# 1946. Largest Number After Mutating Substring

Medium Greedy Array String Leetcode Link

## Problem Description

In this problem, you have a string `num` that represents a large integer and an array `change` of ten integers, where each index of the array corresponds to a digit from 0 to 9. The value at each index `d` in the array is the digit that `d` should be mapped to. You are allowed to mutate a single contiguous substring of `num` by replacing each digit in it according to the mapping defined by the `change` array. The goal is to perform this operation in such a way that the resulting integer, when converted back to a string, is as large as possible. To make it clearer, mutating a substring means selecting a continuous sequence of digits in `num` and replacing each of those digits with their corresponding digit from the `change` array. You can also choose not to mutate any substring at all. A substring here means any contiguous sequence of characters within the string `num`.

## Intuition

To get the largest possible number after mutation, one should start mutating the earliest digit in `num` that can be increased by the corresponding digit in `change`. Once you start mutation, continue to mutate sequentially as long as the digits can be increased or stay the same after replacement. If at any point, a subsequent digit cannot be increased through mutation (i.e., the digit in `num` is greater than the mapping digit in `change`), then stop the mutation process because continuing would result in a smaller number.

The reasoning for starting at the earliest possible digit is to take advantage of the positional value in numbers: a digit towards the left has more value than a digit towards the right. Thus, to maximize the number, we want to increase the value of the leftmost possible digits. Moreover, at the first occurrence where mutating does not increase the number anymore, we should stop. This strategy ensures we get the highest value after a single substring mutation.

The given solution implements this approach by:

1. Converting the string `num` into a list of characters `s` for easy manipulation.
2. Iterating through this list and checking, at each index, if the digit can be increased via mutation.
3. If a digit can be increased, mutating the current digit and continuing to the next digit until a digit is found that would not be increased by mutation, at which point the loop breaks.
4. After the loop, the possibly mutated list of characters `s` is joined back together to form a string representing the largest possible integer after mutation.

## Solution Approach

The algorithm uses a simple linear scan to implement the intuitive approach. Here's how the code executes the solution:

1. First, the string `num` is converted into a list of characters `s`, which allows for indexed access and in-place modification of the individual digits.

2. The algorithm then enters a loop, iterating over each digit in the list `s`. For each digit at index `i`, it is cast to an integer and compared against the digit it would map to in the `change` array.

3. If the digit from the `change` array is greater than the current digit in `s`, this means we can "mutate" this digit to get a larger number. The while loop is then entered to continue mutation as long as the criterion is met (i.e., the digit at the current index is less than or equal to the corresponding digit in the `change` array):

   ```
   while i < len(s) and int(s[i]) <= change[int(s[i])]:
       s[i] = str(change[int(s[i])])
       i += 1
   ```

4. The mutation starts at the current index `i` and continues to the subsequent digits as long as they are less than or equal to their mapped counterparts in the `change` array. This ensures that the number is strictly increasing or staying the same as we continue the mutation.

5. Once a digit is encountered where the `change` array does not provide a greater or equal digit, the while loop is exited (intuitively, mutation ends), and the outer for loop is also exited with a break statement. This ensures we are only mutating one contiguous substring.

6. The list is joined back into a string with `return ''.join(s)`, providing the mutated (or unmutated, if no mutation took place) number as the output.

In this approach, the data structure used is a list for `s`, which allows modifications to be made. No complex algorithms or data patterns are needed—just simple conditional logic and iteration.

String manipulation happens in place within the list of characters, which avoids the need for creating additional strings during the process. This is efficient in terms of space usage, since strings in Python are immutable and creating substrings or modifying strings would otherwise require additional space.

The linear scan enforces that the loop runs with a complexity of O(n), where n is the length of `num`. This makes the algorithm efficient because it scans each digit only once and performs in-place mutations without the overhead of additional string manipulation functions.

### Example Walkthrough

Let's say we are given the string `num = "132"` and the `change` array `change = [9, 8, 5, 0, 3, 2, 1, 2, 6, 4]`. Here is a step-by-step explanation to illustrate how the solution approach would work on this example:

1. Convert `num` to a list `s` of characters for manipulation:

   ```
   s = ['1', '3', '2']
   ```

2. Start iterating from left to right over each character in `s`.

3. For the first digit in `s`, `'1'`, the corresponding digit in the `change` array is `change[1] = 8`. Since `8` is greater than `1`, we can mutate this digit:

   ```
   s = ['8', '3', '2'] // The first digit is mutated to '8'
   ```

4. Continue to the next digit, `'3'`. The corresponding digit in the `change` array is `change[3] = 0`. Since `0` is not greater than `3`, we should stop mutating. However, we should check if the new digit is the same or greater to allow the mutation to continue, but in our case, it's not, so we stop.

5. The algorithm would have ideally continued mutating if the `change` digits were greater than or equal to the current digits in `s`, but since the digit `3` would decrease if we mutate it using the `change` array, the mutation process ends.

6. Finally, the list `s` is joined to form a string, giving us the result:

   ```
   result = '832'
   ```

And that's the final answer. We obtain the largest possible number after mutating the string `num` with the given `change` array using only one continuous substring which, in this case, was just the first digit.

## Python Solution

```python
class Solution:
    def maximumNumber(self, num: str, digit_mappings: List[int]) -> str:
        # Convert the number string to a list of characters for easy manipulation
        num_list = list(num)

        # Traverse through the number's characters
        for index, char in enumerate(num_list):
            # Determine if the current digit can be increased using the provided mapping
            if digit_mappings[int(char)] > int(char):
                # If it can, change the current digit and continue changing subsequent digits
                # until no further changes are beneficial
                while index < len(num_list) and int(num_list[index]) <= digit_mappings[int(num_list[index])]:
                    num_list[index] = str(digit_mappings[int(num_list[index])])
                    index += 1
                # Once a change is not beneficial, break the loop as no further checks are necessary
                break

        # Join the characters back into a string and return the result
        return ''.join(num_list)

```

## Java Solution

```java
class Solution {
    public String maximumNumber(String num, int[] change) {
        // Convert the input string to a character array to manipulate individual numbers.
        char[] digits = num.toCharArray();

        // Iterate through each digit in the character array.
        for (int i = 0; i < digits.length; ++i) {
            // Check if the current digit can be increased according to the change array.
            if (change[digits[i] - '0'] > digits[i] - '0') {
                // Continue to change the current and subsequent digits until it's no longer beneficial.
                while (i < digits.length && digits[i] - '0' <= change[digits[i] - '0']) {
                    // Apply the change to the current digit.
                    digits[i] = (char) (change[digits[i] - '0'] + '0');
                    i++;
                }
                // Once the changing is done, or no longer beneficial, exit the loop.
                break;
            }
        }
        // Return the new string with applied changes.
        return new String(digits);
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // This method returns the maximum number by swapping digits as per the change vector.
    string maximumNumber(string num, const vector<int>& change) {
        int numLength = num.size(); // Store the length of the input number string

        // Iterate over each digit of the number string
        for (int i = 0; i < numLength; ++i) {
            int digit = num[i] - '0'; // Convert character to digit

            // Check if the current digit can be increased by using the change array
            if (change[digit] > digit) {
                // Keep swapping until the change array offers a digit greater than or equal to the current digit
                while (i < numLength && change[digit] >= digit) {
                    num[i] = change[digit] + '0'; // Perform the swap by updating the character in the string
                    ++i; // Move to the next digit

                    // Make sure we are not at the end of the string and update the digit if not
                    if(i < numLength) {
                        digit = num[i] - '0';
                    }
                }
                break; // Once we cannot swap anymore, exit the loop as the string is maximized
            }
        }
        return num; // Return the modified number string which is now the maximum number
    }
};

```

## Typescript Solution

```typescript
// This method returns the maximum number by swapping digits as per the change array.
function maximumNumber(numStr: string, change: number[]): string {
    let numLength = numStr.length; // Store the length of the input number string

    // Convert the string to an array of characters for easy manipulation
    let numArr = numStr.split('');

    // Iterate over each digit of the number array
    for (let i = 0; i < numLength; ++i) {
        // Parse the current character to get the digit
        let digit = parseInt(numArr[i], 10);

        // Check if the current digit can be increased by using the change array
        if (change[digit] > digit) {
            // Keep swapping until the change array offers a digit greater than or equal to the current one
            while (i < numLength && change[digit] >= digit) {
                // Perform the swap by updating the character in the array
                numArr[i] = change[digit].toString();
                ++i; // Move to the next digit

                if (i < numLength) {
                    // Update the digit if we are not at the end of the array
                    digit = parseInt(numArr[i], 10);
                }
            }
            // Once we cannot swap anymore, break out of the loop as the string is now maximized
            break;
        }
    }

    // Join the array of characters back into a string to get the modified number
    return numArr.join('');
}

// Example usage:
// const result = maximumNumber("132", [9, 8, 5, 0, 3, 4, 2, 1, 9, 8]);
// console.log(result); // Output should be "832" after applying the change array.

```

## Time and Space Complexity

### Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the string `num`. The code consists of a single loop that goes through the characters of the string once, starting from the beginning of the string. The inner `while` loop runs only when a digit can be changed to a higher digit based on the `change` array, and it runs at most `n` times in total, not `n` times for each character, since the loop breaks once no more changes can be done. Therefore, in the worst-case scenario, each character is evaluated once, and the change is applied within the same iteration without causing another full scan of the string.

### Space Complexity

The space complexity is `O(n)`, where `n` is the length of the input string `num`. This is due to the conversion of the input string `num` to a list `s`, which creates a separate array of characters of size `n`. Other than that, there are no significant data structures used that would increase the space complexity.