1017. Convert to Base -2

Medium Math Leetcode Link

Problem Description

The problem requires us to convert a given integer `n` into its equivalent representation in base `-2`. Base `-2` numbering system, also known as negabinary, is similar to binary except that the base is `-2` instead of `2`. This means that each digit position is worth `-2` raised to the power of the position index, starting from `0`. The challenge is to represent any integer `n` with digits `0` and `1` in base `-2` such that there are no leading zeros, except when the resulting string should be `"0"`.

Intuition

The solution is derived using a similar approach to converting numbers from decimal to binary, with some modifications due to the negative base. The conventional method to convert a decimal number to binary involves repeatedly dividing the number by `2` and taking the remainder as the next digit. The process continues until the number is completely reduced to 0. In the case of the negabinary system, we continue to divide by `-2`, but we need to handle negative remainders differently.

When converting to negabinary, if the remainder when dividing by `-2` is negative, we adjust the number by adding `1` to prevent the use of negative digits. Since we're working with a negative base, adding `1` effectively subtracts the base from the original number, which in turn cancels out the negative remainder and keeps the number non-negative.

Each division gives us one binary digit, but due to the nature of base `-2`, we sometimes need to increment the quotient to handle the cases where the remainder would otherwise be negative. The pattern of division continues until the dividend (our number `n`) becomes zero. The algorithm appends `'1'` if the expression `n % 2` evaluates to `true` (i.e., remainder is `1` when dividing by `2`), which indicates that our current digit in negabinary is `1`. Otherwise, it appends `'0'`. After each step, `n` is divided by `2` and `k`, which toggles between `1` and `-1`, is used to adjust `n` accordingly. The sequence of `1`s and `0`s is reversed at the end to get the final negabinary representation.

Solution Approach

The `Solution` class defines a single method `baseNeg2`, which takes an integer `n` as input and returns a string representing the negabinary representation of that integer.

Here's how the implementation works:

- We initialize a variable `k` with the value `1`. This will be used to alternate the subtraction value between `1` and `-1` according to the negabinary rules.
- We create an empty list `ans` that will eventually contain the digits of the negabinary number, albeit in reverse order.
- We enter a `while` loop that will keep running until `n` is reduced to zero. On each iteration, we perform the following steps:
  - Check if `n % 2` is `1` or `0` by using the modulo operator. This is equivalent to finding out if there is a remainder when `n` is divided by `2`:
    - If `n % 2` evaluates to `1`, it means the current binary digit should be `'1'`, and we append `'1'` to the `ans` list.
    - Furthermore, we adjust `n` by subtracting `k`. Since the base is negative, when the remainder is `1`, we need to compensate by decrementing `n` by `1` when `k` is `1` or incrementing by `1` when `k` is `-1`. This ensures we are correctly building the negabinary representation.
    - If `n % 2` evaluates to `0`, we simply append `'0'` to the `ans` list. There is no need to adjust `n` since the remainder is `0`.
  - Update `n` to be half of its previous value, which is done by using integer division `n //= 2`. Integer division truncates towards zero, making sure that we are getting closer to zero on each division.
  - Multiply `k` by `-1`. This simple operation alternates the value of `k` between `1` and `-1`, ensuring that we properly adjust `n` in the next iteration to account for the negative base.
- Once the `while` loop exits, we reverse the `ans` list to represent the digits in the correct order since we've been appending them in reverse.
- We then join the list into a string to represent the negabinary number. If the list is empty, which would occur if `n` were `0`, we return `'0'`.

By the end of the loop, the `ans` list will contain the negabinary digits in reverse order, so we need to reverse it to get the correct representation. We perform this reverse operation using `ans[::-1]`. If the number `n` is `0`, then `ans` will be an empty list, in which case we return `'0'`.

The Python `join` function concatenates the elements of the list into a single string, which is the final negabinary representation we return from our method.

This approach cleverly utilizes modular arithmetic and the property of negabinary representation to efficiently convert an integer to its equivalent in base `-2`. No additional data structures besides the list `ans` are used, and the entire conversion happens in a single pass through the loop with an `O(log n)` time complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach by converting the decimal number `6` to its equivalent in base `-2`.

1. Start with `n = 6` and initialize `k = 1`.
2. Initialize an empty list `ans` which will hold the digits of the negabinary number in reverse order.

The while-loop begins:

- Iteration 1:
  - `n % 2` is `0` (since 6 is even), so we append `'0'` to `ans`. (`ans = ['0']`)
  - We do not adjust `n` because there is no remainder.
  - Now, perform integer division `n //= 2`, which gives `n = 3`.
  - Multiply `k` by `-1` to get `k = -1`.
- Iteration 2:
  - `n % 2` is `1`, so we append `'1'` to `ans`. (`ans = ['0', '1']`)
  - Adjust `n` by subtracting `-1` (since `k = -1`), giving us `n = 3 - (-1) = 4`.
  - Perform integer division `n //= 2`, yielding `n = 2`.
  - Multiply `k` by `-1` to get `k = 1`.
- Iteration 3:
  - `n % 2` is `0`, so we append `'0'` to `ans`. (`ans = ['0', '1', '0']`)
  - `n` does not need adjustment as the remainder is `0`.
  - Perform integer division `n //= 2`, which results in `n = 1`.
  - Multiply `k` by `-1` to get `k = -1`.
- Iteration 4:
  - `n % 2` is `1`, so we append `'1'` to `ans`. (`ans = ['0', '1', '0', '1']`)
  - Adjust `n` by subtracting `-1` (since `k = -1`), which makes `n = 1 - (-1) = 2`.
  - Perform integer division `n //= 2`, yielding `n = 1`.
  - Multiply `k` by `-1` to get `k = 1`.
- Iteration 5:
  - `n % 2` is `1`, so we append `'1'` to `ans`. (`ans = ['0', '1', '0', '1', '1']`)
  - Adjust `n` by subtracting `1` (since `k = 1`), which leaves `n = 1 - 1 = 0`.
  - The division step and multiplication of `k` by `-1` are no longer necessary as `n` is now `0`, so the while loop terminates.

Now, `ans = ['0', '1', '0', '1', '1']` holds the negabinary digits in reverse order. To fix this, we reverse `ans` to get `['1', '1', '0', '1', '0']`.

Finally, join all elements of the list `ans` to get the string `'11010'`, which is the negabinary representation of the decimal number `6`.

Thus, the `baseNeg2` method would return `'11010'` when called with the input `6`.

Python Solution

```python
class Solution:
    def baseNeg2(self, n: int) -> str:
        # Initialize a variable to keep track of the power of (-2)
        power_of_neg_2 = 1
        # List to store the digits of the result
        result_digits = []

        # Continue the loop until n becomes 0
        while n:
            # Check if the current bit is set (if n is odd)
            if n % 2:
                # If the bit is set, append '1' to the result digits array
                result_digits.append('1')
                # Since the base is -2, when a '1' is added at this power, we need to subtract
                # the power of (-2) to balance the actual value.
                n -= power_of_neg_2
            else:
                # If the bit is not set, append '0' to the result digits array
                result_digits.append('0')
            # Divide n by 2 (shift right), this will move to the next bit
            n //= 2
            # Multiply the power by -1 because each time we move one position to the left
            # in base -2 notation, the power of the digit alternates signs
            power_of_neg_2 *= -1

        # Since we've been appending digits for the least significant bit (LSB) to the most
        # significant bit (MSB), we need to reverse the array to get the correct order.
        # If there are no digits in the result, return '0'.
        return ''.join(reversed(result_digits)) or '0'

```

Java Solution

```java
class Solution {
    public String baseNeg2(int number) {
        // If number is 0, the base -2 representation is also "0".
        if (number == 0) {
            return "0";
        }

        // StringBuilder used to build the result from right to left.
        StringBuilder result = new StringBuilder();

        // Variable to alternate the sign (k = 1 or k = -1).
        int signAlteration = 1;

        // Continue looping until the number is reduced to 0.
        while (number != 0) {
            // Checking the least significant bit of 'number'.
            if (number % 2 != 0) {
                // If it's 1, append "1" to the result and subtract the sign alteration from number.
                result.append(1);
                number -= signAlteration;
            } else {
                // If it's 0, append "0" to the result (nothing gets subtracted from number).
                result.append(0);
            }

            // Alternate the sign for next iteration.
            signAlteration *= -1;

            // Divide the number by 2 (right shift in base -2 system).
            number /= 2;
        }

        // Since we've been building the result in reverse, we need to reverse the order now.
        return result.reverse().toString();
    }
}

```

C++ Solution

```cpp
#include <algorithm> // For reverse function
#include <string>    // For string class

class Solution {
public:
    // Function returns the base -2 representation of the integer n as a string.
    string baseNeg2(int n) {
        // When n is 0, the base -2 representation is just "0".
        if (n == 0) {
            return "0";
        }

        // Initialize an empty string to build the base -2 representation.
        string baseNeg2Representation;

        // Helper variable to keep track of the alternating signs when dividing by -2.
        int alternatingSignFactor = 1;

        // Process the number n until it becomes 0.
        while (n != 0) {
            // Examine the least significant bit (LSB) to determine whether to place 1 or 0.
            // If LSB is 1 (odd number), add '1' to the representation.
            if (n % 2 == 1) {
                baseNeg2Representation.push_back('1');
                // Adding alternatingSignFactor (which is either +1 or -1) ensures
                // we either increase or decrease n to make n divisible by 2,
                // which helps with the next division.
                n -= alternatingSignFactor;
            }
            // If LSB is 0 (even number), add '0' to the representation.
            else {
                baseNeg2Representation.push_back('0');
            }

            // Negate the alternatingSignFactor to handle base of -2.
            alternatingSignFactor *= -1;

            // Integer division by 2 to remove the processed bit.
            n /= 2;
        }

        // The constructed string needs to be reversed because we started from LSB to MSB.
        reverse(baseNeg2Representation.begin(), baseNeg2Representation.end());

        return baseNeg2Representation;
    }
};

```

Typescript Solution

```typescript
function baseNeg2(n: number): string {
    // If the input number is zero, return '0' as the base -2 representation.
    if (n === 0) {
        return '0';
    }

    // Initialize the power of -2 to 1 (since (-2)^0 is 1).
    let powerOfNegTwo = 1;

    // An array to store the binary digits of the base -2 representation.
    const baseNegTwoDigits: string[] = [];

    // Iterate until n doesn't become zero.
    while (n) {
        // If n is odd, append '1' to the digits array and decrement n by the current powerOfNegTwo.
        if (n % 2) {
            baseNegTwoDigits.push('1');
            n -= powerOfNegTwo;
        } else {
            // If n is even, append '0' to the digits array.
            baseNegTwoDigits.push('0');
        }

        // Alternate the sign of the powerOfNegTwo (-1, 1, -1, 1,...) as we go to higher powers.
        powerOfNegTwo *= -1;

        // Integer division by 2, Math.trunc discards the decimal part.
        n = Math.trunc(n / 2);
    }

    // Reverse the digits and join them to return the string representation of base -2 number.
    return baseNegTwoDigits.reverse().join('');
}

```

Time and Space Complexity

The time complexity of the given code is `O(log(n))`. This is because the loop iterates once for every bit that `n` produces when converging to 0. Since we're halving `n` at every step, albeit with an additional operation to account for negative base representation, the number of iterations grows logarithmically with respect to the input `n`.

The space complexity is `O(log(n))` as well, this is due to the `ans` list that is used to store the binary representation of the number in base -2. The number of digits needed to represent `n` in base -2 scales similarly to the number of iterations, hence the space complexity follows the same logarithmic pattern.