7. Basic Math for DSA Euclidean Algorithm & Key Concepts

Table of Contents

Digit Operations

1. Extract Digits from a Number

Concept: Use modulo 10 (`% 10`) to get the last digit, then divide by 10 (`/ 10`) to remove it.
Time Complexity: $O(\log_{10}(n))$

```C++
void extractDigits(int n) {
    while (n > 0) {
        int lastDigit = n % 10;
        cout << lastDigit << " ";
        n = n / 10;
    }
}
// Input: 7789 → Output: 9 8 7 7
```

2. Count Digits

Method 1: Loop until `n` becomes 0
Method 2: Use `log10(n) + 1`

```cpp
int countDigits(int n) {
    int count = 0;
    while (n > 0) {
        count++;
        n /= 10;
    }
    return count;
}

// Using logarithms (doesn't work for n=0)
int countDigitsLog(int n) {
    return floor(log10(n) + 1);
}
```

Number Reversal & Palindromes

Reverse a Number

Idea: Rebuild number by `reverse = reverse * 10 + lastDigit`

```cpp
int reverseNumber(int n) {
    int reversed = 0;
    while (n > 0) {
        reversed = reversed * 10 + (n % 10);
        n /= 10;
    }
    return reversed;
}
// Input: 1234 → Output: 4321
```

Check Palindrome

Rule: Reverse == Original

```cpp
bool isPalindrome(int n) {
    int original = n;
    int reversed = 0;
    while (n > 0) {
        reversed = reversed * 10 + (n % 10);
        n /= 10;
    }
    return reversed == original;
}
// Input: 121 → Output: true
```

Armstrong Numbers

Definition: Sum of digits^digits_count equals the number.
Example: $371 = 3^3 + 7^3 + 1^3$

```cpp
bool isArmstrong(int n) {
    int original = n;
    int sum = 0;
    int digits = countDigits(n);

    while (n > 0) {
        int digit = n % 10;
        sum += pow(digit, digits);
        n /= 10;
    }
    return sum == original;
}
// Input: 371 → Output: true
```

Divisors & Factors

Find All Divisors

Optimization: Loop up to √n
Time Complexity: O(√n)

```cpp
vector<int> findDivisors(int n) {
    vector<int> divisors;
    for (int i = 1; i <= sqrt(n); i++) {
        if (n % i == 0) {
            divisors.push_back(i);
            if (i != n/i) divisors.push_back(n/i);
        }
    }
    sort(divisors.begin(), divisors.end());
    return divisors;
}
// Input: 36 → Output: 1 2 3 4 6 9 12 18 36
```

Prime Numbers

Check Prime

Optimization: Check divisibility up to √n
Time Complexity: O(√n)

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return false;
    }
    return true;
}
// Input: 11 → Output: true
```

GCD & Euclidean Algorithm

1. Brute Force Approach

```cpp
int gcdBrute(int a, int b) {
    int gcd = 1;
    for (int i = 1; i <= min(a, b); i++) {
        if (a % i == 0 && b % i == 0) gcd = i;
    }
    return gcd;
}
// GCD(20, 15) = 5
```

2. Euclidean Algorithm (Optimal)

Time Complexity: O(log(min(a, b)))

```cpp
int gcdEuclidean(int a, int b) {
    while (a > 0 && b > 0) {
        if (a > b) a %= b;
        else b %= a;
    }
    return a == 0 ? b : a;
}

// Recursive Implementation
int gcdRecursive(int a, int b) {
    return b == 0 ? a : gcdRecursive(b, a % b);
}
```

Key Notes

1. **Digit Extraction**: Use `n % 10` and `n / 10` for $O(\log_{10}(n))$ operations.

2. **Palindrome Check**: Always preserve the original number for comparison.
3. **Divisors**:
   - Check up to √n to reduce time complexity from O(n) → O(√n).
   - Store factors in sorted order for consistency.
4. **Prime Optimization**: No need to check beyond √n for factors.
5. **GCD**:
   - Euclidean Algorithm is **100x faster** than brute force for large numbers.
   - `GCD(a, b) = GCD(b, a % b)` until one number becomes 0.
6. **Edge Cases**:
   - Handle `n = 0` in digit count functions.
   - Return 1 as default GCD for prime number pairs.