2231. Largest Number After Digit Swaps by Parity

Easy Sorting Heap (Priority Queue)

Problem Description

In this problem, we are provided with a positive integer `num` and we're allowed to swap the digits of `num` under one condition: the swapped digits must have the same *parity* – meaning they are either both odd numbers or both even numbers. Our task is to find the maximal value of `num` that can be achieved through any number of such swaps.

Intuition

The key to solving this problem is to realize that the highest value of a number is obtained when its digits are sorted in non-increasing order from left to right. However, because we can only swap digits of the same parity, we need to handle even and odd digits separately.

The strategy is this:

1. Store the frequency (count) of each digit of `num`. This helps us keep track of how many times we can use each digit in the reconstruction of the largest possible number.
2. Iterate over the digits of `num` starting from the least significant digit (rightmost digit).
3. For each digit, we check if there is a larger digit of the same parity that we have not used yet (using our frequency count from step 1). If so, we place that larger digit in the current position to build the largest number.
4. We repeat this process for every digit, thereby creating the largest number possible under the constraint that we can only swap digits of the same parity.

The solution code provided uses a `Counter` from the `collections` module to keep track of the digit frequencies. By iterating over the digits and selecting the largest possible same-parity digit we have available each time, it constructs the solution effectively.

Solution Approach

The code implements the strategy described in the intuition by following these steps:

1. It starts by creating a `Counter` from Python's `collections` module, which is used as a hash map to store the frequency of each digit in the original number.
2. Then, the code iterates over the provided number `num` to populate the `Counter` with the correct frequencies of the digits.
3. With the `Counter` initialized, the solution iterates through the digits of `num` again, this time extracting each digit starting from the least significant digit (rightmost digit).
4. The extracted digit is compared against all available digits in descending order (from 9 to 0) to find a digit of the same parity that is greater and still available for swapping (based on the frequency stored in the `Counter`).
5. The `((v ^ y) & 1)` comparison is used to check if the current digit `v` and the potential digit `y` have the same parity. The XOR operator `^` gives 0 when both numbers have the same parity, and the bitwise AND with 1, `& 1`, filters the result to keep only even comparisons, ensuring parity is maintained.
6. Once a suitable digit `y` is found, it is placed at the current position in the answer by adding `y * t` to `ans`, where `t` is the positional value (1 for the unit's place, 10 for the ten's place, etc.). After a digit is used, its count in the `Counter` is decremented.
7. The loop continues until all digits of the original number have been replaced, building up the largest number possible digit by digit.
8. Finally, the function returns `ans`, which is the largest integer we can get after making all possible parity-preserving digit swaps in the initial number `num`.

This solution intelligently combines the `Counter` data structure with bitwise operations to ensure that swaps only occur between digits with the same parity and greedily selects the largest possible digit for each place value.

Example Walkthrough

Let's consider the number `num = 2736` and walk through the solution approach to find the maximal number that can be achieved through swapping digits of the same parity.

1. Create a `Counter` to store the frequency of each digit. For `num = 2736`, the counter would be `{2: 1, 7: 1, 3: 1, 6: 1}`.
2. Iterate over the digits of `num` starting from the least significant digit. We will process the digits in this order: `6, 3, 7, 2`.
3. Starting with the rightmost digit `6`, we look for the largest even digit available for swap—we can swap it with `2` which is the only even digit smaller than `6`. However, since `6` is already the largest even digit, we leave it as it is.
4. Move to the next digit `3`, an odd digit. The only odd digit larger than `3` available for swap is `7`. Thus, we swap `3` with `7`. Now our number looks like `2763`.
5. Next, we look at `7`. There are no digits larger than `7` that are odd, so we leave it as it is.
6. Finally, we move to the leftmost digit `2`. Since we cannot increase the value of `2` through any even-swaps (`6` is already at the rightmost position), we leave it as it is.
7. Continuing until all digits are checked, we have successfully transformed `2736` into `2763` by making the swap of `3` with `7`.
8. The resulting maximal number is `2763`, which is returned as the solution.

This example demonstrates how, by using a `Counter` to track digits and iterating from right to left to find the highest digit swap of the same parity, we can construct the largest possible number under the parity swap rule. The process emphasizes greedily selecting the best digit for each position while maintaining the condition of parity.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def largestInteger(self, num: int) -> int:
        # Create a counter to track the frequency of digits in the number
        digit_counter = Counter()

        # Extract each digit and update its count in the counter
        temp = num
        while temp:
            temp, digit = divmod(temp, 10)
            digit_counter[digit] += 1

        # Initialize variables to construct the largest number
        temp = num
        largest_number = 0
        place_value = 1  # Used to construct the number digit by digit

        # Construct the largest number with the same even/oddness of digits
        while temp:
            # Extract the rightmost digit from the remaining number
            temp, digit = divmod(temp, 10)

            # Try to find the largest digit with the same even/oddness
            for larger_digit in range(9, -1, -1):
                # Check if both have the same even/oddness by using XOR and if the digit is available
                if ((digit ^ larger_digit) & 1) == 0 and digit_counter[larger_digit] > 0:
                    # Append the largest digit at the current place value
                    largest_number += larger_digit * place_value
                    # Update the place value for the next digit
                    place_value *= 10
                    # Decrease the count of used digit in the counter
                    digit_counter[larger_digit] -= 1
                    break  # Exit the loop after finding the largest suitable digit

        return largest_number
```

Java

```java
class Solution {
    public int largestInteger(int num) {
        // Array to count the occurrences of each digit in the number
        int[] digitCounts = new int[10];
        int tempNum = num; // Temporary variable to manipulate the number without altering the original

        // Count occurrences of each digit
        while (tempNum != 0) {
            digitCounts[tempNum % 10]++;
            tempNum /= 10;
        }

        tempNum = num; // Reset tempNum to original number
        int result = 0;  // This will store the resulting largest integer
        int placeValue = 1; // Used to reconstruct the number by adding digits at the correct place value

        // Iterate over each digit in the number
        while (tempNum != 0) {
            int currentDigit = tempNum % 10; // Current digit of the number from right to left
            tempNum /= 10; // Remove the current digit from tempNum

            // Look for the largest digit not yet used that has the same parity (odd/even) as currentDigit
            for (int nextDigit = 9; nextDigit >= 0; --nextDigit) {
                // Check if 'nextDigit' has the same parity as 'currentDigit' and if it's available
                if (((currentDigit ^ nextDigit) & 1) == 0 && digitCounts[nextDigit] > 0) {
                    digitCounts[nextDigit]--; // Decrease count of the digit as it's now used
                    result += nextDigit * placeValue; // Add digit at correct place value to result
                    placeValue *= 10; // Increase place value for the next iteration
                    break; // Break since we have found the largest digit with same parity
                }
            }
        }

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

C++

```cpp
#include <vector> // Include the vector header for using the vector container
using namespace std;

class Solution {
public:
    // Function to calculate the largest integer formed by swapping digits with same parity
    int largestInteger(int num) {
        vector<int> digitCount(10, 0); // Create a count array initialized to 0 for digits 0-9
        int tempNum = num; // Temporary variable to process the number

        // Count the occurrence of each digit in the number
        while (tempNum > 0) {
            digitCount[tempNum % 10]++;
            tempNum /= 10;
        }

        tempNum = num; // Reset tempNum to process the number again
        int answer = 0; // Initialize the answer which will store the largest integer
        long multiplier = 1; // Initialize the multiplier for the current digit position

        // Construct the largest integer by picking the largest digits that have the same parity
        while (tempNum > 0) {
            int currentDigit = tempNum % 10; // Extract the least significant digit
            tempNum /= 10; // Remove the least significant digit from tempNum

            // Loop to find the largest digit of the same parity
            for (int newDigit = 9; newDigit >= 0; --newDigit) {
                // Check if the newDigit has the same parity as currentDigit and is available (count > 0)
                if ((currentDigit % 2 == newDigit % 2) && digitCount[newDigit] > 0) {
                    digitCount[newDigit]--; // Decrement the count since we're using this digit
                    answer += newDigit * multiplier; // Add the digit to the answer in correct position
                    multiplier *= 10; // Move to the next digit position
                    break; // Break since we found the largest digit of the same parity
                }
            }
        }
        return answer; // Return the answer after processing the entire number
    }
};
```

TypeScript

```typescript
function largestInteger(num: number): number {
    // Convert the given number to an array of its digits.
    let digits = String(num).split('').map(Number);

    // Arrays to hold the odd and even digits separately.
    let oddDigits: number[] = [];
    let evenDigits: number[] = [];

    // Iterate over each digit and classify them into odd or even.
    for (let digit of digits) {
        if (digit % 2 === 1) {
            oddDigits.push(digit);
        } else {
            evenDigits.push(digit);
        }
    }

    // Sort the odd and even digits arrays in ascending order.
    oddDigits.sort((a, b) => a - b);
    evenDigits.sort((a, b) => a - b);

    // Array to hold the rearranged largest integer.
    let largestIntDigits: number[] = [];

    // Construct the largest integer by selecting the largest available odd or even digit.
    for (let digit of digits) {
        if (digit % 2 === 1) {
            // For odd digits, pop the largest remaining odd digit.
            largestIntDigits.push(oddDigits.pop()!); // Using non-null assertion since we are sure the array is not empty.
        } else {
            // For even digits, pop the largest remaining even digit.
            largestIntDigits.push(evenDigits.pop()!); // Same non-null assertion applies here.
        }
    }

    // Convert the array of digits back to a number and return.
    return Number(largestIntDigits.join(''));
}
```

```python
from collections import Counter

class Solution:
    def largestInteger(self, num: int) -> int:
        # Create a counter to track the frequency of digits in the number
        digit_counter = Counter()

        # Extract each digit and update its count in the counter
        temp = num
        while temp:
            temp, digit = divmod(temp, 10)
            digit_counter[digit] += 1

        # Initialize variables to construct the largest number
        temp = num
        largest_number = 0
        place_value = 1  # Used to construct the number digit by digit

        # Construct the largest number with the same even/oddness of digits
        while temp:
            # Extract the rightmost digit from the remaining number
            temp, digit = divmod(temp, 10)

            # Try to find the largest digit with the same even/oddness
            for larger_digit in range(9, -1, -1):
                # Check if both have the same even/oddness by using XOR and if the digit is available
                if ((digit ^ larger_digit) & 1) == 0 and digit_counter[larger_digit] > 0:
                    # Append the largest digit at the current place value
                    largest_number += larger_digit * place_value
                    # Update the place value for the next digit
                    place_value *= 10
                    # Decrease the count of used digit in the counter
                    digit_counter[larger_digit] -= 1
                    break  # Exit the loop after finding the largest suitable digit

        return largest_number
```

Time and Space Complexity

Time Complexity

The given code consists of the following operations:

1. A `while` loop to count the digits (using `Counter`) of the input number `num`. This loop runs as many times as there are digits in `num`. The number of digits in a number is given by `O(log(n))`, where `n` is the input number.
2. Another `while` loop that also runs once for every digit in `num`. Within this loop, there's a `for` loop that could run up to 10 times (the digits 0 through 9) for each digit in the original number to find the largest digit with the same parity (odd or even) that hasn't been used yet.

Therefore, the overall time complexity of the code is governed by these nested loops, giving us:

- Outer loop: `O(log(n))` for the number of digits.
- Inner loop: `O(1)` since it's a constant run of up to 10.

Hence, the combined time complexity is `O(10 * log(n))`, which simplifies to `O(log(n))` since constant factors are dropped in Big O notation.

Space Complexity

The space complexity is determined by the additional space used by the algorithm:

1. `Counter` used to store the frequency of each digit. In the worst case, we store 10 key-value pairs, one for each digit from 0 to 9, so this uses `O(1)` space.
2. Constant amount of extra space used for variables `x`, `v`, `ans`, and `t`.

Thus, the overall space complexity of the code is `O(1)`, as it uses constant extra space regardless of the input size.