# 1742. Maximum Number of Balls in a Box

Easy Hash Table Math Counting

## Problem Description

In this problem, we are situated in a hypothetical ball factory where balls are numbered sequentially starting from `lowLimit` up to `highLimit`. We need to organize these balls into boxes based on a particular rule: the number of the box in which a ball is placed equals the sum of the digits of that ball's number. For example, if we have a ball numbered `321`, it goes into box `3 + 2 + 1 = 6`, while a ball numbered `10` will be put into box `1 + 0 = 1`.

The task is to determine the count of balls in the box that ends up containing the most balls after all the balls numbered from `lowLimit` to `highLimit` have been placed according to the aforementioned rule.

## Intuition

To find the solution to this problem, we employ a straightforward counting approach. We understand from the problem that the maximum ball number can't exceed `10^5`, consequently the sum of the digits of any ball can't be more than `9+9+9+9+9=45` (since `99999` is the largest number less than `10^5` with the most significant digit sum). This observation allows us to safely assume that we do not need to consider box numbers (digit sums) greater than `50`.

With this in mind, we can create an array, `cnt`, to keep track of the number of balls in each box, indexed by the sum of the digits of the ball numbers. This array has a size of `50`, as no digit sum will exceed this value. We then iterate over each ball from `lowLimit` to `highLimit`, calculate the sum of its digits, and increase the respective count in the `cnt` array by one for that digit sum.

Finally, the maximum count in the `cnt` array will tell us the number of balls in the box with the most balls. This constitutes our answer to the problem.

## Solution Approach

The solution approach is direct and leverages the concept of simulation and counting. The primary steps in the solution involved using a simulation algorithm to model the placement of balls into boxes and employing an elementary data structure (an array) to keep track of this simulation.

Here's a breakdown of how it works:

1. **Create a Counting Array**: We initialize an array named `cnt` of size 50, as determined by the largest possible sum of the digits of the ball numbers. This array will serve as a map from the sum of the digits (which we can consider as the "box number") to the count of balls that go into the respective box.
2. **Iterate Ball Numbers**: We iterate over the range from `lowLimit` to `highLimit`, inclusive, representing each ball's number.
3. **Sum of Digits Calculation**: For each ball number, we calculate the sum of its digits. We do this by repeatedly retrieving the last digit using the modulo operator (`x % 10`) and then trimming the last digit off by integer division (`x //= 10`). The sum of these digits represents the box number where the ball will be placed.
4. **Count Balls in Boxes**: Corresponding to the sum of digits, we increment the count in the `cnt` array by one (`cnt[y] += 1`). This effectively places the ball into the correct box and keeps count of the number of balls in each box.
5. **Identify the Box with the Most Balls**: To finalize the solution, we identify the box with the maximum count of balls by taking the maximum value in the `cnt` array (`max(cnt)`). This provides us with the required count of balls in the box that has the highest number of balls.

The choice of data structures and algorithms in this solution is based on both the constraints of the problem (such as the range of ball numbers) and the need for a simple yet efficient method to keep a tally of the balls being placed into boxes. The array serves as an easily accessible structure to increment counts, and its fixed size (derived from the constraints) ensures that the solution is space-efficient. The simulation element in the algorithm models the real-world action of placing balls into boxes, giving a methodical approach to achieving the final goal of the problem.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following limits for the ball numbers: `lowLimit = 5` and `highLimit = 15`. This means we need to organize ball numbers 5 through 15 into boxes based on the sum of digits rule.

1. **Create a Counting Array**: Initiate an array `cnt[50]` to zero to count the balls in each box.
2. **Iterate Ball Numbers**: Start iterating from ball number `5` to `15`.
3. **Sum of Digits Calculation**: Calculate sum of digits for each ball number.
   - Ball 5: Sum of digits is `5`; it goes to box `5`.
   - Ball 6: Sum of digits is `6`; it goes to box `6`.
   - Ball 7: Sum of digits is `7`; it goes to box `7`.
   - Ball 8: Sum of digits is `8`; it goes to box `8`.
   - Ball 9: Sum of digits is `9`; it goes to box `9`.
   - Ball 10: Sum of digits is `1 + 0 = 1`; it goes to box `1`.
   - Ball 11: Sum of digits is `1 + 1 = 2`; it goes to box `2`.
   - Ball 12: Sum of digits is `1 + 2 = 3`; it goes to box `3`.
   - Ball 13: Sum of digits is `1 + 3 = 4`; it goes to box `4`.
   - Ball 14: Sum of digits is `1 + 4 = 5`; it goes to box `5`.
   - Ball 15: Sum of digits is `1 + 5 = 6`; it goes to box `6`.
4. **Count Balls in Boxes**: Increment the count in the `cnt` array for each sum of digits.
   - For ball 5, `cnt[5] += 1`.
   - For ball 6, `cnt[6] += 1`.
   - For ball 7, `cnt[7] += 1`.
   - For ball 8, `cnt[8] += 1`.
   - For ball 9, `cnt[9] += 1`.
   - For ball 10, `cnt[1] += 1`.
   - For ball 11, `cnt[2] += 1`.
   - For ball 12, `cnt[3] += 1`.
   - For ball 13, `cnt[4] += 1`.
   - For ball 14, `cnt[5] += 1` (since `cnt[5]` already has 1, now it is 2).
   - For ball 15, `cnt[6] += 1` (since `cnt[6]` already has 1, now it is 2).
5. **Identify the Box with the Most Balls**: Look for the maximum value in the `cnt` array.
   - After counting, we find that `cnt[5]` and `cnt[6]` each have a count of `2`, which are the highest counts.

So in this example, the boxes that end up with the most balls are boxes `5` and `6`, both containing `2` balls each. This would be the answer returned by the algorithm for our `lowLimit` and `highLimit` range of `5` to `15`.

## Solution Implementation

### Python

```python
class Solution:
    def countBalls(self, low_limit: int, high_limit: int) -> int:
        # Initialize a list to keep track of the count of balls in each box.
        # Assuming the maximum possible sum of the digits is less than 50.
        count = [0] * 50

        # Iterate through each number from low_limit to high_limit, inclusive.
        for num in range(low_limit, high_limit + 1):
            # Calculate the sum of the digits of the current number.
            sum_of_digits = 0
            temp_num = num
            while temp_num:
                sum_of_digits += temp_num % 10
                temp_num //= 10

            # Increment the count corresponding to the box with the index of the sum.
            count[sum_of_digits] += 1

        # Return the count of the most populated box.
        return max(count)
```

### Java

```java
import java.util.Arrays; // Necessary import for using Arrays.stream()

public class Solution {

    // This method counts the maximum number of balls with the same sum of digits
    // between the given lowLimit and highLimit (inclusive).
    public int countBalls(int lowLimit, int highLimit) {
        // Assume the highest possible sum of digits in any ball number is less than 50
        int[] count = new int[50];

        // Iterate through each ball number from lowLimit to highLimit
        for (int i = lowLimit; i <= highLimit; ++i) {
            int sumOfDigits = 0; // Initialize sum of digits for the current ball number

            // Calculate the sum of digits for the current ball number
            for (int number = i; number > 0; number /= 10) {
                sumOfDigits += number % 10; // Add the last digit to the sum
            }

            // Increment the count for the computed sum of digits
            ++count[sumOfDigits];
        }

        // Find the maximum count in the array and return it
        // Uses Java 8 Streams to find the max value in the count array
        return Arrays.stream(count).max().getAsInt();
    }
}
```

### C++

```cpp
#include <algorithm> // Include algorithm header for the max function

class Solution {
public:
    // Function to count the maximum number of balls in a box
    int countBalls(int lowLimit, int highLimit) {
        // Initialize the array to hold counts for each possible box number (max sum of digits from 1 to 99999 is 45)
        int boxCounts[46] = {0}; // Increased to 46 to fit all possible sums of digits
        int maxBalls = 0; // Variable to store the maximum number of balls in a box

        // Iterate over the range from lowLimit to highLimit, inclusive
        for (int i = lowLimit; i <= highLimit; ++i) {
            int boxNumber = 0; // Variable to store the sum of the digits (box number)
            // Calculate boxNumber by summing up the digits of `i`
            for (int x = i; x != 0; x /= 10) {
                boxNumber += x % 10;
            }
            // Increment the ball count for this particular box
            boxCounts[boxNumber]++;
            // Update the maxBalls if the count for current box exceeds the previous maximum
            maxBalls = std::max(maxBalls, boxCounts[boxNumber]);
        }

        return maxBalls; // Return the maximum number of balls in any of the boxes
    }
};
```

### TypeScript

```typescript
// Function to count the maximum number of balls in a box after distributing them according to their number's sum of digits
function countBalls(lowLimit: number, highLimit: number): number {
    // Array to store the count of balls for each possible sum of digits (0-45, so 50 is a safe upper limit)
    const ballCountArray: number[] = Array(50).fill(0);

    // Loop over each number from lowLimit to highLimit
    for (let number = lowLimit; number <= highLimit; ++number) {
        let digitSum = 0; // Variable to hold the sum of digits of the current number

        // Iterate over the digits of the number
        for (let n = number; n > 0; n = Math.floor(n / 10)) {
            digitSum += n % 10; // Add the rightmost digit to the sum
        }

        // Increment the count for the box corresponding to this sum of digits
        ballCountArray[digitSum]++;
    }

    // Return the maximum count of balls from the array
    return Math.max(...ballCountArray);
}
```

```python
class Solution:
    def countBalls(self, low_limit: int, high_limit: int) -> int:
        # Initialize a list to keep track of the count of balls in each box.
        # Assuming the maximum possible sum of the digits is less than 50.
        count = [0] * 50

        # Iterate through each number from low_limit to high_limit, inclusive.
        for num in range(low_limit, high_limit + 1):
            # Calculate the sum of the digits of the current number.
            sum_of_digits = 0
            temp_num = num
            while temp_num:
                sum_of_digits += temp_num % 10
                temp_num //= 10

            # Increment the count corresponding to the box with the index of the sum.
            count[sum_of_digits] += 1

        # Return the count of the most populated box.
        return max(count)
```

## Time and Space Complexity

### Time Complexity

The given code iterates through all the numbers from `lowLimit` to `highLimit` inclusive. The time complexity is determined by two factors: the number of iterations and the complexity of the operations within each iteration.

For each number (`x`) in the range, the inner `while` loop breaks down the number into its digits to calculate the sum of digits (`y`). In the worst case, the number of digits for a number `x` is proportional to the logarithm of `x` to the base 10, or $O(\log 10(x))$. Since the largest possible value of `x` in our range is `highLimit`, we can denote this as $O(\log 10(m))$, where $m = highLimit$.

The number of iterations is $n$, where $n = highLimit - lowLimit + 1$. Therefore, the total time complexity is $O(n * \log 10(m))$ since each iteration's computational overhead is $O(\log 10(m))$.

### Space Complexity

The space complexity is determined by the additional space used by the algorithm besides the input and output. In this case, the only significant additional storage is the `cnt` array, whose size is constant at 50. Therefore, the space complexity is $O(1)$, which means it does not scale with the input size $n$ or the value of `highLimit`.