670. Maximum Swap

Medium Greedy Math

Leetcode Link

Problem Description

In this problem, you are given an integer `num`. Your task is to figure out how to make this number the largest possible by swapping any two of its digits. However, you can only do this once. If the number is already the largest it can be, then no swaps are needed. The aim is to find the maximum value that can be created from `num` by making at most one swap between any two digits.

Intuition

The intuition behind the solution is to find the first pair of digits where a smaller digit precedes a larger digit when traversing from the rightmost digit towards the left. This guarantees that we will increase the value of the number by swapping a smaller digit in a more significant position with a larger digit in a less significant position.

To achieve this efficiently, we can traverse the number's digits from right to left and record the index of the largest digit seen so far at each step (called `d[i]` in the code). After we have this information, we then traverse the array of digits from left to right, looking for the first digit (`s[i]`) that is smaller than the maximum digit that comes after it (`s[d[i]]`). When we find such a digit, we perform the swap—an operation that will result in the maximum possible value after the swap—and then convert the list of digits back into an integer. If no such pair of digits is found, it means the number is already at its maximum value, and no swap is performed.

Here's a step-by-step breakdown of the provided code:

1. Convert the integer `num` to a list of digit characters `s`.
2. Initialize an array `d` that records the index of the largest digit to the right of each digit, including itself.
3. Traverse `s` from right to left (excluding the rightmost digit because a single digit can't be swapped), updating `d` such that `d[i]` holds the index of the largest digit to the right of `i`.
4. Traverse `s` from left to right this time, along with `d`. If a digit is found that's smaller than the largest digit to its right (`s[d[i]]`), perform a swap and break because further swaps are not allowed.
5. Join the list of characters `s` back to form the resulting maximum integer and return it.

This approach ensures we only perform a swap that results in the largest possible increase in the number's value, fulfilling the goal of the problem efficiently.

Solution Approach

To implement the solution given in the code, we will use simple list and array manipulation techniques. The procedure follows a greedy algorithm approach because it makes a locally optimal choice at each step that we hope will lead to a globally optimal solution.

Here's the detailed implementation of the solution:

1. **Conversion to a list of digits**: We start by converting the given integer `num` into a list `s` of its digit characters for easy manipulation.

```
s = list(str(num))
```

2. **Initialization of the largest digit indices array**: An array `d` is initialized with the same length as `s`. Each element at index `i` of `d` is intended to hold the index of the largest digit found to the right of `i`, inclusive of `i` itself.

```
d = list(range(n))
```

3. **Populating the largest digit indices array**: We traverse the digits from right to left, updating `d` such that `d[i]` points to the index of the largest digit found from `i` to the end of the list. This step uses dynamic programming to save the index of the largest digit seen so far at each position.

```
for i in range(n - 2, -1, -1):
    if s[i] >= s[d[i + 1]]:
        d[i] = d[i + 1]
```

4. **Finding and performing the optimal swap**: Once we have our array `d` ready, we traverse `s` from left to right using enumerate to get both the index and the digit. We look for the first digit `s[i]` that is smaller than the maximum to its right, `s[d[i]]`. When we find such a digit, we swap `s[i]` with `s[d[i]]` then immediately break the loop as only one swap is allowed.

```
for i, j in enumerate(d):
    if s[i] < s[j]:
        s[i], s[j] = s[j], s[i]
        break
```

5. **Result**: The list `s` should now represent the digits of the maximum value we can get. We join the list and convert it back to an integer to return it.

```
return int(''.join(s))
```

It's worth noting that the traversal from right to left to populate `d` and the logic used to decide when to swap are both guided by the algorithm's greediness. Greedy algorithms don't always guarantee an optimal solution for every kind of problem, but in this case, because of the problem's constraints (only one swap allowed), the greedy approach works perfectly to give the maximum value after one swap.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose `num` is `2736` and we want to find the largest number possible by swapping at most one pair of digits.

1. **Conversion to a list of digits**: We convert `num` to a list `s` to get `['2', '7', '3', '6']`.
2. **Initialization of the largest digit indices array**: Our array `d` is initialized to be `[0, 1, 2, 3]` with the same length as `s`, which represents indices of each digit.
3. **Populating the largest digit indices array**: We update `d` by traversing from right to left:

   Starting from the second to the last index:

   - At index 2 (`3`), the largest digit to the right is `6` at index 3. So, `d[2] = 3`.
   - At index 1 (`7`), `7` is larger than `3`, hence `d[1]` remains `1`.
   - At index 0 (`2`), `7` is the largest digit to the right, so `d[0] = 1`.

   Now `d` is `[1, 1, 3, 3]`.
4. **Finding and performing the optimal swap**: We use `d` to find the first digit to swap:

   - At index 0, `s[0] = '2'` is smaller than `s[d[0]] = '7'`. So we swap `s[0]` with `s[1]` and break the loop.

   After the swap, `s` becomes `['7', '2', '3', '6']`.
5. **Result**: We join `s` to form `7236`, which is returned as the largest number possible after one swap.

Thus, the solution to increase our number `2736` to its maximum by a single swap is to swap the digits `2` and `7` to get `7236`.

Python Solution

```python
class Solution:
    def maximumSwap(self, num: int) -> int:
        # Convert the number to a list of its digits in string form.
        digits = list(str(num))
        # Get the length of the list of digits.
        length = len(digits)
        # Create a list to keep track of the indices of the digits
        # that should be considered for swapping.
        max_digit_indices = list(range(length))

        # Populate the max_digit_indices list with the index of the maximum digit
        # from the current position to the end of the list.
        for i in range(length - 2, -1, -1):
            # If the current digit is less than or equal to the maximum digit found
            # to its right, update the max_digit_indices for the current position.
            if digits[i] <= digits[max_digit_indices[i + 1]]:
                max_digit_indices[i] = max_digit_indices[i + 1]

        # Loop through each digit to find the first instance where a swap would
        # increase the value of the number. Swap and break the loop.
        for i in range(length):
            max_index = max_digit_indices[i]
            # If the current digit is smaller than the maximum digit to its right,
            # swap the two digits.
            if digits[i] < digits[max_index]:
                # Swap the current digit with the maximum digit found.
                digits[i], digits[max_index] = digits[max_index], digits[i]
                # Only one swap is needed, so break after swapping.
                break

        # Convert the list of digits back to a string and then to an integer.
        return int(''.join(digits))

# The preceding code defines a member function `maximumSwap` within the class `Solution`,
# which takes an integer `num` and returns the maximum value integer obtained by swapping
# two digits once.
```

Java Solution

```java
class Solution {
    public int maximumSwap(int num) {
        // Convert the number to a character array to manipulate single digits
        char[] digits = String.valueOf(num).toCharArray();
        int length = digits.length;
        // Initialize an array to hold the indices of the 'max right' elements
        int[] maxRightIndex = new int[length];

        // Fill the array with the corresponding indices initially
        for (int i = 0; i < length; ++i) {
            maxRightIndex[i] = i;
        }

        // Populate the maxRightIndex array with the index of the greatest digit
        // to the right of each position i, inclusive
        for (int i = length - 2; i >= 0; --i) {
            // Update the index only if the current digit is less than or equal to
            // the maximum digit to the right
            if (digits[i] <= digits[maxRightIndex[i + 1]]) {
                maxRightIndex[i] = maxRightIndex[i + 1];
            }
        }

        // Iterate through each digit to find the first occurrence where the current
        // digit is less than the maximum digit to its right
        for (int i = 0; i < length; ++i) {
            int maxIndex = maxRightIndex[i];
            // If such a digit is found, swap it with the maximum digit to its right
            if (digits[i] < digits[maxIndex]) {
                char temp = digits[i];
                digits[i] = digits[maxIndex];
                digits[maxIndex] = temp;
                // Only the first such swap is needed for the maximum number, so break
                break;
            }
        }

        // Convert the modified character array back to an integer and return
        return Integer.parseInt(new String(digits));
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This method returns the maximum value by swapping two digits of the given number
    int maximumSwap(int num) {
        // Convert the number to a string for easy digit manipulation
        string numStr = to_string(num);
        int n = numStr.size(); // Get the length of the string representation of num
        vector<int> maxIndex(n); // Initialize a vector to store the indices of maximum digits following current

        // Initialize maxIndex with the indices from the string's end to the beginning
        iota(maxIndex.begin(), maxIndex.end(), 0);

        // Populate the maxIndex vector with the index of the maximum digit from current to the end
        for (int i = n - 2; i >= 0; --i) {
            if (numStr[i] <= numStr[maxIndex[i + 1]]) {
                maxIndex[i] = maxIndex[i + 1];
            }
        }

        // Traverse the string and find the first instance where swapping can maximize the number
        for (int i = 0; i < n; ++i) {
            int j = maxIndex[i]; // Get the index of the maximum digit we're considering for swap
            if (numStr[i] < numStr[j]) {
                // If the current digit is less than the max digit found, then swap and break
                swap(numStr[i], numStr[j]);
                break;
            }
        }

        // Convert the modified string back to an integer and return it
        return stoi(numStr);
    }
};
```

Typescript Solution

```typescript
function maximumSwap(num: number): number {
    // Convert the input number into an array of its digits.
    const digits = [];
    while (num != 0) {
        digits.push(num % 10);
        num = Math.floor(num / 10);
    }

    // The length of the digits array.
    const length = digits.length;

    // An array to store the index of the maximum digit encountered so far from right to left.
    const maxIndex = new Array(length);

    for (let i = 0, maxDigitIndex = 0; i < length; i++) {
        // Update the maximum digit index if a larger digit is found.
        if (digits[i] > digits[maxDigitIndex]) {
            maxDigitIndex = i;
        }
        // Assign the current maximum digit's index to the corresponding position in the array.
        maxIndex[i] = maxDigitIndex;
    }

    // Traverse the array from the last (most significant) digit to the first (least significant) digit.
    for (let i = length - 1; i >= 0; i--) {
        // Swap the current digit with the largest digit to its right, if there's any.
        if (digits[maxIndex[i]] != digits[i]) {
            [digits[maxIndex[i]], digits[i]] = [digits[i], digits[maxIndex[i]]];
            // Only the first swap is performed, so we break out of the loop after that.
            break;
        }
    }

    // Reconstruct the number from the array of digits.
    let resultNum = 0;
    for (let i = length - 1; i >= 0; i--) {
        resultNum = resultNum * 10 + digits[i];
    }

    // Return the final number after performing the maximum swap possible.
    return resultNum;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the number of digits in the input number. This is because the code consists of two separate for-loops that iterate over the digits. The first loop goes from the second-to-last digit to the first, updating an array of index positions, and the second loop goes once through the digits to find the first swap opportunity. Both loops have a maximum of `n` iterations, thus resulting in linear time complexity.

The space complexity of the code is also `O(n)` because it stores the digits in a list `s` of size `n` and another list `d` which is also of size `n`. Thus the total space used is proportional to the length of the input number.