526. Beautiful Arrangement

Medium | Bit Manipulation | Array | Dynamic Programming | Backtracking | Bitmask | Leetcode Link

Problem Description

The problem presents the concept of a *beautiful arrangement*, which is a permutation of `n` integers, where each integer is labeled from `1` through `n`. To be considered beautiful, the arrangement must satisfy a condition for each element `perm[i]`: either `perm[i]` is divisible by its position `i` or the position `i` must be divisible by the value of `perm[i]`. The objective is to find out how many such beautiful arrangements can be made for a given integer `n`.

In simpler terms, if we arrange numbers `1` to `n` in a certain order, for each spot `i` in the order, the number that goes in spot `i` must be such that `i` is divisible by that number or vice versa. The task is to count all the possible ways (permutations) we can arrange the numbers satisfying the above condition for each number and its position.

Intuition

To address this problem, we can use a Depth-First Search (DFS) approach to generate all possible permutations and check which of them qualifies as a beautiful arrangement. However, instead of generating all permutations and then checking the condition for each (which would be time-consuming and inefficient), we can incorporate the given condition into the process of generating permutations. This optimizes the search, as we only recurse further into permutations that are potentially beautiful.

The solution involves the following steps:

- Start by creating a list of possible candidates (`match`) for each position `i` in the arrangement that fulfill the given divisibility condition.
- Use recursive DFS to attempt to construct a beautiful arrangement beginning with position `1` and moving forward.
- Maintain an array (`vis`) that keeps track of which values have been used in the current partial arrangement to avoid using any number more than once.
- At each level of recursion, iterate over all viable candidates for the current position `i`, marking them as used (in `vis`), and calling DFS for the next position `i + 1`.
- If we reach a position beyond `n` (`i == n + 1`), it means we've found a full arrangement that satisfies the conditions, so we increment the result counter (`ans`).
- After each DFS call, we backtrack by unmarking the currently tested value as unused (to consider it for other positions in further iterations).

This approach efficiently explores only potential solutions and counts the number of valid beautiful arrangements, rather than generating all permutations and filtering them afterward.

Solution Approach

The solution uses a recursive function `dfs` to explore all possible arrangements while adhering to the constraints of the problem. Here is a breakdown of the implementation, including algorithms, data structures, and design patterns used:

- **Depth-First Search (DFS)**: DFS is a standard algorithm used to traverse all possible paths in a tree or graph-like structure. This is ideal for our case because we want to explore all permutations that satisfy the conditions for a beautiful arrangement. The `dfs` function represents a node in the DFS tree, where each level of recursion corresponds to making a choice for a specific position in the permutation.

- **Backtracking**: This pattern allows us to undo choices that don't lead to a solution and explore other options. The backtracking occurs when we mark a number as visited (`vis[j] = True`), recurse, and then unmark it after the recursive call (`vis[j] = False`). This ensures numbers are freed up for subsequent recursive calls at other tree nodes (positions in the permutation).

- **Boolean Visited Array (`vis`)**: An array of boolean values to keep track of which numbers from `1` to `n` have been used in the current partial arrangement. This is crucial because we cannot repeat a number in the permutation.

- **Pre-computation of Viable Candidates (`match`)**: A list of lists (or a dictionary of lists, if we use Python's `defaultdict`) is prepared before the DFS begins. Each index `i` contains a list of numbers that `i` can be divisible by or that can divide `i`. This pre-computation optimizes our search because we do not have to calculate the divisibility each time we want to put a number in position `i`.

- **Result Counter (`ans`)**: A counter to keep track of the total number of beautiful arrangements found so far. We use a nonlocal variable to allow the nested `dfs` function to modify the outer scope's `ans` variable.

The `dfs` function starts by checking if we have completed an arrangement (i.e., `i == n + 1`), in which case we increment `ans`. Otherwise, we iterate over all numbers that could fit in position `i` (array `match[i]`). If a number hasn't been used yet (`not vis[j]`), we mark it as used and call `dfs(i + 1)` to attempt to place a number at position `i + 1`.

Finally, `dfs(1)` initiates our recursive search starting at the first position in the arrangement, and `return ans` passes back the total count of beautiful arrangements after the search is complete.

By combining a clever DFS that only explores valid paths, with backtracking and a pre-computation of viable candidates, we can solve the problem efficiently.

Example Walkthrough

Let's consider a small example where `n = 3` to illustrate the solution approach. Our task is to count the number of beautiful arrangements possible for the numbers 1, 2, and 3.

First, we precompute the list `match` containing all the candidates for each position that can divide the position or be divided by it:

- For position 1: 1 (since every number is divisible by 1)
- For position 2: 1, 2 (since 2 is divisible by 1 and 2, and 1 is divisible by 1)
- For position 3: 1, 3 (since 3 is divisible by 1 and 3, and 1 is divisible by 1)

This gives us `match[1] = [1]`, `match[2] = [1, 2]`, and `match[3] = [1, 3]`.

Now, we start our DFS with the first position (i=1). Since the only candidate for the first position is 1 (based on `match[1]`), we place 1 in the first position and mark it as visited.

Next, we move on to the second position (i=2). The candidates for the second position are 1 and 2, but since 1 is already used, we can only place 2 in the second position. We mark 2 as visited and proceed.

Now, we are at the third position (i=3). The candidates are 1 and 3; 1 is used, so we place 3 in the third position, marking it as visited.

We've now reached `i = n + 1` (i = 4 in this case), which means we've found a complete arrangement that satisfies the beautiful arrangement condition. The permutation `[1, 2, 3]` is beautiful as:

- 1 is divisible by its position 1
- 2 is divisible by its position 2
- 3 is divisible by its position 3

We increment the result counter `ans`. We then backtrack to explore other possibilities, but here, given `n = 3`, there's no need to backtrack since all the numbers are used.

During the actual process, we would continue this backtracking process, checking all permutations that satisfy the conditions, and each time we complete an arrangement, we would increment `ans`.

Finally, starting `dfs(1)` would explore these permutations, and at the end, `return ans` would hold the total count of beautiful arrangements. In this case, there is only one beautiful arrangement for `n = 3`: `[1, 2, 3]`, so `ans = 1`.

Python Solution

```python
from collections import defaultdict

class Solution:
    def countArrangement(self, n: int) -> int:
        # Helper function for the depth-first search algorithm.
        def dfs(position):
            nonlocal count, n
            # If the position is out of range, a valid arrangement is found.
            if position == n + 1:
                count += 1
                return
            # Try possible numbers for the current position.
            for number in matches[position]:
                # Check if the number is not visited.
                if not visited[number]:
                    # Mark the number as visited.
                    visited[number] = True
                    # Recurse for the next position.
                    dfs(position + 1)
                    # Backtrack, unmark the number as visited for future arrangements.
                    visited[number] = False

        # Initial count of beautiful arrangements.
        count = 0
        # List to check if a number is already used in the arrangement.
        visited = [False] * (n + 1)
        # Dictionary to store all matching numbers for each position.
        matches = defaultdict(list)
        # Populate the matches dictionary with all possible numbers for each position.
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                # Numbers are a match if they are divisible by each other.
                if j % i == 0 or i % j == 0:
                    matches[i].append(j)

        # Start the depth-first search from position 1.
        dfs(1)
        # Return the total count of beautiful arrangements.
        return count

# The Solution can be instantiated and the method can be called as follows:
# solution = Solution()
# beautiful_arrangements_count = solution.countArrangement(n)

```

Java Solution

```java
class Solution {
    public int countArrangement(int n) {
        // Calculate the number of possible states (2^n)
        int maxState = 1 << n;
        // Initialize the array to store the count of valid permutations for each state
        int[] dp = new int[maxState];
        // Base case: there's one way to arrange an empty set
        dp[0] = 1;

        // Iterate through all states from the empty set to the full set
        for (int i = 0; i < maxState; ++i) {
            // Count the number of elements in the current set (state)
            int count = 1;
            for (int j = 0; j < n; ++j) {
                count += (i >> j) & 1;
            }

            // Try to add each element from 1 to N into the current set (state)
            for (int j = 1; j <= n; ++j) {
                // We can only add element 'j' if it's not already present in the set (state)
                // and the position 'count' is divisible by 'j' or vice versa
                if (((i >> (j - 1) & 1) == 0) && (count % j == 0 || j % count == 0)) {
                    // Update the dp value for the state that results from adding 'j' to the current set (state)
                    dp[i | (1 << (j - 1))] += dp[i];
                }
            }
        }

        // The answer is the number of valid permutations for the full set (state)
        return dp[maxState - 1];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int N; // Number of positions (and also numbers to arrange)
    int answer; // The count of valid arrangements
    vector<bool> visited; // Used to check if a number has been used in the arrangement
    unordered_map<int, vector<int>> validMatches; // Map containing numbers which can be placed at index i + 1

    // Method to start counting the number of valid beautiful arrangements
    int countArrangement(int N) {
        this->N = N;
        answer = 0;
        visited.resize(N + 1, false);

        // Precompute the valid matches for all positions 1 to N.
        for (int i = 1; i <= N; ++i) {
            for (int j = 1; j <= N; ++j) {
                if (i % j == 0 || j % i == 0) {
                    validMatches[i].push_back(j);
                }
            }
        }

        // Begin the depth-first search from the first position.
        depthFirstSearch(1);
        return answer;
    }

    // Recursive method used to perform depth-first search for beautiful arrangements
    void depthFirstSearch(int index) {
        // If the index is out of bounds, we've found a valid arrangement.
        if (index == N + 1) {
            ++answer;
            return;
        }

        // Loop through all numbers which are valid to place at the current position
        for (int num : validMatches[index]) {
            // Make sure the number hasn't been used yet.
            if (!visited[num]) {
                visited[num] = true; // Mark the number as used
                depthFirstSearch(index + 1); // Continue search for the next position.
                visited[num] = false; // Backtrack: unmark the number to make it available again
            }
        }
    }
};

```

Typescript Solution

```typescript
// Function to count the number of beautiful arrangements that can be formed using the numbers 1 to n
function countArrangement(n: number): number {
    // 'visited' array to keep track of which numbers have been used in the arrangement
    const visited = new Array(n + 1).fill(false);

    // 'matches' array where each index i contains an array of numbers that are compatible with i
    // in terms of the beautiful arrangement rules (either is a multiple of i or i is a multiple of it)
    const matches = Array.from({ length: n + 1 }, () => new Array<number>());

    // Fill the 'matches' array with appropriate values by checking the compatibility condition
    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= n; j++) {
            if (i % j === 0 || j % i === 0) {
                matches[i].push(j);
            }
        }
    }

    // Variable to store the total number of beautiful arrangements found
    let count = 0;

    // Depth-first search function to explore possible arrangements
    const dfs = (position: number) => {
        // If position is beyond the last index, a valid arrangement has been found
        if (position === n + 1) {
            count++;
            return;
        }
        // Iterate through each number that matches the current position in the arrangement
        for (const number of matches[position]) {
            // If the number has not yet been used in this arrangement
            if (!visited[number]) {
                // Mark the number as used
                visited[number] = true;
                // Continue building the arrangement from the next position
                dfs(position + 1);
                // Backtrack: unmark the number as used to explore other arrangements
                visited[number] = false;
            }
        }
    };

    // Start the DFS process from position 1
    dfs(1);

    // Return the total count of beautiful arrangements found
    return count;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed as follows:

- The code uses a depth-first search (DFS) strategy to generate all possible permutations of numbers from `1 to n`.
- In the worst case, each call to `dfs` function generates `n-i` subsequent calls (where `i` is the current depth of the search), because it iterates through the `match[i]` list which can have up to `n-i` valid candidates for the `i-th` position.
- However, for each `i`, the size of `match[i]` is not the same, and each match requires both `i` and `j` to be divisible by one another, so it's not always `n-i`.
- Since the permutations are unique, every subsequent call to the `dfs` function will have one less option to choose from.
- Therefore, the time complexity has an upper bound of `O(n!)`, as the `dfs` will have to explore all possible permutations in the worst case, but due to the divisibility condition, it is likely to be significantly less in practice. Hence, the strict mathematical representation is elusive but significantly better than `O(n!)` in many cases.

Space Complexity

Regarding the space complexity:

- `vis` array is of size `n + 1`, which occupies `O(n)` space.
- `ans` is a single integer, so it occupies `O(1)` space.
- `match` is a dictionary where each key has a list, and in the worst case, every list could have up to `n` items, making space taken by `match` up to `O(n^2)`.
- The depth of the recursive `dfs` call could go up to `n` in the worst case, which means a recursion stack space of `O(n)`.
- Combining all the space complexities together, we get `O(n^2)` for the `match` dictionary being the dominant term.

Therefore, the overall space complexity of the code is `O(n^2)`.