LeetCode Problem: Digit Product

🔰 Problem Statement

Difficulty: Easy

Tags: Math, Number Theory

Companies: Google, Amazon, Microsoft

💬 Problem Description

Given a non-negative integer `n`, return the **product of all its digits**.

If any digit is `0`, the final product will be `0`.

🔢 Input/Output Specification

Input:

- A single integer `n` where `0 ≤ n ≤ 10^9`

Output:

- Return an integer representing the product of all digits in `n`

📜 Examples

Example 1:

```
Input: n = 123
Output: 6
Explanation: 1 × 2 × 3 = 6
```

Example 2:

```
Input: n = 405
Output: 0
Explanation: 4 × 0 × 5 = 0 (contains zero digit)
```

Example 3:

```
Input: n = 7
Output: 7
Explanation: Single digit returns itself
```

Example 4:

```
Input: n = 0
Output: 0
Explanation: Special case - zero returns zero
```

🔒 Constraints

- `0 ≤ n ≤ 1,000,000,000`
- The input `n` will not contain any non-digit characters
- If `n == 0`, return `0`

💡 Algorithm Explanation

Approach: Digit Extraction

1. **Edge Case**: If `n = 0`, return `0` immediately
2. **Initialize**: Set `product = 1`
3. **Extract Digits**: Use modulo operation (`n % 10`) to get the last digit
4. **Multiply**: Multiply the current product by the extracted digit
5. **Remove Digit**: Integer divide by 10 (`n /= 10`) to remove the last digit
6. **Repeat**: Continue until `n` becomes `0`
7. **Return**: The accumulated product

Time Complexity: O(log n)

- We process each digit once, and the number of digits is logarithmic to the input value

Space Complexity: O(1)

- Only using a constant amount of extra space

🛠️ Multi-Language Implementations

C++ Solution

cpp

```cpp
class Solution {
public:
    int digitProduct(int n) {
        // Edge case: if n is 0, return 0
        if (n == 0) return 0;

        int product = 1;
        while (n > 0) {
            int digit = n % 10;  // Extract last digit
            product *= digit;    // Multiply to product
            n /= 10;             // Remove last digit
        }
        return product;
    }
};

// Alternative recursive approach
class SolutionRecursive {
public:
    int digitProduct(int n) {
        if (n == 0) return 0;
        if (n < 10) return n;
        return (n % 10) * digitProduct(n / 10);
    }
};
```

C Solution

c

```c
#include <stdio.h>

int digitProduct(int n) {
    // Edge case: if n is 0, return 0
    if (n == 0) return 0;

    int product = 1;
    while (n > 0) {
        int digit = n % 10;  // Extract last digit
        product *= digit;    // Multiply to product
        n /= 10;             // Remove last digit
    }
    return product;
}

// Test function
int main() {
    printf("digitProduct(123) = %d\n", digitProduct(123));  // Output: 6
    printf("digitProduct(405) = %d\n", digitProduct(405));  // Output: 0
    printf("digitProduct(7) = %d\n", digitProduct(7));      // Output: 7
    printf("digitProduct(0) = %d\n", digitProduct(0));      // Output: 0
    return 0;
}
```

Java Solution

java

```java
public class Solution {
    public int digitProduct(int n) {
        // Edge case: if n is 0, return 0
        if (n == 0) return 0;

        int product = 1;
        while (n > 0) {
            int digit = n % 10;  // Extract last digit
            product *= digit;    // Multiply to product
            n /= 10;             // Remove last digit
        }
        return product;
    }

    // Alternative string-based approach
    public int digitProductString(int n) {
        if (n == 0) return 0;

        String numStr = String.valueOf(n);
        int product = 1;

        for (char c : numStr.toCharArray()) {
            int digit = Character.getNumericValue(c);
            product *= digit;
        }
        return product;
    }

    // Test method
    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println("digitProduct(123) = " + sol.digitProduct(123)); // 6
        System.out.println("digitProduct(405) = " + sol.digitProduct(405)); // 0
        System.out.println("digitProduct(7) = " + sol.digitProduct(7));     // 7
        System.out.println("digitProduct(0) = " + sol.digitProduct(0));     // 0
    }
}
```

C# Solution

csharp

```csharp
using System;

public class Solution {
    public int DigitProduct(int n) {
        // Edge case: if n is 0, return 0
        if (n == 0) return 0;

        int product = 1;
        while (n > 0) {
            int digit = n % 10;  // Extract last digit
            product *= digit;    // Multiply to product
            n /= 10;             // Remove last digit
        }
        return product;
    }

    // LINQ-based approach (more C#-idiomatic)
    public int DigitProductLinq(int n) {
        if (n == 0) return 0;

        return n.ToString()
                .Select(c => int.Parse(c.ToString()))
                .Aggregate(1, (acc, digit) => acc * digit);
    }
}

// Program class for testing
public class Program {
    public static void Main() {
        Solution sol = new Solution();
        Console.WriteLine($"DigitProduct(123) = {sol.DigitProduct(123)}"); // 6
        Console.WriteLine($"DigitProduct(405) = {sol.DigitProduct(405)}"); // 0
        Console.WriteLine($"DigitProduct(7) = {sol.DigitProduct(7)}");     // 7
        Console.WriteLine($"DigitProduct(0) = {sol.DigitProduct(0)}");     // 0
    }
}
```

Rust Solution

rust

```rust
impl Solution {
    pub fn digit_product(n: i32) -> i32 {
        // Edge case: if n is 0, return 0
        if n == 0 {
            return 0;
        }

        let mut num = n;
        let mut product = 1;

        while num > 0 {
            let digit = num % 10;  // Extract last digit
            product *= digit;      // Multiply to product
            num /= 10;             // Remove last digit
        }
        product
    }

    // Functional approach using iterators
    pub fn digit_product_functional(n: i32) -> i32 {
        if n == 0 {
            return 0;
        }

        n.to_string()
            .chars()
            .map(|c| c.to_digit(10).unwrap() as i32)
            .product()
    }

    // Recursive approach
    pub fn digit_product_recursive(n: i32) -> i32 {
        match n {
            0 => 0,
            1..=9 => n,
            _ => (n % 10) * Self::digit_product_recursive(n / 10),
        }
    }
}

// Test module
#[cfg(test)]
mod tests {
```

```rust
    use super::*;

    #[test]
    fn test_digit_product() {
        assert_eq!(Solution::digit_product(123), 6);
        assert_eq!(Solution::digit_product(405), 0);
        assert_eq!(Solution::digit_product(7), 7);
        assert_eq!(Solution::digit_product(0), 0);
    }
}

// Main function for standalone testing
fn main() {
    println!("digit_product(123) = {}", Solution::digit_product(123)); // 6
    println!("digit_product(405) = {}", Solution::digit_product(405)); // 0
    println!("digit_product(7) = {}", Solution::digit_product(7));     // 7
    println!("digit_product(0) = {}", Solution::digit_product(0));     // 0
}
```

🧪 Test Cases

Input	Expected Output	Explanation
0	0	Special case - zero
1	1	Single digit
12	2	1 × 2 = 2
123	6	1 × 2 × 3 = 6
405	0	Contains zero digit
999	729	9 × 9 × 9 = 729
1000000000	0	Contains zeros

🎯 Key Takeaways

1. **Edge Case Handling**: Always check for `n = 0` first
2. **Digit Extraction**: Use modulo (`%`) and integer division (`/`) operations
3. **Zero Detection**: Any zero digit makes the entire product zero
4. **Multiple Approaches**: Iterative, recursive, and string-based solutions all work
5. **Language Features**: Each language offers unique approaches (LINQ in C#, iterators in Rust)

🔍 Follow-up Questions

1. What if we need to handle negative numbers?
2. How would you modify this to find the sum of digits instead?
3. Can you solve this without converting to string?
4. What's the maximum possible output for the given constraints?

📊 Complexity Analysis Summary

Approach	Time Complexity	Space Complexity
Iterative	O(log n)	O(1)
Recursive	O(log n)	O(log n)
String-based	O(log n)	O(log n)

Note: log n represents the number of digits in the input number.

This document provides comprehensive solutions across multiple programming languages for the Digit Product problem, following LeetCode formatting standards.