561. Array Partition

Easy · Greedy · Array · Counting Sort · Sorting   Leetcode Link

Problem Description

The task is formulated around an optimization problem using an integer array named `nums`. This array has `2n` elements, meaning that its length is an even number. The objective is to pair up the integers into `n` pairs in such a way that the sum of the minimum values from each pair is maximized. In mathematical terms, for pairs `(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)`, we need to find the maximum value of `sum(min(a_i, b_i))` for all `i` from `1` to `n`.

To fulfill the task:

1. We have to create the pairs from the elements in the array.
2. Then, calculate the minimum of each pair.
3. Finally, calculate and return the sum of all these minimum values.

The objective is to construct these pairs wisely so that we get the maximum possible sum of the minimums.

Intuition

The intuition behind the solution relies on the idea of minimizing the loss of the larger numbers and ensuring that we can maximize the sum of the smaller numbers.

A crucial observation is that in each pair `(a, b)`, the minimum value is the one that will be included in our final sum, whereas the larger one is effectively 'wasted' in terms of contributing to the sum. Therefore, we should minimize the waste by making sure that the difference between the paired numbers is as small as possible.

To meet this goal, the following steps are considered:

1. Sort the array in ascending order. Sorting allows us to easily group the elements into pairs in a way that minimizes the differences between the numbers within each pair.
2. Once we have the sorted array, we can pair each element with its adjacent element without worrying about missing out on a possibly smaller or equal pair.
3. Now that we have the pairs as adjacent elements, we know that within each pair, the first element is the smaller one (due to the array being sorted).
4. To find the sum of all minimum elements from each pair, we can add up every second element starting from the first element in the sorted array.

The provided code does exactly that: `sum(sorted(nums)[::2])` sorts the array and then uses slicing to get every second element starting from the first one (which are all the minimums in each pair), and then sums them up to get the result.

Solution Approach

The implementation of the solution is straightforward and leverages Python's built-in functions to achieve the goal efficiently. Here's a step-by-step guide through the algorithm and the patterns used in the solution:

1. **Sorting the Array**: The initial step is to use Python's built-in sorting function with `sorted(nums)`. Sorting is a very common technique in such problems as it often simplifies and provides a structure to the data which can be exploited in a subsequent step. In this case, sorting puts the array in ascending order so that each pair `(a, b)` will have `a` as the minimum.

2. **Slicing the Array**: With the sorted array, we apply slicing `sorted(nums)[::2]`. In Python, slicing is a way to obtain a subsequence of a list or array. The slice `[::2]` specifically means "start from the first element and pick every second element." This slice operation is both compact and efficient, avoiding the need for an explicit loop to go through the array and pick the elements.

3. **Summing the Minimums**: Lastly, by passing the sliced list to `sum()`, Python's built-in function that computes the sum of the numbers in an iterable, we add up the selected numbers. The minimum of each pair is included due to the slicing, and thus the sum of these minimums is maximized.

The algorithm's time complexity is dominated by the sorting operation, which is O(n log n) in the average and worst case scenarios. The slicing and summing are linear operations, O(n), but since n log n grows faster than n, the overall time complexity remains O(n log n).

No additional data structures are used or needed since the problem can be resolved with operations on the provided array. In terms of patterns, it's worth noting that this approach elegantly combines sorting with the properties of a sorted array to avoid a more complex and potentially less efficient approach that might explicitly build and then process pairs of numbers.

The simplicity of the approach is its strength, as it doesn't rely on complex data structures or algorithms beyond what Python provides out of the box. The use of slicing to select every other element of an already sorted array makes the implementation both elegant and intuitive.

Example Walkthrough

Consider the array `nums = [3, 1, 5, 4, 2, 6]` with a length of `2n = 6`, indicating that we need to create `n = 3` pairs. Apply the solution approach to this example:

1. **Sorting the Array**: First, sort the array in ascending order which gives us the sorted array `[1, 2, 3, 4, 5, 6]`. By sorting, the smallest numbers are positioned at the beginning, which will help maximize the sum of the minimums of each pair.

2. **Slicing the Array**: Now, we retrieve every second element starting from the first element using slicing. From our sorted array, this yields `[1, 3, 5]` which represents the minimum values from each potential pair (`(1,2), (3,4), (5,6)`).

3. **Summing the Minimums**: Finally, sum up the elements in the sliced array `[1, 3, 5]`. The sum is `1 + 3 + 5 = 9`. This total is the maximum sum of the minimums that can be obtained by pairing up numbers in the given array.

Following this method, the code `sum(sorted(nums)[::2])` provides a quick and efficient way to reach the solution, which is the sum `9` in this case. By ensuring that we pair each number with its nearest neighbor in the sorted list, we maximize the contribution of the lower number to the sum while minimizing the "waste" due to the higher number being paired with it but not counted in the sum.

Python Solution

```python
from typing import List

class Solution:
    def arrayPairSum(self, nums: List[int]) -> int:
        """
        Find the max sum of min pairs in an array.

        The function takes in an array of 2n integers and we need to pair them up
        in such a way to minimize the difference between the pairs. The sum we want
        to maximize is the sum of the smaller number in each pair.

        :param nums: List[int] - Array of 2n integers
        :return: int - Maximum sum of min pairs
        """

        # Sort the array in non-decreasing order
        sorted_nums = sorted(nums)

        # Take every other element starting from the first element
        # because after sorting, the first element of each pair
        # will be the smaller one
        min_pairs_sum = sum(sorted_nums[::2])

        return min_pairs_sum

# Example usage:
# sol = Solution()
# print(sol.arrayPairSum([1,4,3,2]))  # Output: 4

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting

public class Solution {

    // Function to maximize sum of min(ai, bi) for all pairs (ai, bi)
    public int arrayPairSum(int[] nums) {
        // Sort the array to make pairs of two consecutive elements
        Arrays.sort(nums);

        // Initialize sum to store the final answer
        int sum = 0;

        // Iterate through the array, jumping two steps at a time
        for (int i = 0; i < nums.length; i += 2) {
            // Add the first element of each pair to the sum since it's the minimum
            sum += nums[i];
        }

        // Return the accumulated sum of the min elements of the pairs
        return sum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include the algorithm header to use the std::sort function

// Solution class contains a method to solve the problem.
class Solution {
public:
    // The method arrayPairSum maximizes the minimum pair sum in an array.
    int arrayPairSum(vector<int>& nums) {
        // Sort the input array in non-decreasing order.
        std::sort(nums.begin(), nums.end());

        // Initialize the answer to store the sum of the min elements of the pairs.
        int maxMinPairSum = 0;

        // Iterate over the array, incrementing by 2 to only consider the first element of each pair (since array is sorted).
        for (int i = 0; i < nums.size(); i += 2) {
            // Accumulate the sum by adding the first element of each pair, which is the min of the two.
            maxMinPairSum += nums[i];
        }

        // Return the final sum as the answer.
        return maxMinPairSum;
    }
};

```

Typescript Solution

```typescript
/**
 * Given an integer array `nums` of 2n integers,
 * group these integers into n pairs (a1, b1), (a2, b2), ..., (an, bn)
 * such that the sum of min(ai, bi) for all i is maximized.
 * Return the maximized sum.
 *
 * @param {number[]} nums - The array of 2n integers.
 * @return {number} - The maximized sum of min(ai, bi) for all paired integers.
 */
function arrayPairSum(nums: number[]): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);
    let sum: number = 0;

    // Iterate through the array, increasing by 2 to consider pairs
    for (let i = 0; i < nums.length; i += 2) {
        // Add the minimum of each pair (which is the first element in the sorted pair)
        sum += nums[i];
    }

    // Return the sum of minimums
    return sum;
}

// Example usage:
// const result: number = arrayPairSum([1, 3, 2, 4]);
// console.log(result); // Output would be 4, because 1+3 (min of pair (1,2) + min of pair (3,4)) is the largest sum.

```

Time and Space Complexity

The given Python function `arrayPairSum` finds the sum of min(a[i], a[i+1]) for every pair of elements in the array when the array is sorted. It sorts the array and then sums up elements at even indices (i.e., considering 0-based indexing, indices 0, 2, 4, etc).

Time Complexity:

The time complexity is primarily determined by the sorting function. Python uses Timsort for sorting, which has a time complexity of `O(n log n)` for an average and worst-case scenario, where `n` is the length of the `nums` array.

There is also the summation operation, which iterates over every other element of the sorted array, contributing an additional `O(n/2)`, which simplifies to `O(n)` time. However, since `O(n log n)` dominates `O(n)`, the overall time complexity is `O(n log n)`.

Space Complexity:

The space complexity of the sorting operation depends on the implementation. For Timsort, the worst-case space complexity is `O(n)`, because it might need temporary space to hold elements while merging. Timsort is a hybrid sorting algorithm that requires temporary storage for the merge operations.

However, since the input array itself is sorted in-place and the result is computed using that without requiring extra space, other than what is needed for the sorted array and the sum variable, the space complexity may be considered `O(1)` or constant if the space used by the sorting algorithm is not taken into account, which is typically the case for space complexity analysis in Python where sorting is considered to be in-place.

In summary:

- Time Complexity: `O(n log n)`
- Space Complexity: `O(1)` (disregarding the space used by the sorting algorithm)