Number Theory Cheatsheet: A Comprehensive Guide

Introduction to Number Theory

Number theory is a branch of pure mathematics devoted primarily to the study of integers and integer-valued functions. It's a fascinating field that explores the properties and relationships of numbers, often revealing surprising patterns and deep insights. From ancient times, mathematicians have been captivated by the mysteries of numbers, leading to the development of concepts that are fundamental to modern cryptography, computer science, and many other areas.

This cheatsheet will guide you through some foundational concepts in number theory, including primes, divisors, prime factorization, and the Sieve of Eratosthenes. We'll provide clear explanations, practical C++ code examples, and visual aids to help you grasp these essential ideas.

Primes: The Building Blocks of Numbers

At the heart of number theory are **prime numbers**. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Think of them as the fundamental building blocks for all other natural numbers through multiplication. For example, the number 7 is prime because its only divisors are 1 and 7. On the other hand, 6 is not prime because it can be divided by 1, 2, 3, and 6.

Understanding prime numbers is crucial because of the **Fundamental Theorem of Arithmetic**, which states that every integer greater than 1 is either a prime number itself or can be represented as a product of prime numbers, and this representation is unique (ignoring the order of the factors). This theorem highlights the importance of primes as the 'atoms' of the number system.

How to Check if a Number is Prime: The Naive Approach

A straightforward way to determine if a number `n` is prime is to check every possible integer from 2 up to `n-1`. If any of these integers divide `n` evenly (i.e., leave no remainder), then `n` is not prime. If no such divisor is found after checking all these numbers, then `n` must be prime.

Let's illustrate this with an example. To check if 10 is prime, we would try dividing it by 2, 3, 4, 5, 6, 7, 8, and 9. Since 10 is divisible by 2 (10 / 2 = 5), we immediately know that 10 is not prime.

While simple to understand, this method can be computationally expensive, especially for very large numbers. The time it takes to execute grows linearly with the size of `n`, which is denoted as O(n) time complexity. This means if `n` doubles, the time taken roughly doubles.

Here's the C++ code for this naive approach:

```
bool isPrimeNaive(long long n) {
    // Numbers less than or equal to 1 are not prime.
    if (n <= 1) {
        return false;
    }
    // Iterate from 2 up to n-1
    for (long long i = 2; i < n; i++) {
        // If n is divisible by i, then n is not prime
        if (n % i == 0) {
            return false;
        }
    }
    // If no divisors are found, n is prime
    return true;
}
```

Optimizing Prime Checking: The Square Root Rule

We can significantly improve the efficiency of our prime-checking algorithm by applying a clever mathematical observation known as the **Square Root Rule**. This rule states that if a number `n` has a divisor greater than its square root, it must also have a divisor smaller than its square root. In other words, if `n` is a composite number (not prime), it must have at least one divisor `d` such that `1 < d <= sqrt(n)`.

Consider a number `n` that can be factored into two integers, `a` and `b`, such that `n = a * b`. For example, if `n = 36`, we could have `a = 4` and `b = 9`. Notice that

`sqrt(36) = 6`. In this case, `a = 4` is less than `sqrt(36)`, and `b = 9` is greater than `sqrt(36)`. If we find `a`, we can easily find `b` by `n/a`. The key insight is that we only need to search for divisors up to `sqrt(n)`. If we don't find any divisors up to `sqrt(n)`, then `n` cannot have any divisors greater than `sqrt(n)` either, and thus `n` must be prime.

This optimization drastically reduces the number of checks required. Instead of checking up to `n-1`, we only check up to `sqrt(n)`. This improves the time complexity to O(sqrt(n)), which is much faster for large values of `n`.

Here's the optimized C++ code:

```
bool isPrimeOptimized(long long n) {
    // Numbers less than or equal to 1 are not prime.
    if (n <= 1) {
        return false;
    }
    // Iterate from 2 up to the square root of n
    for (long long i = 2; i * i <= n; i++) {
        // If n is divisible by i, then n is not prime
        if (n % i == 0) {
            return false;
        }
    }
    // If no divisors are found up to sqrt(n), n is prime
    return true;
}
```

This optimized version is the standard approach for checking primality for individual numbers in competitive programming and many practical applications.

Divisors: Finding All Factors of a Number

A **divisor** of an integer `n` is an integer `d` such that `n` can be divided by `d` with no remainder. In simpler terms, if `n` is perfectly divisible by `d`, then `d` is a divisor of `n`. For example, the divisors of 12 are 1, 2, 3, 4, 6, and 12.

Just like with prime checking, we can leverage the **Square Root Rule** to efficiently find all divisors of a number. If `d` is a divisor of `n`, then `n/d` is also a divisor of `n`. This means that divisors always come in pairs. For example, if 2 is a divisor of 12, then 12/2 = 6 is also a divisor. One number in the pair will always be less than or equal to `sqrt(n)`, and the other will be greater than or equal to `sqrt(n)` (unless `n` is a perfect square, in which case `d = n/d = sqrt(n)`).

Therefore, to find all divisors, we only need to iterate from 1 up to `sqrt(n)`. For each `i` in this range, if `i` divides `n` evenly, then both `i` and `n/i` are divisors. We must be careful to add `n/i` only if `i` is not equal to `n/i` (to avoid adding the square root twice when `n` is a perfect square).

This approach allows us to find all divisors in O(sqrt(n)) time complexity.

Here's the C++ code to find all divisors of a number:

```cpp
#include <vector>
#include <algorithm> // For std::sort

std::vector<long long> getDivisors(long long n) {
    std::vector<long long> divs;
    // Iterate from 1 up to the square root of n
    for (long long i = 1; i * i <= n; i++) {
        // If i divides n evenly
        if (n % i == 0) {
            divs.push_back(i); // Add i as a divisor
            // If i is not equal to n/i, then n/i is also a distinct divisor
            if (i != n / i) {
                divs.push_back(n / i);
            }
        }
    }
    // Sort the divisors for a clean output (optional, but good practice)
    std::sort(divs.begin(), divs.end());
    return divs;
}
```

Example: Finding divisors of 36

1. `i = 1`: `36 % 1 == 0`. Add 1 and 36/1 = 36. Divisors: `{1, 36}`
2. `i = 2`: `36 % 2 == 0`. Add 2 and 36/2 = 18. Divisors: `{1, 2, 18, 36}`
3. `i = 3`: `36 % 3 == 0`. Add 3 and 36/3 = 12. Divisors: `{1, 2, 3, 12, 18, 36}`
4. `i = 4`: `36 % 4 == 0`. Add 4 and 36/4 = 9. Divisors: `{1, 2, 3, 4, 9, 12, 18, 36}`
5. `i = 5`: `36 % 5 != 0`. Skip.
6. `i = 6`: `36 % 6 == 0`. Add 6. Since `i == n/i` (6 == 36/6), we only add it once. Divisors: `{1, 2, 3, 4, 6, 9, 12, 18, 36}`

The loop stops because `7 * 7 = 49`, which is greater than 36. The sorted list of divisors for 36 is `{1, 2, 3, 4, 6, 9, 12, 18, 36}`.

Prime Factors: Decomposing Numbers into Primes

Prime factorization is the process of breaking down a composite number into its prime number components. As mentioned earlier, the **Fundamental Theorem of Arithmetic** guarantees that every integer greater than 1 can be uniquely expressed as a product of prime numbers. This unique set of primes is called the prime factorization of the number.

For example, the prime factorization of 12 is `2 * 2 * 3`, which can be written as `2^2 * 3^1`. The prime factors are 2 and 3. This decomposition is fundamental in many areas of number theory and cryptography.

How to Find Prime Factors

We can find the prime factors of a number `n` by repeatedly dividing it by the smallest possible prime numbers. We start with 2 and keep dividing `n` by 2 as long as it is divisible. Then we move to the next prime, 3, and do the same. We continue this process with successive primes.

However, we can make this more efficient by adapting the **Square Root Rule**. We only need to test for prime divisors up to `sqrt(n)`. For each prime `p` we find, we count how many times it divides `n` and then update `n` by dividing it by `p` that many times. If, after dividing by all primes up to `sqrt(n)`, the remaining value of `n` is greater than 1, then this remaining value must also be a prime number.

Let's walk through an example with `n = 84`:

1. Start with `i = 2`. `84` is divisible by `2`. `84 / 2 = 42`. Count for 2 is 1. `n` is now 42.
2. `42` is still divisible by `2`. `42 / 2 = 21`. Count for 2 is 2. `n` is now 21.
3. `21` is not divisible by `2`. Move to `i = 3`.
4. `21` is divisible by `3`. `21 / 3 = 7`. Count for 3 is 1. `n` is now 7.
5. `7` is not divisible by `3`. Move to `i = 4` (but we only need to check primes, so we can optimize this). Let's continue with `i=4` for simplicity of the example. `7` is not divisible by `4`.
6. Move to `i = 5`. `7` is not divisible by `5`.

7. Move to `i = 6`. `7` is not divisible by `6`.
8. Move to `i = 7`. `7` is divisible by `7`. `7 / 7 = 1`. Count for 7 is 1. `n` is now 1.
9. The loop stops. The prime factorization of 84 is `2^2 * 3^1 * 7^1`.

This process can be implemented efficiently in C++ with a time complexity of O(sqrt(n)).

Here's the C++ code for prime factorization:

```
#include <vector>
#include <utility> // For std::pair

std::vector<std::pair<long long, int>> factorize(long long n) {
    std::vector<std::pair<long long, int>> factors;
    // Iterate from 2 up to the square root of n
    for (long long i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            int count = 0;
            // While i divides n, increment the count and divide n by i
            while (n % i == 0) {
                n /= i;
                count++;
            }
            factors.push_back({i, count});
        }
    }
    // If n is still greater than 1, then the remaining n is a prime factor
    if (n > 1) {
        factors.push_back({n, 1});
    }
    return factors;
}
```

This function returns a vector of pairs, where each pair contains a prime factor and its corresponding exponent.

Sieve of Eratosthenes: Finding All Primes Up to a Limit

The **Sieve of Eratosthenes** is an ancient and highly efficient algorithm for finding all prime numbers up to a specified integer limit. It was devised by the Greek mathematician Eratosthenes of Cyrene (c. 276 – c. 195/194 BC). The core idea behind the sieve is elegantly simple: start with a list of consecutive integers and progressively mark (or

cross out) the multiples of each prime number, starting with the first prime number, 2. The numbers that remain unmarked are the primes.

How the Sieve of Eratosthenes Works

Let's imagine we want to find all prime numbers up to 30:

1. **Create a list:** Write down all integers from 2 to 30: `2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30`
2. **Start with the first prime (2):**
    - Circle 2 (it's prime).
    - Cross out all multiples of 2 (4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30).
3. **Move to the next unmarked number (3):**
    - Circle 3 (it's prime).
    - Cross out all multiples of 3 (6, 9, 12, 15, 18, 21, 24, 27, 30). (Some might already be crossed out).
4. **Move to the next unmarked number (5):**
    - Circle 5 (it's prime).
    - Cross out all multiples of 5 (10, 15, 20, 25, 30).
5. **Continue this process:** The next unmarked number is 7. Circle 7 and cross out its multiples (14, 21, 28). We only need to continue this process up to the square root of our limit (sqrt(30) is approximately 5.47). So, after 5, we would check 7, but since 7*7 > 30, we don't need to cross out any more multiples. All remaining unmarked numbers are prime.

The numbers left unmarked are: `2, 3, 5, 7, 11, 13, 17, 19, 23, 29`.

Efficiency of the Sieve

The Sieve of Eratosthenes is highly efficient. Its time complexity is approximately O(n log (log n)), which is very close to linear time. This makes it an excellent choice for generating a list of primes up to a large number, especially when compared to checking each number individually for primality.

C++ Implementation of Sieve of Eratosthenes

```cpp
#include <vector>
#include <iostream>

std::vector<bool> sieve(int n) {
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    std::vector<bool> prime(n + 1, true);

    // Start from p = 2, the first prime number
    for (int p = 2; p * p <= n; p++) {
        // If prime[p] is still true, then it is a prime
        if (prime[p] == true) {
            // Update all multiples of p greater than or
            // equal to the square of it. Multiples less
            // than p*p are already been marked.
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Collect all prime numbers
    // (Optional: you can also return the boolean vector directly)
    // std::vector<int> prime_numbers;
    // for (int p = 2; p <= n; p++) {
    //     if (prime[p]) {
    //         prime_numbers.push_back(p);
    //     }
    // }
    // return prime_numbers;

    return prime; // Return the boolean vector indicating primality
}

/*
Example Usage:
int limit = 30;
std::vector<bool> is_prime = sieve(limit);
std::cout << "Prime numbers up to " << limit << ":\n";
for (int p = 2; p <= limit; p++) {
    if (is_prime[p]) {
        std::cout << p << " ";
    }
}
std::cout << std::endl;
*/
```

Sieve of Eratosthenes Diagram

This diagram visually represents the process of crossing out multiples, leaving only the prime numbers.

Resources

- **Sieve of Eratosthenes Diagram:** Enjoy Teaching with Brenda Kovich
- **General Number Theory Concepts:** Khan Academy - Number Theory
- **C++ Programming Language:** cppreference.com

- **Algorithms and Data Structures:** GeeksforGeeks