1073. Adding Two Negabinary Numbers

Medium String Math

Leetcode Link

Problem Description

The given problem deals with the unusual concept of numbers represented in base -2. Unlike standard positive base numbers, in base -2, the value of each digit increases negatively as we move from least significant bit to most significant bit. That means each bit in the base -2 number can contribute a positive or negative value depending on its position. The challenge is to take two such base -2 numbers, `arr1` and `arr2`, and output the sum of these numbers also in base -2 array format. Just as in binary addition, the sum in base -2 could also involve carrying over or borrowing, but with an additional twist due to the negative base.

The provided arrays `arr1` and `arr2` are the representations of two base -2 numbers where indexes of the array represent the powers of -2 starting from 0 for the least significant bit. Note that there will be no leading zeroes in the numbers given, ensuring the most significant bit is always 1 if the number itself isn't 0.

Intuition

To solve this problem, we need to add two base -2 numbers while carrying over values similar to binary addition, but with modifications for the negative base. We start by setting up a loop from the least significant bit (LSB) to the most significant bit (MSB) of both input arrays, handling cases where one array might be longer than the other. During each iteration, we calculate the sum of the current bits and any previous carry-over.

In base -2, unlike base 2, when we have two '1' bits, adding them would give us '0' and would cause us to 'carry' a -1 to the next more significant bit. This is because 1 + 1 = 2, and in base -2, 2 is represented by a '0' in the current bit and subtracting '1' from the next more significant bit (since it's base -2). We must also handle the scenario where the addition can result in -1, which in base -2 means we need to carry over '+1' to the next bit.

The algorithm in the solution takes the above observations into account. We iterate through the input arrays bit by bit, calculate the sum, adjust the carry-over, and if necessary, continue calculating till all carry-overs are settled. This may sometimes lead to an extension of the resulting array if the carry extends beyond the length of both input arrays. The final step is to strip off any leading zeros to ensure the output conforms to the problem's no leading zero requirement. We then reverse the accumulated result to restore the array to the standard most significant bit to least significant bit format before returning.

Solution Approach

The solution utilizes a straightforward iterative approach, which is a common strategy when dealing with arithmetic problems involving positional number systems. The `while` loop constitutes the bulk of this approach, continuing as long as there are bits left to process in either `arr1` or `arr2`, or there exists a carry-over `c`.

First, we initialize index variables `i` and `j` to point to the LSB (last element) of `arr1` and `arr2` respectively, and a `c` variable to keep track of the carry-over set initially to 0. The variable `ans` is an empty list to store the resulting sum bits in reverse order, due to the traversal from LSB to MSB.

Within the loop, we check if the index `i` or `j` is within the bounds of `arr1` and `arr2`. If the index is out of bounds (i.e., `< 0`), we assume the value of that bit as `0`. The `a` and `b` variables hold the current bit values from `arr1` and `arr2` or `0` if the index is beyond the array length.

Here comes the crucial part: summing up the current bits and the carry-over. We do this by adding `a`, `b`, and `c` and assigning the result to `x`. According to the base -2 rules, if `x` is 2 or larger, we know we've added two '1's and thus we adjust `x` by subtracting 2 and setting the carry to -1, because in base -2, "10" is -2 + 0 which simplifies to -2. Similarly, if `x` equals -1, we set `x` to 1, and increment the carry because in base -2, negative carry-over flips to positive in the next more significant bit.

We append the resulting bit `x` to the `ans` array, decrement `i` and `j` to move to the next more significant bit, and loop continues.

After exiting the loop, trailing zeros are removed from `ans` as they do not affect the value of the number and are not allowed in the output format according to the problem description. The only exception is if the entire number is `0`, accounted for by the condition (while `len(ans) > 1` and `ans[-1] == 0`).

Lastly, we reverse the `ans` list to change the order from LSB-MSB to the required MSB-LSB before returning it as the final sum.

This algorithm is efficient and only requires $O(max(N,M))$ time complexity where N and M are the lengths of `arr1` and `arr2` respectively, as we iterate through each bit once. The space complexity is also $O(max(N,M))$ which is required to store the output.

Example Walkthrough

Let's consider an example to illustrate the solution approach by manually adding two base -2 numbers represented by the arrays `arr1 = [1, 0, 1]` and `arr2 = [1, 1, 1]`.

We will represent the numbers considering the least significant bit is on the left to match the reverse order we work with in the algorithm:

```
1   arr1 = 1 0 1  (which is 1*-2^0 + 0*-2^1 + 1*-2^2 = 1 + 0 - 4 = -3 in decimal)
2   arr2 = 1 1 1  (which is 1*-2^0 + 1*-2^1 + 1*-2^2 = 1 - 2 - 4 = -5 in decimal)
```

Now let's add them step by step as described in the solution approach:

1. Start with a `while` loop, indices `i` and `j` at the last elements, and carry `c` as 0.
2. At first iteration: `a = arr1[i]` which is 1, `b = arr2[j]` which is 1, `c` is 0. Sum `x = a + b + c` is 2. Following base -2 rules, we adjust `x` to 0 and set carry to -1.
3. Append `x = 0` to `ans` and decrement `i` and `j`, `ans = [0]`.
4. Next iteration: `a = 0` from `arr1`, `b = 1` from `arr2`, and carry `c = -1`. Sum is 0. As `x` is negative, we set `x` to 1 and add a positive carry of 1.
5. Append `x = 1` to `ans`. Now, `ans = [0, 1]`.
6. For the last bit (most significant), `a = 1` from `arr1`, `b = 1` from `arr2`, with a positive carry over. We sum up `x = a + b + c` which is 3, reduce it by 2 to get `x = 1` and carry over -1.
7. We append `x = 1` to `ans`. Now, `ans = [0, 1, 1]`.
8. As there are no more elements in the arrays and the carry is -1, we add another iteration and subtract 2 again from the next significant bit, producing a 1 and carry -1.
9. `ans` is updated to `[0, 1, 1, 0]`.
10. Next, carry is -1, resulting in a bit of 1 with a carry of 1 (since -1 + 2 = 1 in base -2), producing a final array of `[0, 1, 1, 0, 1]`.

Before returning the result, we must remove any leading zeros (keeping one if the number is zero) and then reverse the array for the proper base -2 representation:

```
1   After removing leading zeros: [1, 1, 0, 1]
2   After reversing: [1, 0, 1, 1]
```

The final `ans` array `[1, 0, 1, 1]` represents the sum of `arr1` and `arr2` in base -2, which is the number 4-1+3 in decimal. As one can check, -3 + (-5) = -8, and -8 in base -2 can be depicted as `1*-2^0 + 0*-2^1 + 1*-2^2 + 1*-2^3 = 1 + 0 - 4 + 8 = 5`, which is the sum we expected.

Python Solution

```python
from typing import List

class Solution:
    def add_negabinary(self, arr1: List[int], arr2: List[int]) -> List[int]:
        # Initialize pointers for arr1 and arr2
        pointer_arr1, pointer_arr2 = len(arr1) - 1, len(arr2) - 1

        # Initialize carry to 0
        carry = 0

        # Initialize result as an empty list
        result = []

        # Loop until we have processed both arrays and any carry left
        while pointer_arr1 >= 0 or pointer_arr2 >= 0 or carry:
            # Get the current digits or 0 if we have passed the beginning
            digit_arr1 = 0 if pointer_arr1 < 0 else arr1[pointer_arr1]
            digit_arr2 = 0 if pointer_arr2 < 0 else arr2[pointer_arr2]

            # Calculate the new digit and adjust carry if necessary
            digit_sum = digit_arr1 + digit_arr2 + carry
            carry = 0  # reset carry
            if digit_sum == 2:
                # When sum is 2 or more we have to subtract 2 and add -1 to carry for negabinary
                digit_sum = 0
                carry -= 1
            elif digit_sum == -1:
                # When sum is -1 in negabinary, we have to add one to the digit and subtract from carry
                digit_sum = 1
                carry += 1

            # Append the result digit to the result list
            result.append(digit_sum)

            # Move the pointers backwards
            pointer_arr1, pointer_arr2 = pointer_arr1 - 1, pointer_arr2 - 1

        # Remove leading zeros from the result list except the last 0
        while len(result) > 1 and result[-1] == 0:
            result.pop()

        # Reverse the result to get the correct order since we added digits from the least significant
        return result[::-1]
```

Java Solution

```java
class Solution {
    public int[] addNegabinary(int[] arr1, int[] arr2) {
        // Initialize indices for the last elements of arr1 and arr2
        int i = arr1.length - 1;
        int j = arr2.length - 1;

        // Prepare a list to store the result
        List<Integer> result = new ArrayList<>();

        // 'carry' will keep track of the value to carry over to the next digit
        for (int carry = 0; i >= 0 || j >= 0 || carry != 0; --i, --j) {
            // Retrieve or default to zero if the index is less than zero
            int digitArr1 = i < 0 ? 0 : arr1[i];
            int digitArr2 = j < 0 ? 0 : arr2[j];

            // Calculate the sum of the current digits and the carry
            int sum = digitArr1 + digitArr2 + carry;
            carry = 0; // Reset the carry

            // Adjust the sum and carry for the negabinary system rules
            if (sum == 2) {
                sum = 0;
                carry = -1; // In negabinary, carrying over '2' results in adding '-1' to the next more significant digit
            } else if (sum == -1) {
                sum = 1;
                carry = 1; // In negabinary, having a sum of '-1' requires converting to '1' and carrying '1' over
            }

            // Add the calculated sum to the results list
            result.add(sum);
        }

        // Remove leading zeros, except for the situation where the sum is exactly '0'
        while (result.size() > 1 && result.get(result.size() - 1) == 0) {
            result.remove(result.size() - 1);
        }

        // Since the current result is in reverse order, reverse it to obtain the correct order
        Collections.reverse(result);

        // Convert the list of Integers to a primitive int array
        return result.stream().mapToInt(x -> x).toArray();
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to add two negabinary numbers
    std::vector<int> addNegabinary(std::vector<int>& arr1, std::vector<int>& arr2) {
        int firstIndex = arr1.size() - 1; // Set the starting index for arr1
        int secondIndex = arr2.size() - 1; // Set the starting index for arr2
        std::vector<int> result; // Vector to store the result
        int carry = 0; // Initialize the carry to 0

        // Iterate while either arr1 or arr2 has digits left, or carry is non-zero
        while (firstIndex >= 0 || secondIndex >= 0 || carry) {
            int firstValue = firstIndex < 0 ? 0 : arr1[firstIndex]; // Get arr1 digit or 0 if index is negative
            int secondValue = secondIndex < 0 ? 0 : arr2[secondIndex]; // Get arr2 digit or 0 if index is negative
            int sum = firstValue + secondValue + carry; // Calculate the sum with carry

            // Reset the carry for the next calculation
            carry = 0;

            // Adjust the sum and carry for the next digit if necessary
            if (sum == 2) {
                sum = 0;
                carry = -1;
            } else if (sum == -1) {
                sum = 1;
                carry = 1;
            }

            // Add the calculated sum to the result
            result.push_back(sum);

            // Decrement indices for next loop iteration
            --firstIndex;
            --secondIndex;
        }

        // Remove any leading zeros (but keep one zero if the result is zero)
        while (result.size() > 1 && result.back() == 0) {
            result.pop_back();
        }

        // Reverse the result to get the correct ordering of digits
        std::reverse(result.begin(), result.end());

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

Typescript Solution

```typescript
function addNegabinary(arr1: number[], arr2: number[]): number[] {
    let indexArr1 = arr1.length - 1; // Start from the end of the first array
    let indexArr2 = arr2.length - 1; // Start from the end of the second array
    const result: number[] = []; // Initialize the result array
    let carry = 0; // Initialize the carry variable, which will store the carryover during addition

    // Loop until both arrays are processed or there is a carry
    while (indexArr1 >= 0 || indexArr2 >= 0 || carry) {
        // Get the value from arr1 or 0 if indexArr1 is less than 0
        const value1 = indexArr1 < 0 ? 0 : arr1[indexArr1];
        // Get the value from arr2 or 0 if indexArr2 is less than 0
        const value2 = indexArr2 < 0 ? 0 : arr2[indexArr2];

        // Perform the addition with the values and the carry
        let sum = value1 + value2 + carry;
        carry = 0; // Reset carry to 0 for the next iteration

        // Correct the sum and update the carry according to negabinary rules
        if (sum >= 2) {
            sum -= 2; // Subtract 2 when sum is 2 or more
            carry = -1; // Set carry to -1 when we're in negabinary
        } else if (sum == -1) {
            sum = 1; // If left sum to 1 when it is -1
            carry = 1; // Carry over 1
        }

        // Prepend the calculated digit to the result array
        result.unshift(sum);

        // Move to the next digits
        indexArr1--;
        indexArr2--;
    }

    // Trimming leading zeros, except for the last 0 which represents the number zero
    while (result.length > 1 && result[0] === 0) {
        result.shift();
    }

    return result; // Return the computed negabinary number as an array of digits
}
```

Time and Space Complexity

The code provided sums two negabinary numbers, represented as lists `arr1` and `arr2`. The time complexity and space complexity of the code are as follows:

Time Complexity

The time complexity is governed by the length of the two input lists. Let `n` be the length of the longer list (`arr1` or `arr2`). The main loop runs at most `n` times if one list is smaller than the other, and additional iterations occur for the carry `c` handling.

Therefore, the time complexity of the code is $O(n)$, where n is the length of the longer input list. The while loops at the end trim leading zeros and take at most $O(n)$ time in the worst case (when all bits are zeros except the first bit).

Space Complexity

The space complexity is determined by the additional space used to store the result. The `ans` list is built to store the sum of the two numbers, with space for an additional carry if necessary.

Thus, the space complexity of the code is also $O(n)$, where n is the length of the longer input list. This accounts for the space needed for the `ans` list. It is essential to note that this does not include the space used to store the input lists themselves; it is the extra space required by the algorithm.