Number Theory 1

Contents:

- ✔ Primality Test
- ✔ Sieve of Eratosthenes
- ✔ Prime factorization of a number
- ✔ Modular Arithmetic
- ✔ Modular Inverse
- ✔ Binary Exponentiation
- ✔ Euclidean GCD & LCM
- ✔ Binomial Coefficient nCr
- ✔ Euler Totient Function

Primality Test:
Check If a number n is prime or not.

O(n) - Brute force approach:

Check for every integer from 2 to n-1, if it divides n.

- Observation: Factors always occur in pairs.
- If (a, b) is a factor pair, a*b=n and a <= sqrt(n) <= b.
- For every composite number there will always be a factor from 2 to sqrt(n)
- It’s sufficient to check for all integers from 2 to sqrt(n) if it divides n

O(sqrt n) - Optimized Approach:

Check for every integer from 2 to sqrt(n), if it divides n.

```
bool isPrime(int x) {
    for (int d = 2; d * d <= x; d++) {
        if (x % d == 0)
            return false;
    }
    return true;
}
```

Find all Prime number from 1 to n (1 <= n <= 1e6)

Traditional Approach will make n*sqrt(n) operation ~1e9 operations in worst case. Can we do better?

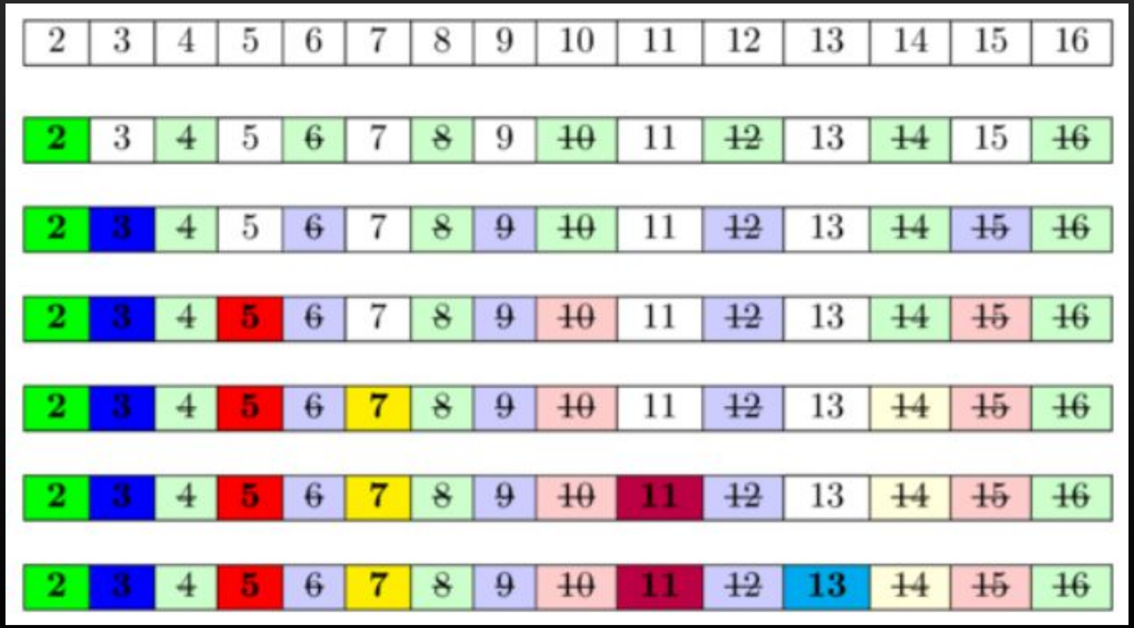
Sieve Of Eratosthenes:

An algorithm for finding all the prime numbers in a segment [1:n] in just O(n*loglogn) operations
How small is O(n*loglogn)

Idea: Don’t check for all numbers, multiples of prime numbers are always composite.

- Mark multiples of prime numbers as composite
- Complexity – n*log(log(n)) - [How]?

N = 16 Dry Run



Implementation:

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
    if (is_prime[i] && (long long)i * i <= n) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

Additional Algorithms:

- Segmented Sieve – for finding primes between large numbers.
- Linear Sieve – finding primes less than 1e7 in O(n), also used for computing smallest prime factor of a number

You can read about the above 2 algorithms if you're fascinated by this area of Number Theory but they won't be required unless you're a Master or above on Codeforces.

Prime Factorization of a number

Trial Division Method O(sqrt(n)):

- If n is composite, it will have factors in pair, and one factor from each pair will be less than sqrt(n) and the other can be found using the first factor.
- If n is prime, it will have just 1 prime factor that is n.

Implementation:

```
vector<long long> trial_division1(long long n) {
    vector<long long> factorization;
    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}
```

Using Sieve - Precomputation: O(nloglogn), Query Time: O(logn)

Precomputing smallest prime factor for each n from 1 to N

```
void computeSmallestPrime(int N){
    smallestPrime.resize(N + 1, -1);
    for(int i = 2; i <= N; i++){
        if(smallestPrime[i] == -1){
            smallestPrime[i] = i;
            for(int j = i * i; j <= N; j += i){
                if(smallestPrime[j] == -1 || smallestPrime[j] > i)
                    smallestPrime[j] = i;
            }
        }
    }
}
```

Query Function

```
vector<int> queryFactorisation(int n){
    vector<int> primeFactorisation;
    while(n > 1){
        int prime = smallestPrime[n];
        primeFactorisation.pb(prime);
        while(n % prime == 0){
            n /= prime;
        }
    }
    return primeFactorisation;
}
```

Binary Modular Exponentiation

Modular Binary Exponentiation in O(logn):

Instead of multiplying linearly, multiply by squaring.

Example: find 3^13

13 can be written as 1101 or 8+4+1.

3^13 can be written as 3^(8+4+1) = 3^8 * 3^4 * 3^1.

Implementation:

```
long long binpow(long long a, long long b) {
    if (b == 0)
        return 1;
    long long res = binpow(a, b / 2);
    if (b % 2)
        return res * res * a;
    else
        return res * res;
}
```

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

Modular Arithmetic:

What does the expression $a \equiv b \pmod{m}$ signify?

Modular Congruences:
Number a and b which leaves the same remainder when divided by some integer m.
Example –

$19 \equiv 44 \bmod 5$

$23 \equiv 3 \bmod 4$

Important Properties of modular arithmetic: (mod is distributive over addition, subtraction and multiplication)

- $(a + b) \bmod m \equiv ((a \bmod m) + (b \bmod m)) \bmod m$

- $(a - b) \bmod m \equiv ((a \bmod m) - (b \bmod m) + m) \bmod m$

- $(a * b) \bmod m \equiv ((a \bmod m) * (b \bmod m)) \bmod m$

Remember: $(a / b) \bmod m$ is not $\equiv ((a \bmod m) / (b \bmod m)) \bmod m$
Mod is <u>not</u> distributive over division.

For division we use something called a modular multiplicative inverse.

Modular multiplicative inverse (mod_inv) :

There are 2 faster ways of calculating mod_inv of a number.

- Extended Euclidean algorithm.
- Fermat’s little theorem.

Though the extended Euclidean algorithm is more versatile and sometimes slightly faster, the Fermat's little theorem method is more popular and simpler, but it works only when m is prime.

Fermat’s little theorm: Let mod_inv(a)=b We want to find a number b such that a*b mod m = 1
We know,

    a^m mod m = a mod m
    a^(m-1) mod m = 1
    a*a^(m-2) mod m = 1

Hence b = a^(m-2) mod m.

How to calculate a^(m - 2) mod m -> Binary Modular Exponentiation as discussed before

Euclidean GCD

Originally, the Euclidean algorithm was formulated as follows: subtract the smaller number from the larger one until one of the numbers is zero.

Instead of subtracting multiple times to get to the remainder, we can use mod. The relation then becomes:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Implementation: Time complexity: O(log(min(a, b))) - Proof

```
// recursive approach
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

```
// iterative approach
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

Binomial Coefficient nCr.

Find nCr(n, r) in log n:

Dp approach for finding nrc will be discussed later.

Using Fermat's little theorem:

- We know that nCr = n! / (r! * (n-r)!)
- We can precompute n! for all integers from [1, n].
- nCr can run out of bound very quickly so most of the time the answer is returned modulo some prime number.
- nCr = n! / (r! * (n-r)!) mod m
- We can use fermat little theorem for finding mod_inv.
- nCr = n! mod m * mod_inv(r!) mod m * mod_inv((n-r)!) mod m.

Implementation:

```
void computeFact(ll n, ll p) {
    fact.resize(n+1);
    fact[0] = 1;
    for(ll i=1; i<=n; i++) {
        fact[i] = fact[i-1]*i%p;
    }
}
ll mod_inv(ll n, ll p) {
    return bin_expo(n, p-2, p);
}
```

```
computeFact(n, p);
ll ncr(ll n, ll r, ll p) { // return nCr mod p
    if(n<r) return 0;
    if(r==0) return 1;
    return fact[n]*mod_inv(fact[r], p)%p*mod_inv(fact[n-r], p)%p;
}
```

Count number of integers from [1, n] which are coprime to n.

Euler Totient Function: phi(n) – returns count of integers in [1, n] which are coprime to n.

- If p is a prime number, then $\gcd(p, q) = 1$ for all $1 \le q < p$. Therefore we have:

$$\phi(p) = p - 1.$$

- If p is a prime number and $k \ge 1$, then there are exactly p^k/p numbers between 1 and p^k that are divisible by p. Which gives us:

$$\phi(p^k) = p^k - p^{k-1}.$$

- If a and b are relatively prime, then:

$$\phi(ab) = \phi(a) \cdot \phi(b).$$

If $n = {p_1}^{a_1} \cdot {p_2}^{a_2} \cdots {p_k}^{a_k}$, where p_i are prime factors of n,

$$\begin{aligned}
\phi(n) &= \phi({p_1}^{a_1}) \cdot \phi({p_2}^{a_2}) \cdots \phi({p_k}^{a_k}) \\
&= \left({p_1}^{a_1} - {p_1}^{a_1-1}\right) \cdot \left({p_2}^{a_2} - {p_2}^{a_2-1}\right) \cdots \left({p_k}^{a_k} - {p_k}^{a_k-1}\right) \\
&= p_1^{a_1} \cdot \left(1 - \frac{1}{p_1}\right) \cdot p_2^{a_2} \cdot \left(1 - \frac{1}{p_2}\right) \cdots p_k^{a_k} \cdot \left(1 - \frac{1}{p_k}\right) \\
&= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)
\end{aligned}$$

Source: Cp Algorithms

Phi(n) $= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$

Implementation Using factorization in O(sqrt(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;
}
```

Phi(n) $= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$

Calculating phi(n) for all numbers from 1 to n using sieve in O(n*loglogn)

```
void phi_1_to_n(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) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

Number Theory Functions that I like to keep handy for contests

```
ll gcd(ll a, ll b) {if (b > a) {return gcd(b, a);} if (b == 0) {return a;} return gcd(b, a %
ll expo(ll a, ll b, ll mod) {ll res = 1; while (b > 0) {if (b & 1)res = (res * a) % mod; a =
void extendgcd(ll a, ll b, ll*v) {if (b == 0) {v[0] = 1; v[1] = 0; v[2] = a; return ;} exten
ll mminv(ll a, ll b) {ll arr[3]; extendgcd(a, b, arr); return arr[0];} //for non prime b
ll mminvprime(ll a, ll b) {return expo(a, b - 2, b);}
bool revsort(ll a, ll b) {return a > b;}
ll combination(ll n, ll r, ll m, ll *fact, ll *ifact) {ll val1 = fact[n]; ll val2 = ifact[n
void google(int t) {cout << "Case #" << t << ": ";}
vector<ll> sieve(int n) {int*arr = new int[n + 1](); vector<ll> vect; for (int i = 2; i <= n
ll mod_add(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a + b) % m) + m) % m;}
ll mod_mul(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a * b) % m) + m) % m;}
ll mod_sub(ll a, ll b, ll m) {a = a % m; b = b % m; return (((a - b) % m) + m) % m;}
ll mod_div(ll a, ll b, ll m) {a = a % m; b = b % m; return (mod_mul(a, mminvprime(b, m), m)
ll phin(ll n) {ll number = n; if (n % 2 == 0) {number /= 2; while (n % 2 == 0) n /= 2;} for
ll getRandomNumber(ll l, ll r) {return uniform_int_distribution<ll>(l, r)(rng);}
```