Solution Review: Problem Challenge 3

We'll cover the following

Frequency Stack (hard)

Design a class that simulates a Stack data structure, implementing the following two operations:

1. `push(int num)`: Pushes the number 'num' on the stack.
2. `pop()`: Returns the most frequent number in the stack. If there is a tie, return the number which was pushed later.

Example:

```
After following push operations: push(1), push(2), push(3), push(2), push(1), push(2), push(5)

1. pop() should return 2, as it is the most frequent number
2. Next pop() should return 1
3. Next pop() should return 2
```

Solution

This problem follows the Top 'K' Elements pattern, and shares similarities with Top 'K' Frequent Numbers.

We can use a **Max Heap** to store the numbers. Instead of comparing the numbers we will compare their frequencies so that the root of the heap is always the most frequently occurring number. There are two issues that need to be resolved though:

1. How can we keep track of the frequencies of numbers in the heap? When we are pushing a new number to the **Max Heap**, we don't know how many times the number has already appeared in the **Max Heap**. To resolve this, we will maintain a **HashMap** to store the current frequency of each number. Thus whenever we push a new number in the heap, we will increment its frequency in the HashMap and when we pop, we will decrement its frequency.
2. If two numbers have the same frequency, we will need to return the number which was pushed later while popping. To resolve this, we need to attach a sequence number to every number to know which number came first.

In short, we will keep three things with every number that we push to the heap:

```
1. number // value of the number
2. frequency // current frequency of the number when it was pushed to the heap
3. sequenceNumber // a sequence number, to know what number came first
```

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```python
from heapq import *


class Element:

  def __init__(self, number, frequency, sequenceNumber):
    self.number = number
    self.frequency = frequency
    self.sequenceNumber = sequenceNumber

  def __lt__(self, other):
    # higher frequency wins
    if self.frequency != other.frequency:
      return self.frequency > other.frequency
    # if both elements have same frequency, return the element that was pushed later
    return self.sequenceNumber > other.sequenceNumber


class FrequencyStack:
  sequenceNumber = 0
  maxHeap = []
  frequencyMap = {}

  def push(self, num):
    self.frequencyMap[num] = self.frequencyMap.get(num, 0) + 1
    heappush(self.maxHeap, Element(
      num, self.frequencyMap[num], self.sequenceNumber))
    self.sequenceNumber += 1

  def pop(self):
    num = heappop(self.maxHeap).number
    # decrement the frequency or remove if this is the last number
    if self.frequencyMap[num] > 1:
      self.frequencyMap[num] -= 1
    else:
      del self.frequencyMap[num]

    return num


def main():
  frequencyStack = FrequencyStack()
  frequencyStack.push(1)
  frequencyStack.push(2)
  frequencyStack.push(3)
  frequencyStack.push(2)
  frequencyStack.push(1)
  frequencyStack.push(2)
  frequencyStack.push(5)
  print(frequencyStack.pop())
  print(frequencyStack.pop())
  print(frequencyStack.pop())


main()

```

Run Save Reset

Time complexity

The time complexity of `push()` and `pop()` is $O(logN)$ where 'N' is the current number of elements in the heap.

Space complexity

We will need $O(N)$ space for the heap and the map, so the overall space complexity of the algorithm is $O(N)$.