# 2151. Maximum Good People Based on Statements

Hard | Bit Manipulation | Array | Backtracking | Enumeration | Leetcode Link

## Problem Description

In this LeetCode problem, we're given a set of statements made by a group of `n` people, about each other's trustworthiness. Specifically, the input is a 2D integer array `statements` where:

- `statements[i][j]` = 0 means person `i` claims person `j` is bad (untruthful).
- `statements[i][j]` = 1 means person `i` claims person `j` is good (truthful).
- `statements[i][j]` = 2 means person `i` has not made any claim about person `j`.

Individuals are not permitted to make statements about themselves, thus `statements[i][i]` is always 2. The objective is to determine the maximum number of people that can be considered 'good' based on these statements. A 'good' person always tells the truth, while a 'bad' person may lie or tell the truth.

## Intuition

The intuition behind the solution is to utilize a brute-force approach, trying every possible combination of 'good' and 'bad' assignments for the people. Since a 'good' person always tells the truth, their statements can be used to cross-verify which people can be good and which cannot. If a 'good' person says another person is good, then both must be good; if they say someone is bad, then that person cannot be considered good in any combination that includes the first person as a good individual.

The solution uses bit masking to represent each combination of 'good' and 'bad' people. Each bit in a mask represents a person, where `1` can denote 'good' and `0` can denote 'bad'. There are `2^n` possible combinations (masks) to check, where `n` is the number of people.

The function `check(mask)` takes a mask as an argument and iterates over all people. It checks the statements of those who are 'good' (bits set to `1`) in the current mask. For each 'good' person's statements, it checks whether they align with the current assumption of who's good and bad. For instance, if a 'good' person claims another person is good, but the mask marks them as bad (or vice versa), the current mask is invalid, and the function returns 0. Otherwise, if no conflicts are found, it returns the number of 'good' people in the current mask (the count of `1`'s in the mask).

The main function calls `check(mask)` for all `2^n` combinations and returns the maximum number of good people found among all valid masks that don't lead to contradictions.

This brute-force solution works well for smaller `n`, as it explores all possibilities and ensures that the maximum number of 'good' people is found according to the statements made.

## Solution Approach

The implementation is grounded on a bit manipulation strategy. It uses an integer to represent a set of truth assignments to individuals, where the `i-th` bit of the integer corresponds to the `i-th` person.

Here is a step-by-step walkthrough of the solution:

1. **Brute Force Enumeration**: We generate all possible combinations of each person being 'good' or 'bad' by iterating through integers from `1` to `2^n - 1`, inclusive. Here, `n` is the number of people. The reason we start from `1` is because `0` would mean that nobody is good, which is not a valid scenario for us to consider.

2. **Checking Each Combination**: For each combination (represented by `mask`), we implement a function called `check(mask)`. This function is responsible for validating the combination against the statements made.

3. **Validating the 'Good' People**: Within `check(mask)`, we traverse over each person (`i`) by examining the current bit in the mask: `(mask >> i) & 1`. If this results in `1`, the person is assumed 'good' in this particular combination. Then, we check each statement `i` has made about other people (`j`).

4. **Statement Verification**: If person `i` has made a definitive statement about person `j` (i.e., `statements[i][j]` is 0 or 1), we check the truthfulness of `i`'s statement against the current combination (`mask`). If a 'good' `i` made a statement that contradicts the current assumption about `j`, the function returns `0`, indicating this mask (or combination) is not valid.

5. **Counting 'Good' People**: If no contradictions are found for a 'good' person `i`, the count increments, tallying the number of 'good' people in this combination.

6. **Finding the Maximum**: Finally, the main solution function calls `check(mask)` for each possibility and keeps track of the maximum number of 'good' people found from all the valid combinations. It uses the built-in Python function `max()` to find the maximum over all return values from `check(mask)`.

By utilizing bit masks to represent sets and checking all possible combinations, the algorithm finds the optimal solution that matches the problem's constraints. This approach, while not the most time-efficient for large `n`, ensures accuracy by not overlooking any potential solution when `n` is within a manageable size.

### Example Walkthrough

Suppose we have a group of 3 people, and we are given the following `statements` array:

```
statements = [
  [2, 1, 0],  // Person 0's statements about Person 1 and 2
  [2, 2, 1],  // Person 1's statements about Person 0 and 2
  [1, 0, 2]   // Person 2's statements about Person 0 and 1
]
```

Here's how we would apply the solution approach:

1. **Brute Force Enumeration**: Since there are 3 people, we have `2^3 = 8` combinations to consider. We will ignore the mask `000` (where nobody is good) and check `001` to `111`.

2. **Checking Each Combination**: Let's consider the mask `011`, indicating Person 0 is 'bad' while Persons 1 and 2 are 'good'. We use the `check(mask)` function to determine if this mask is a valid representation of the 'good' people.

3. **Validating the 'Good' People**: In the `check(mask)` function, we start with Person 0 being 'bad', so we skip and move to Persons 1 and 2.

4. **Statement Verification**: For Person 1 (who is 'good' in this mask), their statement about Person 2 is that they are 'good' (`statements[1][2]` is 1), which aligns with our mask. However, we do not have a statement from Person 1 about Person 0, so we cannot verify Person 1's truthfulness this way.

   Moving to Person 2, they state that Person 0 is 'good' (`statements[2][0]` is 1), but our mask assumes Person 0 is 'bad', creating a contradiction. Person 2 also claims that Person 1 is 'bad' (`statements[2][1]` is 0), which contradicts our mask where Person 1 is 'good'.

5. **Counting 'Good' People**: Since there's a contradiction in Person 2's statements, we cannot count any 'good' people under this mask, so `check(mask)` returns 0.

6. **Finding the Maximum**: We perform the same `check(mask)` for all other combinations. Let's take another example with the mask `101`, where Persons 0 and 2 are 'good', and Person 1 is 'bad'. Person 0 claims Person 1 is 'good' which is a contradiction since we assumed them to be 'bad'. Therefore, `check(mask)` also returns 0 for this mask.

By going through all combinations, we may find that the mask `110` yields the maximum number of 'good' people without contradiction — Person 0 claims Person 2 is 'bad', and Person 2 claims Person 0 is 'good', which both align with this mask. In this scenario, both Persons 0 and 1 can be considered good based on their statements and each other's statements.

Therefore, the final result given by the main function would be `2`, as this is the maximum number of 'good' people found across all valid combinations. The function would iterate through all the masks, evaluate each one using the `check(mask)` function, and conclude that the combination `110` presents the maximum number of good people without causing any conflicts.

## Python Solution

```python
class Solution:
    def maximumGood(self, statements):
        # Helper function to check and count the number of 'good' people based on the given mask
        def check(mask):
            count_good = 0  # Initialize count of 'good' people
            # Iterate through each person's statements
            for idx, statement in enumerate(statements):
                # Check if the bit in the mask for this person is set to 1 (indicating 'good')
                if (mask >> idx) & 1:
                    # Iterate through this person's statements about others
                    for jdx, value in enumerate(statement):
                        # If the statement is not 'unknown' and conflicts with the mask, return 0
                        if value < 2 and ((mask >> jdx) & 1) != value:
                            return 0
                    # If there is no conflict, increment the count of 'good' people
                    count_good += 1
            return count_good  # Return the count of 'good' people for this mask

        # Iterate over all possible combinations of 'good' and 'bad' people and find the max count of 'good' people
        # 1 << len(statements) gives the total number of combinations for 'good' / 'bad' assignments
        max_good_people = max(check(mask) for mask in range(1, 1 << len(statements)))
        return max_good_people

# The List type need to be imported from typing if you want to use it in type hints
from typing import List

# Then you could use it as follows:
class Solution:
    def maximumGood(self, statements: List[List[int]]) -> int:
        # ... rest of the code remains the same

```

## Java Solution

```java
class Solution {
    // Method to find the maximum number of people that can be classified as good
    public int maximumGood(int[][] statements) {
        int maxGoodPersons = 0; // Initialize the count of maximum good persons to zero.

        // Iterate through all possible subsets of persons using a mask
        for (int mask = 1; mask < (1 << statements.length); ++mask) {
            // Check the current subset and update the maximum if applicable
            maxGoodPersons = Math.max(maxGoodPersons, checkSubset(mask, statements));
        }

        return maxGoodPersons; // Return the maximum number of good persons found
    }

    // Helper method to check if a given subset of persons is valid and calculate the count
    private int checkSubset(int mask, int[][] statements) {
        int goodPersonCount = 0; // Initialize the count of good persons in this subset to zero.
        int personCount = statements.length; // The total number of persons

        // Iterate over each person to check if they can be classified as good, based on the current subset
        for (int i = 0; i < personCount; ++i) {
            // Check if the current person (i) is included in the subset (good) as per the mask
            if (((mask >> i) & 1) == 1) {

                // Iterate over all the statements made by this person
                for (int j = 0; j < personCount; ++j) {
                    int statementValue = statements[i][j]; // Statement about person j made by person i

                    // If the statement value is less than 2 (0 or 1), check consistency with subset
                    // If there is an inconsistency, the subset is not valid and we return 0
                    if (statementValue < 2 && ((mask >> j) & 1) != statementValue) {
                        return 0;
                    }
                }

                // If all statements by person i are consistent, increment the count of good persons
                ++goodPersonCount;
            }
        }

        // Return the total number of good persons in this subset if it is valid
        return goodPersonCount;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // Finds the maximum number of people who can be good based on their statements
    int maximumGood(vector<vector<int>>& statements) {
        int maxGood = 0; // Stores the maximum number of 'good' people found

        // Iterate over all possible combinations of 'good' and 'bad' people
        for (int mask = 1; mask < (1 << statements.size()); ++mask) {
            // Compare current count with the new count returned by check function
            maxGood = max(maxGood, check(mask, statements));
        }

        return maxGood; // Return the maximum count of 'good' people possible
    }

    // Checks if a given combination (mask) of 'good' people is consistent with their statements
    int check(int mask, vector<vector<int>>& statements) {
        int countGood = 0; // Counter for the number of 'good' people in the current combination
        int n = statements.size(); // Number of people

        // Iterate over each person
        for (int i = 0; i < n; ++i) {
            if ((mask >> i) & 1) { // Check if the person is 'good' in the current mask
                // If the person is 'good', check all of their statements
                for (int j = 0; j < n; ++j) {
                    int statementValue = statements[i][j]; // Statement about person j by person i

                    // If statementValue < 2, it means person i has made a definitive statement.
                    // Check if the statement is consistent with our current 'good'/'bad' combination
                    if (statementValue < 2 && ((mask >> j) & 1) != statementValue) {
                        return 0; // Inconsistency found; this is not a valid combination
                    }
                }
                countGood++; // Increment the count of 'good' people for this combination
            }
        }

        return countGood; // Return the number of 'good' people for this combination if consistent
    }
};

```

## Typescript Solution

```typescript
// Helper function to count the number of 'good' persons that can be inferred based on the current mask.
// The mask represents a set of people we assume to be 'good' (where each bit corresponds to a person).
function checkValidity(mask: number, statements: number[][], peopleCount: number): number {
    let goodCount = 0; // Track number of 'good' people.
    for (let i = 0; i < peopleCount; ++i) {
        // If the i-th bit of mask is set, consider the i-th person 'good'.
        if ((mask >> i) & 1) {
            for (let j = 0; j < peopleCount; ++j) {
                const statementValue = statements[i][j];
                // If a 'good' person makes a definite statement (true or false) about another person,
                // check if that statement conflicts with our current assumptions. If it does, return 0 as
                // this configuration of 'good' people is not possible.
                if (statementValue < 2 && ((mask >> j) & 1) != statementValue) {
                    return 0;
                }
            }
            // If we reach here, person 'i' is 'good' without contradictions.
            goodCount++;
        }
    }
    // Return the total count of 'good' people in this configuration.
    return goodCount;
}

// Main function to find the maximum number of 'good' people based on their statements.
function maximumGood(statements: number[][]): number {
    const peopleCount = statements.length; // Total number of people.
    let maxGood = 0; // Keep track of the maximum number of 'good' people found so far.

    // Iterate over all possible sets of 'good' people represented by masks.
    // (1 << peopleCount) gives us the number of possible masks, as each person can be good or not.
    for (let mask = 1; mask < (1 << peopleCount); ++mask) {
        // Update the maximum good count if the current mask leads to a higher number of 'good' people.
        maxGood = Math.max(maxGood, checkValidity(mask, statements, peopleCount));
    }
    // Return the maximum number of 'good' people that can be deduced without contradictions.
    return maxGood;
}

```

## Time and Space Complexity

The provided Python code defines a function `maximumGood` which finds out the maximum number of people who can be considered "good" based on their statements about themselves and others. The solution employs a brute force approach to check every possible combination of "good" and "bad" people and then validate these combinations against the given statements.

### Time Complexity

The time complexity of the code is determined by the number of possible combinations of people being good or bad, since we are iterating over all possible combinations using a bit mask. Given `n` people, there are `2^n` possible combinations (since for each person, there is a binary decision to be either good or bad). For each combination, we are checking all statements made by all `n` persons, and each person makes `n` statements or observations. This results in a nested loop running `n` times for each of the `2^n` combinations.

Thus, the time complexity is `O(n * 2^n)`, where `n` is the number of people.

### Space Complexity

The space complexity of the code is quite straightforward. The only additional space used is for the variable `cnt`, which is an integer used to keep count of the number of "good" people in the current combination, and the space used in the recursive process stack. This does not scale with `n` and hence is `O(1)`.

However, it's important to note that the code uses recursive calls to the `check` function. The maximum depth of this recursion would be `n` in the worst case (although in the `check` function there are no recursive calls, it's important to consider for more general cases). Therefore, if we would consider the function call stack in the analysis, the space complexity would be `O(n)` for the recursion call stack space. However, since in the provided code `check` is more of an iterative function, we consider the space complexity to be `O(1)`.