Number Theory

Tawhid Bin Omar

Number Theory Fundamentals

Number theory studies properties of integers and is fundamental to competitive programming, especially for modular arithmetic, primes, and combinatorics.

1 GCD and LCM

Greatest Common Divisor

Theorem

Euclidean Algorithm: $\gcd(a, b) = \gcd(b, a \bmod b)$

```cpp
// Recursive GCD
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// Iterative GCD
int gcd_iterative(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Using C++17 built-in
#include <numeric>
int g = gcd(a, b);
```

Complexity

Time: $O(\log \min(a, b))$

Least Common Multiple

```cpp
// LCM using GCD
long long lcm(long long a, long long b) {
    return (a / gcd(a, b)) * b;
}

// Using C++17 built-in
#include <numeric>
long long l = lcm(a, b);
```

Key Concept

Important: Always compute as $(a/\gcd(a, b)) \times b$ to avoid overflow, not $a \times b/\gcd(a, b)$.

Extended Euclidean Algorithm

Find integers x, y such that $ax + by = \gcd(a, b)$.

```cpp
// Returns gcd and fills x, y
int extendedGCD(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    int x1, y1;
    int g = extendedGCD(b, a % b, x1, y1);

    x = y1;
    y = x1 - (a / b) * y1;

    return g;
}

// Usage
int x, y;
int g = extendedGCD(a, b, x, y);
// Now: a*x + b*y = g
```

2 Prime Numbers

Primality Test

```cpp
// O(sqrt(n)) primality test
bool isPrime(long long n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0)
        return false;

    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }
    return true;
}
```

Sieve of Eratosthenes

Complexity

Time: $O(n \log \log n)$ | **Space:** $O(n)$

```cpp
// Find all primes up to n
vector<bool> sieve(int n) {
    vector<bool> isPrime(n + 1, true);
    isPrime[0] = isPrime[1] = false;

    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }
    return isPrime;
}

// Get list of primes
vector<int> getPrimes(int n) {
    vector<bool> isPrime = sieve(n);
    vector<int> primes;

    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
        }
    }
    return primes;
}
```

Segmented Sieve

For finding primes in range $[L, R]$ where R is large.

```cpp
// Primes in [L, R], R - L <= 10^6
vector<bool> segmentedSieve(long long L,
    long long R) {
    long long lim = sqrt(R);
    vector<bool> mark(lim + 1, true);
    vector<long long> primes;

    // Find primes up to sqrt(R)
    for (long long i = 2; i <= lim; i++) {
        if (mark[i]) {
            primes.push_back(i);
            for (long long j = i * i; j <= lim;
                j += i)
                mark[j] = false;
        }
    }

    // Sieve [L, R]
    vector<bool> isPrime(R - L + 1, true);

    for (long long p : primes) {
        long long start = max(p * p,
            (L + p - 1) / p * p);

        for (long long j = start; j <= R; j += p) {
            isPrime[j - L] = false;
        }
    }

    if (L == 1) isPrime[0] = false;

    return isPrime;
}
```

Prime Factorization

```cpp
// O(sqrt(n)) factorization
vector<pair<int,int>> primeFactors(int n) {
    vector<pair<int,int>> factors;

    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            int count = 0;
            while (n % i == 0) {
                n /= i;
                count++;
            }
            factors.push_back({i, count});
        }
    }

    if (n > 1) {
        factors.push_back({n, 1});
    }

    return factors;
}

// Using precomputed smallest prime factor
vector<int> spf(MAXN);

void computeSPF(int n) {
    for (int i = 1; i <= n; i++)
        spf[i] = i;

    for (int i = 2; i * i <= n; i++) {
        if (spf[i] == i) { // i is prime
            for (int j = i * i; j <= n; j += i) {
                if (spf[j] == j)
                    spf[j] = i;
            }
        }
    }
}

vector<int> getFactors(int n) {
    vector<int> factors;
    while (n > 1) {
        factors.push_back(spf[n]);
        n /= spf[n];
    }
    return factors;
}
```

Count Divisors

```cpp
// Count divisors in O(sqrt(n))
int countDivisors(int n) {
    int count = 0;
    for (int i = 1; i * i <= n; i++) {
        if (n % i == 0) {
            count++;
            if (i * i != n)
                count++;
        }
    }
    return count;
}

// Count divisors using prime factorization
// If n = p1^a1 * p2^a2 * ... * pk^ak
// divisors = (a1+1) * (a2+1) * ... * (ak+1)
int countDivisorsPF(int n) {
    auto factors = primeFactors(n);
    int count = 1;
    for (auto [p, exp] : factors) {
        count *= (exp + 1);
    }
    return count;
}

// Sum of divisors
long long sumDivisors(int n) {
    auto factors = primeFactors(n);
    long long sum = 1;

    for (auto [p, a] : factors) {
        // (p^(a+1) - 1) / (p - 1)
        long long term = (pow(p, a + 1) - 1) / (p -
            1);
```

```
        sum *= term;
    }
    return sum;
}
```

3 Modular Arithmetic

Basic Operations

```
const int MOD = 1e9 + 7;

// Modular addition
int addMod(int a, int b) {
    return ((a % MOD) + (b % MOD)) % MOD;
}

// Modular subtraction
int subMod(int a, int b) {
    return ((a % MOD) - (b % MOD) + MOD) % MOD;
}

// Modular multiplication
long long mulMod(long long a, long long b) {
    return ((a % MOD) * (b % MOD)) % MOD;
}
```

Modular Exponentiation

Complexity

Time: $O(\log n)$

```
// Compute (a^b) % MOD
long long power(long long a, long long b,
    long long mod = MOD) {
    long long res = 1;
    a %= mod;

    while (b > 0) {
        if (b & 1)
            res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}
```

Modular Inverse

Theorem

Fermat's Little Theorem: If p is prime and $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \pmod{p}$.
Therefore: $a^{-1} \equiv a^{p-2} \pmod{p}$

```
// Modular inverse (MOD must be prime)
long long modInv(long long a, long long mod = MOD) {
    return power(a, mod - 2, mod);
}

// Modular division
long long divMod(long long a, long long b) {
    return (a * modInv(b)) % MOD;
}

// Using Extended GCD (works for any coprime)
long long modInvGCD(long long a, long long mod) {
    int x, y;
    int g = extendedGCD(a, mod, x, y);
    if (g != 1) return -1; // inverse doesn't exist

    return (x % mod + mod) % mod;
}
```

Modular Factorial

```cpp
const int MAXN = 1e6 + 5;
long long fact[MAXN], invFact[MAXN];

// Precompute factorials
void precomputeFactorials(int n) {
    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % MOD;
    }

    invFact[n] = modInv(fact[n]);
    for (int i = n - 1; i >= 0; i--) {
        invFact[i] = (invFact[i+1] * (i+1)) % MOD;
    }
}

// nCr modulo MOD
long long nCr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return (fact[n] * invFact[r] % MOD) *
           invFact[n - r] % MOD;
}

// nPr modulo MOD
long long nPr(int n, int r) {
    if (r < 0 || r > n) return 0;
    return (fact[n] * invFact[n - r]) % MOD;
}
```

4 Chinese Remainder Theorem

Theorem

CRT: Given:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

If m_i are pairwise coprime, unique solution exists modulo $M = \prod m_i$.

```cpp
// Solve x ≡a (mod m), x ≡b (mod n)
// where gcd(m, n) = 1
long long crt2(long long a, long long m,
               long long b, long long n) {
    int x, y;
    extendedGCD(m, n, x, y);

    long long M = m * n;
    long long result = (a * n % M * y % M +
                        b * m % M * x % M) % M;
    return (result + M) % M;
}

// General CRT for multiple equations
long long crt(vector<long long>& a,
              vector<long long>& m) {
    int n = a.size();
    long long result = a[0];
    long long lcm_val = m[0];

    for (int i = 1; i < n; i++) {
        result = crt2(result, lcm_val, a[i], m[i]);
        lcm_val = lcm(lcm_val, m[i]);
    }

    return result;
}
```

5 Euler's Totient Function

Key Concept

$\phi(n)$ = count of integers in $[1, n]$ coprime to n.

Theorem

If $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, then:

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

```cpp
// Compute phi(n)
int phi(int n) {
    int result = n;

    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }

    if (n > 1)
        result -= result / n;

    return result;
}

// Sieve for phi values up to n
vector<int> phiSieve(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) { // i is prime
            for (int j = i; j <= n; j += i) {
                phi[j] -= phi[j] / i;
            }
        }
    }
    return phi;
}
```

Theorem

Euler's Theorem: If $\gcd(a, n) = 1$, then:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

6 Linear Diophantine Equations

Problem

Find integer solutions to $ax + by = c$.

Key Concept

Solutions exist $\iff \gcd(a,b) \mid c$.
If (x_0, y_0) is one solution, then all solutions are:

$$x = x_0 + k \cdot \frac{b}{\gcd(a,b)}, \quad y = y_0 - k \cdot \frac{a}{\gcd(a,b)}$$

for integer k.

Solution

```
// Find one solution to ax + by = c
bool solveDiophantine(int a, int b, int c,
                      int& x, int& y) {
   int g = extendedGCD(abs(a), abs(b), x, y);

   if (c % g != 0)
      return false; // no solution

   x *= c / g;
   y *= c / g;

   if (a < 0) x = -x;
   if (b < 0) y = -y;

   return true;
}

// Count solutions in range [minX, maxX]
int countSolutions(int a, int b, int c,
                   int minX, int maxX) {
   int x, y;
   if (!solveDiophantine(a, b, c, x, y))
      return 0;

   int g = gcd(a, b);
   int dx = b / g;

   // Find valid k range
   // x + k*dx in [minX, maxX]
   int k1 = (minX - x + dx - 1) / dx;
   int k2 = (maxX - x) / dx;

   if (k1 > k2) return 0;
   return k2 - k1 + 1;
}
```

7 Combinatorics

Binomial Coefficients

```
// Pascal's Triangle - compute all C(n,k)
vector<vector<long long>> pascalTriangle(int n) {
   vector<vector<long long>> C(n + 1,
      vector<long long>(n + 1, 0));

   for (int i = 0; i <= n; i++) {
      C[i][0] = 1;
      for (int j = 1; j <= i; j++) {
         C[i][j] = C[i-1][j-1] + C[i-1][j];
      }
   }
   return C;
}

// C(n, k) with overflow prevention
long long nCr_safe(int n, int k) {
   if (k > n - k) k = n - k;

   long long result = 1;
   for (int i = 0; i < k; i++) {
      result *= (n - i);
      result /= (i + 1);
   }
   return result;
}
```

Catalan Numbers

Key Concept

$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\cdot n!}$
Applications: Parentheses, binary trees, paths

```
// Catalan numbers modulo MOD
long long catalan(int n) {
   return (nCr(2*n, n) * modInv(n + 1)) % MOD;
}

// DP approach
vector<long long> catalanDP(int n) {
   vector<long long> cat(n + 1, 0);
   cat[0] = cat[1] = 1;

   for (int i = 2; i <= n; i++) {
      for (int j = 0; j < i; j++) {
         cat[i] = (cat[i] +
            cat[j] * cat[i-1-j]) % MOD;
      }
   }
   return cat;
}
```

Derangements

Count of permutations where no element is in its original position.

```
// D(n) = (n-1) * (D(n-1) + D(n-2))
long long derangements(int n) {
   if (n == 0) return 1;
   if (n == 1) return 0;

   long long d0 = 1, d1 = 0;
   for (int i = 2; i <= n; i++) {
      long long d2 = (i - 1) * (d0 + d1) % MOD;
      d0 = d1;
      d1 = d2;
   }
   return d1;
}
```

8 Matrix Exponentiation

For computing recurrences in $O(\log n)$ time.

```
typedef vector<vector<long long>> Matrix;

Matrix multiply(Matrix& A, Matrix& B, int n) {
   Matrix C(n, vector<long long>(n, 0));
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         for (int k = 0; k < n; k++) {
            C[i][j] = (C[i][j] +
               A[i][k] * B[k][j]) % MOD;
         }
      }
   }
   return C;
}

Matrix matrixPower(Matrix A, long long p) {
   int n = A.size();
```

```
    Matrix result(n, vector<long long>(n, 0));

    // Identity matrix
    for (int i = 0; i < n; i++)
        result[i][i] = 1;

    while (p > 0) {
        if (p & 1)
            result = multiply(result, A, n);
        A = multiply(A, A, n);
        p >>= 1;
    }
    return result;
}

// Fibonacci using matrix exponentiation
long long fibMatrix(long long n) {
    if (n <= 1) return n;

    Matrix M = {{1, 1}, {1, 0}};
    Matrix result = matrixPower(M, n - 1);

    return result[0][0];
}
```

9 Important Sequences

Fibonacci Identities

- $F_{n+m} = F_n F_{m+1} + F_{n-1} F_m$
- $F_{2n} = F_n(2F_{n+1} - F_n)$
- $\gcd(F_m, F_n) = F_{\gcd(m,n)}$
- $F_n^2 + F_{n+1}^2 = F_{2n+1}$

Sum Formulas

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

10 Number Theoretic Problems

Trailing Zeros in n!

Problem

Count trailing zeros in $n!$

Solution

```
// Count factors of 5 in n!
int trailingZeros(int n) {
    int count = 0;
    while (n > 0) {
        n /= 5;
        count += n;
    }
    return count;
}
```

Power of Prime in n!

```
// Count power of prime p in n! (Legendre)
int powerInFactorial(int n, int p) {
    int count = 0;
    while (n > 0) {
        n /= p;
        count += n;
    }
    return count;
}
```

Modular Fibonacci

Problem

Compute $F_n \bmod m$ for large n ($n \leq 10^{18}$).

Key Concept

Pisano Period: Fibonacci sequence is periodic modulo any integer m. For $m \leq 10^6$, period $\leq 6m$.

Solution

```
// Find Pisano period
int pisanoPeriod(int m) {
    int prev = 0, curr = 1;

    for (int i = 0; i < m * m; i++) {
        int temp = (prev + curr) % m;
        prev = curr;
        curr = temp;

        if (prev == 0 && curr == 1)
            return i + 1;
    }
    return 0;
}

// Fibonacci mod m for large n
int fibMod(long long n, int m) {
    if (n <= 1) return n;

    int period = pisanoPeriod(m);
    n %= period;

    int prev = 0, curr = 1;
    for (int i = 2; i <= n; i++) {
        int temp = (prev + curr) % m;
        prev = curr;
        curr = temp;
    }
    return curr;
}
```

Modular Square Root

```
// Find x such that x^2 ≡n (mod p)
// Works when p is prime and p ≡3 (mod 4)
long long modSqrt(long long n, long long p) {
    if (power(n, (p - 1) / 2, p) != 1)
        return -1; // no solution

    return power(n, (p + 1) / 4, p);
}
```

Binary Exponentiation Tricks

```cpp
// Compute a^b for very large b stored as string
int powerLargeExponent(int a, string b, int mod) {
    int result = 1;
    a %= mod;

    for (char c : b) {
        int digit = c - '0';

        // result = result^10 * a^digit
        result = power(result, 10, mod);
        result = (result * power(a, digit, mod)) %
            mod;
    }
    return result;
}
```

Practice Problems

GCD & LCM:

- CSES - Common Divisors
- Codeforces 1436C - Binary Search
- LeetCode 1071 - Greatest Common Divisor of Strings

Primes:

- CSES - Prime Multiples
- SPOJ - PRIME1 (Segmented Sieve)
- ProjectEuler - Problem 7, 10

Modular Arithmetic:

- CSES - Exponentiation
- CSES - Binomial Coefficients
- Codeforces 300C - Beautiful Numbers

Combinatorics:

- CSES - Counting Towers
- LeetCode 62 - Unique Paths
- AtCoder ABC - Typical DP Contest

Advanced:

- CSES - Counting Divisors
- CSES - Fibonacci Numbers
- Codeforces 17A - Noldbach problem

Number theory is the foundation of cryptography, hashing, and many algorithmic optimizations.