2974. Minimum Number Game

Easy | Array | Sorting | Simulation | Heap (Priority Queue)

Problem Description

In this problem, we have a simple game involving two players, Alice and Bob, and an integer array `nums` that has an even number of elements. The game follows a specific order of operations in each round:

1. Alice starts by removing the smallest element from the `nums` array.
2. Bob follows by also removing the now smallest element from the `nums` array.
3. Bob then adds the element he removed to the array, `arr`.
4. Alice adds the element she removed to the array, `arr`.

This process repeats until the `nums` array is empty. The task is to simulate this game and return the final state of the `arr` array.

To solve this problem, you need to think about how to efficiently track and remove the smallest elements and in which order they should be added to the `arr` array.

Intuition

The core of the solution lies in consistently selecting the two smallest elements from the `nums` array with each round. To do this efficiently, we use a data structure called a min-heap, which allows us to access and remove the smallest element in an optimally fast manner.

The intuition behind using a min-heap is that it maintains the elements of `nums` in a way that we can always quickly extract the minimum element. After initializing the heap with the elements of `nums`, we repeatedly perform the following actions:

1. Remove the smallest element `a` (Alice's move).
2. Remove the next smallest element `b` (Bob's move).
3. Add element `b` to the `arr` array first (Bob's addition to `arr`).
4. Add element `a` to the `arr` array second (Alice's addition to `arr`).

This effectively simulates the game's rules, and we continue this process until the `nums` array (min-heap) is empty. The resulting `arr` array will be the final output of the algorithm, which is the order in which elements were appended to `arr` by Bob and Alice according to the game's rules.

Solution Approach

The solution to this problem is implemented using a priority queue, which is a type of data structure that always gives priority to the element with the least value.

In Python, the priority queue can easily be implemented using the `heapq` module, which turns a regular list into a min-heap. Below is a step-by-step breakdown of the solution approach:

1. **Heapify the Array:** We start by turning the original array `nums` into a min-heap using the `heapify` function from the `heapq` module. The `heapify` process reorders the array into a heap, so that `heappop()` can be used in the subsequent steps to always get the smallest item efficiently.
2. **Initialize the Result Array:** We define an empty list called `ans`, which will serve as the `arr` array where Alice and Bob will append their removed elements based on the game's rules.
3. **Simulate the Game Rounds:** a. In a loop that runs as long as there are elements in `nums`, we simulate a single round by performing two `heappop` operations to get the two smallest elements from `nums`, `a` and `b`. Here, `a` will be the smallest element and `b` will be the second smallest. b. We append `b` to the `ans` list first (since Bob appends his choice to `arr` first), and then append `a`.
4. **Final Output:** After the loop ends (when `nums` is empty), we return the `ans` list. This list is now the final state of the `arr` array after all rounds of the game have been played according to the rules specified in the problem description.

Here are the core lines of code that correspond to each step enclosed in "`":

- Convert nums into a min-heap: `heapify(nums)`
- Loop until nums is empty, simulating the game rounds:

```
while nums:
    a, b = heappop(nums), heappop(nums)
    ans.append(b)
    ans.append(a)
```

- Return the result: `return ans`

In conclusion, the use of a min-heap is crucial because it allows us to simulate the rules of the game in an efficient way since we are repeatedly needing to access and eliminate the minimum elements from the array.

Example Walkthrough

Let's illustrate how this solution approach works with a small example. Assume we have the following integer array, `nums`:

```
[3, 1, 4, 2]
```

First, we convert `nums` into a min-heap using `heapify(nums)`. For our small-sized array, the heapify process might not change the arrangement much, but for a larger dataset, the heap would significantly help in extracting the minimal elements efficiently.

Now our array, represented as a min-heap, looks like this:

```
[1, 2, 4, 3]
```

Here's how we simulate the game rounds:

1. `heappop(nums)` removes `1` (Alice's move) and the heap now looks like `[2, 3, 4]`.
2. `heappop(nums)` again removes `2` (Bob's move), resulting in a heap of `[3, 4]`.
3. We append `2` to `ans` first (Bob's addition to `arr`), so `ans` now looks like `[2]`.
4. We append `1` to `ans` (Alice's addition to `arr`), making `ans` look like `[2, 1]`.

The `nums` heap after the first round is now `[3, 4]`. We repeat the process:

1. `heappop(nums)` removes `3` (Alice's move), and the heap is `[4]`.
2. `heappop(nums)` removes the last element `4` (Bob's move), and the heap is now empty.
3. We append `4` to `ans` (Bob's addition to `arr`), so `ans` now looks like `[2, 1, 4]`.
4. Finally, we append `3` to `ans` (Alice's addition to `arr`), resulting in `ans` being `[2, 1, 4, 3]`.

The `nums` array is now empty, so our process is complete. The final state of the `arr` array returned by our function is `[2, 1, 4, 3]`.

In this example, we followed the steps of the solution approach exactly, prioritizing the removal of the smallest elements and appending them to the result array `ans` in the order specified by the game's rules. The use of a min-heap made the process of selecting the smallest elements straightforward and efficient.

Solution Implementation

Python

```python
from heapq import heapify, heappop # Import necessary functions from the heapq module

class Solution:
    def numberGame(self, numbers):
        """
        Reorder numbers from a min-heap such that each element from the heap
        is appended in alternating order to a new list.

        :param numbers: List[int]
        :return: List[int]
        """
        # Convert the list into a heap in place
        heapify(numbers)

        # Initialize an empty list to hold the reordered elements
        reordered_numbers = []

        # Continue until the heap is empty
        while numbers:
            # Pop the two smallest elements from the heap.
            # Note that the list must have an even number of elements,
            # otherwise, the second pop operation would fail when the list becomes empty.
            first_num = heappop(numbers)
            second_num = heappop(numbers)

            # Append the numbers to the reordered list in alternating order
            reordered_numbers.append(second_num)
            reordered_numbers.append(first_num)

        # Return the list with elements reordered
        return reordered_numbers
```

Java

```java
import java.util.PriorityQueue;

class Solution {

    // Method to reorder the numbers in a specific pattern.
    public int[] numberGame(int[] numbers) {
        // Create a min-heap priority queue to order numbers in increasing order.
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Add all numbers in the input array into the priority queue.
        for (int num : numbers) {
            minHeap.offer(num);
        }

        // Create an array to store the answer.
        int[] answer = new int[numbers.length];
        // Initialize an index counter to zero.
        int index = 0;

        // Continue this process until the priority queue is empty.
        while (!minHeap.isEmpty()) {
            // Poll (remove) the smallest element from the queue.
            int first = minHeap.poll();
            // Check if there's another number to pair with the polled number.
            if (!minHeap.isEmpty()) {
                // Poll the next smallest element to pair with the first.
                int second = minHeap.poll();
                // Place the second item first in the answer array.
                answer[index++] = second;
            }
            // Place the first item after the second item in the answer array.
            answer[index++] = first;
        }

        // Return the re-ordered array as the answer.
        return answer;
    }
}
```

C++

```cpp
#include <vector>
#include <queue> // Required for priority_queue
using namespace std;

class Solution {
public:
    // Renamed the method to reflect standard naming practices and added comments
    vector<int> numberGame(vector<int>& nums) {
        // Use a min-heap to store the numbers in increasing order
        priority_queue<int, vector<int>, greater<int>> minHeap;

        // Add all numbers from the input vector to the min-heap
        for (int num : nums) {
            minHeap.push(num);
        }

        // Prepare an answer vector to store the results
        vector<int> answer;

        // Iterate until the min-heap is empty
        while (!minHeap.empty()) {
            int smaller = minHeap.top(); // Get the smallest number
            minHeap.pop(); // Remove the smallest number from the min-heap

            // Since the problem statement is not clear about what should be done if there is
            // an odd number of elements in the heap, it's important to check if the heap is
            // not empty before trying to access the top element again.
            if (!minHeap.empty()) {
                int larger = minHeap.top(); // Get the next smallest number
                minHeap.pop(); // Remove it from the min-heap

                answer.push_back(larger); // Add the larger of the two to the answer
                answer.push_back(smaller); // Then add the smaller
            } else {
                // If there is an odd number of elements, the last element won't have a pair.
                // Assuming we're to add it directly to the answer, as per the original code,
                // which would otherwise throw an error if not handled.
                answer.push_back(smaller);
            }
        }

        // Return the final populated answer vector
        return answer;
    }
};
```

TypeScript

```typescript
// Import the MinPriorityQueue class from the 'typescript-collections' library.
// Note: you must have this library installed for this code to work.
import { MinPriorityQueue } from 'typescript-collections';

// Takes in an array of numbers and arranges them in a specific order.
// The smallest two numbers are dequeued from the priority queue;
// the second-dequeued (larger) number is placed before the first-dequeued (smaller) number in the output array.
// This process repeats until the priority queue is empty.
// @param nums - The array of numbers to be processed.
// @returns An array of numbers arranged according to the specific game rule.
function numberGame(nums: number[]): number[] {
    // Create a new minimum priority queue to store the numbers.
    const priorityQueue = new MinPriorityQueue<number>();

    // Add each number to the priority queue.
    for (const num of nums) {
        priorityQueue.enqueue(num);
    }

    // Initialize the array that will hold the re-arranged numbers.
    const result: number[] = [];

    // Continue processing until the priority queue is empty.
    while (!priorityQueue.isEmpty()) {
        // Dequeue the smallest number.
        const smallerNumber = priorityQueue.dequeue().element;

        // Check if there is another number to pair with the one dequeued.
        // If the queue is empty, push the last number into the result and break the loop.
        if (priorityQueue.isEmpty()) {
            result.push(smallerNumber);
            break;
        }

        // Dequeue the next smallest number.
        const largerNumber = priorityQueue.dequeue().element;

        // Add the second (larger) number before the first (smaller) one into the result array.
        result.push(largerNumber, smallerNumber);
    }

    // Return the re-arranged array of numbers.
    return result;
}
```


```python
from heapq import heapify, heappop # Import necessary functions from the heapq module

class Solution:
    def numberGame(self, numbers):
        """
        Reorder numbers from a min-heap such that each element from the heap
        is appended in alternating order to a new list.

        :param numbers: List[int]
        :return: List[int]
        """
        # Convert the list into a heap in place
        heapify(numbers)

        # Initialize an empty list to hold the reordered elements
        reordered_numbers = []

        # Continue until the heap is empty
        while numbers:
            # Pop the two smallest elements from the heap.
            # Note that the list must have an even number of elements,
            # otherwise, the second pop operation would fail when the list becomes empty.
            first_num = heappop(numbers)
            second_num = heappop(numbers)

            # Append the numbers to the reordered list in alternating order
            reordered_numbers.append(second_num)
            reordered_numbers.append(first_num)

        # Return the list with elements reordered
        return reordered_numbers
```


Time and Space Complexity

The time complexity of the code provided operates in two primary steps, heapification of the `nums` list and the subsequent pop operations. Heapification is completed in `O(n)` time. After heapification, the while loop pops two elements for every iteration until the heap is empty. Since there are `n/2` iterations and each pop operation from a heap is `O(log n)`, the cumulative time complexity for the pop operations is `O(n/2 * 2 * log n) = O(n * log n)`. Therefore, the overall time complexity of the code is `O(n + n * log n)`, which simplifies to `O(n * log n)` as `n * log n` dominates `n`.

The space complexity of the code is largely governed by the `ans` array that at most will contain `n` elements (if all elements from the input list `nums` are added to it). The in-place heapify operation does not consume additional space proportional to the input. Thus, the space complexity is `O(n)`.