405. Convert a Number to Hexadecimal

Easy | Bit Manipulation | Math

Leetcode Link

Problem Description

The LeetCode problem requires us to convert an integer to its hexadecimal representation. The process should work for both positive and negative integers. When dealing with the hexadecimal system, instead of using the decimal digits 0-9, we also use the letters 'a' to 'f' to represent the values 10 to 15. For negative integers, the standard method of conversion is using the two's complement representation.

Here are key points to consider:

- The output should be a string.
- Each hexadecimal character is a representation of 4 bits.
- The letters in the output should all be lowercase.
- The output should not have any leading zeroes unless the number is '0'.
- Built-in library methods for direct conversion are not allowed, implying that the solution should manually perform the conversion.

Intuition

The solution uses bitwise operations and direct character mapping to convert an integer to its hexadecimal equivalent.

Here's the reasoning behind the solution:

1. Since we cannot use built-in library methods, we create a string `chars` containing hexadecimal characters '0' to 'f' which map to their corresponding decimal values from 0 to 15.
2. The given number is processed from its most significant hex digit to its least significant digit, checking 4 bits at a time (which corresponds to a single hex digit). This is achieved by using bitwise operations:
   - The right-shift operation (`>>`) is used to bring the set of 4 bits we want to convert into the least significant position.
   - The bitwise AND operation (`&`) with `0xF` (which is `1111` in binary) is used to isolate these 4 bits.
3. For each set of 4 bits, we look up the corresponding hexadecimal digit from our predefined string `chars` and add it into an array `s`.
4. We use a flag (`f` or `x != 0`) to avoid adding leading zeros to our result array `s`. This way, zero(es) will be appended only if there are non-zero digits already in the array, or the current digit is itself non-zero.
5. Since we're going from most significant hex digit to least significant, we don't need to reverse the array `s` at the end, we simply join it to form the result string.

The core of the solution hinges on the concept that a hexadecimal number is just a representation of a binary number with a grouping of 4 bits at a time and that two's complement binary representation for negative numbers will automatically handle the sign.

Solution Approach

The implementation of the solution uses bit manipulation, which is a powerful technique in computer programming for tasks involving low-level data processing. Bit manipulation operations are used to perform operations directly on the binary digits or bits of a number. In the case of converting an integer to its hexadecimal representation, we particularly use bitwise shifting and bitwise AND operations.

Here's a step-by-step breakdown of how the solution approach works:

1. **Special Case for Zero:** The function begins by checking if `num` is zero. If it is, the function immediately returns `'0'` since, in hexadecimal, `0` is represented as `0`.
2. **Character Mapping for Hexadecimal Digits:** A string `chars`, which contains all the hexadecimal digits from `0` to `9` and `a` to `f`, is used as a map. This is the data structure used for converting a 4-bit binary number to its hexadecimal character.
3. **Array Initialization:** An empty list `s` is initialized to accumulate the resulting hexadecimal characters. This list will act as a stack to hold the characters before joining them into a final string.
4. **Iterating Over Each Hex Digit:** The main for-loop processes the integer 4 bits at a time, which corresponds to one hex digit. The loop iterates 8 times since an integer in most systems is 32 bits and 32/4 = 8 hex digits. The range in the loop, `range(7, -1, -1)`, indicates the iteration goes from the most significant hex digit to the least significant.
5. **Extracting Hex Digits Using Bit Manipulation:**
   - Bitwise Right Shift: The number `num` is right-shifted `4 * i` times, bringing the current hex digit to the least significant position.
   - Bitwise AND with `0xF`: The operation `(num >> (4 * i)) & 0xF` then isolates these 4 bits.
6. **Avoiding Leading Zeros:** To avoid leading zeros in the output, there is a check to see if the list `s` is empty or if the current hex digit `x` is non-zero before appending the character to the `s` list. This way, we are sure that the final string will not contain unnecessary leading zeros.
7. **Appending to the Result List:** Once the correct hex digit is determined, it is appended to the list `s`.
8. **Building the Result String:** After the loop finishes processing all hex digits, the list `s` is joined (`''.join(s)`) to form the hexadecimal string, which is then returned as the result.

The solution makes judicious use of bit manipulation to convert each group of 4 bits into its hexadecimal character equivalent without ever processing the entire number as a whole. This localized processing is ideal for handling large integers and ensures compatibility with the two's complement representation for negative integers.

Example Walkthrough

Let's walk through an example to understand how the solution approach works. Consider the decimal number `26` as our example, which needs to be converted into a hexadecimal string.

Following our step-by-step solution:

1. **Special Case for Zero:** First, we check if the number is zero. In our case, `26` is not zero, so we can proceed to the next steps.
2. **Character Mapping for Hexadecimal Digits:** We have a string `chars = '0123456789abcdef'`, which helps us map each 4-bit binary number to its hex character.
3. **Array Initialization:** We initialize an empty list `s = []` that will hold each hex character before we join them.
4. **Iterating Over Each Hex Digit:** The `for` loop will iterate 8 times since a 32-bit integer has 8 hex digits. But for simplicity (our number is small), we only need to consider iterations that don't lead to a `0` result from `(num >> (4 * i)) & 0xF`. This means for our example, `26`, we only need the last two iterations, `i=1` and `i=0`.
5. **Extracting Hex Digits Using Bit Manipulation:**
   - For `i=1`:
     - Right shift `26` by `4 * 1 = 4`; `26 >> 4` gives `1`.
     - Bitwise AND with `0xF`: `(1) & 0xF` results in `1`.
   - Since `s` is empty and `1` is non-zero, we should append its hex character to `s`. So, we look up `1` in `chars`, which gives us `'1'`, and we append it: `s += ['1']`.
   - For `i=0`:
     - Right shift `26` by `4 * 0 = 0`; `26 >> 0` gives `26`.
     - Bitwise AND with `0xF`: `(26) & 0xF` results in `10` (since binary 26 is `11010` and `1010` is `10` in decimal).
   - Since `10` is non-zero and `s` is not empty, we look up `10` in `chars`, yielding `'a'`. Append it to `s`: `s += ['a']`.
6. **Avoiding Leading Zeros:** As explained in the steps, we only appended characters since we did not encounter a situation where a zero would be a leading character.
7. **Appending to the Result List:** At this point, we have `s = ['1', 'a']`.
8. **Building the Result String:** Lastly, we join the list into a string, resulting in `'1a'`, which is the hexadecimal representation of `26`.

Upon running this process on the given number `26`, we've successfully converted it to its hexadecimal representation `'1a'` using the explained bit manipulation technique.

Python Solution

```python
class Solution:
    def toHex(self, num: int) -> str:
        # If the number is 0, return '0' directly
        if num == 0:
            return '0'

        # String containing hexadecimal characters for conversion
        hex_chars = '0123456789abcdef'

        # List to store hexadecimal characters
        hex_string = []

        # We extract 8 bits at a time from the integer, and we process from the MSB to the LSB
        # On a 32-bit architecture, we process the integer as 8 groups of 4 bits.
        for i in range(7, -1, -1):
            # Extract 4 bits by shifting right (4 * position) and masking with 0xF to get the value
            current_bits = (num >> (4 * i)) & 0xF

            # If the hex_string list is non-empty or the current_bits are non-zero, append the corresponding hex character
            # This check also prevents leading zeros from being included in the output
            if hex_string or current_bits != 0:
                hex_string.append(hex_chars[current_bits])

        # Join the list into a string and return it as the hexadecimal representation
        return ''.join(hex_string)
```

Java Solution

```java
public class Solution {

    // Method to convert a given integer to a hexadecimal string
    public String toHex(int num) {

        // If the number is zero, then the hexadecimal string is simply "0"
        if (num == 0) {
            return "0";
        }

        // StringBuilder to build the hexadecimal string
        StringBuilder hexBuilder = new StringBuilder();

        // Loop to process the number until all hexadecimal digits are obtained
        while (num != 0) {

            // Extract the last 4 bits of the number
            int last4Bits = num & 15; // 15 in hexadecimal is 0xF

            // If the value of last 4 bits is less than 10, append the corresponding decimal value as character
            if (last4Bits < 10) {
                hexBuilder.append(last4Bits);
            } else {
                // If the value is 10 or above, convert it to a hexadecimal character from 'a' to 'f'
                hexBuilder.append((char) (last4Bits - 10 + 'a'));
            }

            // Shift the number 4 bits to the right to process the next hexadecimal digit
            // Using the unsigned right shift operator ">>>" to handle negative numbers
            num >>>= 4;
        }

        // Reverse the StringBuilder contents to get the right hexadecimal string order and return it
        return hexBuilder.reverse().toString();
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    string toHex(int num) {
        // Check base condition: if num is 0, return "0"
        if (num == 0) {
            return "0";
        }

        // Initialize the output string
        string hexString = "";

        // Iterate over each hex digit from the most significant to least significant
        for (int i = 7; i >= 0; --i) {
            // Extract current hex digit from the number
            int hexDigit = (num >> (4 * i)) & 0xf;

            // Skip leading zeros
            if (hexString.size() > 0 || hexDigit != 0) {
                // Convert the current digit to its hex char representation
                char hexChar = hexDigit < 10 ? (char)(hexDigit + '0') : (char)(hexDigit - 10 + 'a');
                // Append the hex character to the hex string
                hexString += hexChar;
            }
        }

        // Return the complete hex string
        return hexString;
    }
};
```

Typescript Solution

```typescript
function toHex(num: number): string {
    // Check base condition: if num is 0, return "0"
    if (num === 0) {
        return "0";
    }

    // Initialize the output string
    let hexString = "";

    // Iterate over each hex digit from the most significant to least significant
    for (let i = 7; i >= 0; --i) {
        // Extract current hex digit from the number
        const hexDigit = (num >> (4 * i)) & 0xf;

        // Skip leading zeros
        if (hexString.length > 0 || hexDigit !== 0) {
            // Convert the current digit to its hex char representation
            const hexChar = hexDigit < 10 ? String.fromCharCode(hexDigit + '0'.charCodeAt(0)) :
                                            String.fromCharCode(hexDigit - 10 + 'a'.charCodeAt(0));
            // Append the hex character to the hex string
            hexString += hexChar;
        }
    }

    // Return the complete hex string
    return hexString;
}
```

Time and Space Complexity

Time Complexity

The given code performs a fixed number of iterations regardless of the size of the input `num` since it iterates 8 times corresponding to each hex digit of a 32-bit integer. The shift operation `>>` and the bitwise AND operation `&` take constant time, and the append operation for a Python list also takes constant time on average. Therefore, the time complexity is $O(1)$.

Space Complexity

The space complexity is determined by the additional space used by the algorithm. Here, the space is used to store the hex characters in the `chars` string and the `s` list which is used to build the final hex string. The `chars` string is of a fixed size, and the `s` list grows to a maximum length of 8 since a 32-bit integer can have at most 8 hex digits. Therefore, the space required does not depend on the input number, and the space complexity is also $O(1)$.