3075. Maximize Happiness of Selected Children

Medium Greedy Array Sorting

Problem Description

In this problem, we are presented with `n` children standing in a queue, each with a certain "happiness" value. We aim to maximize the total happiness by selecting `k` children. Notably, each time a child is chosen, the happiness value of every unselected child decreases by `1`, if their happiness is positive. The challenge is that as we select children, the unselected ones become less happy, so we need to carefully pick the sequence to make the most out of the happiness we can gather. The goal is to devise a strategy to get the maximum sum of happiness values by selecting `k` children from the queue.

Intuition

To maximize the total happiness from selecting `k` children, we need to grab the opportunity of the highest happiness values first. One optimal strategy is to prioritize children with higher happiness values before their potential happiness diminishes due to the selection of others. This calls for a greedy approach where sorting comes to play a vital role.

By sorting the children in descending order based on their happiness, we ensure that we select the happiest child first. Remember, each time we pick a child, subsequent children's happiness reduces by `1`—but only if they're still positive. So, we greedily select from the top, decrementing the happiness value by the number of children already selected, symbolized by `i`.

The solution approach involves:

1. Sorting the children based on their happiness values in descending order.
2. Iteratively selecting `k` children from the sorted list.
3. For each selected child, computing the effective happiness as the current happiness minus the count of previously selected children.
4. Ensuring that if decrementing makes the happiness value negative, we consider it as zero since happiness cannot be negative by the problem's constraint.
5. Summing up all the effective happiness values of the selected children to achieve the maximum sum.

Implementing this approach yields the maximum sum of happiness we can obtain from the given queue of children.

Solution Approach

To implement the solution effectively, we use the following algorithmic steps and Python built-in features aligned with the Greedy approach and the specified behavior of the problem:

1. **Sorting**: We start by sorting the `happiness` array in descending order. This is crucial because we want to select children with higher happiness values first, as each selection will decrease the potential happiness we can collect from the remaining children.

   ```
   happiness.sort(reverse=True)
   ```

   By sorting in reverse order, we guarantee that `happiness[0]` holds the highest value, `happiness[1]` the second-highest, and so on.

2. **Iterative Selection**: We iterate over the first `k` children in the sorted `happiness` array. This is because, according to our Greedy strategy, these will be the children from whom we can extract the most happiness before their values start to decrease.

   ```
   for i, x in enumerate(happiness[:k]):
   ```

   Here, `enumerate` is a built-in Python function that provides both the index `i` and the value `x` during the iteration. The index `i` represents how many children have been previously selected and thus by how much the currently considered child's happiness will be reduced.

3. **Happiness Calculation and Summation**:
   - We calculate the adjusted happiness for each selected child. The adjustment is based on the assumption that each child's happiness decreases by `1` for every child already selected, which corresponds to the index `i`.
   - We use the `max` function to ensure that we do not consider negative happiness values because the problem states that happiness cannot be negative even after decrementing.

   ```
   x -= i
   ans += max(x, 0)
   ```

   The adjusted happiness (`x - i`) is added to the `ans`, which accumulates the total happiness over the selection of `k` children.

4. **Returning the Result**: After processing the `k` children, we have the maximum sum of happiness in `ans`, which we return as the result.

   ```
   return ans
   ```

Altogether, the algorithm uses sorting and a single pass through the first `k` elements of the sorted array to compute the maximum sum of happiness. The Greedy approach ensures that, at each step, we make the locally optimal choice by picking the child with the maximum possible happiness at that turn.

Example Walkthrough

Let's walk through an example to demonstrate the solution approach. Suppose we have `n = 5` children with happiness values given by the array `happiness = [5, 3, 1, 2, 4]`, and we want to maximize the total happiness by selecting `k = 3` children from the queue.

Here are the steps we'll follow, as per the solution approach:

Step 1: Sorting First, we sort the `happiness` array in descending order:

```
happiness.sort(reverse=True)  # [5, 4, 3, 2, 1]
```

Now, the `happiness` array becomes `[5, 4, 3, 2, 1]`.

Step 2: Iterative Selection Next, we iterate over the first `k` children in the sorted `happiness` array. Applying the Greedy strategy, we'll pick the children with the highest happiness first:

```
for i, x in enumerate(happiness[:3]):  # i goes from 0 to 2, x will be 5, then 4, then 3
```

Step 3: Happiness Calculation and Summation On each iteration, we calculate the effective happiness (`x - i`) and ensure it does not go below zero:

- First iteration (`i = 0`, `x = 5`): `x - i` is `5 - 0`, which equals `5`. So we add `5` to the total happiness.
- Second iteration (`i = 1`, `x = 4`): `x - i` is `4 - 1`, which equals `3`. We add `3` to the total happiness.
- Third iteration (`i = 2`, `x = 3`): `x - i` is `3 - 2`, which equals `1`. We add `1` to the total happiness.

The `ans` keeps accumulating these values, and we use the `max` function to add only non-negative values:

```
ans = 0
ans += max(5 - 0, 0)  # ans = 5
ans += max(4 - 1, 0)  # ans = 5 + 3 = 8
ans += max(3 - 2, 0)  # ans = 8 + 1 = 9
```

Step 4: Returning the Result After processing `k = 3` children, we have the total happiness:

```
return ans  # returns 9
```

The resulting maximum sum of happiness values for selecting `k = 3` children is `9`. The selection process prioritized children with the highest happiness value first and adjusted for each subsequent selection. This example confirms that our Greedy approach effectively maximizes the total happiness achieved.

Solution Implementation

Python

```python
class Solution:
    def maximumHappinessSum(self, happiness: List[int], k: int) -> int:
        # Sort the happiness values in descending order.
        happiness.sort(reverse=True)
        max_happiness = 0 # Initialize the maximum happiness sum.

        # Loop through the first k elements after sorting.
        for i in range(k):
            # Since a person's happiness value can decrease with each additional person,
            # we subtract the current index from the happiness value,
            # however, we should not go below 0.
            decreased_happiness = happiness[i] - i
            if decreased_happiness > 0:
                max_happiness += decreased_happiness

        # Return the sum of the maximized happiness values.
        return max_happiness
```

Java

```java
class Solution {
    public long maximumHappinessSum(int[] happiness, int k) {
        // Sort the array so that we can easily pick the largest elements
        Arrays.sort(happiness);

        // Initialize the sum of happiness to 0
        long totalHappiness = 0;

        // Iterate through the array starting from the end to get the largest values
        for (int i = 0; i < k; ++i) {
            // Calculate the current happiness after decrementing based on index i
            // Using n - 1 - i to pick the k largest elements in a sorted array
            int currentHappiness = happiness[happiness.length - 1 - i] - i;

            // Sum only the positive values of current happiness after the decrement
            totalHappiness += Math.max(currentHappiness, 0);
        }

        // Return the total happiness calculation
        return totalHappiness;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for std::sort

class Solution {
public:
    // Function to calculate the maximum happiness sum using the provided happiness vector and integer k
    long long maximumHappinessSum(vector<int>& happiness, int k) {
        // Sort the happiness vector in non-increasing order
        sort(happiness.rbegin(), happiness.rend());

        // Initialize a variable to store the accumulated maximum happiness sum
        long long maxHappinessSum = 0;

        // Iterate over the first k elements of the sorted vector
        for (int i = 0, n = happiness.size(); i < k; ++i) {
            // Calculate the modified happiness value by subtracting the index
            int modifiedHappiness = happiness[i] - i;

            // If the modified happiness value is positive, add it to the sum
            // Otherwise, add zero (do not decrease the sum)
            maxHappinessSum += std::max(modifiedHappiness, 0);
        }

        // Return the accumulated maximum happiness sum
        return maxHappinessSum;
    }
};
```

TypeScript

```typescript
// This function calculates the maximum happiness sum from an array of happiness points.
// It considers the 'k' people with the highest happiness points after applying a specific penalty.
// The penalty subtracts the person's index from their happiness points (starting from 0).

function maximumHappinessSum(happiness: number[], k: number): number {
    // Sort the happiness array in descending order.
    happiness.sort((a, b) => b - a);
    // Initialize a variable to keep track of the total happiness sum.
    let totalHappiness = 0;

    // Loop through the first 'k' elements in the sorted happiness array.
    for (let i = 0; i < k; ++i) {
        // Calculate the penalty by subtracting the index (0-based) from the happiness points.
        const penalizedHappiness = happiness[i] - i;
        // Add the greater of penalizedHappiness or 0 to the totalHappiness sum
        // to ensure that negative values do not reduce the overall happiness.
        totalHappiness += Math.max(penalizedHappiness, 0);
    }
    // Return the total happiness sum after processing the first 'k' elements.
    return totalHappiness;
}
```

```python
class Solution:
    def maximumHappinessSum(self, happiness: List[int], k: int) -> int:
        # Sort the happiness values in descending order.
        happiness.sort(reverse=True)
        max_happiness = 0 # Initialize the maximum happiness sum.

        # Loop through the first k elements after sorting.
        for i in range(k):
            # Since a person's happiness value can decrease with each additional person,
            # we subtract the current index from the happiness value,
            # however, we should not go below 0.
            decreased_happiness = happiness[i] - i
            if decreased_happiness > 0:
                max_happiness += decreased_happiness

        # Return the sum of the maximized happiness values.
        return max_happiness
```

Time and Space Complexity

The time complexity of the `maximumHappinessSum` function is composed of two parts: sorting the `happiness` list and iterating over a slice of this list.

- Sorting the list requires `O(n * log n)` time, where `n` is the length of the `happiness` list. The Python `sort()` method uses the Timsort algorithm, which has this complexity.
- Iterating over the first `k` elements of the sorted list takes `O(k)` time, because we're only looking at a subset of the list, which contains `k` elements.

Adding these two components together, the total time complexity is `O(n * log n + k)`.

As for the space complexity, the sort operation can be done in-place, but Timsort requires additional temporary space for its operation. This temporary space is `O(log n)` because of the way the algorithm divides the list and merges sorted sublists. There is no additional significant space usage since only a few extra variables are created, and these do not depend on the size of the input list.