667. Beautiful Arrangement II

Medium Array Math

Problem Description

The problem provides us with two integers `n` and `k`. We are required to construct a list of `n` distinct positive integers, each ranging from `1` to `n`. The list needs to be such that the absolute differences between consecutive elements form a list that has exactly `k` distinct integers. This means that if we say our list is `answer = [a1, a2, a3, ... , an]`, then the list formed by calculating the absolute differences `[|a1 - a2|, |a2 - a3|, |a3 - a4|, ..., |an-1 - an|` should contain `k` unique values.

The task is to find any such list `answer` that satisfies these conditions and return it. This is an interesting problem as it mixes elements of combinatorics, construction, and the understanding of what patterns can emerge from the difference operations.

Intuition

To understand the solution approach, let's keep in mind what we're aiming for: distinct absolute differences. A key observation here is that the largest difference we can get is `n - 1`, which happens when you subtract `1` from `n`. To get `k` distinct differences, we can start by forming the sequence in such a way that it includes the largest differences first, which can be achieved by starting at `1`, then jumping to `n`, then `2`, and so on, alternating between the 'lows' and 'highs'.

For example, if `n` is 10 and `k` is 4, you would start with `[1, 10, 2, 9...]`, because the differences are `[9, 8, 7...]` which covers 3 of the `k` distinct differences we need.

The provided solution alternates between the lowest and highest numbers not yet in the answer, which naturally creates the largest possible differences and starts to get smaller with each pair added to the list. Once we have created `k` differences, we continue the pattern in order to meet the list length `n`, but at this point, we no longer need to create new unique differences, just to maintain the pre-existing pattern.

Notice the alternating pattern in the for-loop where depending on the parity of `i`, we append either `l` (low end of unused numbers) or `r` (high end of unused numbers). The further for-loop picking up from `k` to `n` continues the pattern based on the last number added to ensure we finish with a pattern that still has only `k` unique differences.

Through this process, we're able to construct the desired list, satisfying the condition for `k` distinct differences, using a simple and efficient approach.

Solution Approach

The implementation provided in the reference solution uses a thoughtful pattern to satisfy the condition of `k` distinct differences. To walk you through it:

1. The first `for-loop` iterates `k` times. During each iteration, it alternates between the lowest and highest numbers not yet added to `ans`. This is determined by checking the parity of `i`, the loop index. If it's even, we append and increment the `l` (the next smallest available integer) to `ans`. If it's odd, we append and decrement `r` (the next largest available integer) to `ans`. This loop effectively creates a "zig-zag" pattern in `ans` that guarantees the differences between adjacent numbers in `ans` have `k` distinct values.
2. Once we have enough distinct differences, the objective of creating `k` distinct integers has been met, and we only need to fill the rest of `ans` with the remaining numbers in a way that doesn't create new distinct differences. This is done with the second `for-loop` which starts iterating from `k` up to `n-1`. The appending in this loop continues the established pattern based on the last number added to `ans` before this loop began.
   - If `k` is even, then the pattern must continue with the decreasing order, hence `r` is appended and decremented.
   - If `k` is odd, then the pattern must continue with increasing order, therefore `l` is appended and incremented.

By these steps, a valid `ans` with the right properties is constructed and ready to be returned as the solution.

Here is a high-level breakdown of the algorithm:

- Initialize two pointers, `l` and `r`, to the smallest (1) and largest (`n`) numbers within the specified range respectively.
- Alternate between appending `l` and `r` to `ans`, increasing `l` and decreasing `r` appropriately, until `k` distinct differences have been created.
- Continue filling `ans` with the remaining numbers, ensuring the previously established pattern is maintained, until `ans` has a length of `n`.
- Return `ans` as the final answer.

Example Walkthrough

Let's illustrate the solution approach with a small example. Let's use `n = 5` and `k = 3`. We need to create a list of 5 distinct positive integers, each between 1 and 5, and the absolute differences of consecutive elements should have 3 distinct values.

1. Initialize two pointers, `l = 1` and `r = 5` (smallest and largest numbers).
2. Start the first for-loop to iterate `k` times. The goal here is to alternate between `l` and `r` to get distinct differences.
   - For the first iteration (`i = 0` is even), append `l` to `ans` and increment `l`. `ans = [1]`, `l = 2`, `r` remains `5`.
   - For the second iteration (`i = 1` is odd), append `r` to `ans` and decrement `r`. `ans = [1, 5]`, `l` remains `2`, `r = 4`.
   - For the third iteration (`i = 2` is even), append `l` to `ans` and increment `l`. `ans = [1, 5, 2]`, `l = 3`, `r` remains `4`.
3. Now, `ans` contains `1, 5, 2`. The absolute differences so far are `[4, 3]`. We have our `k = 3` distinct differences, because when we append the next number, it will either be a 3 or a 2, giving us the third difference (which would be `1`), without creating a *fourth* difference.
4. Determine the pattern to finish the last part of the list. We've placed `5` and `2`, and `k` is odd, so we continue with this increasing pattern (because the last movement from `5` to `2` was a decrease).
   - Append `l` to `ans` and increment `l`. `ans = [1, 5, 2, 3]`, `l = 4`, `r` remains `4`.
   - At this point, appending `l` or `r` would get the same number (4 in this case), so we can just add the remaining number to the list.
5. You finish with `ans = [1, 5, 2, 3, 4]`. The absolute differences of consecutive elements are `[4, 3, 1, 1]` which have three distinct values: `4, 3`, and `1`.

This example demonstrates how the pattern works by maximizing differences first and then following the pattern to fill the rest of the list without introducing new distinct differences. The final list `ans = [1, 5, 2, 3, 4]` satisfies the provided conditions.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def construct_array(self, n: int, k: int) -> List[int]:
        # Initialize pointers for the smallest and largest elements
        left, right = 1, n

        # This list will store our answer
        result = []

        # First phase: creating k distinct differences
        for i in range(k):
            # If i is even, append from the left side (increasing numbers)
            if i % 2 == 0:
                result.append(left)
                left += 1
            # If i is odd, append from the right side (decreasing numbers)
            else:
                result.append(right)
                right -= 1

        # Second phase: filling up the rest of the array
        for i in range(k, n):
            # If k is even, fill the rest of the array with decreasing numbers
            if k % 2 == 0:
                result.append(right)
                right -= 1
            # If k is odd, fill the rest of the array with increasing numbers
            else:
                result.append(left)
                left += 1

        # Return the result array
        return result

# Example usage
sol = Solution()
print(sol.construct_array(10, 4))  # This will print an array of size 10 with exactly 4 distinct differences.
```

Java

```java
class Solution {
    public int[] constructArray(int n, int k) {
        // Initialize left and right pointers to the start and end of the range
        int left = 1, right = n;
        // Create an array to store the result
        int[] result = new int[n];

        // Fill the first part of the array with a 'k' difference pattern
        for (int i = 0; i < k; ++i) {
            // Alternate between the lowest and highest unused numbers
            result[i] = (i % 2 == 0) ? left++ : right--;
        }

        // Complete the rest of the array
        for (int i = k; i < n; ++i) {
            // If 'k' is even, decrement from right; otherwise, increment from left
            // This ensures that the difference is no more than 'k'
            result[i] = (k % 2 == 0) ? right-- : left++;
        }

        // Return the constructed array
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> constructArray(int n, int k) {
        // 'left' and 'right' are used to keep track of the smallest and largest elements remaining
        int left = 1, right = n;
        vector<int> result(n); // This will be our final result array

        // The first part of the sequence should have 'k' distinct differences
        for (int i = 0; i < k; ++i) {
            // If 'i' is even, choose from the smallest values, else choose from the largest
            result[i] = (i % 2 == 0) ? left++ : right--;
        }

        // The remaining part of the sequence should be either increasing or decreasing
        for (int i = k; i < n; ++i) {
            // If 'k' is even, keep decreasing, else keep increasing.
            // This keeps the absolute difference to 1, as required for elements after the initial 'k'
            result[i] = (k % 2 == 0) ? right-- : left++;
        }

        return result; // Return the final constructed array
    }
};
```

TypeScript

```typescript
// This function creates an array with a unique set of integers that have k different absolute
// differences. The array is of length 'n', and the differences range between 1 and k.
function constructArray(n: number, k: number): number[] {
    let leftNumber = 1;     // Initialize the starting value for the low end
    let rightNumber = n;    // Initialize the starting value for the high end
    const answer = new Array<number>(n); // Initialize the answer array with length n

    // Fill the first k elements of the array with the pattern to ensure k unique differences
    for (let i = 0; i < k; ++i) {
        // Alternate between the low and high end numbers to create the different differences
        answer[i] = i % 2 === 0 ? leftNumber++ : rightNumber--;
    }

    // Fill the remaining elements of the array
    // If k is even, continue decrementing from the rightNumber
    // If k is odd, continue incrementing from the leftNumber
    for (let i = k; i < n; ++i) {
        answer[i] = k % 2 === 0 ? rightNumber-- : leftNumber++;
    }

    return answer; // Return the constructed array
}

// Example usage:
// constructArray(10, 4) might return [1, 10, 2, 9, 3, 4, 5, 6, 7, 8] (k unique differences from 1 to 4)
```

```python
from typing import List

class Solution:
    def construct_array(self, n: int, k: int) -> List[int]:
        # Initialize pointers for the smallest and largest elements
        left, right = 1, n

        # This list will store our answer
        result = []

        # First phase: creating k distinct differences
        for i in range(k):
            # If i is even, append from the left side (increasing numbers)
            if i % 2 == 0:
                result.append(left)
                left += 1
            # If i is odd, append from the right side (decreasing numbers)
            else:
                result.append(right)
                right -= 1

        # Second phase: filling up the rest of the array
        for i in range(k, n):
            # If k is even, fill the rest of the array with decreasing numbers
            if k % 2 == 0:
                result.append(right)
                right -= 1
            # If k is odd, fill the rest of the array with increasing numbers
            else:
                result.append(left)
                left += 1

        # Return the result array
        return result

# Example usage
sol = Solution()
print(sol.construct_array(10, 4))  # This will print an array of size 10 with exactly 4 distinct differences.
```

Time and Space Complexity

Time Complexity

The provided function consists of two for loops. The first loop runs `k` times, where `k` is a parameter. The second loop starts from `k` and runs until `n`. Therefore, the total number of operations is `k + (n - k) = n`, which means that every element from 1 to `n` is visited exactly once. This gives us a time complexity of `O(n)`.

Space Complexity

The space complexity of the function is determined by the space required to store the output list, `ans`, which contains `n` elements since the function constructs an array of `n` unique integers. No additional data structures that grow with the input size are used. Thus, the space complexity of the function is `O(n)`.