1675. Minimize Deviation in Array

Hard · Greedy · Array · Ordered Set · Heap (Priority Queue) · Leetcode Link

Problem Description

In this problem, we are provided with an array called `nums` consisting of `n` positive integers. The main objective is to perform certain operations on the elements of the array to achieve the minimum possible "deviation."

Deviation is defined as the maximum difference between any two elements in the array. We want to minimize this deviation by performing operations on the array elements according to these rules:

1. If an element is even, we can divide it by `2`.
2. If an element is odd, we can multiply it by `2`.

These operations can be applied to any element any number of times. The goal is to find out the smallest possible deviation that can be obtained after performing these operations.

Intuition

To solve this problem, we need to consider the effects of the defined operations on the elements of the array. Multiplying an odd number by `2` will always make it even, and dividing an even number by `2` can potentially make it smaller and closer to other numbers in the array. However, an odd number cannot be reduced further by these operations after it has been doubled. The intuition behind the solution exploits these properties.

Here are the steps to arrive at the intuition:

1. To minimize the deviation, we would like all the numbers to be as close to each other as possible.
2. Increasing the smallest element or decreasing the largest element would bring us closer to the goal.
3. Since we can't decrease an odd number and doubling it gives us more future flexibility (as the result is an even number), we initially double all the odd numbers in the array.
4. Once all numbers are even, we have the option to reduce the largest elements by dividing them by `2`. This is the key operation, as it can potentially lower the deviation.
5. We should keep reducing the largest number until we can no longer reduce it (i.e., until it becomes odd).
6. We need to keep track of the smallest number encountered during the process since the deviation depends on both the smallest and largest numbers in the array.
7. A heap (priority queue) is perfect for this purpose because it allows us to efficiently extract the largest element and to add new elements.

By following these steps, we can incrementally decrease the array's deviation until we are left with the minimal possible deviation.

The solution code implements this intuition in Python, using a min-heap (by storing the negative of the values) to always pop the current largest element (most negative value) and implementing the described operations to reduce the deviation.

Solution Approach

The implementation of the solution follows a well-structured approach using a min-heap data structure to efficiently manage the elements while minimizing the deviation. The steps in the solution code can be broken down as follows:

1. Initialize an empty list called `h` which will serve as our min-heap, and a variable `mi` to store the minimum value in `nums`, initially set to infinity.

2. Iterate over the array `nums` and for each value `v` in `nums`:

   - If `v` is odd (which is tested using the bitwise AND operator `v & 1`), it is doubled using the left shift operator (`v <<= 1`). This is done to ensure that we have the flexibility to divide it later on.
   - Add the negative of each value to the heap, `h`. We use the negative because Python's `heapq` module provides a min-heap that gives the smallest value. By inserting the negative of our values, we can simulate the behavior of a max-heap, which is necessary to efficiently retrieve the largest element.
   - Update `mi` to be the minimum between `mi` and the current value `v`.
3. Transform the list `h` into a heap in place using `heapify(h)`.

4. Initialize a variable `ans` to store the minimum deviation, it's set to the difference between the smallest value (`mi`) and the largest value (`-h[0]`, which we take the negative of because we stored negative values in the heap).

5. Loop until the current largest element (`h[0]`, the first element in the heap) is odd. Since we are storing negatives, for an actual element to be even, its negative must be divisible by 2, which we check by evaluating `h[0] % 2 == 0`. Inside the loop:

   - Extract the largest element from the heap by popping the heap (`x = heappop(h)`), divide `x` by `2` (since `x` is negative, dividing by two makes it larger in absolute value but smaller in actual value since negative), and push it back into the heap (`heappush(h, x)`).
   - Update `mi` if the new value `-x` (actual value due to negation) is smaller.
   - Adjust `ans` to be the new minimum deviation if needed, which is the difference between `-h[0]` (since we stored negatives, this now represents the current largest actual value) and `mi`.
6. Finally, return `ans` as the result, which holds the minimum deviation after performing the operations.

The approach efficiently tracks the largest and smallest values in the heap, allowing the solution to converge to the minimum deviation by selectively doubling odd numbers once and halving even numbers until they become odd.

Example Walkthrough

Let's consider the array `nums` with the following integers: `[6, 2, 3, 4]`. We'll walk through each step of the solution approach to demonstrate how we minimize the deviation.

Step 1 & 2: Convert all odd numbers to even by doubling and initialize the min-heap:

- Since `3` is odd, we double it to get `6`.
- Now all elements are even: `[6, 2, 6, 4]`.
- We add their negative values to our min-heap `h`: `[-6, -2, -6, -4]`
- Our minimum value `mi` is `2` (the smallest element in `nums`).

Step 3: Heapify the list `h` to convert it into a min-heap. Heap `h` looks like `[-6, -4, -6, -2]` after heapifying (min-heap of negative values effectively acts as a max-heap for their absolute values).

Step 4: Initialize `ans` for the minimum deviation, which is `max(nums) - mi`, which equates to `6 - 2` = `4`.

Step 5: Loop until the largest element is odd (in terms of absolute values for stored negatives). The largest element is `-6` (actual value `6`).

- Pop `-6` from the heap, divide it by `2` (absolute value operation), and get `-3` (reflecting actual value `3`), which is then pushed back onto the heap.
- The new heap is `[-4, -2, -6, -3]` when re-heapified. The smallest value so far remains `2`. The largest element is now `-4` (actual value `4`), and `ans` is potentially updated to `4 - 2` = `2`.
- The process continues, and now `-4` is the current largest element in the heap. Pop it out, divide it by `2`, and we get `-2`. Now the heap has the elements `[-3, -2, -6, -2]`. The smallest value we have is still `2`, and `ans` remains `2`.
- None of the remaining elements can now be divided by `2` since they are all odd (in terms of absolute values), so the loop ends.

Step 6: The final `ans` reflects our minimum deviation, which remains `2`.

Through this example, we can see how the solution makes use of doubling the odd numbers and halving the even numbers while keeping track of the maximum and minimum values using a heap structure to arrive at the smallest possible deviation.

Python Solution

```python
import heapq

class Solution:
    def minimumDeviation(self, nums: List[int]) -> int:
        # Initialize a max heap (using negative values because Python has a min heap by default)
        max_heap = []
        # Initialize the minimum value with infinity to find the minimum more easily later
        min_value = float('inf')

        # Convert all integers to their potential maximum values and find the initial minimum
        for value in nums:
            if value % 2 == 1:  # If the value is odd
                value <<= 1  # Double it to make it even (which can later be halved)
            max_heap.append(-value)  # Add negative value to max_heap to maintain max heap property
            min_value = min(min_value, value)  # Update the minimum value if necessary

        # Transform the list `max_heap` into a heap in-place
        heapq.heapify(max_heap)

        # Initial answer is the difference between the largest (smallest negative) and the smallest value
        answer = -max_heap[0] - min_value

        # While the smallest negative value (largest original value) on the heap is even
        # it can be halved to potentially reduce deviation
        while max_heap[0] % 2 == 0:
            largest_neg = heapq.heappop(max_heap)  # Pop the largest element
            new_value = largest_neg // 2  # Halve it (dividing a negative number by 2 gives a smaller negative number)
            heapq.heappush(max_heap, new_value)  # Push the new halved value back on the heap
            min_value = min(min_value, -new_value)  # update min_value in regards to this new value
            # Update the answer with the new smallest deviation possible
            answer = min(answer, -max_heap[0] - min_value)

        # Once the largest value in the max_heap is odd, we can't make any more moves to reduce deviation.
        return answer

```

Java Solution

```java
class Solution {
    public int minimumDeviation(int[] nums) {
        // Create a Priority Queue which sorts in descending order
        PriorityQueue<Integer> queue = new PriorityQueue<>((a, b) -> b - a);

        // Initialize minimum value to the largest possible integer
        int minElement = Integer.MAX_VALUE;

        // Pre-process the array
        for (int value : nums) {
            // If the value is odd, multiply by 2 to convert it to even as the deviation operation.
            if (value % 2 == 1) {
                value <<= 1;
            }
            // Add the processed value to the queue
            queue.offer(value);
            // Update the minimum value encountered so far
            minElement = Math.min(minElement, value);
        }

        // Calculate the initial deviation
        int deviation = queue.peek() - minElement;

        // While the largest element in the queue is even
        while (queue.peek() % 2 == 0) {
            // Divide the largest element by 2 (which is an allowed operation)
            int topElement = queue.poll() / 2;

            // Add the reduced element back to the queue
            queue.offer(topElement);

            // Update the minimum element after dividing the largest element
            minElement = Math.min(minElement, topElement);

            // Update deviation if a smaller one is found
            deviation = Math.min(deviation, queue.peek() - minElement);
        }

        // Return the minimum deviation found
        return deviation;
    }
}

```

C++ Solution

```cpp
#include <queue>
#include <vector>
#include <climits> // for INT_MAX

class Solution {
public:
    int minimumDeviation(vector<int>& nums) {
        // Initialize the minimum found so far with the maximum possible integer value
        int min_value = INT_MAX;

        // Use a max priority queue to keep track of the current max value
        priority_queue<int> max_queue;

        // Preprocess the numbers in the initial vector
        for (int value : nums) {
            // If the number is odd, multiply it by 2 (to make it even)
            if (value % 2 != 0) value *= 2;

            // Push the possibly altered value onto the priority queue
            max_queue.push(value);

            // Update the minimum found so far
            min_value = min(min_value, value);
        }

        // Calculate the initial deviation between the max value and the min_value
        int deviation = max_queue.top() - min_value;

        // While the maximum element in max_queue is even
        while (max_queue.top() % 2 == 0) {
            // Take the top (max) element and divide it by 2
            int max_value = max_queue.top() / 2;

            // Remove the top element from the priority queue
            max_queue.pop();

            // Push the new divided value back onto the priority queue
            max_queue.push(max_value);

            // Update the minimum value if necessary
            min_value = min(min_value, max_value);

            // Update the deviation between current max and the min_value
            deviation = min(deviation, max_queue.top() - min_value);
        }

        // Return the minimum deviation found
        return deviation;
    }
};

```

Typescript Solution

```typescript
// Import the necessary elements for the priority queue implementation
import PriorityQueue from 'ts-priority-queue';

function minimumDeviation(nums: number[]): number {
    // Initialize the minimum found so far with the maximum safe integer value
    let minValue: number = Number.MAX_SAFE_INTEGER;

    // Use a max priority queue to keep track of the current max value
    // Comparator function for max priority queue (reversing arguments for max behavior)
    let maxQueue = new PriorityQueue<number>({
        comparator: function(a, b) { return b - a; }
    });

    // Preprocess the numbers in the initial array
    nums.forEach(value => {
        // If the number is odd, multiply it by 2 (to make it even)
        if (value % 2 !== 0) value *= 2;

        // Push the possibly altered value onto the priority queue
        maxQueue.queue(value);

        // Update the minimum found so far
        minValue = Math.min(minValue, value);
    });

    // Calculate the initial deviation between the max value and the minValue
    let deviation: number = maxQueue.peek() - minValue;

    // While the maximum element in maxQueue is even
    while (maxQueue.peek() % 2 === 0) {
        // Take the top (max) element and divide it by 2
        let maxValue: number = maxQueue.dequeue() / 2;

        // Push the new divided value back onto the priority queue
        maxQueue.queue(maxValue);

        // Update the minimum value if necessary
        minValue = Math.min(minValue, maxValue);

        // Update the deviation between current max and the minValue
        deviation = Math.min(deviation, maxQueue.peek() - minValue);
    }

    // Return the minimum deviation found
    return deviation;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by the following operations:

1. The loop that processes each element to identify if it's odd and to potentially double it, which runs in `O(n)`, where `n` is the size of the input list `nums`.
2. The `heapify` operation on the list `h` which has a time complexity of `O(n)`.
3. The `while` loop, which has a variable number of iterations depending on the values in the heap. In the worst case, each element may be divided by 2 until it becomes odd, which can happen at most `log(max_element)` times for each element. Since each iteration involves a `heappop` and `heappush`, both of which have `O(log(n))` complexity, the total for this part is `O(n * log(n) * log(max_element))`.

Adding these up, we get:

- Initial loop and heapify: `O(2n) = O(n)`
- While loop: `O(n * log(n) * log(max_element))`

Since `log(max_element)` will be at most around 30 for integers within the 32-bit range, we can treat it as a constant factor, simplifying our worst-case time complexity to `O(n * log(n))`.

Space Complexity

The space complexity of the code is determined by the additional space used, which are:

1. The heap `h`, which contains at most `n` elements, so `O(n)`.
2. The variable `mi` and other temporary variables, which use a fixed amount of space and thus can be considered `O(1)`.

Therefore, the total space complexity is `O(n)` due to the heap.