# Question

Given a **non-empty** array of decimal digits representing a non-negative integer, increment one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contains a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

**Example 1:**

**Input:** digits = [1,2,3]

**Output:** [1,2,4]

**Explanation:** The array represents the integer 123.

**Example 2:**

**Input:** digits = [4,3,2,1]

**Output:** [4,3,2,2]

**Explanation:** The array represents the integer 4321.

**Example 3:**

**Input:** digits = [0]

**Output:** [1]

**Constraints:**

* 1 <= digits.length <= 100
* 0 <= digits[i] <= 9

# Solution

#### **Overview**

"Plus One" is a subset of the problem set "Add Number", which shares the same solution pattern.

All these problems could be solved in linear time, and the question here is how to solve it without using the addition operation or how to solve it in constant space complexity.

The choice of algorithm should be based on the format of input. Here we list a few examples:

1. Integers

Usually the addition operation is not allowed for such a case. Use Bit Manipulation Approach. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).

1. Strings

Use bit by bit computation. Note, sometimes it might not be feasible to come up a solution with the constant space for languages with immutable strings, e.g. for Java and Python. Here is an example: [Add Binary](https://leetcode.com/articles/add-binary/).

1. Linked Lists

Sentinel Head + Schoolbook Addition with Carry. Here is an example: [Plus One Linked List](https://leetcode.com/articles/plus-one-linked-list/).

1. Arrays (also the current problem)

Schoolbook addition with carry.

Note that, a straightforward idea to convert everything into integers and then apply the addition could be risky, especially for the implementation in Java, due to the potential integer overflow issue.

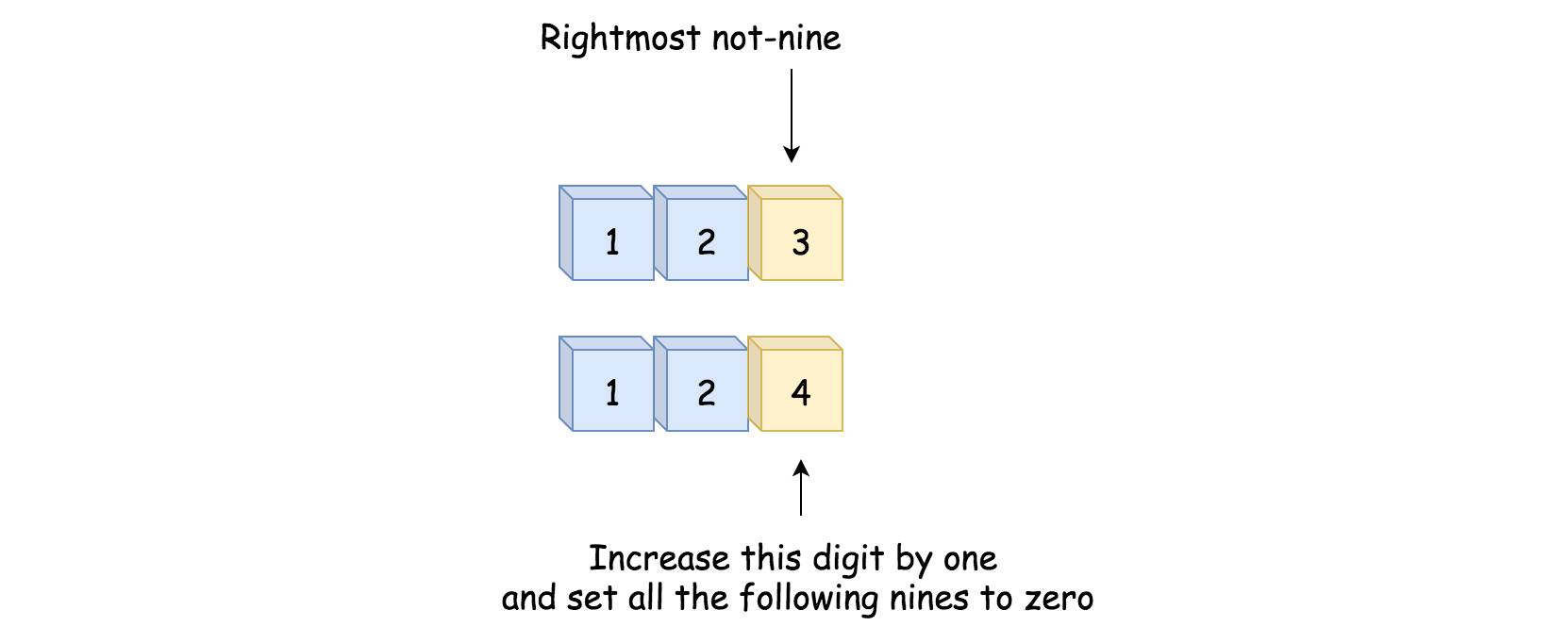
As one can imagine, once the array gets long, the result of conversion cannot fit into the data type of Integer, or Long, or even [BigInteger](https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html).

#### **Approach 1: Schoolbook Addition with Carry**

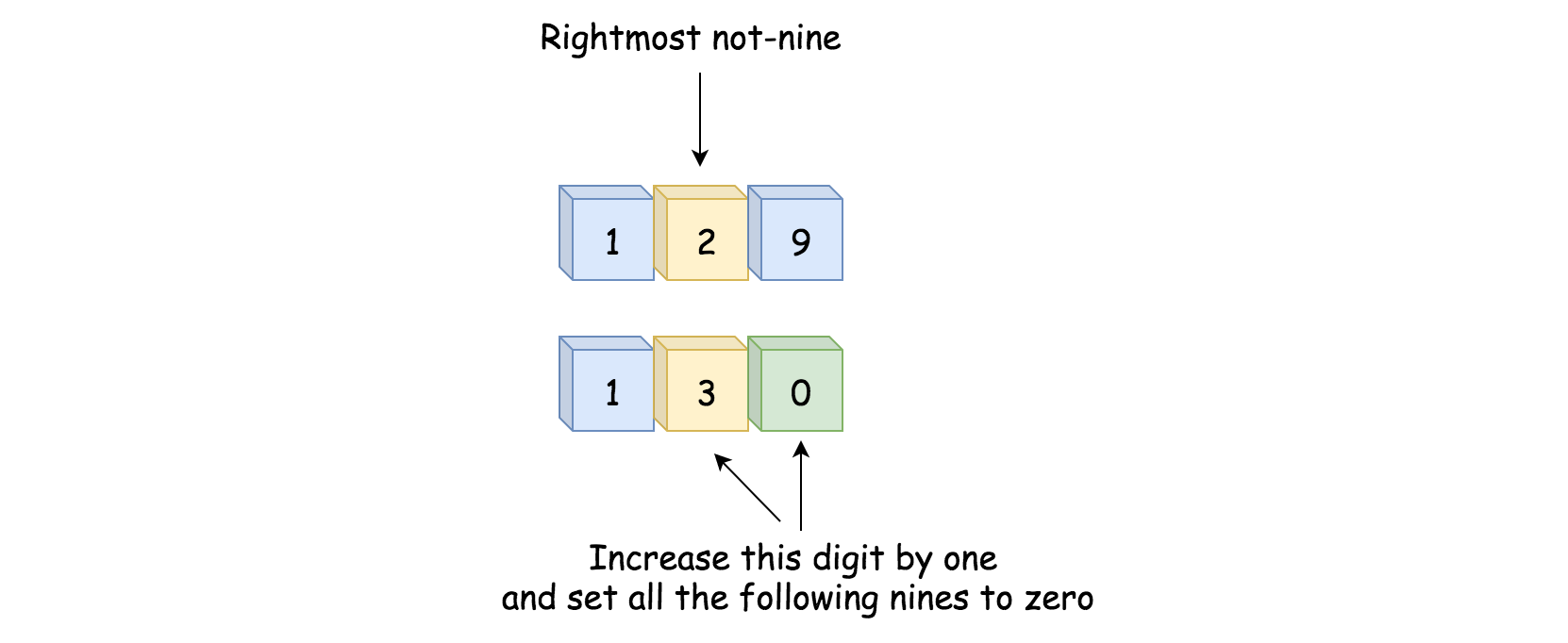
**Intuition**

Let us identify the rightmost digit which is not equal to nine and increase that digit by one. All the following consecutive digits of nine should be set to zero.

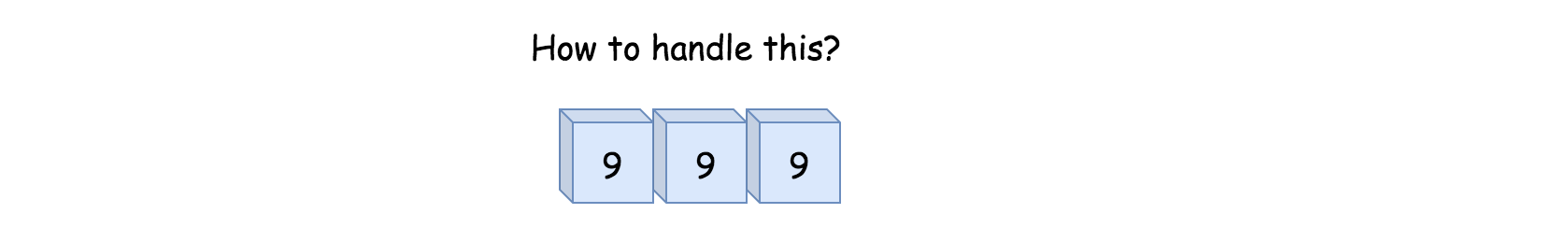
Here is the simplest use case which works fine.



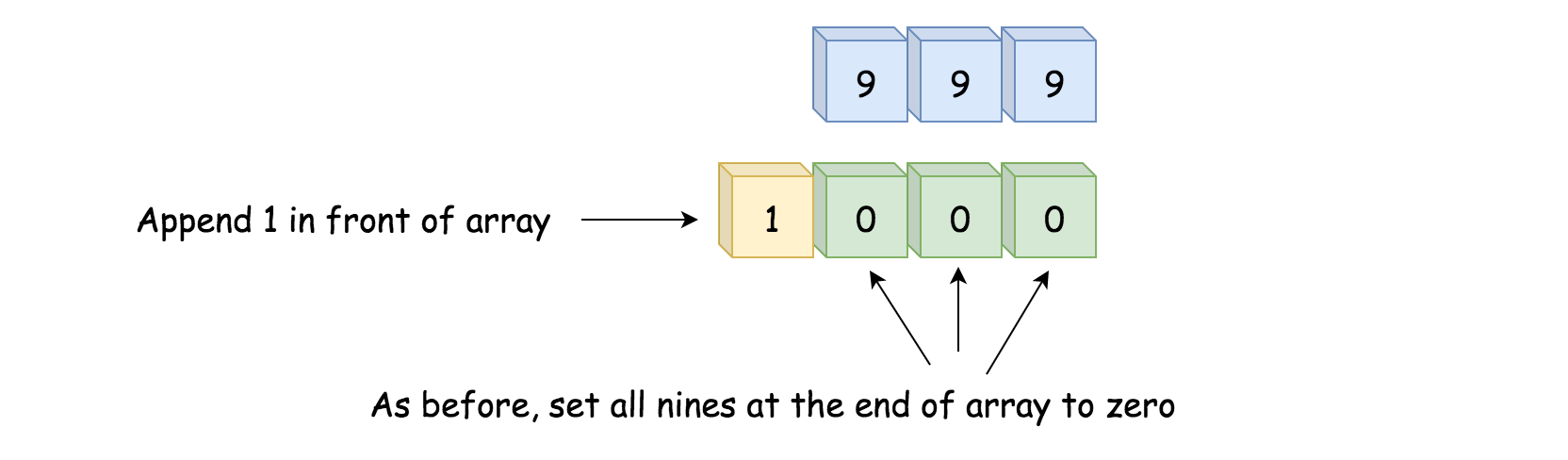
Here is a slightly complicated case which still passes.



And here is the case which breaks everything, because all the digits are nines.



In this case, we need to set all nines to zero and append 1 to the left side of the array.



**Algorithm**

* Move along the input array starting from the end of array.
* Set all the nines at the end of array to zero.
* If we meet a not-nine digit, we would increase it by one. The job is done - return digits.
* We're here because **all** the digits were equal to nine. Now they have all been set to zero. We then append the digit 1 in front of the other digits and return the result.

**Implementation**

1 / 7

|  |
| --- |
| **class Solution {**  **public int[] plusOne(int[] digits) {**  **int n = digits.length;**  **// move along the input array starting from the end**  **for (int idx = n - 1; idx >= 0; --idx) {**  **// set all the nines at the end of array to zeros**  **if (digits[idx] == 9) {**  **digits[idx] = 0;**  **}**  **// here we have the rightmost not-nine**  **else {**  **// increase this rightmost not-nine by 1**  **digits[idx]++;**  **// and the job is done**  **return digits;**  **}**  **}**  **// we're here because all the digits are nines**  **digits = new int[n + 1];**  **digits[0] = 1;**  **return digits;**  **}**  **}** |

**Complexity Analysis**

Let N*N* be the number of elements in the input list.

* Time complexity: \mathcal{O}(N)O(*N*) since it's not more than one pass along the input list.
* Space complexity: \mathcal{O}(N)O(*N*)
  + Although we perform the operation **in-place** (i.e. on the input list itself), in the worst scenario, we would need to allocate an intermediate space to hold the result, which contains the N+1*N*+1 elements. Hence the overall space complexity of the algorithm is \mathcal{O}(N)O(*N*).