Module 2 Quiz 1 2nd Attempt Editorial

1. Question:

Which stack operation allows you to view the top element without deleting it from the stack?

a) Push
b) Pop
c) Peek
d) Clear

Correct Answer: c) Peek

Explanation:
In a stack, which operates on a Last-In-First-Out (LIFO) principle, the "peek" operation is designed to inspect the top element without altering the stack. "Push" adds an element to the top, "pop" removes the top element, and "clear" empties the entire stack. Since the question specifies inspecting without removing, "peek" is the correct choice.

2. Question:

If an algorithm's runtime is expressed as 3n + 5, how is its time complexity represented in Big O notation?

a) O(1)
b) O(log n)
c) O(n)
d) $O(n^2)$

Correct Answer: c) O(n)

Explanation:
Big O notation describes the upper bound of an algorithm's runtime, focusing on the dominant term as n grows large. For the expression 3n + 5, the linear term 3n grows proportionally with n, while the constant 5 becomes negligible for large n. Thus, the time complexity simplifies to O(n), indicating

linear growth. O(1) is constant time, O(log n) is logarithmic, and $O(n^2)$ is quadratic—none of which match this runtime.

3. Question:

An algorithm sorts an array of size n in O(n log n) time, then performs a single pass through the array in n steps. What is the overall time complexity in Big O notation?

a) O(n)
b) O(n log n)
c) $O(n^2)$
d) O(1)

Correct Answer: b) O(n log n)

Explanation:
When combining time complexities, we take the dominant term. The sorting step takes O(n log n) time, which is typical for efficient sorting algorithms like Merge Sort or Quick Sort. The subsequent iteration through the array takes O(n) time. Since O(n log n) grows faster than O(n) as n increases, the overall complexity is determined by the sorting step, making it O(n log n). $O(n^2)$ would apply to slower sorting methods, and O(n) or O(1) underestimate the combined effort.

4. Question:

What is a primary difference between Selection Sort and Bubble Sort?

a) Selection Sort repeatedly swaps adjacent elements, while Bubble Sort finds the minimum element in each pass
b) Selection Sort identifies the minimum element in each pass, while Bubble Sort swaps adjacent elements when they are out of order
c) Selection Sort uses recursion exclusively, while Bubble Sort is purely iterative
d) Both algorithms create partially sorted sublists from the beginning to the end in each iteration

Correct Answer: b) Selection Sort identifies the minimum element in each pass, while Bubble Sort swaps adjacent elements when they are out of order

Explanation:
Selection Sort works by finding the smallest (or largest) element in the unsorted portion of the array and placing it in its final position in each pass, minimizing swaps. Bubble Sort, conversely, compares and swaps adjacent elements repeatedly until the array is sorted, potentially requiring many swaps per pass. Option a reverses these roles, c is incorrect as both are typically iterative, and d is false since Bubble Sort doesn't guarantee a sorted sublist after each pass in the same way Selection Sort does.

5. Question:

An algorithm's worst-case time complexity is O(n log n), but on average, it uses half the number of comparisons. What is its average-case time complexity in Big O notation?

a) O(n)
b) O(n log n)
c) O(n^2)
d) O(√n)

Correct Answer: b) O(n log n)

Explanation:
Big O notation focuses on the growth rate of an algorithm, not the exact number of operations. If the worst-case complexity is O(n log n) and the average case involves half the comparisons, the growth rate remains proportional to n log n because halving is a constant factor (e.g., ½ * n log n). In Big O, constant coefficients are ignored, so the average-case complexity is still O(n log n). Options like O(n) or O(√n) suggest a fundamentally different growth rate, which isn't supported by the problem.

Question : Count such pairs

```
def count_pairs_with_sum(arr, target_sum):
    count = 0
    seen = {}

    for num in arr:
        complement = target_sum - num
        if complement in seen:
            count += seen[complement]
        if num in seen:
            seen[num] += 1
        else:
            seen[num] = 1

    print(count)
```

Explanation

1. **Function Definition**

   ```
   def count_pairs_with_sum(arr, target_sum):
   ```

   - The function takes an array (`arr`) of integers and a target sum (`target_sum`).
2. **Initialization**

   ```
   count = 0
   seen = {}
   ```

   - `count` keeps track of how many pairs sum to `target_sum` .
   - `seen` is a dictionary (hash map) to store the frequency of each number encountered so far in `arr` .
3. **Iterating Over Each Number**

   ```
   for num in arr:
       complement = target_sum - num
       ...
   ```

   - We loop through each integer `num` in `arr` .
   - `complement` is the value needed so that `num + complement = target_sum` .

4. **Checking for Complement**

```python
if complement in seen:
    count += seen[complement]
```

   - If the needed `complement` has already been seen, it means we can form a valid pair with the current `num`.
   - We increase `count` by the frequency of `complement` in `seen` (because there might be multiple occurrences of `complement`).

5. **Updating the `seen` Dictionary**

```python
if num in seen:
    seen[num] += 1
else:
    seen[num] = 1
```

   - After checking for `complement`, we record the current `num` in the dictionary.
   - If `num` is already in `seen`, we increment its count.
   - Otherwise, we add it to `seen` with an initial frequency of 1.

6. **Printing the Result**

```python
print(count)
```

   - After processing the entire array, the total count of pairs whose sum equals `target_sum` is printed.

How It Works Overall

- As we iterate through `arr`, we use the `seen` dictionary to look up how many times the matching `complement` (i.e., `target_sum - num`) has already appeared.
- Each time we find that `complement` exists, we add the frequency of `complement` to our `count`.
- By the end of the loop, `count` contains the number of distinct pairs that sum to `target_sum`.
- This approach runs in **O(n)** time, since we only make a single pass through the array, using constant-time dictionary lookups.

Question : Middle Node

```
# Definition of Linked List Node

# class Node:
#       def __init__(self, data):
#               self.data = data
#               self.next = None

# Complete the function below

class Solution:
    def middleNode(self, head):

        slow_ptr = head
        fast_ptr = head
        while fast_ptr is not None and fast_ptr.next is not None:
            slow_ptr = slow_ptr.next
            fast_ptr = fast_ptr.next.next
        return slow_ptr.data
```

Explanation of the Code

1. **Pointers Initialization**

   ```
   slow_ptr = head
   fast_ptr = head
   ```

   - We initialize two pointers:
     - `slow_ptr` moves one node at a time.
     - `fast_ptr` moves two nodes at a time.
2. **Traversing the List**

   ```
   while fast_ptr is not None and fast_ptr.next is not None:
       slow_ptr = slow_ptr.next
       fast_ptr = fast_ptr.next.next
   ```

   - We continue iterating as long as `fast_ptr` and `fast_ptr.next` are valid (i.e., not `None`).

- In each loop:
    - `slow_ptr` advances by one node (`slow_ptr = slow_ptr.next`).
    - `fast_ptr` advances by two nodes (`fast_ptr = fast_ptr.next.next`).
3. **Stopping Condition**
    - When `fast_ptr` reaches the end of the list (or the node before the last, depending on the list length), the loop stops.
    - At this point, `slow_ptr` will be at the **middle** of the list.
4. **Returning the Middle Node’s Data**

    ```
    return slow_ptr.data
    ```

    - We return the data value (`slow_ptr.data`) of the middle node.

How the Fast & Slow Pointer Method Works

- By moving one pointer (`slow_ptr`) at half the speed of the other (`fast_ptr`), by the time the faster pointer reaches the end, the slower pointer will be exactly in the middle.
- This approach is **O(n)** in time complexity (we only do one pass through the list) and **O(1)** in extra space.

Question : War of Minions

```python
def remainingMinions(n, s):
    st = []
    for i in range(n):
        if not st:
            st.append(s[i])
        else:
            if st[-1] == s[i]:
                st.pop()
            else:
                st.append(s[i])
    if not st:
        return "-1"
    else:
        str = ''.join(st[::-1])
        return str[::-1]

n = int(input())
s = input()
print(remainingMinions(n, s))
```

Explanation

1. **Function Definition:** `remainingMinions(n, s)`
    - The function takes:
        - `n` : The length of the string.
        - `s` : A string of length `n` representing minions.
2. **Using a Stack** `st`

    ```python
    st = []
    ```

    - We use `st` as a stack to keep track of characters.
3. **Iterating Through Each Character**

```
for i in range(n):
    if not st:
        st.append(s[i])
    else:
        if st[-1] == s[i]:
            st.pop()
        else:
            st.append(s[i])
```

- **Case 1: `st` is empty** (`not st`):
    - Push the current character (`s[i]`) onto the stack.
- **Case 2: `st` is not empty**:
    - Compare the top of the stack (`st[-1]`) with the current character (`s[i]`).
        - If they are the **same**, pop the top of the stack (they “cancel out”).
        - If they are **different**, push the new character onto the stack.

> **Interpretation**:
> This process repeatedly removes adjacent pairs of the same character. If the top of the stack is the same as the new character, we pop; otherwise, we push.

4. **Checking if the Stack is Empty**

```
if not st:
    return "-1"
```

- After processing all characters, if the stack is empty, we return `"-1"` , meaning no minion survived.

5. **Returning the Remaining Minions**

```
else:
    str = ''.join(st[::-1])
    return str[::-1]
```

- If the stack is not empty, we first reverse the stack (`st[::-1]`), join it into a string, and then reverse that string again.
- Mathematically, reversing twice gets us back to the original order of `st` . Thus, effectively, the code returns all surviving minions in the order they ended up in the stack.

6. **Main Execution**

```
n = int(input())
s = input()
print(remainingMinions(n, s))
```

  - We read `n`, the length of the string, and then the string `s`.
  - We call `remainingMinions(n, s)` and print the result.

How It Works Overall

- As we iterate through the string `s`, we use a stack-based cancellation approach:
  - If the top of the stack and the new character are the same, they cancel out and we pop from the stack.
  - Otherwise, we push the new character.
- In the end, any characters left in the stack are the “surviving” minions.
- If no characters survive, we return `"-1"`. Otherwise, we return the string formed by the stack.