2757. Generate Circular Array Values

Medium

Leetcode Link

Problem Description

In this LeetCode problem, we're asked to simulate a circular traversal through an array using a generator function. A circular array means that after reaching the end of the array, the next element is again the first element of the array, and similarly, if we are at the beginning of the array and we need to go back, we should end up at the end of the array.

We are given an array `arr` and an initial `startIndex` from where to start yielding elements when the generator's `next` method is called. For every subsequent call to `next`, a `jump` value is provided, which determines the number of steps to move from the current position. If `jump` is positive, we move forward, and if `jump` is negative, we move backward. Due to the circular nature of the array, these movements might wrap around the beginning or the end of the array. The problem is to correctly calculate the next index considering these wraps.

Intuition

The key to solving this problem is understanding how to handle the circular aspect of the array traversal. We need to yield the element at `startIndex` first, then update the index in a circular manner every time we receive a new `jump` value.

We do this by using modular arithmetic. Adding the `jump` value to `startIndex` and then taking the remainder of dividing by the length of the array (`n`) ensures that we cycle through indices `0` to `n-1`. However, in JavaScript and TypeScript, if `jump` is negative, using the modulo operator directly would give us a negative index. To handle the negative indices correctly, we add `n` to the sum and then take the modulo again. This effectively rotates the indices in a circular fashion while keeping them within the valid range of array indices.

The use of a generator function simplifies the iterative yielding of values, handling state between the yields, and accepting the `jump` input each time the generator resumes.

Solution Approach

The main algorithm in the solution is based on the concept of a generator function in TypeScript, which allows us to lazily produce a sequence of values on demand using the `yield` keyword. In our case, the sequence of values is the elements of the array, produced according to the circular traversal defined by the `jump` values.

Here's a step-by-step breakdown of the solution approach, explaining the algorithm, data structures, and pattern used:

1. A generator function named `cycleGenerator` is declared, which takes two parameters: an array `arr` and a starting index `startIndex`.
2. The length of the array is stored in a constant `n`. We use `n` for modulo operations to ensure indices are kept within bounds. The array's length will not change during the execution of the generator, so calculating it once is efficient.
3. The generator enters an infinite `while (true)` loop. This loop will continue to produce values every time the generator is resumed with a `gen.next()` call.
4. Inside the loop, the current element corresponding to `startIndex` is yielded via the `yield` statement. When `yield` is executed, the generator function is paused, and the yielded value (in this case, `arr[startIndex]`) is returned back to the caller.
5. On subsequent `next` calls, a `jump` value is passed, and the generator function resumes. The next index is calculated by adding the `jump` to `startIndex` and then applying modulo `n` to keep the index within bounds. As JavaScript's modulo can yield negative results, `(startIndex + jump) % n` is added to `n` and modulo `n` is taken again to ensure the index is positive.

```
startIndex = (((startIndex + jump) % n) + n) % n;
```

6. The calculation `((startIndex + jump) % n)` can yield a negative index if `jump` is negative. By adding `n` and taking the modulo again `((... + n) % n)`, we guarantee that the final index is in the range `[0, n-1]`.
7. The updated `startIndex` value will be used in the next iteration to yield the next element, and the process repeats each time `gen.next(jump)` is called.

The algorithm leverages the efficiency of generators for state management between yields and the simplicity of modular arithmetic to handle circular indexing. No additional data structures are needed since we directly operate on the given array and modify the index accordingly.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Assume we have an array `arr = [10, 20, 30, 40, 50]` and we want to start our circular traversal from `startIndex = 2`, which corresponds to the value `30` in the array.

1. We create a generator using the `cycleGenerator` function and initialize it with our array and starting index:

```
const gen = cycleGenerator([10, 20, 30, 40, 50], 2);
```

2. When we first call `gen.next().value`, it will yield the value at `startIndex = 2`, which is `30`. The generator is now paused at the `yield` keyword.
3. Next, we call `gen.next(2).value`, meaning we want to jump two places forward from index `2`. Since our array has 5 elements, index `4` is `50`, which is what the generator yields this time.
4. If we call `gen.next(1).value` now, we want to jump one more place forward, but since we're at the end of the array, we wrap around to the beginning, so the generator yields the first element of the array which is `10`.
5. To jump backwards, we can pass a negative value to `next`, such as `gen.next(-3).value`. If our current position was index `0`, we will wrap around to the end of the array and move two places back, ending up at index `2`. Given our array, the returned value would be `30`.

In terms of the implementation details, here's how each step would operate:

- The array's length `n` is determined (`n = 5` in this case).
- The infinite `while` loop begins execution, preparing to yield values upon each `next` invocation.
- Yield the value `arr[startIndex]` where `startIndex = 2` during the first call, then use calculation for subsequent indexes.
- For each subsequent call to `next`, accept a `jump` value and calculate the new index with the modulo operation outlined in the solution.
- If `jump` is positive, the new position is found easily with `(startIndex + jump) % n`.
- If `jump` is negative, the modulo operation may yield a negative result. In this case, the expression `((startIndex + jump) % n + n) % n` ensures that the resulting index is positive and within the valid range.

The efficiency of this solution is in its simplicity: it uses fundamental programming concepts like loops and modular arithmetic to solve the problem of circular array traversal without the need for complex data structures or additional memory overhead.

Python Solution

```python
# A generator function that creates an infinite cycle over the input list.
# You can jump to any index in the list by providing the "jump" value when calling next().
# @param arr The list to be cycled through.
# @param start_index The index at which to start the cycle.
# @returns A generator that yields values from the list.
def cycle_generator(arr, start_index):
    # Determine the length of the list to handle the cycling logic.
    n = len(arr)

    # Start an infinite loop to allow the cycling.
    while True:
        # Yield the current element of the list and receive the jump value from the next() call.
        jump = yield arr[start_index]

        # Calculate the new start_index by adding the jump value and using modulo operation
        # to ensure it wraps around the list. The additional +n ensures the result is non-negative.
        start_index = ((start_index + jump) % n + n) % n

# Example usage:
# Create an instance of the generator, starting at index 0 of the provided list.
gen = cycle_generator([1, 2, 3, 4, 5], 0)

# Calling next() without a parameter to retrieve the first value, which will be at the starting index.
print(next(gen))   # Outputs: 1

# Calling gen.send(jump) with a parameter to jump to the next index in the cycle.
print(gen.send(1))  # Outputs: 2 (jumps 1 index forward)

# Again calling gen.send(jump) with a different parameter to jump to subsequent indices in the cycle.
print(gen.send(2))  # Outputs: 4 (jumps 2 indices forward from the current position)

# Demonstrate the cycling behavior with a jump that exceeds the list's length.
print(gen.send(6))  # Outputs: 5 (jumps 6 indices forward, cycling back to the end of the list)

```

Java Solution

```java
import java.util.function.IntUnaryOperator;

public class CycleIterator implements IntUnaryOperator {
    private final int[] array;
    private int currentIndex;

    // Constructor to initialize the iterator with the array and the starting index.
    public CycleIterator(int[] array, int startIndex) {
        this.array = array;
        // Normalize the start index in case it's negative or greater than the array length.
        this.currentIndex = ((startIndex % array.length) + array.length) % array.length;
    }

    // The method to get the next element. Also receives a "jump" value to move the current index.
    public int next(int jump) {
        // Retrieve the current element.
        int currentElement = array[currentIndex];
        // Update the currentIndex with the jump, making sure it's within the array bounds.
        currentIndex = (((currentIndex + jump) % array.length) + array.length) % array.length;
        // Return the current element.
        return currentElement;
    }

    // Java's applyAsInt method to adhere to the IntUnaryOperator functional interface.
    @Override
    public int applyAsInt(int operand) {
        return next(operand);
    }
}

// Example usage:
public class CycleGeneratorExample {
    public static void main(String[] args) {
        // Create an instance of CycleIterator, starting at index 0 of the provided array.
        CycleIterator iterator = new CycleIterator(new int[]{1, 2, 3, 4, 5}, 0);

        // Get the first value, which will be at the starting index.
        System.out.println(iterator.next(0)); // Outputs: 1

        // Next(jump) with a parameter to jump to the next index in the cycle.
        System.out.println(iterator.next(1)); // Outputs: 2

        // Again, call next(jump) with a different jump value.
        System.out.println(iterator.next(2)); // Outputs: 4

        // Demonstrate the cycling behavior with a jump that exceeds the array's length.
        System.out.println(iterator.next(6)); // Outputs: 5
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <vector>
#include <functional>

// A class representing a generator that produces a cycle over the input vector.
class CycleGenerator {
private:
    std::vector<int> arr;  // The vector to be cycled through.
    size_t startIndex;     // The current starting index within the vector.

public:
    // Construct the generator with a given vector and a start index.
    CycleGenerator(const std::vector<int>& arr, size_t startIndex) : arr(arr), startIndex(startIndex) {}

    // Function to get the next value in the cycle.
    // Optionally, jump to a different index in the vector.
    int next(int jump = 0) {
        // Determine the length of the vector to handle the cycling logic.
        const size_t n = arr.size();

        // Get the current element to yield.
        int currentValue = arr[startIndex];

        // Calculate the new startIndex by adding the jump value.
        // Using modulo operation to wrap around if necessary.
        // The additional +n and modulo is used to ensure the result is non-negative.
        startIndex = (((startIndex + jump) % n) + n) % n;

        // Return the current value.
        return currentValue;
    }
};

// Example usage:
int main() {
    // Create an instance of the generator starting at index 0 of the provided vector.
    CycleGenerator gen({1, 2, 3, 4, 5}, 0);

    // Retrieve the first value, which will be at the starting index.
    std::cout << gen.next() << std::endl;  // Outputs: 1

    // Jump to the next index in the cycle by providing a jump value.
    std::cout << gen.next(1) << std::endl; // Outputs: 2 (jumps 1 index forward)

    // Jump to subsequent indices in the cycle with a different jump value.
    std::cout << gen.next(2) << std::endl; // Outputs: 4 (jumps 2 indices forward from the current position)

    // Demonstrate the cycling behavior with a jump that exceeds the vector's length.
    std::cout << gen.next(6) << std::endl; // Outputs: 5 (jumps 6 indices forward, cycling back to the end of the vector)

    return 0;
}

```

Typescript Solution

```typescript
// A generator function that creates an infinite cycle over the input array.
// You can jump to any index in the array by providing the "jump" value when calling next().
// @param arr The array to be cycled through.
// @param startIndex The index at which to start the cycle.
// @returns A generator that yields values from the array.
function* cycleGenerator(arr: number[], startIndex: number): Generator<number, void, number> {
    // Determine the length of the array to handle the cycling logic.
    const n = arr.length;

    // Start an infinite loop to allow the cycling.
    while (true) {
        // Yield the current element of the array.
        const jump = yield arr[startIndex];

        // Calculate the new startIndex by adding the jump value and applying modulo operation.
        // The additional +n and modulo is used to ensure the result is non-negative.
        startIndex = (((startIndex + jump) % n) + n) % n;
    }
}

// Example usage:
// Create an instance of the generator, starting at index 0 of the provided array.
const gen = cycleGenerator([1,2,3,4,5], 0);

// Calling next() without a parameter to retrieve the first value, which will be at the starting index.
console.log(gen.next().value);  // Outputs: 1

// Calling next(jump) with a parameter to jump to the next index in the cycle.
console.log(gen.next(1).value); // Outputs: 2 (jumps 1 index forward)

// Again calling next(jump) with a different parameter to jump to subsequent indices in the cycle.
console.log(gen.next(2).value); // Outputs: 4 (jumps 2 indices forward from the current position)

// Demonstrate the cycling behavior with a jump that exceeds the array's length.
console.log(gen.next(6).value); // Outputs: 5 (jumps 6 indices forward, cycling back to the end of the array)

```

Time and Space Complexity

Time Complexity

The time complexity of each call to `gen.next()` is `O(1)`, presuming that modular arithmetic and yield operations are constant-time operations. This is because there is only a single step each time the generator resumes after yielding—an update to `startIndex` using arithmetic operations.

Space Complexity

The space complexity of the generator function is `O(1)`. No additional space is required proportional to the input size or the number of iterations because the state is encapsulated within the generator's internal variables, and no new memory allocation is occurring within the generator function after it's been instantiated.