2551. Put Marbles in Bags

Hard Greedy Array Sorting Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, you are tasked with distributing a set of marbles, each with a certain weight, into `k` bags. The distribution must follow certain rules:

- **Contiguity**: The marbles in each bag must be a contiguous subsegment of the array. This means if marbles at indices `i` and `j` are in the same bag, all marbles between them must also be in that bag.
- **Non-Empty Bags**: Each of the `k` bags must contain at least one marble.
- **Cost of a Bag**: The cost of a bag that contains marbles from index `i` to `j` (inclusive) is the sum of the weights at the start and end of the bag, that is `weights[i] + weights[j]`.

Your goal is to find the way to distribute the marbles so that you can calculate the maximum and minimum possible scores, where the score is defined as the sum of the costs of all `k` bags. The result to be returned is the difference between the maximum and minimum scores possible under the distribution rules.

Intuition

To find the maximum and minimum scores, we need to think about what kind of distributions will increase or decrease the cost of a bag.

For the **maximum** score, you want to maximize the cost of each bag. You can do this by pairing the heaviest marbles together because their sum will be higher. Conversely, for the **minimum** score, you would pair the lightest marbles together to minimize the sum.

The intuition behind the solution is grounded on the understanding that the cost of a bag can only involve the first and the last marble in it due to the contiguity requirement. Thus, to minimize or maximize the score, you should look at possible pairings of marbles.

A solution approach involves the following steps:

1. Precompute the potential costs of all possible bags by pairing each marble with the next one, as you will always have to pick at least one boundary marble from each bag.
2. Sort these potential costs. The smallest potential costs will be at the beginning of the sorted array, and the largest will be towards the end.
3. To get the **minimum score**, sum up the smallest `k-1` potential costs, which correspond to the minimum possible cost for each of the bags except one.
4. To get the **maximum score**, sum up the largest `k-1` potential costs, which correspond to the maximum possible cost for each of the bags except one.
5. The difference between the maximum and minimum scores will provide the result.

The Python solution provided uses the precomputed potential costs and sums the required segments of it to find the maximum and minimum scores, and then computes their difference.

Solution Approach

The implementation of the solution in Python involves the following steps:

1. **Pairwise Costs**: Compute the potential costs of making a bag by pairing each marble with its immediate successor. This is done using the expression `a + b for a, b in pairwise(weights)`, which computes the cost for all possible contiguous pairs of marbles and represents the possible costs of bags if they were to contain only two marbles.

2. **Sorting Costs**: Once we have all the potential costs, sort them in ascending order using `sorted()`. This gives us an array where the smallest potential costs are at the start of the array and the largest potential costs are at the end.

3. **Calculating Minimum Score**: To obtain the minimum score, sum up the first `k-1` costs from the sorted array with `sum(arr[: k - 1])`. Each of these costs represents the cost of one of the bags in the minimum score distribution, excluding the last bag, because `k-1` bags will always have neighboring marbles from the sorted pairwise costs as boundaries, and the last bag will include all remaining marbles which might or might not follow the pairing rule.

4. **Calculating Maximum Score**: To get the maximum score, sum up the last `k-1` costs from the sorted array with `sum(arr[len(arr) - k + 1:])`. Each of these costs represents the cost of one of the bags in the maximum score distribution, excluding the last bag, similar to the minimum score calculation but from the other end due to the sorted order.

5. **Computing the Difference**: Finally, the difference between the maximum and the minimum scores is computed by `return sum(arr[len(arr) - k + 1 :]) - sum(arr[: k - 1])`, which subtracts the minimum score from the maximum score to provide the desired output.

This solution uses a greedy approach that ensures the maximum and minimum possible costs by making use of sorted subsegments of potential costs. The data structures used are simple arrays, and the sorting of the pairwise costs array is the central algorithmic pattern that enables the calculation of max and min scores efficiently.

Example Walkthrough

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

Suppose we have an array of marble weights `[4, 2, 1, 3]` and we want to distribute them into `k = 2` bags following the given rules.

Step 1: **Pairwise Costs**

First, we compute the potential costs for all pairwise marbles:

- Bag with marbles at index 0 and 1: `weights[0] + weights[1] = 4 + 2 = 6`
- Bag with marbles at index 1 and 2: `weights[1] + weights[2] = 2 + 1 = 3`
- Bag with marbles at index 2 and 3: `weights[2] + weights[3] = 1 + 3 = 4`

So the pairwise costs are `[6, 3, 4]`.

Step 2: **Sorting Costs**

We sort these costs in ascending order: `[3, 4, 6]`.

Step 3: **Calculating Minimum Score**

To get the minimum score, we need to consider `k-1 = 1` smallest pairwise cost, so we sum up the first `k-1` costs: `3`.

Step 4: **Calculating Maximum Score**

To get the maximum score, we consider the `k-1` largest pairwise costs from the sorted list, so the last `k-1` cost: `6`.

Step 5: **Computing the Difference**

The difference between the maximum and minimum scores is `6 - 3 = 3`. Therefore, the difference between the highest and the lowest possible scores with the given distribution rules is `3`.

In terms of the actual marble distributions:

- The minimum score distribution would be having one bag with marbles at indices `1` and `2`, and the other bag getting the rest (`4+2+1+3 = 10`, but the last bag's cost is just `4+3=7`). Total minimum score: `3 + 7 = 10`.
- The maximum score distribution would be one bag with marbles at indices `0` and `1`, and another bag with the rest (`2+1+3 = 6`, but the last bag's cost is just `2+3=5`). Total maximum score: `6 + 5 = 11`.

This example clearly illustrates the steps outlined in the solution approach, showing how the precomputed pairwise costs, when sorted and summed appropriately, can yield the minimum and maximum scores, and thus the difference between them.

Python Solution

```python
from itertools import pairwise  # Import pairwise from itertools for Python versions >= 3.10

class Solution:
    def putMarbles(self, weights, k):
        # Generate all possible pairwise sums of weights
        pairwise_sums = sorted(a + b for a, b in pairwise(weights))

        # Calculate the sum of the largest k pairwise sums
        sum_of_largest_k = sum(pairwise_sums[-k:])

        # Calculate the sum of the smallest k-1 pairwise sums
        sum_of_smallest_k_minus_1 = sum(pairwise_sums[:k-1])

        # Return the difference between the sum of the largest k and smallest k-1 pairwise sums
        return sum_of_largest_k - sum_of_smallest_k_minus_1
```

It's important to note that the `pairwise` function is available in the `itertools` module for Python versions 3.10 and later. If you are using an earlier version of Python, the `pairwise` function won't be available, and you would need to define it manually:

```python
from itertools import tee

def pairwise(iterable):
    # pairwise('ABCDEFG') --> AB BC CD DE EF FG
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // Method to calculate the sum based on the given weights and integer k
    public long putMarbles(int[] weights, int k) {
        // Get the number of weights
        int numWeights = weights.length;

        // Create an array to store the sum of adjacent weights
        int[] adjacentSums = new int[numWeights - 1];

        // Calculate the sum of adjacent weights and store in the array
        for (int i = 0; i < numWeights - 1; ++i) {
            adjacentSums[i] = weights[i] + weights[i + 1];
        }

        // Sort the array of adjacent sums
        Arrays.sort(adjacentSums);

        // Initialize answer to 0
        long answer = 0;

        // Calculate the sum of the largest k - 1 elements
        // and subtract the sum of the smallest k - 1 elements from it
        for (int i = 0; i < k - 1; ++i) {
            answer -= adjacentSums[i];
            answer += adjacentSums[numWeights - 2 - i];
        }

        // Return the final answer
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include necessary headers

class Solution {
public:
    long long putMarbles(std::vector<int>& weights, int k) {
        // Finding the size of the input vector weights.
        int numWeights = weights.size();

        // Create a new vector to store the sum of adjacent weights.
        std::vector<int> adjacentSum(numWeights - 1);

        // Calculate the sum of adjacent weights and fill the adjacentSum vector.
        for (int i = 0; i < numWeights - 1; ++i) {
            adjacentSum[i] = weights[i] + weights[i + 1];
        }

        // Sort the adjacentSum vector in non-decreasing order.
        std::sort(adjacentSum.begin(), adjacentSum.end());

        // Initialize ans to store the final result.
        long long ans = 0;

        // Calculate the answer by picking k - 1 smallest elements and k - 1 largest elements from sorted adjacentSum.
        for (int i = 0; i < k - 1; ++i) {
            ans -= adjacentSum[i]; // Subtract the k - 1 smallest elements.
            ans += adjacentSum[numWeights - 2 - i]; // Add the k - 1 largest elements.
        }

        // Return the final computed result.
        return ans;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculate the difference between the sum of the heaviest and lightest marbles after k turns.
 *
 * @param {number[]} weights - An array of integers representing the weights of the marbles.
 * @param {number} k - The number of turns.
 * @return {number} The difference in weight between the selected heaviest and lightest marbles.
 */
function putMarbles(weights: number[], k: number): number {
    // Number of weights provided.
    const numWeights = weights.length;

    // Will hold the sum of pairs of weights.
    const pairSums: number[] = [];

    // Calculate the sum of each pair of adjacent weights.
    for (let i = 0; i < numWeights - 1; ++i) {
        pairSums.push(weights[i] + weights[i + 1]);
    }

    // Sort the pair sums in ascending order.
    pairSums.sort((a, b) => a - b);

    // This will hold the computed result.
    let difference = 0;

    // Sum the difference between the heaviest and lightest pair sums for k - 1 turns.
    for (let i = 0; i < k - 1; ++i) {
        // The heaviest pair is at the back of the sorted array.
        // As the index in JavaScript is 0-based, the (n-i-2) gives us the element just before the last i elements.
        difference += pairSums[numWeights - i - 2] - pairSums[i];
    }

    // Return the calculated difference.
    return difference;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be broken down as follows:

1. `sorted(a + b for a, b in pairwise(weights))`:
    - First, the `pairwise` function creates an iterator over adjacent pairs in the list `weights`. This is done in O(N) time, where N is the number of elements in `weights`.
    - The generator expression `a + b for a, b in pairwise(weights)` computes the sum of each pair, resulting in a total of O(N) operations since it processes each pair once.
    - The `sorted` function then sorts the sums, which has a time complexity of O(N log N), where N is the number of elements produced by `pairwise`, which is N - 1. However, we simplify this to O(N log N) for the complexity analysis.

Combining these steps, the time complexity is primarily dominated by the sorting step, resulting in an overall time complexity of `O(N log N)`.

2. `sum(arr[len(arr) - k + 1 :])` and `sum(arr[: k - 1])`:
    - Both `sum(...)` operations are performed on slices of the sorted list `arr`. The number of elements being summed in each case depends on `k`, but in the worst case, it will sum up to N - 1 elements (the length of `arr`), which is O(N).

The combination of sorting and summing operations results in a total time complexity of `O(N log N)`, with the sorting step being the most significant factor.

Space Complexity

The space complexity of the code can be examined as follows:

1. The sorted array `arr` is a new list created from the sums of pairwise elements, which contains N - 1 elements. Hence, this requires `O(N)` space.

2. The slices made in the `sum` operations do not require additional space beyond the list `arr`, as slicing in Python does not create a new list but rather a view of the existing list.

Therefore, the overall space complexity of the code is `O(N)` due to the space required for the sorted list `arr`.