1877. Minimize Maximum Pair Sum in Array

Medium Greedy Array Two Pointers Sorting

Leetcode Link

Problem Description

In this problem, we are given an array `nums` with an even number of elements. Our task is to form pairs of elements from this array. Each pair is a combination of two items from the array `(a, b)`, and the pair sum is defined as `a + b`. After pairing all the elements, there will be many pair sums, and we want to find the pair with the maximum sum, which we'll call the maximum pair sum.

However, the twist is that we have to pair the elements in such a way that this maximum pair sum is as small as possible compared to all different pairings. That is, we are trying to minimize the maximum. The output of the function will be this minimized maximum pair sum.

For example, if the array is `[3, 5, 2, 3]`, there are multiple ways to form pairs, but if we pair `3` with `5` and `2` with `3`, we get pairs `(3,5)` and `(2,3)`, which results in pair sums `8` and `5`. Here, `8` is the maximum pair sum, and there is no way to pair these elements to get a lower maximum pair sum, so the output will be `8`.

Intuition

To arrive at the solution, we need to use a greedy approach. The greedy insight for minimizing the maximum pair sum is to pair the smallest elements available with the largest ones. If we were to pair large elements with other large elements, it would definitely increase the maximum pair sum more than necessary.

To implement this strategy, we first sort the array. By sorting, we can easily access the smallest and the largest elements and pair them together. Specifically, we pair the first element (the smallest after sorting) with the last element (the largest), the second element with the second to last, and so on.

Once we have paired the numbers in this way, all that remains is to find the maximum pair sum from these optimal pairs. This can be done by looking at the pair sums resulting from each first half element and its corresponding second half partner. We take the maximum of these sums to get the minimized maximum pair sum.

The intuition for the solution can be summarized as follows:

- Sort the array to efficiently pair smallest and largest elements.
- Pair elements from the start of the array with elements from the end.
- The maximum pair sum from these pairs will be the minimized maximum.

By following this approach, we ensure that we do not have any unnecessarily large pair sums, while still adhering to the constraints of the problem.

Solution Approach

The provided solution approach leverages two well-known concepts: sorting and the greedy method.

Here's the step-by-step implementation of the solution:

1. **Sort the Array:** We begin by sorting the array `nums` in ascending order. This puts the smallest numbers at the start of the array and the largest numbers at the end.

   ```
   1 nums.sort()
   ```

   Sorting is essential because it allows us to directly access the smallest and largest numbers without having to search for them, thereby enabling the next steps of the greedy approach. The built-in `.sort()` method in Python is utilized, which sorts the list in-place.

2. **Pair Smallest with Largest:** Once the array is sorted, we iterate through the first half of the array. For each element `x` in the first half, we find its optimal pair by selecting the element at the symmetrical position in the second half of the array (`nums[n - i - 1]`, where `i` is the index of `x` and `n` is the total number of elements in the array).

   Using the greedy insight, the smallest element is paired with the largest, the second smallest with the second largest, and so on. This is achieved by iterating over the first half of the sorted array and finding the corresponding element in the second half.

3. **Compute and Return Maximum Pair Sum:** For each such pair, we compute the pair sum and find the maximum among all these sums. This maximum is the minimized maximum pair sum we want to return.

   ```
   1 return max(x + nums[n - i - 1] for i, x in enumerate(nums[: n >> 1]))
   ```

   Here, the use of `n >> 1` is a bit manipulation trick that is equivalent to integer division by 2, effectively finding the midpoint of the array. The `enumerate` function is used in a loop to get both the index `i` and the value `x` of the elements from the first half of `nums`. With these, we calculate the pair sums and pass them to the `max()` function which finds the maximum pair sum, fulfilling our original goal.

This approach is incredibly efficient because by sorting the array only once, we are then able to pair elements in a single pass without extra loops or complex logic. The overall time complexity is dominated by the sorting step, which is typically `O(n log n)` where `n` is the length of the array, making this solution scalable and effective for large inputs.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `[4, 1, 2, 7]`.

1. **Sort the Array:** As a first step, we sort the array `nums` in ascending order. This will arrange the numbers so that we have the smallest elements at the start and the largest at the end.

   ```
   1 Original array: [4, 1, 2, 7]
   2 Sorted array: [1, 2, 4, 7]
   ```

2. **Pair Smallest with Largest:** After sorting, we pair the first element in the array with the last, and the second element with the second to last, and so on. This is done using the greedy approach, to minimize the maximum pair sum.

   Pairs are as follows:

   ```
   1 Pair 1: (1, 7)
   2 Pair 2: (2, 4)
   ```

   These pairs are made by pairing the first half of the sorted array `[1, 2]` with the reversed second half `[7, 4]`.

3. **Compute and Return Maximum Pair Sum:** We compute the pair sum for each pair and identify the maximum of those sums.

   ```
   1 Pair sum 1: 1 + 7 = 8
   2 Pair sum 2: 2 + 4 = 6
   ```

   The maximum of these sums is `8`, which is the answer.

The outcome of this approach is that the array `[1, 2, 4, 7]` yields a minimized maximum pair sum of `8`. By using this greedy method and sorting beforehand, we ensured that we didn't unnecessarily increase the maximum pair sum and met our goal in an efficient manner.

Python Solution

```python
class Solution:
    def minPairSum(self, nums: List[int]) -> int:
        # Sort the list in non-decreasing order
        nums.sort()

        # Get the length of the list
        n = len(nums)

        # Calculate the maximum pair sum.
        # The ith smallest number is paired with the ith largest number to minimize the maximum pair sum.
        # This is done for the first half of the sorted list, then the maximum of these sums is found.
        max_pair_sum = max(x + nums[n - i - 1] for i, x in enumerate(nums[: n // 2]))

        # Return the maximum pair sum
        return max_pair_sum

```

Java Solution

```java
class Solution {
    public int minPairSum(int[] nums) {
        // Sort the array to have numbers in ascending order.
        Arrays.sort(nums);

        // Initialize a variable to store the maximum pair sum found so far.
        int maxPairSum = 0;

        // Calculate the number of pairs - half the length of the array
        int numPairs = nums.length / 2;

        // Iterate over the first half of the sorted array to form pairs with elements from both ends
        for (int i = 0; i < numPairs; i++) {
            // Calculate the sum of the current pair
            int currentPairSum = nums[i] + nums[nums.length - i - 1];

            // Update maxPairSum if the current pair sum is greater than the maxPairSum found so far
            maxPairSum = Math.max(maxPairSum, currentPairSum);
        }

        // Return the maximum pair sum found
        return maxPairSum;
    }
}
```

C++ Solution

```cpp
#include <vector>  // Required for using the vector container.
#include <algorithm>  // Required for the sort() and max() functions.

class Solution {
public:
    // Function to find the minimum possible maximum pair sum in an array.
    int minPairSum(vector<int>& nums) {
        // Sort the array in non-decreasing order.
        sort(nums.begin(), nums.end());

        // Initialize the variable that will store the minimum possible maximum pair sum.
        int minMaxPairSum = 0;

        // Calculate the size of the array.
        int n = nums.size();

        // Iterate over the first half of the array.
        for (int i = 0; i < n / 2; ++i) {
            // For each element in the first half, pair it with the corresponding
            // element in the second half, and then update minMaxPairSum with the maximum
            // sum of these pairs seen so far.
            minMaxPairSum = max(minMaxPairSum, nums[i] + nums[n - i - 1]);
        }

        // Return the minimum possible maximum pair sum found.
        return minMaxPairSum;
    }
};
```

Typescript Solution

```typescript
/**
 * Calculates the min pair sum of an array when paired optimally.
 * Pairs are formed such that the largest and smallest numbers are paired
 * and the resulting max pair sum is the smallest possible.
 *
 * @param {number[]} nums - The input array of numbers.
 * @return {number} - The minimum possible maximum pair sum.
 */
function minPairSum(nums: number[]): number {
    // Sort the array in non-decreasing order.
    nums.sort((a, b) => a - b);

    let maxPairSum = 0; // Initialize the maximum pair sum as 0.
    const arrayLength = nums.length; // Get the length of the array.

    // Loop over the first half of the array to create pairs.
    for (let i = 0; i < arrayLength >> 1; ++i) {
        // Calculate the current pair sum.
        const currentPairSum = nums[i] + nums[arrayLength - 1 - i];

        // Update the maximum pair sum if current is larger.
        maxPairSum = Math.max(maxPairSum, currentPairSum);
    }

    // Return the maximum pair sum after pairing all elements optimally.
    return maxPairSum;
}

// Example usage:
// const nums = [1, 5, 2, 3];
// const result = minPairSum(nums);
// console.log(result); // Output would be 7.

```

Time and Space Complexity

Time Complexity:

The main operations in the code include:

1. Sorting the list: `nums.sort()`, which usually has a time complexity of O(n log n), where n is the length of the list.
2. A list comprehension to calculate the pair sums and find the maximum: This operation goes through roughly half of the sorted list, leading to a complexity of O(n/2), which simplifies to O(n).

Combining these, the overall time complexity is dominated by the sort operation, thus it is O(n log n).

Space Complexity:

The space complexity is determined by:

1. The sorted `nums` list does not require additional space as the sort is done in-place.
2. The list comprehension creates a generator, which doesn't create a new list; it just iterates through the existing items.

Therefore, aside from the input list, the code uses a fixed amount of space, giving us a space complexity of O(1).