Module 2 Quiz 1 Editorial

1. Question:

An algorithm takes a constant time C for every operation and performs n^2 operations. In Big O notation, its time complexity is:

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

Correct Answer: b)

Explanation:
If an algorithm performs n^2 operations, and each operation is constant time C, the total time complexity is proportional to n^2.

2. Question:

What is the primary difference between Selection Sort and Bubble Sort in terms of how they operate on the list?

a) Selection Sort swaps only adjacent elements, while Bubble Sort can swap any two elements
b) Bubble Sort always swaps the first and last elements, while Selection Sort swaps all elements
c) Selection Sort selects the largest element and places it in its correct position, while Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order
d) There is no difference; both algorithms operate identically

Correct Answer: c)

Explanation:
We look for the largest element during each pass and move it to its correct position at the end of the list. By contrast, Bubble Sort continually compares adjacent pairs and swaps them if they are out of order, effectively “bubbling” the largest element to the end in each pass.

3. Question:

Which of the following best describes a stack?

a) First In First Out (FIFO)
b) Last In First Out (LIFO)
c) Both FIFO and LIFO
d) None of the above

Correct Answer: b)

Explanation:
A stack is a LIFO structure, meaning the last item pushed (added) onto the stack is the first one to be popped (removed).

4. Question:

If you are implementing two stacks in a single list (array), one from the front and one from the back, what indicates that the stack space is fully used?

a) When the front stack has more elements than the back stack
b) When the back stack has more elements than the front stack
c) When the top of the front stack meets the top of the back stack
d) When the list is empty

Correct Answer: c)

Explanation:
If two stacks grow towards each other within the same list, the space is fully used (or an overflow condition) when the top pointers meet or cross.

5. Question:

Consider the following Python code snippet for Bubble Sort on a list named `arr` of length `n`:

```
for i in range(n):
    for j in range(0, n - i - 1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
```

What is the time complexity of this Bubble Sort algorithm in the worst case?

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

Correct Answer: c)

Explanation:
Bubble Sort compares adjacent pairs in nested loops, leading to an $O(n^2)$ time complexity in the worst case.

Question : Flower Management

```
tc = int(input())
for _ in range(tc):
    m, n = list(map(int, input().split()))
    arr = list(map(int, input().split()))

    count = 0
    i = 0

    while i < m:
        if arr[i] == 0:
            # Check if the left plot is empty or doesn't exist (i.e., i == 0)
            left = (i == 0 or arr[i - 1] == 0)

            # Check if the right plot is empty or doesn't exist (i.e., i == m - 1)
            right = (i == m - 1 or arr[i + 1] == 0)

            # If both left and right plots are empty, place a flower here
            if left and right:
                count += 1
                # Move 2 steps ahead to avoid placing a flower in the adjacent plot
                i += 2
            else:
                i += 1
        else:
            i += 1

    if count >= n:
        print("Yes")
    else:
        print("No")
```

Explanation of the Code

1. **Reading the Number of Test Cases**

```
tc = int(input())
```

- This reads an integer `tc` representing how many times we need to run the flower-placing logic.

2. **Looping Over Each Test Case**

```
for _ in range(tc):
    m, n = list(map(int, input().split()))
    arr = list(map(int, input().split()))
```

- For each test case:
  - `m` is the size of the flowerbed array (the number of plots).
  - `n` is the number of new flowers we want to plant.
  - `arr` is a list of length `m`, where each element can be:
    - `0` (an empty plot),
    - `1` (a plot where a flower already exists).

3. **Initializing Counters**

```
count = 0
i = 0
```

- `count` will track how many flowers we can successfully place.
- `i` will be used to iterate through the `arr` list.

4. **While Loop to Check Plots**

```
while i < m:
    if arr[i] == 0:
        ...
    else:
        i += 1
```

- We iterate through each plot in `arr` using `i` as the current index.

5. **Checking Empty Plots**

```
if arr[i] == 0:
    left = (i == 0 or arr[i - 1] == 0)
    right = (i == m - 1 or arr[i + 1] == 0)

    if left and right:
        count += 1
        i += 2
    else:
        i += 1
```

- When we encounter an empty plot (`arr[i] == 0`), we check the **left** and **right** neighbors:
  - **Left** is empty if `i == 0` (no left neighbor) or `arr[i - 1] == 0` .
  - **Right** is empty if `i == m - 1` (no right neighbor) or `arr[i + 1] == 0` .
- If both sides are empty, we can plant a flower at `i` , so we:
  - Increment `count` (we've planted one flower).
  - Jump `i` by 2 to skip the next index, ensuring we don't violate the "no adjacent flowers" rule.
- If the spot is not suitable, we simply move `i` by 1 to check the next plot.

6. **If the Plot Already Has a Flower**

```
else:
    i += 1
```

- If `arr[i] == 1` , we just move to the next index, since we cannot place a flower here.

7. **Check Final Flower Count**

```
if count >= n:
    print("Yes")
else:
    print("No")
```

- After processing the entire flowerbed, if the number of flowers we managed to place (`count`) is **at least** `n` , we print `"Yes"` .
- Otherwise, we print `"No"` .

How It Works Overall

- For each test case, we read the flowerbed configuration and how many new flowers we need to plant.
- We scan through the array:
  - Whenever we find an empty spot, we check if it's valid to place a flower (i.e., its neighbors are also empty).
  - If valid, we place a flower and skip the next spot to avoid adjacency conflicts.
- By the end, if we have placed **at least** as many flowers as required, we print `"Yes"` ; otherwise, `"No"` .

Question : New Year Celebration

```
cold = []
ice = []
for i in range(int(input())):
    x = list(map(int, input().split()))
    if len(x) == 2:
        if x[0] == 1:
            ice.append(x[1])
        else:
            cold.append(x[1])
    else:
        if x[0] == 3:
            if len(ice) == 0:
                print(-1)
            else:
                print(ice[0])
        elif x[0] == 4:
            if len(cold) == 0:
                print(-1)
            else:
                print(cold[-1])
        else:
            if len(ice) == 0:
                print(-1)
            else:
                cold.append(ice.pop(0))
```

Explanation

1. **List Initialization**

   ```
   cold = []
   ice = []
   ```

   - We have two lists:
     - `ice` to represent a **queue** (FIFO structure) for ice-cream orders.
     - `cold` to represent a **stack** (LIFO structure) for cold drinks.

2. **Reading the Number of Queries**

```
for i in range(int(input())):
```

- The code first reads an integer which tells us how many queries (operations) we need to process.

3. **Reading Each Query**

```
x = list(map(int, input().split()))
```

- Each query is read as a list of integers.
- Depending on the length of `x`, we can have different types of operations:
  - If `len(x) == 2`, it means the query is either `1 X` or `2 X`.
  - Otherwise, if `len(x) == 1`, it means the query is one of `3`, `4`, or `5`.

4. **Handling `len(x) == 2`**

```
if len(x) == 2:
    if x[0] == 1:
        ice.append(x[1])
    else:
        cold.append(x[1])
```

- **Query `1 X`**: Add item `X` to the `ice` list (queue).
  - This simulates enqueueing `X` to the queue.
- **Query `2 X`**: Add item `X` to the `cold` list (stack).
  - This simulates pushing `X` onto the stack.

5. **Handling `len(x) == 1`**
   - Here, `x[0]` can be `3`, `4`, or `5`.

a. Query `3`:

```
if x[0] == 3:
    if len(ice) == 0:
        print(-1)
    else:
        print(ice[0])
```

- If the queue `ice` is empty, print `-1`.
- Otherwise, print the front element of the queue (which is `ice[0]`).

b. Query `4`:

```
elif x[0] == 4:
    if len(cold) == 0:
        print(-1)
    else:
        print(cold[-1])
```

- If the stack `cold` is empty, print `-1`.
- Otherwise, print the top element of the stack (which is `cold[-1]`).

c. Query `5` (implicitly the `else` case):

```
else:
    if len(ice) == 0:
        print(-1)
    else:
        cold.append(ice.pop(0))
```

- If the queue `ice` is empty, print `-1`.
- Otherwise, remove the front item from the queue (`ice.pop(0)`) and **push** it onto the `cold` stack (`cold.append(...)`).

How It Works Overall

- We maintain two data structures:
  - **`ice`** (list used as a queue): Items are added to the end (`ice.append(...)`) and removed from the front (`ice.pop(0)`).
  - **`cold`** (list used as a stack): Items are added to the end (`cold.append(...)`) and accessed/removed from the end (`cold[-1]`).
- **Operations**:
  i. **`1 X`**: Enqueue `X` to `ice`.
  ii. **`2 X`**: Push `X` onto `cold`.
  iii. **`3`**: Print the front of `ice` (or `-1` if empty).
  iv. **`4`**: Print the top of `cold` (or `-1` if empty).
  v. **`5`**: Pop from the front of `ice` and push onto `cold` (or `-1` if `ice` is empty).
- This setup simulates two counters: one for ice-cream (`ice` queue) and one for cold drinks (`cold` stack). Queries allow adding to each counter, peeking at them, or transferring from the ice queue to the cold stack.

Question : Rotate Linked List

```
# Definition of Linked List Node

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

# Complete the function below

class Solution:

    def rotateRight(self,head,k):

        n = 0;
        temp = head;
        while(temp):
            temp = temp.next;
            n += 1;

        # Reduce k so it doesn't exceed the length of the list
        k = k % n;

        # Perform the rotation k times
        for i in range(k):
            prev = None;
            curr = head;

            # Traverse to the end of the list
            while(curr.next != None):
                prev = curr;
                curr = curr.next;

            # 'curr' is now the last node, 'prev' is the node before the last
            prev.next = None;  # Break the link before the last node
            curr.next = head;  # Link the last node to the front
            head = curr;       # Update the head to be the last node

        return head;
```

Explanation

1. **Finding the Length of the List**

```
n = 0;
temp = head;
while(temp):
    temp = temp.next;
    n += 1;
```

- We initialize a counter `n` to `0`.
- We traverse the linked list using a temporary pointer `temp`, incrementing `n` for each node until we reach the end.
- After this loop, `n` holds the total number of nodes in the linked list.

2. **Adjusting `k`**

```
k = k % n;
```

- If `k` is larger than the length of the list, rotating the list more than `n` times would be redundant (every `n` rotations brings the list back to its original state).
- We therefore take `k` modulo `n` to handle cases where `k >= n`.

3. **Rotating the List `k` Times**

```
for i in range(k):
    prev = None;
    curr = head;
    while(curr.next != None):
        prev = curr;
        curr = curr.next;

    prev.next = None;
    curr.next = head;
    head = curr;
```

- **Outer Loop** (`for i in range(k)`): We will perform the 1-step rotation process exactly `k` times.
- **Inner Loop**: We start from the head (`curr = head`) and move forward until `curr.next` is `None`.
  - At the end of this traversal, `curr` points to the **last node** in the list, and `prev` points to the node just before the last node.
- **Re-linking**:

- `prev.next = None;` removes the last node from its current position by making the second-to-last node the new tail.
- `curr.next = head;` places the last node at the **front** of the list.
- `head = curr;` updates the list's head pointer to this newly moved node.

4. **Return the New Head**

```
return head;
```

- After `k` such single-step rotations, we return the updated `head` of the list.

Time Complexity Note

- This approach performs a **1-step rotation** `k` times. Each 1-step rotation requires traversing the list (in the worst case), leading to a time complexity of approximately **O(n*k)**.
- There is a more optimal solution that runs in **O(n)** by connecting the tail to the head to form a cycle, then breaking at the correct spot. However, the above code correctly implements a simpler, step-by-step rotation approach.