TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Fall 2024 (@ Section 1 | Tuesdays and Thursdays 5PM - 7PM PT)
Personal Member ID#: **111572**

Unit 6 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as a reference while you solve the breakout problems for Unit 6. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 6. In addition to the material below, you are expected to know any required concepts from previous units. You may also find advanced concepts and bonus concepts included at the bottom of this page helpful for solving problem set questions, but you are not required to know them for the unit assessment.

Standard Concepts

Python Syntax

Chained Assignment

In Python, chained assignment is a convenient shorthand syntax that allows us to assign multiple variables to the same value with a single line of code.

Chained assignment can help improve readability when initializing or updating multiple variables to the same value.

Example Usage:

```
x = y = z = 3
print(x) # Prints 3
print(y) # Prints 3
print(z) # Prints 3
```

Break Statements

The `break` keyword is used to break out of a loop early. When we add a break statement, the loop mediately terminates. **Try it**

Example Usage:

```
# Example 1: While Loop
count = 1
while count <= 10:
    if count == 5:
        break  # Break the loop if count reaches 5
    print(count)
    count += 1

# Example 2: For Loop
numbers = [1, 3, 5, 7, 9, 2, 4]
for number in numbers:
    if number % 2 == 0:
        print(f"First even number found: {number}")
        break
```

Linked Lists

Multiple Pass Technique

Sometimes a linked list must be traversed multiple times in order to solve a problem. This is common when we need to extract some information about the list before we begin manipulating it. For example, there are many problems that require us to know the length of a linked list in order to solve the overall problem.

Example Usage:

The following function returns the values of a linked list as pairs in a list of tuples. If the linked list has an odd length, the values cannot be evenly paired and the function returns `None`.

```
Example Input List: 1->2->3->4
Example Input: head = 1
Expected Return Value: [(1, 2), (3, 4)]

Example Input List: 1->2->3
Example Input: head = 1
Expected Return Value: None
```

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

def pair_nodes(head):
    # Check if the list is empty
    if not head:
        return None

    # Pass 1: Get the length of the list
    current = head
    length = 0
    while current:
        length += 1
        current = current.next

    # If the length is odd, return None
    if length % 2 != 0:
        return None

    # Pass 2: If the length is even, create tuples of pairs
    pairs = []
    current = head
    while current:
        # Since we know the list length is even, current.next should always be valid here
        pairs.append((current.value, current.next.value))
        current = current.next.next  # Move to the next pair

    return pairs
```

Because we perform a constant number of traversals and each traversal is done in series, the time complexity of a function is not affected by conducting multiple consecutive passes over the list. If the length of our input list is `n`, the above function `pair_nodes()` performs `O(n + n)` operations which reduces to `O(2n)` which can be further reduced to `O(n)` operations.

Variations of this technique may also nest loops or extract certain traversals into helper functions. For example, we can rewrite our `pair_nodes()` function so that getting the length of the list is encapsulated in a helper function.

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

# Helper function: Returns the length of the list
def get_length(head):

    # Pass 1: Get the length of the list
    current = head
    length = 0
    while current:
        length += 1
        current = current.next
    return length

def pair_nodes(head):
    # Check if the list is empty
    if not head:
        return None

    # If the length is odd, return None
    if get_length(head) % 2 != 0: # Call helper function
        return None

    # Pass 2: If the length is even, create tuples of pairs
    pairs = []
    current = head
    while current:
        # Since we know the list length is even, current.next should always be valid here
        pairs.append((current.value, current.next.value))
        current = current.next.next  # Move to the next pair

    return pairs
```

Temporary Head Technique

When we solve linked list problems, we often encounter edge cases that involve the head of a linked list. Common edge cases involving the head of a linked list include:

- Deleting the current head of a linked list
- Adding a new element to the front of a linked list
- Reducing a linked list containing only one node to an empty list
- Adding a new node to an empty list

A common technique for handling these edge cases is creating a temporary head node. To apply this technique, an extra node object is created to serve as the temporary head of the list. We can use the temporary head as a placeholder that allows us to easily add items to an empty linked list or manipulate the actual head of a list as if it were any other node.

Example Usage:

The following function accepts the head of a linked list and a value and deletes the first node in the list with the specified value. It returns the head of the modified list.

```
Example Input List: 1->2->4
Example Input: head = 1, val = 2
Expected Return Value: 1
Expected Return List: 1->4
```

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

def delete_node(head, val):
    temp_head = Node("temp") # Initialize a temporary head node
    temp_head.next = head    # Point the temporary head at the input list

    # Traverse the list
    previous = temp_head
    current = head
    while current:
        if current.value == val:          # If the current node is the node to delete
            previous.next = current.next  # Delete the node
            break                         # Break out of the loop

        previous = current
        current = current.next
    return temp_head.next # Return the head of the input list
```

Because we created a temporary head, deleting the head of the list can be handled in the same way that deleting any other element in the list is handled. We can also proceed directly to traversing the list without checking if `head` is `None`, because the existence of the temporary head means there is at least one node int he list.

▶ **Deleting a node without a temporary head**

▶ ⚠️ **Temp Node Terminology**

Slow-Fast Pointer Technique

The slow-fast pointer technique, also called the tortoise and hare algorithm, is a common technique used to solve linked list and other data structure problems that involve cyclic structures. It involves using two pointers that move at different speeds to solve problems such as cycle detection, finding the middle of a list, or determining the meeting point within a cycle.

To apply the technique, create two pointers:

1. **Slow Pointer**: Increment one step at a time.
2. **Fast Pointer**: Increment two steps at a time.

Example Usage:

The following function returns `True` if a linked list has a cycle and `False` otherwise.

Linked List with a Cycle

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

def has_cycle(head):
    if not head:
        return False

    slow = head  # Starts at the head
    fast = head  # Also starts at the head

    while fast and fast.next:
        slow = slow.next          # Move slow pointer by one
        fast = fast.next.next     # Move fast pointer by two

        if slow == fast:          # If they meet, there's a cycle
            return True

    return False  # If fast reaches the end, no cycle
```

Think about this algorithm as if you had a walker and a runner moving around a track. If the walker and runner start at the same time and the runner moves twice as fast as the walker, eventually the runner will lap the walker. If the walker and runner were instead moving along a straight path, the walker would never see the runner again!

Source: via GIPHY

If you think of the slow pointer as the walker and the fast pointer as the runner, we can apply the same logic to say that if the slow pointer and fast pointer have the same value, the fast pointer is "lapping" the slow pointer and there must be a cycle in the list.