2293. Min Max Game

Easy Array Simulation

Leetcode Link

Problem Description

You have an array of integers, `nums`, whose length is guaranteed to be a power of `2`. A specific algorithm needs to be applied to this array until only one number is left. The steps of the algorithm are as follows:

1. If the length of `nums` is 1, stop.
2. Create a new array named `newNums` of length `n / 2`, where `n` is the current length of `nums`.
3. Fill `newNums` by iterating through `nums` in pairs. For every even index `i`, set `newNums[i]` to the smaller number in the pair from `nums`. For every odd index `i`, set `newNums[i]` to the larger number in the pair.
4. Replace `nums` with `newNums`.
5. Repeat the process starting from step 1.

The goal is to return the single remaining number after repeating this process.

Intuition

Understanding the algorithm is critical. You repeatedly shrink the `nums` array by half until it consists of a single element. Each iteration consists of pairwise comparisons. For even-indexed pairs, you select the smaller number, and for odd-indexed pairs, you select the larger number.

The approach involves:

1. Using a loop to reduce the array size until one element is left.
2. Iterating over the current `nums` to create the `newNums` based on the even and odd index rule.
3. Overwriting `nums` with `newNums` after each iteration.

The transformation defined by the algorithm ensures that after each iteration, the size of `nums` is halved. The bitwise operations (`n >>= 1`, `i <<= 1`, `i |= 1`) in the solution are efficient ways to perform calculations needed for indexing and updating the array. The expression `n >>= 1` halves `n`, `i <<= 1` doubles `i` (used to find the base index of a pair), and `i |= 1` finds the immediate next index (the second element of the pair).

By representing pairs of elements as `a` and `b`, the code is more readable and easier to work with. The condition `if i % 2 == 0` checks if the index `i` is even, using the smaller value of the pair, otherwise using the larger value for odd indices.

This algorithmic process is inherently divide-and-conquer in nature, breaking down the problem into smaller pieces and then combining the results. Since the length of the array is a power of 2, we are assured that we will end with exactly one element.

Solution Approach

The solution uses a simple while loop to repeatedly apply the steps of the algorithm as follows:

- The condition for the while loop `while n > 1` checks whether there is more than one element left in `nums`. If only one element is left, we have reached the end of the process.
- Inside the loop, the operation `n >>= 1` efficiently halves `n`. This operation is equivalent to `n = n // 2` but is quicker since it is a bitwise operation.
- A for loop `for i in range(n)` is then used to iterate through the first `n` elements of the array and perform the pairwise comparison and assignment to `newNums`. Instead of explicitly creating a `newNums` array, the solution saves space by reusing the first half of the original `nums` array to store the new values.
- The variables `a` and `b` represent the elements of the current pair being considered, where `a` is at the index `i << 1` (double the index `i`) and `b` is at the index `i << 1 | 1` (one more than double the index `i`). The use of bitwise operations makes the index calculation efficient.
- The condition `if i % 2 == 0` checks if the index `i` is even, and if so, the code uses `min(a, b)` to select the smaller of the two elements. If `i` is odd, the code uses `max(a, b)` to select the larger of the two elements.
- After iterating through all pairs, the new values are already stored in the first half of `nums`, and we can repeat the process until only one element remains.

There are no additional data structures used in the implementation aside from basic variables and the input array, which highlights the in-place nature of this solution. This implementation benefits from space efficiency by not needing extra arrays. The solution emphasizes an understanding of bitwise manipulation (`>>`, `<<`, `|`) to perform operations on indices, a common pattern for algorithms with a binary nature or when dealing with arrays where the size is a power of 2.

The idea of the solution is quite straightforward once you understand how the array evolves after each round of the algorithm, and no complex data structures or algorithms are needed aside from a couple of loops and simple conditions.

Example Walkthrough

Let's walk through a hypothetical example to better understand the solution approach.

Suppose the input array is `[3, 1, 4, 2, 8, 6, 5, 7]`. The length of this array is `8`, which is a power of `2` (2^3).

Here's how we apply the algorithm step by step:

Iteration 1:

- The current length of `nums` is `8`. We need to create a `newNums` array of length `8 / 2 = 4`.
- We iterate through `nums` in pairs: `(3, 1)`, `(4, 2)`, `(8, 6)`, `(5, 7)`.
- We fill `newNums` by selecting the smaller number for the first pair and the larger number for the second pair: `newNums = [1, 4, 8, 7]`.

Iteration 2:

- The reduced `nums` is now `[1, 4, 8, 7]`.
- We halve the length again for `newNums`, which will now have a length of `2`.
- Iterating in pairs: `(1, 4)`, `(8, 7)`.
- Following the even-odd index rule, `newNums` becomes: `[1, 8]`.

Iteration 3:

- Now `nums` is `[1, 8]`.
- We halve the length again, and since it will be `1`, this will be the final iteration.
- There's only one pair: `(1, 8)`.
- Following the even-odd index rule, `newNums` ends up as just `[1]`.

After the third iteration, `nums` is reduced to a single element array `[1]`. The algorithm stops here as the length of `nums` is 1. The remaining single number, `1`, is the answer and will be returned by the algorithm.

In summary, the sequence of `nums` arrays through the iterations would look like this:

1. `[3, 1, 4, 2, 8, 6, 5, 7]`
2. `[1, 4, 8, 7]`
3. `[1, 8]`
4. `[1]`

The result, which is the last remaining number after all reductions, is `1`.

Python Solution

```python
from typing import List

class Solution:
    def minMaxGame(self, nums: List[int]) -> int:
        # Get the length of the nums list
        n = len(nums)

        # Continually reduce the list size until only one element is left
        while n > 1:
            # The size is halved because each pair of elements will be reduced to one element
            n //= 2

            # Iterate over the list in pairs
            for i in range(n):
                # Retrieve the pair of elements to consider
                first_element = nums[2 * i]  # The even-indexed element
                second_element = nums[2 * i + 1] # The odd-indexed next element of the pair

                # If the index of the pair is even, store the minimum of the pair,
                # otherwise, store the maximum
                if i % 2 == 0:
                    nums[i] = min(first_element, second_element)
                else:
                    nums[i] = max(first_element, second_element)

        # Once the list is reduced to one element, return that element as the result
        return nums[0]
```

Java Solution

```java
class Solution {
    // Method to calculate the minimum or maximum number alternatively until
    // there's only one number left in the array.
    public int minMaxGame(int[] nums) {
        // Continue the process until the length of the current array
        // is reduced to one.
        for (int n = nums.length; n > 1;) {
            n >>= 1; // Divide the size of the array by 2, round down if it is odd.
            // Traverse the array for the current size.
            for (int i = 0; i < n; ++i) {
                // Find the elements from the original array that will be compared.
                int firstElement = nums[i << 1];    // Equivalent to two times i.
                int secondElement = nums[(i << 1) | 1]; // Equivalent to two times i plus one.

                // For even indices, assign the minimum of the two elements.
                // For odd indices, assign the maximum of the two elements.
                // The bitwise AND operator "&" checks if i is even or odd.
                nums[i] = (i & 1) == 0 ? Math.min(firstElement, secondElement) : Math.max(firstElement, secondElement);
            }
        }

        // At the end, there's only one element left in the array, which is
        // the result of the min-max game.
        return nums[0];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to find the min-max game result from a given vector of integers.
    int minMaxGame(vector<int>& nums) {
        // Loop until we have only one element left in 'nums'.
        for (int size = nums.size(); size > 1;) {
            // Divide the size by two to determine the new size after pairing elements.
            size >>= 1;

            // Iterate through the elements in pairs.
            for (int i = 0; i < size; ++i) {
                // Determine the pair of elements to play the game with.
                int firstElement = nums[i << 1];
                int secondElement = nums[(i << 1) | 1];

                // Update the element at position 'i' with the minimum if 'i' is even
                // or the maximum if 'i' is odd as per game rules.
                nums[i] = i % 2 == 0 ? min(firstElement, secondElement) : max(firstElement, secondElement);
            }
        }
        // Once we have reduced the list to a single element, return it.
        // It is the result of the min-max game.
        return nums[0];
    }
};
```

Typescript Solution

```typescript
function minMaxGame(nums: number[]): number {
    // Continue to iterate until only one element remains in the nums array.
    for (let newSize = nums.length; newSize > 1; ) {
        // Divide the size of the array by 2 for the new iteration.
        newSize >>= 1;

        // Iterate over each pair of elements to apply min or max operations.
        for (let index = 0; index < newSize; ++index) {
            // Get the pair of elements to compare.
            const firstElement = nums[index << 1];
            const secondElement = nums[(index << 1) | 1];

            // If the index is even, assign minimum of the pair to the current position.
            // If the index is odd, assign maximum of the pair to the current position.
            nums[index] = index % 2 === 0 ? Math.min(firstElement, secondElement) : Math.max(firstElement, secondElement);
        }
    }
    // After reducing the nums array to one element, return that element (result of the min-max game).
    return nums[0];
}
```

Time and Space Complexity

The given code performs a series of actions to transform an initial list of numbers into just one number according to the rules of the "MinMax Game". For each round, the number of elements in the list is halved, and the new elements are either the minimum or maximum of pairs of the original elements, depending on their position.

Time Complexity:

The time complexity of this code is analyzed based on the number of operations needed to reach the final number. Initially, there are `n` elements, and in each subsequent round, the number of elements is halved. This halving continues until there is just one element left. The number of rounds is equal to `log2(n)` since we are halving the element count successively.

In each round, the operations performed are constant-time comparisons and assignments. For the first round, there are `n/2` operations, for the second round `n/4`, then `n/8`, and so on, until we perform just one operation. When you sum these operations, it forms a geometric series: `n/2 + n/4 + n/8 + ... + 1`. The sum of this geometric series converges to `n`, which means the total number of operations will be `O(n)`.

Space Complexity:

The space complexity is determined by the amount of extra space used. The given code modifies the input list `nums` in place, and does not use any additional space that grows with the input size. The space used for variables is constant, and does not depend on the size of the input.

Hence, the space complexity is `O(1)`, indicating constant space complexity.