1700. Number of Students Unable to Eat Lunch

Easy Stack Queue Array Simulation

Leetcode Link

Problem Description

In this problem, we have a queue of students and a stack of sandwiches. The sandwiches can be of two types, represented by `0` (circular) and `1` (square). Each student in the queue has a preference for either circular or square sandwiches.

Students are served in the order they are standing in the queue. If the student at the front of the queue prefers the type of sandwich that is on top of the sandwich stack, they take the sandwich and leave the queue. However, if they do not prefer the type of sandwich on top of the stack, they pass on the sandwich and move to the end of the queue.

The process stops when the sandwich on the top of the stack is not preferred by any of the students left in the queue, meaning no more sandwiches can be served, and some students will be unable to eat. The task is to find the number of students who will not be able to eat a sandwich.

Intuition

The key insight into solving this problem lies in recognizing that if a certain type of sandwich could not be taken by any student in the queue, no further progress can be made. This is because the students' order and preferences do not change.

We start with two counts: how many students prefer circular sandwiches and how many prefer square ones. We process the sandwich stack from the top (`i = 0` being the top of the stack). For each sandwich, we check if there is a student in the queue who prefers that type of sandwich. If there is, we decrease the count of students waiting for that type of sandwich. If there isn't, it means all remaining students prefer the other type of sandwich and will be unable to eat. At this point, we can return the number of students who prefer the other type of sandwich.

This approach ensures we check the selection possibility for each sandwich without actually moving students around in the queue, which results in a time-efficient solution.

The solution implementation is straightforward: we keep counts of students' preferences and iterate through the sandwich stack. When we encounter a sandwich with no matching student preference, we stop and return the count of the other type of sandwich. If we go through the whole stack, it means all students got their preferred sandwich, and we return `0`.

Solution Approach

The solution provided uses the `Counter` class from Python's `collections` module to efficiently keep track of the count of students' sandwich preferences.

Here's a step-by-step explanation of the implementation:

1. A `Counter` object called `cnt` is created from the `students` list. This will give us a dictionary-like object where the keys are `0` and `1`, and the values are the count of students preferring circular (`0`) and square (`1`) sandwiches respectively.
2. The solution then iterates over the `sandwiches` stack with a for-loop. For each sandwich type `v` at the top of the stack:

   a. The solution checks if `cnt[v]` equals `0`. If it is, this means there are no students left that prefer the current type of sandwich on top of the stack. Since the students who prefer the other type of sandwich won't take the current one, the solution returns `cnt[v ^ 1]`. The expression `v ^ 1` is a bitwise XOR operation that toggles between `0` and `1`, effectively giving us the count of the other sandwich preference.

   b. If `cnt[v]` does not equal `0`, which means there is at least one student who prefers the type of sandwich on top of the stack, the solution decrements the count of that sandwich preference by `1` since that student takes the sandwich and leaves the queue.
3. After the for-loop, if all sandwiches have been successfully served to students, the final return value is `0`, meaning no students are left unable to eat.

The `for-loop` in this approach works effectively because it processes the sandwiches in the natural order they are being served, and the `Counter` keeps track of the dynamic preference count without the need to manage the queue of students manually.

By running this efficient loop and conditions, the algorithm ensures a linear time complexity proportional to the number of sandwiches (or equivalently the number of students), which is $O(n)$ where n is the number of students or the length of the `sandwiches` array.

Therefore, the algorithm and the usage of the `Counter` data structure work together to provide a solution that is straightforward, readable, and efficient in both time and space.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Suppose we have the following queue of students and stack of sandwiches:

```
1 Students preferences (queue): [1, 0, 0, 1, 1]
2 Sandwiches stack (top to bottom): [0, 1, 0, 1, 1]
```

Each `1` in the students' queue represents a student who prefers square sandwiches and each `0` represents a student who prefers circular sandwiches.

Step 1:

- We initialize a `Counter` object from the students' preferences: `cnt = Counter([1, 0, 0, 1, 1])`, resulting in `cnt = {0: 2, 1: 3}`. This means 2 students prefer circular sandwiches and 3 prefer square ones.

Step 2:

- Now, we start iterating over the `sandwiches` stack from the top:

Iteration 1:

- The top of the stack is a circular sandwich (`0`). `cnt[0]` is not `0`, so there's a student who prefers this sandwich.
- We decrement `cnt[0]` by `1`, leaving `cnt` as `{0: 1, 1: 3}`.

Iteration 2:

- Next sandwich is square (`1`). `cnt[1]` is not `0`, indicating a matching student preference.
- We decrement `cnt[1]` by `1`, making `cnt {0: 1, 1: 2}`.

Iteration 3:

- Another circular sandwich (`0`) is on top. `cnt[0]` is not `0`, so a student will take this sandwich.
- We decrement `cnt[0]` by `1` and now `cnt` is `{0: 0, 1: 2}`.

Iteration 4:

- The next sandwich is square (`1`). `cnt[1]` is not `0`, so a student prefers and takes it.
- We decrement `cnt[1]` to `1`, updating `cnt` to `{0: 0, 1: 1}`.

Iteration 5:

- The last sandwich is also square (`1`). `cnt[1]` is not `0`.
- We decrement `cnt[1]` to `0`, final `cnt` is `{0: 0, 1: 0}`.

Step 3:

- We have successfully iterated over all the sandwiches, and the `cnt` for both types of sandwiches is `0`, meaning each student got their preference.

Final step:

- The final return value is `0` since no students are left unable to eat.

This small example demonstrates how we can efficiently determine that all students will be able to eat by just keeping track of the counts of their sandwich preferences and decrementing the respective counts as we match students with sandwiches. No actual rearrangement of the queue is required. If at any point we discovered a sandwich on top that matches none of the students' remaining preferences, we would return the number of students left with the non-matching preference immediately.

Python Solution

```python
from collections import Counter

class Solution:
    def countStudents(self, students: List[int], sandwiches: List[int]) -> int:
        # Count the number of students preferring each type of sandwich
        count = Counter(students)

        # Loop through the sandwiches queue
        for sandwich in sandwiches:
            # If no student wants the current type of sandwich,
            # return the number of students who want the other type
            if count[sandwich] == 0:
                # The 'sandwich ^ 1' toggles between 1 and 0
                return count[sandwich ^ 1]

            # Decrement the count for the current sandwich type
            count[sandwich] -= 1

        # If all sandwiches match the students' preferences return 0
        return 0
```

Java Solution

```java
class Solution {
    public int countStudents(int[] students, int[] sandwiches) {
        // Array to count the number of students who prefer each type of sandwich
        // cnt[0] for students who prefer sandwich type 0, and cnt[1] for sandwich type 1
        int[] count = new int[2];

        // Count the number of students preferring each type of sandwich
        for (int student : students) {
            count[student]++;
        }

        // Iterate through the sandwiches
        for (int sandwich : sandwiches) {
            // If no students want the current type of sandwich, break and return the number of students
            // who want the other type of sandwich
            if (count[sandwich] == 0) {
                // The other type is obtained by XORing the current type with 1
                // If current is 0, 0^1 will be 1 (the other type)
                // If current is 1, 1^1 will be 0 (the other type)
                return count[sandwich ^ 1];
            }
            // Decrement the count as one student takes a sandwich of the type they prefer
            count[sandwich]--;
        }

        // If all students got their preferred sandwiches, return 0
        return 0;
    }
}
```

C++ Solution

```cpp
#include <vector> // make sure to include the vector header for vector support

class Solution {
public:
    // Function to count the number of students who can't get a sandwich
    int countStudents(vector<int>& students, vector<int>& sandwiches) {
        // Initialize a count array to store the number of students preferring each type of sandwich
        int count[2] = {0};

        // Count the number of students preferring each type of sandwich (0 or 1)
        for (int student : students) {
            count[student]++;
        }

        // Iterate through the sandwiches queue
        for (int sandwich : sandwiches) {
            // If no students prefer the current type of sandwich, break and return the count of the other type
            if (count[sandwich] == 0) {
                return count[1 - sandwich]; // use 1 - sandwich to get the opposite type
            }
            // Otherwise, decrement the count of students who prefer the current type of sandwich
            count[sandwich]--;
        }

        // If all sandwiches can be taken by students, return 0
        return 0;
    }
};
```

Typescript Solution

```typescript
// Function to count the number of students who will not be able to get their preferred sandwich
function countStudents(students: number[], sandwiches: number[]): number {
    // Initialize a count array where index 0 represents students who prefer sandwich type 0,
    // and index 1 represents students who prefer sandwich type 1.
    const sandwichPreferenceCount = [0, 0];

    // Count the number of students who prefer each type of sandwich
    for (const studentPreference of students) {
        sandwichPreferenceCount[studentPreference]++;
    }

    // Iterate over the sandwich types in the order they are to be served
    for (const sandwichType of sandwiches) {
        // If there are no students left preferring the current sandwich type,
        // return the count of students preferring the other type
        if (sandwichPreferenceCount[sandwichType] === 0) {
            return sandwichPreferenceCount[sandwichType ^ 1];
        }
        // Otherwise, decrement the count of students who prefer the current sandwich type
        sandwichPreferenceCount[sandwichType]--;
    }

    // If all students can be served with their preferred sandwich type,
    // return 0 indicating no students are left unserved
    return 0;
}
```

Time and Space Complexity

The given Python code snippet implements a function to count the number of students who won't be able to eat their preferred type of sandwich. The time and space complexities of the code are as follows:

- **Time complexity:**

The function iterates over each sandwich in the `sandwiches` list, which has a time complexity of $O(n)$, where n is the number of sandwiches (or students, since they are the same). The operation `cnt[v]` and the decrement operation `cnt[v] -= 1` both operate in constant time, $O(1)$. Therefore, the overall time complexity of the function is $O(n)$.

- **Space complexity:**

The space complexity is primarily defined by the space required to store the counter for the students. The `Counter` object stores an entry for each unique value in the `students` list, which, in this case, has only two possible values (0 and 1). Therefore, the `Counter` object space is constant, $O(1)$. Hence, the space complexity of the function is $O(1)$.