1646. Get Maximum in Generated Array

Easy Array Dynamic Programming Simulation

Problem Description

In this problem, you need to work with a special array generation rule to find the maximum element in the generated array. An integer `n` is given which determines the size of an integer array `nums` of length `n + 1`. The array `nums` is constructed following specific rules:

- `nums[0]` is set to `0`.
- `nums[1]` is set to `1`.
- For each even index `2i` where `2 <= 2i <= n`, the value is `nums[2i] = nums[i]`.
- For each odd index `2i + 1` where `2 <= 2i + 1 <= n`, the value is `nums[2i + 1] = nums[i] + nums[i + 1]`.

Your goal is to return the maximum integer value that exists in the array `nums`.

Intuition

Intuitively, the problem can be solved by directly applying the given generation rules to construct the array and then find the maximum value in it. Since the maximum number generated is a result of the addition in the sequence, one would expect it to appear at an odd index because all odd indices are created by summing two elements of the array.

We can observe that each element is derived from previously calculated values. This gives us a hint that we should approach the problem iteratively, constructing the array one number at a time and making use of previously computed elements. This iterative approach ensures that we use O(n) time complexity because each element calculation requires constant time.

Additionally, we can see that the array follows a repetitive pattern based on even and odd indices, which means we can use bitwise operations for efficiency. Notably, when working with an index `i`, `i >> 1` is equivalent to `i//2`, which we need for calculating even indices. For odd indices, we use `i >> 1` (the same as `i//2`) and `(i >> 1) + 1` (the same as `i//2 + 1`).

Using this approach, we create and fill the array `nums` up to index `n` and simply return the maximum value in the array as our solution. Here space complexity is O(n) because we need to store `n + 1` elements in the array `nums`.

Solution Approach

The implementation of the solution follows a straightforward approach based on the rules for generating the array. This solution uses the iterative method to populate the array based on the two conditions for even and odd indices. Let's break down the solution step by step:

1. Initialize the `nums` array with `n + 1` elements, as the length of the array is determined by the input `n`. Initialize all elements to `0`.
2. Since the first two elements `nums[0]` and `nums[1]` are already given by the problem (0 and 1, respectively), we manually set them. This serves as a base case for building up the rest of the array.
3. Iterate over the range from `2` to `n`, inclusive. We divide this process into two cases:
   - When `i` is even (`i % 2 == 0`), we set `nums[i]` to `nums[i >> 1]`. The bitwise right shift operation "`i >> 1`" effectively divides `i` by 2.
   - When `i` is odd, we set `nums[i]` to the sum of `nums[i >> 1]` and `nums[(i >> 1) + 1]`, essentially summing the elements at indices `i//2` and `i//2 + 1`.
4. After we've populated the `nums` array, we return the maximum integer from it using Python's built-in `max()` function.

In terms of data structures, we only use a list to store the sequence, and no additional data structures are needed. As the process involves simple arithmetic and assignment operations, the algorithm doesn't make use of complex patterns. It's an imperative, step-by-step generation of the sequence values using conditions for even and odd integers, followed by a quest for the maximum value.

The time complexity of this solution is O(n), as we iterate over the range once and the max function also traverses the list with complexity O(n). The space complexity is O(n) as well, due to the storage requirements for the `nums` array.

Here is the implementation of our solution:

```
class Solution:
    def getMaximumGenerated(self, n: int) -> int:
        if n < 2:
            return n
        nums = [0] * (n + 1)
        nums[1] = 1
        for i in range(2, n + 1):
            # Populate nums[i] based on even/odd index
            nums[i] = nums[i >> 1] if i % 2 == 0 else nums[i >> 1] + nums[(i >> 1) + 1]
        # Return the max value from the generated array
        return max(nums)
```

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `n = 7`. We want to generate the `nums` array according to the rules and find the maximum element.

1. We initialize `nums` with `n + 1` (8) elements: `nums = [0, 0, 0, 0, 0, 0, 0, 0]`.
2. We set `nums[0]` to `0` and `nums[1]` to `1` as given: `nums = [0, 1, 0, 0, 0, 0, 0, 0]`.
3. We start iterating from index `2` to `7`:
   1. For `i = 2` (even), we use the formula `nums[2] = nums[2 >> 1]`, which is `nums[2] = nums[1]`, so `nums[2] = 1`.
   2. For `i = 3` (odd), `nums[3] = nums[3 >> 1] + nums[(3 >> 1) + 1]`, which is `nums[3] = nums[1] + nums[2]`, so `nums[3] = 2`.
   3. For `i = 4` (even), `nums[4] = nums[4 >> 1]`, which is `nums[4] = nums[2]`, so `nums[4] = 1`.
   4. For `i = 5` (odd), `nums[5] = nums[5 >> 1] + nums[(5 >> 1) + 1]`, which is `nums[5] = nums[2] + nums[3]`, so `nums[5] = 3`.
   5. For `i = 6` (even), `nums[6] = nums[6 >> 1]`, which is `nums[6] = nums[3]`, so `nums[6] = 2`.
   6. For `i = 7` (odd), `nums[7] = nums[7 >> 1] + nums[(7 >> 1) + 1]`, which is `nums[7] = nums[3] + nums[4]`, so `nums[7] = 3`.
5. Now our `nums` array looks like this: `nums = [0, 1, 1, 2, 1, 3, 2, 3]`.
6. The final step is to return the maximum value in `nums`, which is `3`.

This example follows the solution approach step by step to generate the `nums` array and then uses the built-in `max()` function to find the maximum element, which is the objective of our problem. The unit of work done for each iteration illustrates the O(n) time complexity of generating the array. The space complexity is also illustrated with the array holding `n + 1` = 8 elements in this example.

Solution Implementation

Python

```
class Solution:
    def get_maximum_generated(self, n: int) -> int:
        # If the input is less than two, return the input as it is.
        if n < 2:
            return n

        # Initialize the array with zeros and set the second element to one.
        generated_nums = [0] * (n + 1)
        generated_nums[1] = 1

        # Generate the array using the given rules.
        for i in range(2, n + 1):
            if i % 2 == 0:
                # For even indices, the value at the index is equal to
                # the value in the array at half the index.
                generated_nums[i] = generated_nums[i // 2]
            else:
                # For odd indices, the value at the index is the sum of
                # the values in the array at half the index and one more than half.
                generated_nums[i] = generated_nums[i // 2] + generated_nums[i // 2 + 1]

        # Return the maximum value from the generated array.
        return max(generated_nums)
```

Java

```
import java.util.Arrays;

class Solution {

    public int getMaximumGenerated(int n) {
        // Handle the base cases where n is 0 or 1.
        if (n < 2) {
            return n;
        }

        // Initialize an array to store the generated numbers.
        int[] generatedNumbers = new int[n + 1];
        // According to the problem statement, nums[1] should be 1.
        generatedNumbers[1] = 1;

        // Populate the array based on the given rules.
        for (int i = 2; i <= n; ++i) {
            // If i is even, the number is generated using the formula nums[i/2].
            if (i % 2 == 0) {
                generatedNumbers[i] = generatedNumbers[i / 2];
            } else {
                // If i is odd, the number is generated using the sum of nums[i/2] and nums[i/2 + 1].
                generatedNumbers[i] = generatedNumbers[i / 2] + generatedNumbers[i / 2 + 1];
            }
        }

        // Return the maximum number from the array using Java Streams.
        return Arrays.stream(generatedNumbers).max().getAsInt();
    }
}
```

C++

```
class Solution {
public:
    // Function to compute the maximum value in the generated array based on the given rules.
    int getMaximumGenerated(int n) {
        // Handle the base case where n is less than 2.
        if (n < 2) {
            return n;
        }

        // Initialize the array with enough space to hold values up to index n.
        vector<int> generatedNums(n + 1);

        // The first two values are given.
        generatedNums[0] = 0;
        generatedNums[1] = 1;

        // Populate the array based on the given rule:
        // If i is even, then generatedNums[i] = generatedNums[i / 2].
        // If i is odd, then generatedNums[i] = generatedNums[i / 2] + generatedNums[i / 2 + 1].
        for (int i = 2; i <= n; ++i) {
            if (i % 2 == 0) {
                // i is even: use the formula for even indices.
                generatedNums[i] = generatedNums[i / 2];
            } else {
                // i is odd: use the formula for odd indices.
                generatedNums[i] = generatedNums[i / 2] + generatedNums[i / 2 + 1];
            }
        }

        // Find and return the maximum element in the array.
        return *max_element(generatedNums.begin(), generatedNums.end());
    }
};
```

TypeScript

```
function getMaximumGenerated(n: number): number {
    // If the input is 0, the maximum generated value is 0.
    if (n === 0) {
        return 0;
    }

    // Create an array initialized with zeros to store the generated values.
    const generatedArray: number[] = new Array(n + 1).fill(0);
    // Base case: the second element in the array is always 1.
    generatedArray[1] = 1;

    // Loop over each index starting from 2 and populate the array following the rules.
    for (let index = 2; index <= n; index++) {
        if (index % 2 === 0) {
            // If the index is even, the value is the same as the value at half the index.
            generatedArray[index] = generatedArray[index / 2];
        } else {
            // If the index is odd, the value is the sum of values at the floor of half the index and one more than that.
            const halfIndex = Math.floor(index / 2);
            generatedArray[index] = generatedArray[halfIndex] + generatedArray[halfIndex + 1];
        }
    }

    // Find and return the maximum value in the generated array.
    return Math.max(...generatedArray);
}
```

```
class Solution:
    def get_maximum_generated(self, n: int) -> int:
        # If the input is less than two, return the input as it is.
        if n < 2:
            return n

        # Initialize the array with zeros and set the second element to one.
        generated_nums = [0] * (n + 1)
        generated_nums[1] = 1

        # Generate the array using the given rules.
        for i in range(2, n + 1):
            if i % 2 == 0:
                # For even indices, the value at the index is equal to
                # the value in the array at half the index.
                generated_nums[i] = generated_nums[i // 2]
            else:
                # For odd indices, the value at the index is the sum of
                # the values in the array at half the index and one more than half.
                generated_nums[i] = generated_nums[i // 2] + generated_nums[i // 2 + 1]

        # Return the maximum value from the generated array.
        return max(generated_nums)
```

Time and Space Complexity

The given Python code generates an array of integers according to specific generation rules and returns the maximum value in the array for a given `n`.

- Time Complexity: The time complexity of the code is `O(n)`. This is because the for loop runs from 2 to `n`, and each operation inside the loop (calculating `nums[i]` and referencing previously computed values) takes constant time.
- Space Complexity: The space complexity of the code is also `O(n)`. This is due to the allocation of a list `nums` of size `n + 1`, where each element is initialized and potentially modified as the for loop executes. No auxiliary space that depends on `n` is used besides this list.