179. Largest Number

Medium | Greedy | Array | String | Sorting

Leetcode Link

Problem Description

The problem presents us with a list of non-negative integers called `nums`. We're tasked with rearranging these integers to create the largest possible number when they're concatenated together. It's important to realize that simply sorting the integers in descending order won't work because when concatenating numbers, the order in which we join them affects the final result. For example, '9' and '34' form a larger number when concatenated as '934' rather than '349'. As a final result can be extremely large, we must return it as a string instead of an integer.

Intuition

To find the correct order to concatenate the numbers to form the largest number, we need to determine a sorting strategy that compares combinations of two numbers. This strategy should allow us to understand which two numbers will produce a larger number when placed one after the other.

The intuition here is to design a custom sorting function that compares two strings—representations of the numbers—and decides which combination (either `a + b` or `b + a`) creates a larger number. For instance, when comparing `9` and `34`, we check if `934` is larger than `349`, so we can position `9` before `34` in the final arrangement.

We apply this custom sorting method to all pairs of numbers in `nums` to sort the entire list in such a way that concatenating the sorted elements yields the largest number.

The last caveat to address is when the list of numbers begins with one or multiple '0's due to the sorting, which would lead to a result starting with '0' (like '00' or '000'). The correct output should just be '0'. To solve this, we check the first element of the sorted array—if it's '0', we return '0' as the result, otherwise, we join all numbers into a string and return it.

Solution Approach

The solution implements a sorting strategy that targets the specific needs of the problem. Since we need to arrange the numbers to form the largest number possible when concatenated, we convert the integers to strings to allow for comparison concatenation.

Here's the step by step approach:

1. Convert each integer in `nums` to a string. This allows for easy concatenation and comparison.
2. Sort the string numbers using a custom comparator. This comparator is implemented with a lambda function that takes two string numbers `a` and `b` and uses `cmp_to_key` to convert the comparator result into a key for the sort function.
3. The lambda function defined for comparison returns `1` if `a + b < b + a`, meaning `b` should come before `a`, and `-1` if `a + b >= b + a`. The `cmp_to_key` method transforms this comparator to a function that can be used with the list's sort method.
4. If the first number is '0', it implies that all numbers are zero due to the sorting property, so we return '0'. Otherwise, we concatenate all numbers in a single string. This step ensures we don't end up with leading zeroes in the result.
5. Finally, join all string representations of the numbers to form the largest number and return it.

The key to this solution is the custom sort comparator, which isn't straightforward because we're comparing combinations of the numbers. The `cmp_to_key` from Python's `functools` module is used to convert the lambda function into a key function suitable for the sort method.

Overall, the algorithm's efficiency comes from the sorting, which operates in `O(n log n)` time complexity, where `n` is the number of elements in `nums`. The conversion to strings and the final concatenation of the sorted list have a lower time complexity and therefore don't dominate the time complexity of the whole algorithm.

Example Walkthrough

Let's illustrate the solution approach with a simple example using the list `nums = [3, 30, 34, 5, 9]`.

1. We begin by converting each integer in `nums` to a string so we'll end up with `["3", "30", "34", "5", "9"]`.

2. Next, we sort these strings using our custom comparator. This is how the comparisons would look:

   - Compare `3` vs `30`: Since `330` > `303`, `3` should come before `30`.
   - Compare `3` vs `34`: Since `334` > `343`, `3` should come before `34`.
   - Compare `3` vs `5`: Since `35` < `53`, `5` should come before `3`.
   - Compare `3` vs `9`: Since `39` < `93`, `9` should come before `3`.

   Continuing with the rest of the list in a similar fashion, our custom comparator will end up ordering our list as `["9", "5", "34", "3", "30"]`. Notice how `9` is placed before `5`, even though `9` is smaller than `5`, because `95` is larger than `59`.

3. We check if the first element is '0'. If so, the entire list would be zeros, and we would return '0'. In this case, it's not, so we proceed to the next step.

4. Finally, we concatenate all numbers in our sorted array to form the largest number, which is `"9534330"`.

By using this method, we ensure that we consider the best placement of each number to achieve the highest numerical value when the elements are concatenated. The resulting string, `"9534330"`, is the largest number that can be formed from the array.

Python Solution

```python
from functools import cmp_to_key

class Solution:
    def largest_number(self, nums: List[int]) -> str:
        # Convert all integers to strings for easy concatenation
        nums_as_str = [str(num) for num in nums]

        # Sort the list of strings, defining a custom comparator
        # that compares concatenated string combinations
        nums_as_str.sort(key=cmp_to_key(self._compare_numbers))

        # Special case: if the highest number is "0", the result is "0"
        # (happens when all numbers are zeros)
        if nums_as_str[0] == "0":
            return "0"

        # Join the numbers to form the largest number
        return "".join(nums_as_str)

    def _compare_numbers(self, a: str, b: str) -> int:
        # Custom comparator for sorting:
        # If the concatenation of a before b is less than b before a,
        # then we say a is "greater than" b in terms of the custom sorting.
        # That is, return -1 to indicate a should come before b.
        if a + b < b + a:
            return 1
        else:
            return -1

```

Java Solution

```java
class Solution {
    public String largestNumber(int[] nums) {
        // Create a list of strings to store the numbers as strings
        List<String> stringNumbers = new ArrayList<>();

        // Convert each integer in the array to a string and add it to the list
        for (int num : nums) {
            stringNumbers.add(String.valueOf(num));
        }

        // Sort the list using a custom comparator
        // The custom comparator defines the order based on the concatenation result
        // This ensures that the largest number is formed after sorting
        stringNumbers.sort((str1, str2) -> (str2 + str1).compareTo(str1 + str2));

        // After sorting, if the largest number is "0", it means all numbers were zeros
        // In that case, return "0"
        if ("0".equals(stringNumbers.get(0))) {
            return "0";
        }

        // Join all the strings in the list to get the final largest number representation
        return String.join("", stringNumbers);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    // Takes a vector of integers and returns the largest number possible
    // by concatenating their string representations
    string largestNumber(vector<int>& nums) {
        // Convert integers to strings
        vector<string> stringNums;
        for (int num : nums) {
            stringNums.push_back(to_string(num));
        }

        // Sort the strings based on a custom comparison function
        sort(stringNums.begin(), stringNums.end(),
             [](const string& a, const string& b) {
                 // The 'larger' number is the one that leads to a bigger
                 // concatenated string
                 return a + b > b + a;
             });

        // If after sorting the biggest number is "0",
        // the result is "0" (because all numbers are zeros)
        if (stringNums[0] == "0") {
            return "0";
        }

        // Build the largest number by concatenating sorted strings
        string result;
        for (const string& numStr : stringNums) {
            result += numStr;
        }

        return result;
    }
};

```

Typescript Solution

```typescript
function largestNumber(nums: number[]): string {
  // Convert integers to strings
  const stringNums: string[] = nums.map(num => num.toString());

  // Custom comparison function for sorting
  const compare = (a: string, b: string): number => {
    const concat1 = a + b;
    const concat2 = b + a;
    // If concat1 is 'greater' than concat2 in terms of
    // lexicographic order, it should come first
    return concat1 > concat2 ? -1 : (concat1 === concat2 ? 0 : 1);
  };

  // Sort the strings based on the custom comparison function
  stringNums.sort(compare);

  // Check if the largest element is "0", the result must be "0"
  // since all numbers would be zeros
  if (stringNums[0] === "0") {
    return "0";
  }

  // Build the largest number by concatenating sorted strings
  const result = stringNums.join('');

  return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity mainly depends on the sorting function. In the above code, the sorting function uses a custom comparator which is used to decide the order based on the concatenation of two numbers as strings.

- Converting all integers in the list to strings takes `O(N)` time, where `N` is the number of elements in `nums`.
- Sorting in Python usually has a time complexity of `O(N log N)`. However, the comparator itself is a lambda function that concatenates strings, which takes `O(k)` time, where `k` is the average length of the strings (or the average number of digits in the numbers). Therefore, the sorting step takes `O(N log N * k)`.
- The join operation on the sorted strings takes `O(N * k)` time.

Hence, the overall time complexity is `O(N log N * k)`, where `N` is the number of elements in `nums`, and `k` is the average number of digits in those elements.

Space Complexity

The space complexity of the code also has several components:

- Creating a new list of strings requires `O(N * k)` space.
- The sort operation might use additional space for its internal operations. In the worst case, the sorting algorithm might take `O(N * k)` space.
- The output string takes `O(N * k)` space.

Thus, the overall space complexity would be `O(N * k)`, taking into consideration the space used by the list of strings and the potential internal space used by the sorting algorithm.