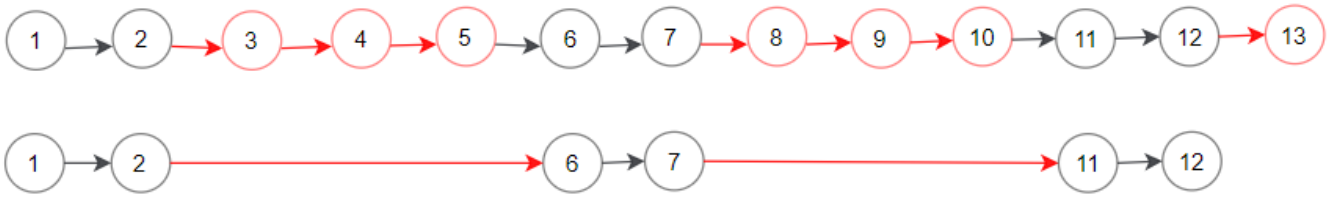
Given the head of the linked list `dna_strand` and two integers `m` and `n`, write a function `edit_dna_sequence()` that simulates the selective deletion of nucleotides in a DNA sequence. You will: - Start at the beginning of the DNA strand. - Retain the first `m` nucleotides from the current position. - Remove the next `n` nucleotides from the sequence. - Repeat the process until the end of the DNA strand is reached.

Return the head of the modified DNA sequence after removing the mentioned nucleotides.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class Node:
   def __init__(self, value, next=None):
       self.value = value
       self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def edit_dna_sequence(dna_strand, m, n):
    pass
```

Example Usage:




```
dna_strand = Node(1, Node(2, Node(3, Node(4, Node(5, Node(6, Node(7, Node(8, Node(9,

print_linked_list(selective_trail_clearing(dna_strand, 2, 3))
```

Example Output:

```
1 -> 2 -> 6 -> 7 -> 11 -> 12
Explanation: Keep the first (m = 2) nodes starting from the head of the linked List
(1 -> 2) show in black nodes.
Delete the next (n = 3) nodes (3 -> 4 -> 5) show in red nodes.
Continue with the same procedure until reaching the tail of the Linked List.
```

Problem 2: Protein Folding Loop Detection

As a biochemist, you're studying the folding patterns of proteins, which are represented as a sequence of amino acids linked together. These proteins sometimes fold back on themselves, creating loops that can impact their function.

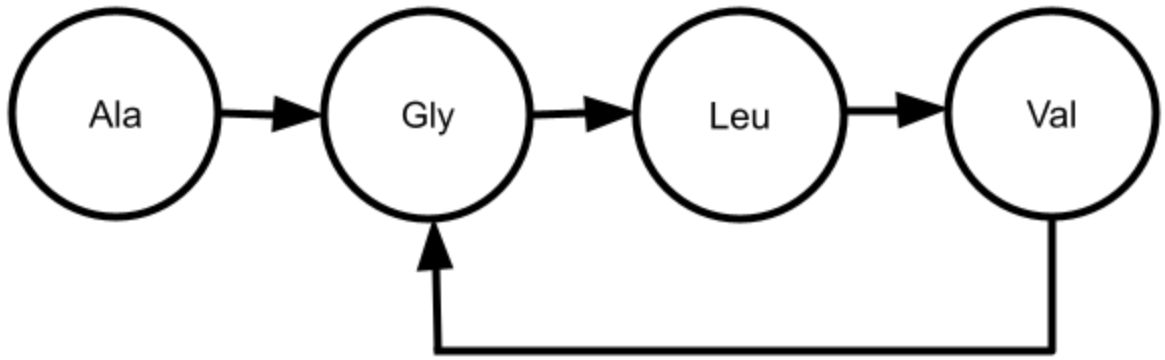
Given the head of a linked list `protein` where each node in the linked list represents an amino acid in the protein, return an array with the `value` s of any cycle in the list. A linked list has a cycle if at some point in the list, the node's next pointer points back to a previous node in the list.

The `value` s may be returned in any order.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

def cycle_length(protein):
    pass
```

Example Usage:




```
protein_head = Node('Ala', Node('Gly', Node('Leu', Node('Val'))))
protein_head.next.next.next.next = protein_head.next

print(cycle_length(protein_head))
```

Example Output:

```
['Gly', 'Leu', 'Val']
```

▶ 💡 **Hint: Slow and Fast Pointers**

▶ 💡 **Hint: Multiple Pass Technique**

Problem 3: Segmenting Protein Chains for Analysis

As a biochemist, you are analyzing a long protein chain represented by a singly linked list, where each node is an amino acid. For a specific experiment, you need to split this protein chain into `k` consecutive segments for separate analysis. Each segment should be as equal in length as possible, with no two segments differing in size by more than one amino acid.

The segments should appear in the same order as the original protein chain, and segments earlier in the list should have a size greater than or equal to those occurring later. If the protein chain cannot be evenly divided, some segments may be an empty list.

*W*rite a function `split_protein_chain()` that takes the head of the linked list `protein` and an integer `k`, and returns an array of `k` segments.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    if not head:
        print("Empty List")
        return
        current = head
        while current:
                print(current.value, end=" -> " if current.next else "\n")
                current = current.next

def split_protein_chain(protein, k):
    pass
```

Example Usage:

```
protein1 = Node('Ala', Node('Gly', Node('Leu', Node('Val', Node('Pro', Node('Ser', N
protein2 = Node('Ala', Node('Gly', Node('Leu', Node('Val'))))

parts = split_protein_chain(protein1, 3)
for part in parts:
    print_linked_list(part)

parts = split_protein_chain(protein2, 5)
for part in parts:
    print_linked_list(part)
```

Example Output:

```
Ala -> Gly -> Leu
Val -> Pro -> Ser
Thr -> Cys
Example 1 Explanation: The input list has been split into consecutive parts with siz
and earlier parts are a larger size than later parts.

Ala
Gly
Leu
Val
Empty List
Example 2 Explanation: The input list has been split into consecutive parts with siz
Because k is one greater than the length of the input list, the last segment is an e
```

Problem 4: Maximum Protein Pair Stability

You are analyzing the stability of protein chains, which are represented by a singly linked list where each node contains an integer stability value. The chain has an even number of nodes, and for each node `i` (0-indexed), its "twin" is defined as node `(n-1-i)`, where n is the length of the linked list.

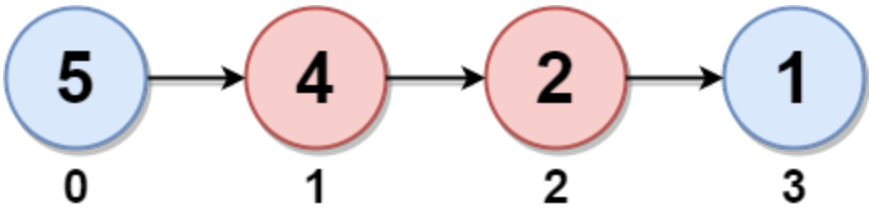
Write a function `max_protein_pair_stability()` that accepts the `head` of a linked list, and determines the maximum "twin stability sum," which is the sum of the stability values of a node and its twin.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

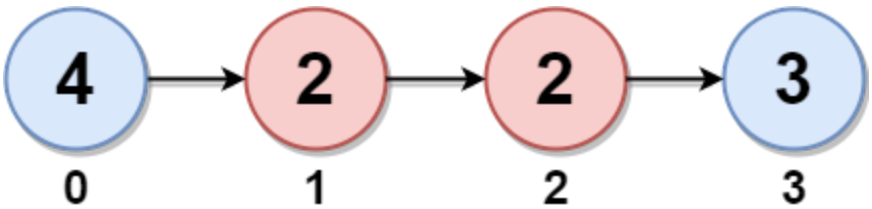
```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def max_protein_pair_stability(head):
    pass
```

Example Usage:

Example 1



Example 2




```
head1 = Node(5, Node(4, Node(2, Node(1))))
head2 = Node(4, Node(2, Node(2, Node(3))))

print(max_protein_pair_stability(head1))
print(max_protein_pair_stability(head2))
```

Example Output:

```
6
Example 1 Explanation:
Nodes 0 and 1 are the twins of nodes 3 and 2, respectively. All have twin sum = 6.
There are no other nodes with twins in the linked list.
Thus, the maximum twin sum of the linked list is 6.

7
Explanation:
The nodes with twins present in this linked list are:
- Node 0 is the twin of node 3 having a twin sum of 4 + 3 = 7.
- Node 1 is the twin of node 2 having a twin sum of 2 + 2 = 4.
Thus, the maximum twin sum of the linked list is max(7, 4) = 7.
```

▶ 💡 **Hint: Identifying Subproblems**

Problem 5: Grouping Experiments

You have a list of experiment results for two types of experiments conducted in alternating order represented by a singly linked list. Each node in the list corresponds to an experiment result, and the position of the result in the 1-indexed sequence determines whether it is odd or even.

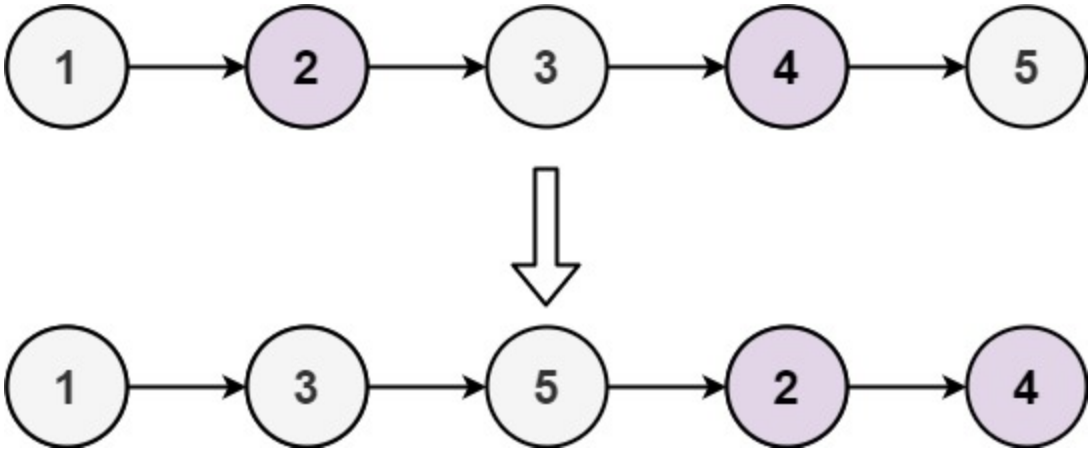
Given the head of the linked list, `exp_results`, reorganize the experiment results so that all results in odd positions are grouped together first, followed by all results in even positions. The relative order of the results within the odd group and the even group must remain the same as the original sequence. The first result in the list is considered to be odd, the second result is even, and so on. Return the head of the reorganized list.

Your solution must have `O(1)` space complexity and `O(n)` time complexity.

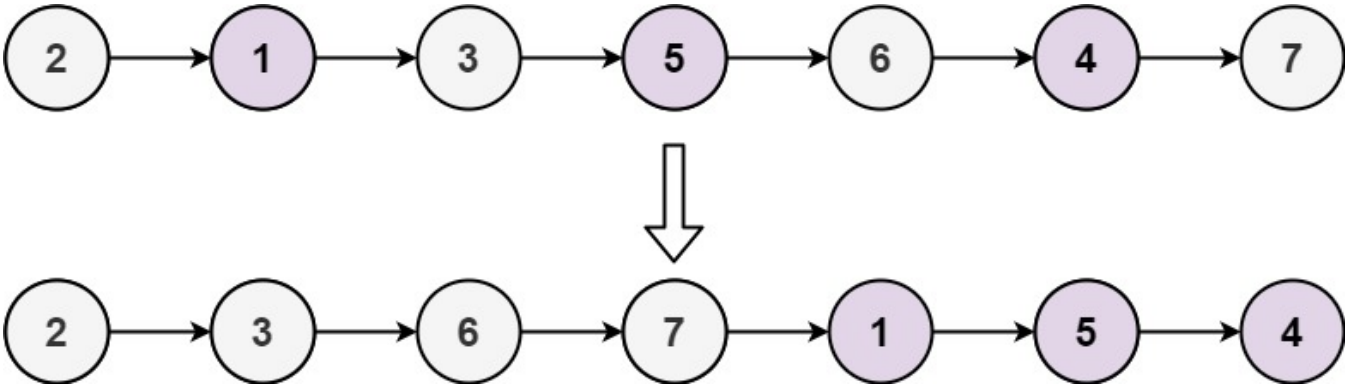
```python
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
       print(current.value, end=" -> " if current.next else "\n")
       current = current.next

def odd_even_experiments(exp_results):
    pass
```

Example Usage:

Example 1



Example 2




```
experiment_results1 = Node(1, Node(2, Node(3, Node(4, Node(5)))))
experiment_results2 = Node(2, Node(1, Node(3, Node(5, Node(6, Node(4, Node(7)))))))

print_linked_lists(odd_even_experiments(experiment_results1))
print_linked_lists(odd_even_experiments(experiment_results2))
```

Example Output:

```
1 -> 3 -> 5 -> 2 -> 4
2 -> 3 -> 6 -> 7 -> 1 -> 5 -> 4
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Linked List Game

As the judge of the game show, you are given the `head` of a linked list of **even** length containing integers.

Each **odd-indexed** node contains an odd integer and each **even-indexed** node contains an even integer.

We call each even-indexed node and its next node a **pair**, e.g., the nodes with indices `0` and `1` are a pair, the nodes with indices `2` and `3` are a pair, and so on.

For every pair, we compare the values of the nodes in the pair:

- If the odd-indexed node is higher, the `"Odd"` team gets a point.
- If the even-indexed node is higher, the `"Even"` team gets a point.

Write a function `game_result()` that returns the name of the team with the higher points, if the points are equal, return `"Tie"`.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class Node:
        def __init__(self, value, next=None):
                self.value = value
                self.next = next

# For testing
def print_linked_list(head):
        current = head
        while current:
                print(current.value, end=" -> " if current.next else "\n")
                current = current.next

def game_result(head):
    pass
```

Example Usage:

```
game1 = Node(2, Node(1))
game2 = Node(2, Node(5, Node(4, Node(7, Node(20, Node(5))))))
game3 = Node(4, Node(5, Node(2, Node(1))))

print(game_result(game1))
print(game_result(game2))
```

Example Output:

```
Even
Example 1 Explanation: There is only one pair in this linked list and that is (2,1).
Since 2 > 1, the Even team gets the point.
Hence, the answer is "Even".

Odd
Example 2 Explanation: There are 3 pairs in this linked list.
Let's investigate each pair individually:
(2,5) -> Since 2 < 5, The Odd team gets the point.
(4,7) -> Since 4 < 7, The Odd team gets the point.
(20,5) -> Since 20 > 5, The Even team gets the point.
The Odd team earned 2 points while the Even team got 1 point and the Odd team has th
Hence, the answer is "Odd".

Tie
Example 3 Explanation: There are 2 pairs in this linked list.
Let's investigate each pair individually:
(4,5) -> Since 4 < 5, the Odd team gets the point.
(2,1) -> Since 2 > 1, the Even team gets the point.
Both teams earned 1 point.
Hence, the answer is "Tie".
```

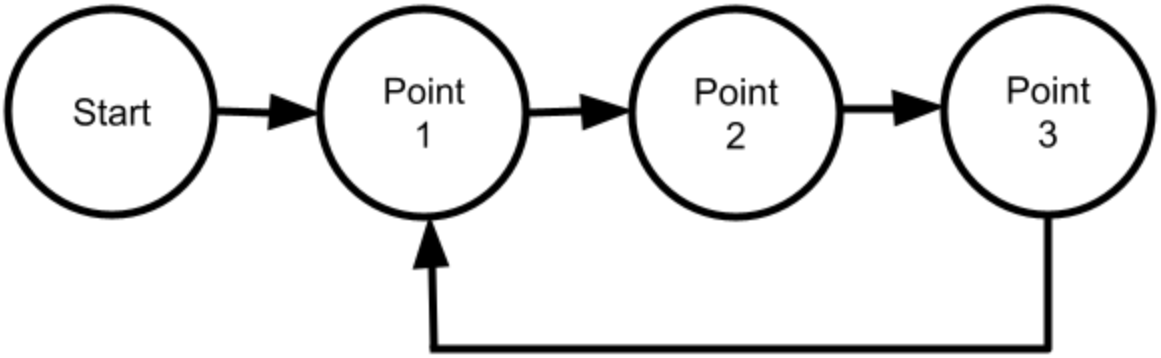
Problem 2: Cycle Start

On your marks, get set, go! Contestants in the game show are racing along a path that contains a loop, but there's a hidden mini challenge: they aren't told where along the path the loop begins. Given the head of a linked list, `path_start` where each node represents a point in the path, return the value of the node at the start of the loop. If no loop exists in the path, return `None`.

A linked list has a cycle or loop if at some point in the list, the node's next pointer points back to a previous node in the list.

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

def cycle_start(path_start):
        pass
```

Example Usage:




```
path_start = Node('Start', Node('Point 1', Node('Point 2', Node('Point 3'))))
path_start.next.next.next.next = path_start.next
print(cycle_start(path_start))
```

Example Output:

```
Point 1
```

▶ 💡 **Hint: Slow and Fast Pointers**

▶ 💡 **Hint: Multiple Pass Technique**

Problem 3: Fastest Wins!

Contestants, today's challenge is to sort a linked list of items the fastest! The catch - you have to follow a certain technique or you're disqualified from the round. You'll start with an unsorted lineup, and with each step, you'll move one item at a time into its proper position until the entire lineup is perfectly ordered.

Given the `head` of a linked list, sort the items using the following procedure:

- Start with the first item: The sorted section initially contains just the first item. The rest of the items await their turn in the unsorted section.
- Pick and Place: For each step, pick the next item from the unsorted section, find its correct spot in the sorted section, and place it there.
- Repeat: Continue until all items are in the sorted section.

Return the head of the sorted linked list.

As a preview, here is a graphical example of the required technique (also known as the insertion sort algorithm). The partially sorted list (black) initially contains only the first element in the list. One element (red) is removed from the input data and inserted in-place into the sorted list with each iteration.

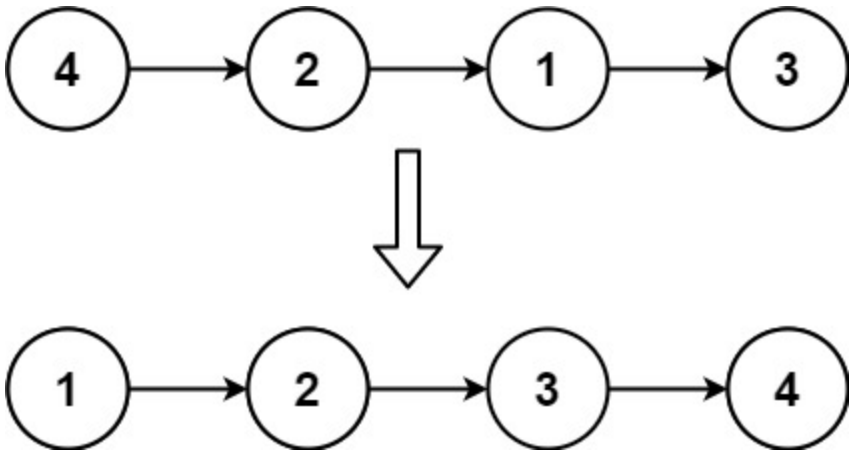
6 5 3 1 8 7 2 4

When you have finished your sorting, receive bonus points for evaluating the time and space complexity of your solution. To get full points, you must define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

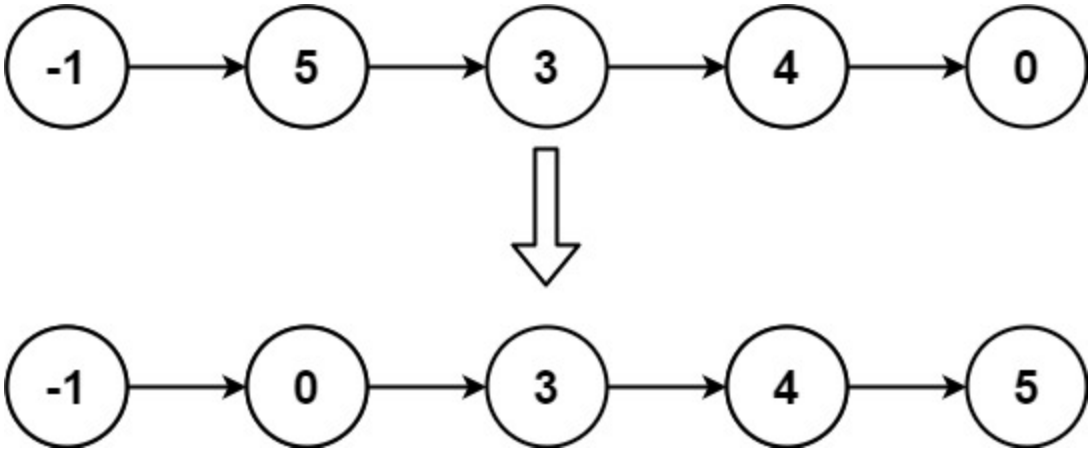
```
class Node:
        def __init__(self, value, next=None):
                self.value = value
                self.next = next

# For testing
def print_linked_list(head):
        current = head
        while current:
                print(current.value, end=" -> " if current.next else "\n")
                current = current.next

def sort_list(head):
    pass
```

Example Usage:

Example 1



Example 2




```
head1 = Node(4, Node(2, Node(1, Node(3))))
head2 = Node(-1, Node(5, Node(3, Node(4, Node(0)))))

print_linked_list(sort_list(head1))
print_linked_list(sort_list(head2))
```

Example Output:

```
1 -> 2 -> 3 -> 4
-1 -> 0 -> 3 -> 4 -> 5
```

▶ 💡 **Hint: Temporary Head Technique**

Problem 4: Calculate Prize Money

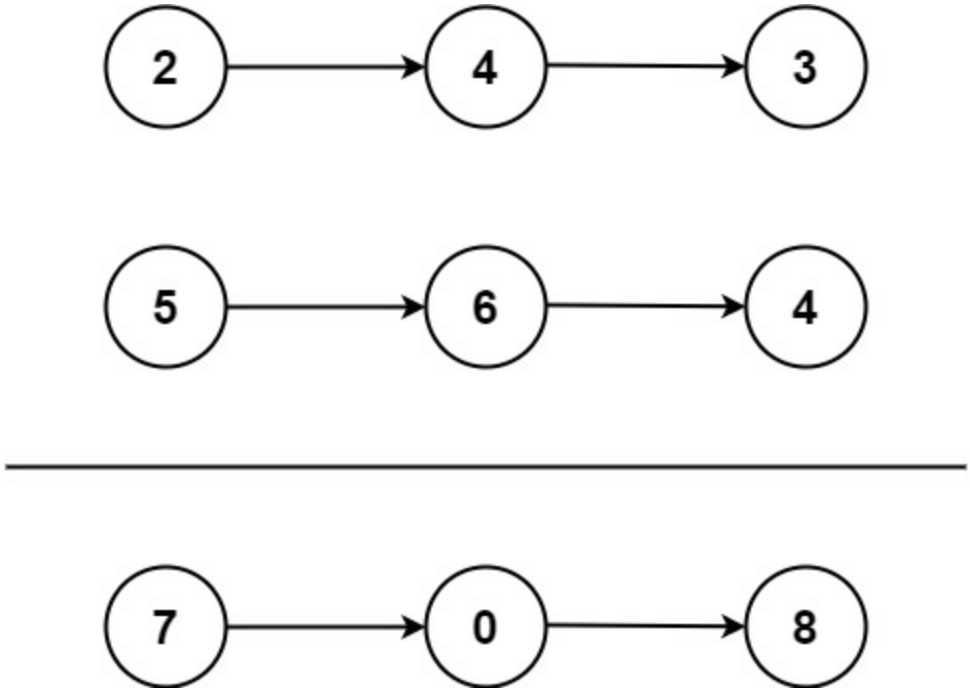
In the game show, contestants win prize money for each of the challenges they participate in. Write a function `get_total_prize()` that accepts the heads of two non-empty linked lists, `prize_a` and `prize_b`, representing two non-negative integers. The digits are stored in reverse order and each node represents a single digit. The function should add the two numbers and return the sum of the prize money as a linked list.

The digits of the sum should also be stored in reverse order with each node containing a single digit.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def add_two_numbers(head_a, head_b):
    pass
```

Example Usage:




```
head_a = Node(2, Node(4, Node(3))) # 342
head_b = Node(5, Node(6, Node(4))) # 465

print_linked_list(add_two_numbers(head_a, head_b))
```

Example Output:

```
7 -> 0 -> 8
Explanation: 342 + 465 = 807
```

Problem 5: Next Contestant to Beat

You are given the head of a linked list `contestant_scores` with `n` nodes where each node represents the current score of a contestant in the game.

For each node in the list, find the value of the contestant with the next highest score. That is, for each score, find the value of the first node that is next to it and has a strictly larger value than it.

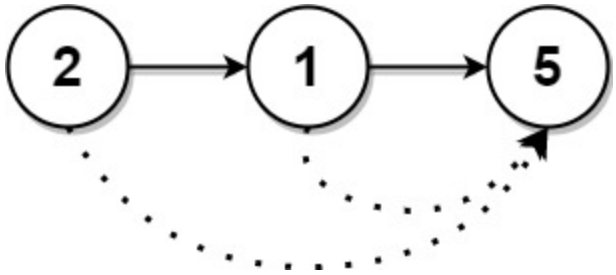
Return an integer array `answer` where `answer[i]` is the value of the next greater node of the `ith` node (1-indexed). If the `ith` node does not have a next greater node, set `answer[i] = 0`.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

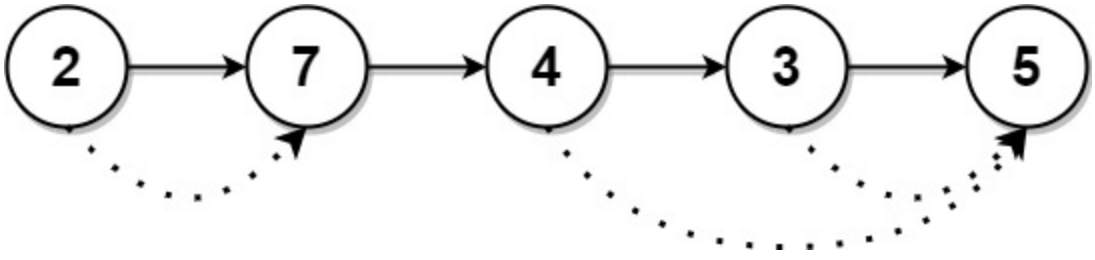
```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
        current = head
        while current:
                print(current.value, end=" -> " if current.next else "\n")
                current = current.next

def next_highest_scoring_contestant(contestant_scores):
    pass
```

Example Usage:

Example 1



Example 2




```
contestant_scores1 = Node(2, Node(1, Node(5)))
contestant_scores2 = Node(2, Node(7, Node(4, Node(3, Node(5)))))

print(next_highest_scoring_contestant(contestant_scores1))
print(next_highest_scoring_contestant(contestant_scores2))
```

Example Output:

```
[5, 5, 0]
[7, 0, 5, 5, 0]
```

▶ 💡 **Hint: Blast from the Past!**

Close Section