# 66. Plus One

Easy Array Math

## Problem Description

You are provided with an array of integers named `digits`, where each element of the array represents a digit from a larger integer number. The first element in the array corresponds to the most significant digit, and the last element is the least significant digit. This array is a non-negative integer and does not contain any leading zeros.

The task is to simulate the increment of this large number by one (as if you were adding 1 to a normal integer) and return the array format of the new integer after this increment. If the number was `123`, represented by the array `[1,2,3]`, adding one should make it `124`, and hence the output should be `[1,2,4]`. This is relatively straightforward for numbers that don't result in a carry-over after addition, but special attention is needed for numbers like `999`, where adding `1` results in `1000`, requiring a change in the number of digits.

## Intuition

To increment the integer, we should add `1` to the last element (the least significant digit) and then deal with any carry that might result from this operation. We start from the end of the array and work our way backward to the beginning because that's how addition would naturally carry over in multi-digit numbers. For each digit, we add `1` and then take the modulus with `10` to find its new value after incrementing. If the new value isn't `0`, there's no carry, and we can return our result immediately.

For example, if our number is `129`, we start by adding `1` to `9`, getting `10`, take `10 % 10` and get `0`, which we place as the new least significant digit. Since the result is `0`, there's a carry over to the next digit. We repeat the process with `2`, getting `3` after addition, and since `3 % 10` is `3`, we now have `[1, 3, 0]` and there's no further carry over, so we can return this result.

However, if we encounter a number like `998` and add `1`, after modulus operation till the first element, all elements would turn to `0`. When this happens, and we reach the beginning of the array, it means there is a carry out from the most significant digit. In this case, we need to represent an extra digit for the carry. We achieve this by placing a `1` at the beginning of the array. So the final result will be `[1, 0, 0, 0]`.

The main idea is that addition might require either modification of the existing digits or in some cases, it may require us to extend the array to accommodate an additional digit because of a carry that overflows from the most significant digit.

## Solution Approach

The solution implements a straightforward arithmetic approach that mirrors how we naturally do addition with carrying.

- The solution starts by getting the length of the given `digits` list, referred to as `n`.
- A for loop is then initiated, starting from the last element (`n - 1`) and moving backwards (`-1`) towards the first element (`0`). The loop decrements by 1 each time, iterating over the array from least significant digit to the most significant digit.
- Inside the loop, we increment the current digit by 1 (`digits[i] += 1`), and then immediately apply a modulo operation with 10 (`digits[i] %= 10`). The modulo operation is crucial: it effectively resets a `10` to `0` which is needed for the carry. If it does not become `0`, then we know no carry is needed and we can return the digits array as the result.
- If the current digit does indeed become `0` after the operation, it indicates a carry over to the next most significant digit, so we loop to the next iteration to process the carry.
- In the event that we have gone through the entire array and each digit has become `0` (like when we have all `9`s), there will still be a carry over remaining. This is caught after the loop ends, and we handle it by creating a new list with a `1` followed by all the `0`s from the original `digits` list.

From a data structures and algorithms perspective:

1. **Data Structures Used**: We use a list (array) to represent the large integer. This allows us to access and modify each digit in constant time.
2. **Algorithm Pattern**: The for loop used in the solution represents a simple linear iteration, which is a common pattern for traversing arrays or lists. This iteration pattern is combined with arithmetic logic to simulate digit-by-digit addition with carry.
3. **Time Complexity**: The time complexity of this algorithm is O(n) where `n` is the number of digits. This is because in the worst-case scenario, each digit will be visited once.
4. **Space Complexity**: The space complexity is O(1) if we don't count the space taken by the output (or O(n) if we do, because in the worst case, we create a new array with one more digit than the input). This is because the algorithm modifies the digits in-place and only uses a fixed amount of extra space.

The elegance of the solution lies in its direct imitation of manual digit-by-digit addition, and its sophisticated handling of carry-over while using minimal additional space.

### Example Walkthrough

To illustrate the solution approach, let's take a small example. Suppose we have the integer `459` represented by the array `digits = [4, 5, 9]`. Let's walk through the steps to increment this integer by one, step by step:

1. **Initialize**: Start by noting the length of the `digits` array, which is `n = 3` in this case.
2. **Loop through digits backwards**: We start with the last digit (least significant digit), so `i = n - 1 = 2`.
3. **Increment and mod operation**: We add `1` to the digit at index `2`, which is `9`. Thus, `digits[2]` becomes `9 + 1 = 10`. Apply the modulo operation, `digits[2] %= 10` results in `0`. Since the result is `0`, we know there is a carry to the next digit.
4. **Move to the next digit**: Decrement `i` to `1` and again apply the increment and modulo operation. Now `digits[1]` is `5`, and we add `1` resulting in `6`. Perform `digits[1] %= 10` which is `6 % 10 = 6`, and since the result is not `0`, there is no carry, so we can stop the process. The array now is `[4, 6, 0]`.
5. **Final output**: As we have a non-zero result at step 4, we can conclude the increment process and return the updated array. The incremented integer is `460`, and the array is `[4, 6, 0]`.

The solution is simple and closely follows the steps you perform when adding numbers manually. If we had a number that caused a carry out from the most significant digit (like `999`), after the loop, we would need to add an extra digit to the array, resulting in `[1, 0, 0, 0]`. But in our example, the result was straightforward, as only the least significant digit and the second significant digit were affected and no resizing of the array was necessary.

## Python Solution

```python
from typing import List

class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        # Determine the length of the input list
        num_digits = len(digits)

        # Iterate over the digits list from the end to the beginning
        for i in range(num_digits - 1, -1, -1):
            # Increment the current digit by 1
            digits[i] += 1
            # Perform modulo operation to handle the carry - if it's 10, it becomes 0
            digits[i] %= 10
            # If after the modulo the digit is not 0, no carry-over is needed
            # Return the list as is
            if digits[i] != 0:
                return digits

        # If all digits were '9', we exited the loop without returning
        # We need an additional '1' at the beginning of the list
        return [1] + digits

```

## Java Solution

```java
class Solution {
    public int[] plusOne(int[] digits) {
        // Get the number of digits in the array
        int length = digits.length;

        // Iterate over the digits starting from the least significant digit (LSD)
        for (int i = length - 1; i >= 0; --i) {
            // Increase the current digit by one
            digits[i]++;

            // If after the increment the digit is 10, it means it should be 0, and we carry over 1.
            // But if it's not 10 after the increment, we can return the result immediately.
            digits[i] %= 10;
            if (digits[i] != 0) {
                return digits;
            }
        }

        // If we're here, it means that we had a carry out from the most significant digit (MSD)
        // which requires us to expand the array. Example: 999 + 1 = 1000
        int[] result = new int[length + 1];
        result[0] = 1; // Set the first element to 1, the rest are default to 0

        // Return the result which has an additional digit
        return result;
    }
}
```

## C++ Solution

```cpp
#include <vector> // Include necessary header

class Solution {
public:
    // Function to add one to a number represented as a vector of digits
    std::vector<int> plusOne(std::vector<int>& digits) {
        // Work backwards through the digits of the number
        for (int i = digits.size() - 1; i >= 0; --i) {
            digits[i]++; // Add one to the current digit
            digits[i] %= 10; // If the current digit is 10, wrap around to 0

            // If the current digit is not zero, we can return the result
            // since no further carry would be needed
            if (digits[i] != 0) {
                return digits;
            }
        }

        // If we're here, the number was a series of 9's, such as 9999 which turned into 0000
        // and we need to add an extra digit at the beginning
        digits.insert(digits.begin(), 1);

        return digits; // Return the result
    }
};
```

## Typescript Solution

```typescript
function plusOne(digits: number[]): number[] {
    const length = digits.length; // Length of the input array of digits.

    // Loop through the digits array starting from the end.
    for (let index = length - 1; index >= 0; index--) {
        digits[index]++; // Increment the current digit.

        // If the incremented digit is less than 10, no carry is needed.
        if (digits[index] < 10) {
            return digits;
        }

        // If the digit is now 10 (after increment), set it to 0 as we carry over 1 to next digit.
        digits[index] = 0;
    }

    // If all digits were 9, the loop completes and we need an additional place value.
    // Prepend 1 to the digits array to account for the carry.
    return [1, ...digits];
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of this function is `O(n)`, where `n` is the number of digits in the input list. The reason for this is that there is a single loop which iterates over the list of digits in reverse, from last to first. In the worst case scenario, if we need to add 1 to the leftmost digit, we traverse the entire length of the list.

### Space Complexity

The space complexity of the function is `O(1)`. This result is because the operation modifies the input list in-place without using any additional space that scales with the size of the input. The only additional space used is for storing the length of the list and the loop index. However, a new list of length `n+1` is created and returned when there is a carryover that affects the most significant digit (i.e., when an additional digit is needed to represent the result, such as turning [9, 9, 9] into [1, 0, 0, 0]). Even though a new list is created in this scenario, it does not affect the asymptotic space complexity, which remains `O(1)` with respect to the input size.