932. Beautiful Array

Medium Array Math Divide and Conquer

Problem Description

We need to construct a `beautiful array` for a given integer `n`. An array is defined to be beautiful if it satisfies two conditions:

1. The array is a permutation of the integers in the range `[1, n]`. That means it contains all the integers from 1 to `n`, each one exactly once.
2. For every pair of indices `0 <= i < j < n`, there is no index `k` such that `i < k < j` and `2 * nums[k] == nums[i] + nums[j]`. In other words, no element is the average of any other two distinct elements not directly next to it.

Our goal is to find at least one such array that meets these criteria.

Intuition

The problem might seem complex at first because the second condition appears to require checking many possible combinations of triplet indices. However, there is a neat recursive approach that can simplify this problem significantly, and it's based on the following observations:

1. If an array is beautiful, then so are its subarrays. So if we have a beautiful array, any subarray extracted from it will also fulfill the beauty condition.
2. If we divide the array into odd and even elements, none of the even elements can be the average of two odd elements, and vice versa. This is because the sum of two odd numbers is an even number, and the average will also be an odd number. Similarly, the average of two even numbers is an even number. Since we don't mix odds and evens, we don't need to worry about condition 2 when merging them back together.

Leveraging these insights, the solution employs divide and conquer strategy, where it recursively generates the left half of the array with odd elements and the right half with even elements. The array generated from the left recursive call is comprised of all odds (by multiplying by 2 and then subtracting 1), and the array from the right recursive call is comprised of all evens (by multiplying by 2). Combining these two beautiful arrays yields a new beautiful array.

Here are the steps of the solution approach:

1. Define a base case for the recursive function: when `n == 1`, return the array `[1]` because it trivially satisfies both conditions.
2. For `n > 1`, make two recursive calls:
   - The first call is for `((n + 1) >> 1)`, which ensures we cover all numbers when `n` is odd.
   - The second call is for `(n >> 1)`, which is straightforward since we are dividing `n` by 2.
3. Convert the left array into odd elements and the right array into even elements.
4. Return the concatenation of the left and right arrays, knowing that they are both beautiful and won't violate conditions when merged, due to the aforementioned observations.

When going step by step through this recursive approach, we are able to build a beautiful array that fulfills the problem's requirements.

Solution Approach

The solution takes advantage of divide and conquer strategy along with recursion. Here's how the implementation is carried out step-by-step, referring to the code provided as a reference:

1. **Base Case**:
   - The recursion starts with the simplest scenario, which is when `n == 1`. In this case, the function just returns a single-element array containing only `[1]` since it's trivially beautiful.
2. **Divide Step**:
   - The array needs to be split into two, one with the odds and one with the evens. The division is not made by slicing the existing array but by reducing the problem size and calling the `beautifulArray` function recursively.
   - The function is called twice recursively, first for the left side with `((n + 1) >> 1)`, and second for the right side with `(n >> 1)`.
     - `>>` is the bitwise right shift operator which effectively divides the number by 2, and in the case of `((n + 1) >> 1)`, it ensures that we account for an additional element in case `n` is odd.
3. **Conquer Step**:
   - The left and right arrays are generated by the recursive calls and are guaranteed to be beautiful by the recursive hypothesis.
   - The next step is to adjust the values for both arrays:
     - For the left array, each element is made odd by the transformation `x * 2 - 1`
     - For the right array, each element is made even by the transformation `x * 2`
4. **Combine Step**:
   - As per the mathematical observations, the even-indexed elements and odd-indexed elements do not interfere with the beauty condition when merged.
   - With this assurance, the function combines (`left + right`) the two arrays which have been separately ensured to be beautiful. Due to the observation that the even elements cannot be the average of two odd elements, this ensures that the combined array is also beautiful.
5. **Data Structures and Patterns**:
   - The main data structure used is the array or list in Python.
   - The pattern used is divide and conquer, which simplifies the problem by breaking it down into smaller subproblems, in this case, creating smaller "beautiful" arrays and then combining them together.

The solution does not require the use of any complex algorithms or additional data structures, relying instead on the mathematical properties of the numbers involved and a straightforward recursive strategy. By ensuring that each recursive call returns a beautiful array, the final result by concatenation is also guaranteed to be a beautiful array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the recursive divide and conquer strategy for constructing a beautiful array. We will use `n = 5` for our example.

1. **Starting Point** Since `n` is 5, it is greater than 1. We need to make two recursive calls to handle the odds and the evens separately.
2. **First Recursive Call (Odds)**
   - We calculate `((n + 1) >> 1)` which in this case is `((5 + 1) >> 1)` = `3`.
   - The recursive call for `beautifulArray(3)` will now be made.
   - Inside this call, since `3 > 1`, two more recursive calls are made:
     - First for `beautifulArray(((3 + 1) >> 1))` which is `beautifulArray(2)`.
     - Second for `beautifulArray((3 >> 1))` which is `beautifulArray(1)`.
   - For `beautifulArray(2)`, we will have another set of recursive calls for 1 (beautifulArray(1)), which just returns `[1]`. The odd transformation `x * 2 - 1` for `[1]` will give us `[1]`.
   - Since `beautifulArray(1)` is the base case, it returns `[1]` as mentioned. The even transformation `x * 2` will give us `[2]`.
   - After the transformations and combining the results, we get the left part as `[1, 2]`.
3. **Second Recursive Call (Evens)**
   - Now we calculate `(n >> 1)` which is `(5 >> 1)` = `2`.
   - The recursive call for `beautifulArray(2)` will now be made.
   - Inside this call, since `2 > 1`, two more recursive calls are made:
     - First for `beautifulArray(((2 + 1) >> 1))` which is `beautifulArray(1)` and we get `[1]`. The odd transformation is `x * 2 - 1` which will give us `[1]`.
     - Second for `beautifulArray((2 >> 1))` which is `beautifulArray(1)` and we also get `[1]`. The even transformation is `x * 2` which will give us `[2]`.
   - After the transformations and combining the results, we get the right part as `[3, 4]`.
4. **Combine Step**
   - Now we have our left and right arrays which are `[1, 2]` and `[3, 4]` respectively.
   - We need to adjust values for both to ensure odds and evens:
     - For the left part, we apply the odd transformation: `[1 * 2 - 1, 2 * 2 - 1]` which results in `[1, 3]`.
     - For the right part, we apply the even transformation: `[3 * 2, 4 * 2]` which results in `[6, 8]`.
   - However, notice that after applying transformations, elements in the right part exceed `n = 5`. We should only include elements up to `n`, which means we take the elements `[2, 4]` (`[6, 8]` is out of the range `[1, n]`).
5. **Concatenation**
   - The left and right arrays are combined to form the beautiful array: `[1, 3] + [2, 4]` which gives us `[1, 3, 2, 4]`.
6. **Final Adjustment**
   - We still need to include all integers from 1 to `n`, which means we need to insert the missing element `5`. Since `5` is an odd number, it can seamlessly be added to the end of our existing array without violating the beautiful array conditions.
   - The final beautiful array is: `[1, 3, 2, 4, 5]`.

Thus, using the divide and conquer approach and ensuring that we only combine elements that won't violate the beautiful array condition, we've constructed a beautiful array for `n = 5`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def beautifulArray(self, n: int) -> List[int]:
        # Base case: if n is 1, return an array containing just the number 1
        if n == 1:
            return [1]

        # Recursively generate the left half of the array with the next odd n
        left_half = self.beautifulArray((n + 1) // 2)
        # Recursively generate the right half of the array with the next even n
        right_half = self.beautifulArray(n // 2)

        # Double each element of the left half and subtract 1 to keep all elements odd
        left_half = [element * 2 - 1 for element in left_half]
        # Double each element of the right half to keep all elements even
        right_half = [element * 2 for element in right_half]

        # Combine the left and right halves to form the beautiful array
        return left_half + right_half
```

Java

```java
class Solution {
    public int[] beautifulArray(int n) {
        // Base case: if the array size is 1, return an array with a single element 1
        if (n == 1) {
            return new int[] {1};
        }

        // Recursively call the function for the first half, rounded up
        int[] leftHalf = beautifulArray((n + 1) >> 1);

        // Recursively call the function for the second half
        int[] rightHalf = beautifulArray(n >> 1);

        // Create an array to hold the beautiful array of size n
        int[] result = new int[n];

        int index = 0; // Initialize the index for the result array

        // Fill the result array with odd numbers by manipulating the left half
        for (int element : leftHalf) {
            result[index++] = element * 2 - 1;
        }

        // Fill the result array with even numbers by manipulating the right half
        for (int element : rightHalf) {
            result[index++] = element * 2;
        }

        // Return the compiled beautiful array
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> beautifulArray(int n) {
        // Base case - if n is 1, return an array containing just the number 1
        if (n == 1) return {1};

        // Recursive call to construct the left half of the beautiful array
        // for the numbers less than or equal to (n + 1) / 2
        vector<int> leftHalf = beautifulArray((n + 1) >> 1);

        // Recursive call to construct the right half of the beautiful array
        // for the numbers less than or equal to n / 2
        vector<int> rightHalf = beautifulArray(n >> 1);

        // Initialize the beautiful array for n elements
        vector<int> result(n);
        int currentIndex = 0;

        // Populate the beautiful array with odd numbers by manipulating elements of the left half
        for (int& element : leftHalf) {
            result[currentIndex++] = element * 2 - 1;
        }

        // Populate the rest of the beautiful array with even numbers by manipulating elements of the right half
        for (int& element : rightHalf) {
            result[currentIndex++] = element * 2;
        }

        // Return the completed beautiful array
        return result;
    }
};
```

TypeScript

```typescript
function beautifulArray(n: number): number[] {
    // Base case - if n is 1, return an array containing just the number 1
    if (n === 1) return [1];

    // Recursive call to construct the left half of the beautiful array
    // for numbers less than or equal to (n + 1) / 2
    let leftHalf: number[] = beautifulArray(Math.ceil(n / 2));

    // Recursive call to construct the right half of the beautiful array
    // for numbers less than or equal to n / 2
    let rightHalf: number[] = beautifulArray(Math.floor(n / 2));

    // Initialize the beautiful array for n elements
    let result: number[] = new Array(n);
    let currentIndex: number = 0;

    // Populate the beautiful array with odd numbers by manipulating elements of the left half
    for (let element of leftHalf) {
        result[currentIndex++] = element * 2 - 1;
    }

    // Populate the rest of the beautiful array with even numbers by manipulating elements of the right half
    for (let element of rightHalf) {
        result[currentIndex++] = element * 2;
    }

    // Return the completed beautiful array
    return result;
}
```

```python
from typing import List

class Solution:
    def beautifulArray(self, n: int) -> List[int]:
        # Base case: if n is 1, return an array containing just the number 1
        if n == 1:
            return [1]

        # Recursively generate the left half of the array with the next odd n
        left_half = self.beautifulArray((n + 1) // 2)
        # Recursively generate the right half of the array with the next even n
        right_half = self.beautifulArray(n // 2)

        # Double each element of the left half and subtract 1 to keep all elements odd
        left_half = [element * 2 - 1 for element in left_half]
        # Double each element of the right half to keep all elements even
        right_half = [element * 2 for element in right_half]

        # Combine the left and right halves to form the beautiful array
        return left_half + right_half
```

Time and Space Complexity

The given Python code generates a beautiful array for a given integer `n`. A beautiful array is an array where for every `i < j < k`, `A[i] * A[k] != 2 * A[j]`. The approach is to recursively split the problem into two halves: generating a beautiful array for the odd numbers and for the even numbers, and then combining these arrays.

Time Complexity

The recursion happens by dividing the problem size roughly in half at each step: once for numbers `> n/2` that will be twiced and decreased by one (odd numbers) and once for numbers `<= n/2` that will be just twiced (even numbers). If `T(n)` represents the time complexity of the `beautifulArray` function, this gives us a recurrence relation similar to the one for the merge sort algorithm, which is `T(n) = 2 * T(n/2) + O(n)`.

At each level of recursion, we perform an operation proportional to `n` (specifically, multiplying and unit decrementing/incrementing each element in the arrays). Since the number of levels of the recursion would be `O(logn)`, multiplying numbers at each level gives us `O(nlogn)`.

Therefore, the overall time complexity of this algorithm is `O(nlogn)`.

Space Complexity

For the space complexity, each recursive call requires its own space to store left and right subarrays, which are of size `n`. The space needed will be the height of the recursion tree, which is `O(logn)`, times the largest width of the tree, which is `O(n)` at the bottom level.

This leads to a space complexity of `O(nlogn)` as well, because we have to consider the space used by all levels of the recursion stack until the base case is reached.