2611. Mice and Cheese

Medium Greedy Array Sorting Heap (Priority Queue) Leetcode Link

Problem Description

The problem describes a scenario with two mice and n types of cheese. Each type of cheese has a specific point value assigned to it, which varies depending on which mouse eats it (`reward1` for the first mouse and `reward2` for the second mouse). The objective is to maximize the total points scored by both mice under the constraint that the first mouse must eat exactly k types of cheese, with each cheese type being consumed by only one of the mice. The inputs provided are two arrays `reward1` and `reward2`, representing the point values for each type of cheese for each mouse, along with an integer k that specifies the number of cheese types the first mouse must eat.

The goal of the problem is to determine the maximum number of points that can be achieved by optimally distributing the cheese between the two mice.

Intuition

To arrive at the solution, we can use a greedy approach that aims to maximize the total points gained by both mice each time we decide which mouse eats a particular cheese. This means we need to make choices that will benefit our total score the most at each step.

The key insight is to look at the difference in points each mouse would gain for each type of cheese, i.e., `reward1[i] - reward2[i]`. If this difference is large, it means that giving the cheese to the first mouse is very advantageous compared to giving it to the second mouse, and vice versa.

By sorting the cheese in descending order of this difference, we can ensure that the first mouse eats the cheese that provides the greatest relative advantage over the second mouse. This way, when we give k types of cheese to the first mouse, we're maximizing the point difference that those k types contribute.

Once we sort the cheese accordingly, the first k pieces in this order are eaten by the first mouse, and the rest are eaten by the second mouse. This strategy ensures that the accumulated points are maximized, resulting in the highest possible score.

The given Python code implements this intuition by first sorting the cheese indices based on the point difference and calculating the total points by summing up the appropriate `reward1` and `reward2` values according to the sorted indices.

Solution Approach

The Python solution utilizes a greedy algorithm combined with sorting to decide the optimal distribution of cheese between the two mice. Let's walk through the algorithm step by step:

1. Calculate the difference in score for each cheese when eaten by the first mouse compared to the second mouse. This is done by subtracting the value of `reward2[i]` from `reward1[i]` for each cheese `i`.
2. Sort the indices of the cheese based on this difference in descending order. The sorting criterion, specifically `lambda i: reward1[i] - reward2[i]`, means that the cheese types are ordered such that those with the higher score for the first mouse come first.
3. Determine which cheese should be eaten by which mouse:
   - The first k cheeses (after sorting) will be eaten by the first mouse because they represent the cases where the first mouse gains significantly compared to the second mouse. The sum of `reward1[i]` for these k cheeses is calculated using `sum(reward1[i] for i in idx[:k])`.
   - The remaining cheeses are eaten by the second mouse. The sum of `reward2[i]` for the rest is calculated using `sum(reward2[i] for i in idx[k:])`.
4. Add the points from step 3 to get the total maximum points.

No additional data structures are needed other than the list used for sorting the indices. The pattern used in the solution is straightforward as it revolves around the greedy choice of optimizing local decisions (which mouse should eat each cheese) to lead to a globally optimized solution (maximum points).

The overall time complexity of the algorithm is determined mainly by the sorting step, which is O(n log n), where n is the number of cheeses. The rest of the operations are linear in complexity, resulting in an efficient solution approach that scales well with the number of cheeses.

In summary, the implementation details involve creating a sorted list of indices based on the difference in points and utilizing a greedy strategy to maximize the total points by distributing the cheeses in an optimal manner.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose there are five types of cheese and the first mouse must eat exactly $k=2$ types of cheese. The points values are as follows:

- For the first mouse (`reward1`): [5, 20, 15, 10, 25]
- For the second mouse (`reward2`): [10, 30, 5, 15, 10]

Following the steps described in the solution approach:

1. We first calculate the score difference for each cheese:
   - Cheese 1: 5 - 10 = -5
   - Cheese 2: 20 - 30 = -10
   - Cheese 3: 15 - 5 = 10
   - Cheese 4: 10 - 15 = -5
   - Cheese 5: 25 - 10 = 15

   The difference array is: [-5, -10, 10, -5, 15].

2. Next, we sort the indices of the cheeses based on the score difference in descending order:
   - Cheese 5: 15
   - Cheese 3: 10
   - Cheese 1: -5
   - Cheese 4: -5
   - Cheese 2: -10

   The sorted indices are: [4, 2, 0, 3, 1].

3. Now, we assign cheese to each mouse based on the sorted indices:
   - The first mouse eats cheeses 5 and 3 ($k=2$), so the points for the first mouse are: `reward1[4] + reward1[2] = 25 + 15 = 40`.
   - The second mouse eats the remaining cheeses (1, 4, 2), so the points for the second mouse are: `reward2[0] + reward2[3] + reward2[1] = 10 + 15 + 30 = 55`.
4. The total maximum points that can be achieved by optimally distributing the cheese between the two mice is the sum of the points from step 3: `40 (first mouse) + 55 (second mouse) = 95 points`.

The example demonstrates that by following the solution approach, the total points scored by both mice have been maximized through an optimal distribution of the cheese types, in accordance with their preferences.

Python Solution

```python
from typing import List

class Solution:
    def miceAndCheese(self, reward1: List[int], reward2: List[int], k: int) -> int:
        # Determine the number of cheese pieces
        num_cheese_pieces = len(reward1)

        # Sort the indices based on the difference between rewards from reward1 and reward2
        # Sorting in descending order since we take the largest differences first
        sorted_indices = sorted(range(num_cheese_pieces), key=lambda i: reward1[i] - reward2[i], reverse=True)

        # Calculate the total reward for the first 'k' mice using reward1
        reward_for_first_k = sum(reward1[i] for i in sorted_indices[:k])

        # Calculate the total reward for the rest of the mice using reward2
        reward_for_remaining = sum(reward2[i] for i in sorted_indices[k:])

        # Return the sum of rewards for all mice
        return reward_for_first_k + reward_for_remaining
```

Java Solution

```java
class Solution {
    public int miceAndCheese(int[] rewardsPath1, int[] rewardsPath2, int k) {
        // Get the number of positions where rewards are placed
        int positionsCount = rewardsPath1.length;

        // Create an array to store the indices of the positions
        Integer[] positionIndices = new Integer[positionsCount];
        for (int i = 0; i < positionsCount; ++i) {
            positionIndices[i] = i;
        }

        // Sort the indices based on the difference between the two paths
        Arrays.sort(positionIndices, (idx1, idx2) -> rewardsPath1[idx2] - rewardsPath2[idx2] - (rewardsPath1[idx1] - rewardsPath2[idx

        // Initialize the answer to start calculating the maximum reward
        int maxReward = 0;

        // Choose the first 'k' rewards from Path 1 based on the sorted indices
        for (int i = 0; i < k; ++i) {
            maxReward += rewardsPath1[positionIndices[i]];
        }

        // Choose the remaining rewards from Path 2
        for (int i = k; i < positionsCount; ++i) {
            maxReward += rewardsPath2[positionIndices[i]];
        }

        // Return the total maximum reward that can be obtained
        return maxReward;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int miceAndCheese(vector<int>& rewards1, vector<int>& rewards2, int k) {
        int size = rewards1.size(); // Store the size of the rewards vectors

        // Create a vector to track indices and initialize it
        vector<int> indices(size);
        iota(indices.begin(), indices.end(), 0);  // iota fills indices with sequential values starting with 0

        // Sort the indices based on the difference in rewards between the first mouse and the second mouse
        sort(indices.begin(), indices.end(), [&](int i, int j) {
            // Compare differences between rewards for two indices i, j
            return rewards1[i] - rewards2[i] > rewards1[j] - rewards2[j];
        });

        int totalReward = 0;  // Initialize the total reward accumulated

        // Accumulate the maximum possible rewards by selecting k mice for rewards1 vector
        for (int i = 0; i < k; ++i) {
            totalReward += rewards1[indices[i]];
        }

        // Accumulate the remaining rewards by selecting mice for rewards2 vector
        for (int i = k; i < size; ++i) {
            totalReward += rewards2[indices[i]];
        }

        return totalReward;  // Return the total accumulated reward
    }
};
```

Typescript Solution

```typescript
function miceAndCheese(reward1: number[], reward2: number[], k: number): number {
    // Determine the length of the reward arrays
    const length = reward1.length;

    // Create an index array to keep track of the original positions
    const indices = Array.from({ length: length }, (_, index) => index);

    // Sort the indices based on the difference between corresponding rewards
    // The sort will prioritize the mice with higher reward difference in reward1 over reward2
    indices.sort((indexA, indexB) => (reward1[indexB] - reward2[indexB]) - (reward1[indexA] - reward2[indexA]));

    let totalReward = 0; // Initialize total reward to 0

    // Collect the top 'k' rewards from reward1 based on the sorted index positions
    for (let i = 0; i < k; ++i) {
        totalReward += reward1[indices[i]];
    }

    // Collect the remaining rewards from reward2 based on the sorted index positions
    for (let i = k; i < length; ++i) {
        totalReward += reward2[indices[i]];
    }

    // Return the total reward collected by the mice
    return totalReward;
}
```

Time and Space Complexity

The time complexity of the given code is $O(n * \log n)$. This is because the main operation affecting the time complexity is the sorting of indexes. The `sorted()` function in Python uses the Timsort algorithm, which has a time complexity of $O(n * \log n)$ for the worst case. The key for the sorting operation is the difference between `reward1[i]` and `reward2[i]`, but this does not change the time complexity of the sort.

The space complexity of the code is $O(n)$. The extra space is mainly used to store the sorted indexes in the list `idx`. This list has the same length as the input lists (`reward1` and `reward2`), so it takes $O(n)$ space. The slicing operations and `sum()` calls do not add any additional space complexity beyond what is required for the output, and temporary variables used in the sum() function are considered $O(1)$ space.