1968. Array With Elements Not Equal to Average of Neighbors **Greedy Array Sorting** Medium

# **Problem Description**

way that no element is equal to the average of its immediate neighbors. To clarify, for every index i of the rearranged array (1 <= i < nums.length - 1), the value at nums[i] should not be equal to (nums[i-1] + nums[i+1]) / 2. The task is to return any version of the nums array that satisfies this condition.

You are provided with an array nums which consists of unique integers. The goal is to rearrange the elements in the array in such a

Intuition The key observation to arrive at the solution is that if we arrange the numbers in a sorted manner and then alternate between the

## smaller and larger halves of the sorted array, we can ensure that no element will be the average of its neighbors. This is because

the numbers are distinct and sorted, so a number from the smaller half will always be less than the average of its neighbors, and a number from the larger half will always be greater than the average of its neighbors. Therefore, first, we sort nums. Then, we divide the sorted array into two halves - the first half containing the smaller elements and the second half containing the larger elements. We then interleave these two halves such that we first take an element from the

first half, followed by an element from the second half, and so on until we run out of elements. Here's how the thought process breaks down: 1. Sort the array to easily identify the smaller and larger halves.

2. Identify the middle index to divide the sorted array into two halves. 3. Create an empty array ans to store the final rearranged sequence.

4. Iterate through each of the elements in the first half and alternate by adding an element from the second half. 5. Return the rearranged array which should now meet the requirements of the problem.

## **Solution Approach** The solution is pretty straightforward once we understand the intuition behind the problem. The approach makes use of the

Here's a step-by-step explanation of the implementation with reference to the provided solution code:

### We start by sorting the array nums. Sorting is done in-place and can use the TimSort algorithm (Python's default sorting algorithm), which has a time complexity of O(n log n), where n is the number of elements in the array.

ans = []

return ans

**Example Walkthrough** 

nums = [1, 3, 2, 5, 4, 6]

m = (6 + 1) >> 1 => 3

the second half, and repeat:

for i in range(3): # range(m)

The final output for the initial nums array is:

return ans  $\Rightarrow$  [1, 4, 2, 5, 3, 6]

**Python** 

class Solution:

nums.sort()

ans = []

# from typing import List

input\_nums = [6,2,0,9,7]

# Initialize the class instance

# Example input list of numbers

solution\_instance = Solution()

n = len(nums)

m = (n + 1) // 2

for i in range(m):

if i + m < n:

ans.append(nums[i])

# Demonstrating the use of Solution class:

# Get the rearranged list from the rearrangeArray method

public int[] rearrangeArray(int[] nums) {

// Find the length of the array

int midIndex = (length + 1) >> 1;

int[] rearranged = new int[length];

rearranged[i] = nums[j];

if (j + midIndex < length) {</pre>

function rearrangeArray(nums: number[]): number[] {

nums.sort((a, b) => a - b);

const rearranged: number[] = [];

for (let i = 0; i < mid; ++i) {

rearranged.push(nums[i]);

// Sort the elements of the nums array in non-decreasing order.

// Calculate the middle index based on the size of nums to divide

// Check if the symmetric position in the second half exists,

// Add the current element from the first half to the rearranged array.

// Declare an array to store the rearranged elements.

// the array into two halves. Equivalent to (n+1)/2

// Loop through the first half of the sorted array.

// if so, add it to the rearranged array.

rearranged.push(nums[secondHalfIndex]);

const secondHalfIndex = i + mid;

if (secondHalfIndex < nums.length) {</pre>

// Return the array with rearranged elements.

def rearrangeArray(self, nums: List[int]) -> List[int]:

# Sort the list of numbers in ascending order

# Initialize an empty list to store the answer

ans.append(nums[i + m])

# Get the rearranged list from the rearrangeArray method

rearranged\_nums = solution\_instance.rearrangeArray(input\_nums)

# Iterate over the first half of the sorted list

# Add the current element from the first half to the answer list

# Add the corresponding element from the second half to the answer list

# Assuming 'List' has already been imported from 'typing' module otherwise, add the following line:

# Check if there is a corresponding element in the second half

const mid = Math.floor((nums.length + 1) / 2);

int length = nums.length;

rearranged\_nums = solution\_instance.rearrangeArray(input\_nums)

// Sort the array to arrange elements in ascending order

// Initialize a new array to store the rearranged elements

for (int i = 0, j = 0; i < length; i += 2, j++) {

rearranged[i + 1] = nums[j + midIndex];

// Compute the mid-index accounting for both even and odd length arrays

// Loop to interleave elements from the two halves of the sorted array

ans.append(nums[i]) # Add from the first half

When we iterate through, the ans array gets filled as follows:

def rearrangeArray(self, nums: List[int]) -> List[int]:

# Find the middle index, rounded up to account for odd lengths

# Add the current element from the first half to the answer list

# Assuming 'List' has already been imported from 'typing' module otherwise, add the following line:

# Check if there is a corresponding element in the second half

# Sort the list of numbers in ascending order

# Equivalent to ceil function: m = ceil(n / 2)

# Initialize an empty list to store the answer

# Iterate over the first half of the sorted list

# Get the length of the nums list

 $\circ$  i = 0: Append nums [0] which is 1 and then nums [0 + 3] which is 4, so ans = [1, 4].

average of its immediate neighbors, and this is returned as the result.

 $\circ$  i = 1: Append nums [1] which is 2 and then nums [1 + 3] which is 5, so ans = [1, 4, 2, 5].

 $\circ$  i = 2: Append nums [2] which is 3 and then nums [2 + 3] which is 6, so ans = [1, 4, 2, 5, 3, 6].

nums.

array is:

ans.append(nums[i + m])

nums.sort() Calculate the middle index m of the array. It represents the starting index of the second half of the sorted array. If the array is

sorting algorithm, which is a very common and powerful technique in many different problems to create order from disorder.

of odd length, the middle index will include one more element in the first half. m = (n + 1) >> 1

The >> operator is a right bitwise shift, which is equivalent to dividing by two, but since we are dealing with arrays that are zero-indexed, (n + 1) ensures that the first half includes the middle element when n is odd. For even n, this does not change the result. An empty list ans is initialized to store the final rearranged sequence of nums.

to ans. Then, check if the corresponding index in the second half i + m is within bounds (i + m < n), and if so, add nums [i + m < n], and if so, add nums [i + m < n], and if so, add nums [i + m < n]. m] to ans. This ensures that elements are alternated from both halves. for i in range(m): ans.append(nums[i]) if i + m < n:

Use a for-loop to iterate over the indices of the first half of nums, which runs from 0 to m - 1 inclusive. For each i, add nums[i]

The completed ans list is now a rearranged version of nums that satisfies the problem conditions and is returned as the result.

By implementing this solution, we ensure that elements from the lower half and the upper half of the sorted array alternate in the final arrangement, which prevents any element from being the average of its neighbors due to the distinct and sorted nature of

Following the steps in the approach: We start by sorting the array nums, which gives us: nums.sort() => [1, 2, 3, 4, 5, 6]

Calculate the middle index m. There are 6 elements (n = 6), so the middle index will be:

Let's walk through a small example to illustrate the solution approach described above using an array nums. Suppose our input

### This means that the first half is [1, 2, 3] and the second half is [4, 5, 6]. An empty list ans is initialized to store the final rearranged sequence of nums.

ans = []

if i + 3 < 6: # Check if the second half index is within bounds ans.append(nums[i + 3]) # Add from the second half

The reordered list ans is now [1, 4, 2, 5, 3, 6] and satisfies the problem condition where no element is equal to the

The process effectively rearranges the original array preventing any value from being the average of its neighbors by interleaving

Use a for-loop to iterate over the indices of the first half of nums. We take an element from the first half, then an element from

```
the first and second halves of the sorted list.
Solution Implementation
```

# Add the corresponding element from the second half to the answer list ans.append(nums[i + m]) # Return the rearranged list return ans

```
Java
class Solution {
```

Arrays.sort(nums);

# Print the rearranged list

print(rearranged\_nums)

```
// Return the rearranged array
        return rearranged;
C++
#include <vector>
#include <algorithm>
class Solution {
public:
   // Method to rearrange the elements of the array such that each positive integer
   // is followed by a negative integer and vice-versa.
    vector<int> rearrangeArray(vector<int>& nums) {
        // First, sort the elements of the nums array in non-decreasing order.
        sort(nums.begin(), nums.end());
       // Declare a vector to store the rearranged elements.
        vector<int> rearranged;
       // Calculate the middle index based on the size of nums to divide
       // the array into two halves.
        int mid = (nums.size() + 1) >> 1; // Equivalent to <math>(n+1)/2
        // Loop through the first half of the sorted array.
        for (int i = 0; i < mid; ++i) {
            // Add the current element from the first half to the rearranged vector.
            rearranged.push_back(nums[i]);
            // If the symmetric position in the second half exists, add it to the rearranged vector.
            if (i + mid < nums.size()) rearranged.push_back(nums[i + mid]);</pre>
       // Return the vector with rearranged elements.
        return rearranged;
```

// Place the j-th element from the first half into the i-th position of the rearranged array

// Check if there's an element to pair from the second half and place it next to the element from the first half

```
# Get the length of the nums list
n = len(nums)
# Find the middle index, rounded up to account for odd lengths
# Equivalent to ceil function: m = ceil(n / 2)
m = (n + 1) // 2
```

ans = []

return ans

# from typing import List

input\_nums = [6,2,0,9,7]

**Time Complexity** 

# Initialize the class instance

# Example input list of numbers

solution\_instance = Solution()

for i in range(m):

if i + m < n:

ans.append(nums[i])

# Return the rearranged list

# Demonstrating the use of Solution class:

nums.sort()

class Solution:

return rearranged;

**}**;

**TypeScript** 

```
# Print the rearranged list
print(rearranged_nums)
Time and Space Complexity
```

2. The for loop runs for m = (n + 1) >> 1 iterations, which is essentially n/2 iterations for large n. Each iteration executes constant time operations, making this part O(n/2) which simplifies to O(n).

The space complexity is determined by the additional space used besides the input nums array.

The time complexity of the code consists of the sort operation and the loop that rearranges the elements.

Hence, the overall time complexity is dominated by the sort operation:  $0(n \log n)$ . **Space Complexity** 

1. Sorting the nums array takes  $O(n \log n)$  time, where n is the number of elements in nums.

1. The ans list, which contains n elements, requires 0(n) space. 2. There are also constant-size extra variables like n, m, and i which use 0(1) space.

Thus, the total space complexity is O(n) for the ans list.