Aula 8 · Teoria dos Números: Fatoração, Crivos, Algoritmo de Euclides, Teorema Chinês do Resto, Exponenciação de Matrizes e mais

Desafios de Programação

Fernando Kiotheka Victor Alflen

UFPR

27/07/2022

Apresentação de Teoria dos Números

Maratonista só pensa em número primo...

- Teoria dos números é uma área da matemática que estuda os números inteiros e funções com números inteiros.
- Cientistas da computação são bem fãs de números inteiros, porque é o que conseguimos representar nos computadores sem problemas (a história é diferente com os números reais).
- O registro mais antigo desse estudo pode ser traçado à Mesopotamia, 1800 BCE, em uma tablete (de argila) com uma lista de trios pitagóricos ($a^2 + b^2 = c^2$).
- Um dos mais importantes problemas não resolvidos na matemática é a Hipótese de Riemann que implicaria em resultados sobre a distribuição de números primos.
- **Sempre** cai na maratona.

O Teorema Fundamental da Aritmética

O Teorema Fundamental da Aritmética afirma que todo inteiro $N > 1$ pode ser representado como um produto único de números primos não necessariamente distintos:

$$50 = 2 \cdot 5 \cdot 5 = 2^1 \cdot 5^2$$

Nomeamos isso a fatoração prima de um número.

Fatoração por divisão por tentativa

- Método que testa os números $\leq \sqrt{N}$ para ver se é divisível.
- Ao encontrar um fator, divide o N quantas vezes for possível.
- No pior caso (por exemplo se o número for um primo gigante), a complexidade é de $\mathcal{O}(\sqrt{N})$.

Implementação de fatoração por divisão por tentativa

Código `factorize.cpp`

```cpp
#include <bits/stdc++.h>
using namespace std; using ll = long long;
void factorize /* O(sqrt(n)) */ (ll n) {
    for (ll i = 2; i*i <= n; i++)
        while (n % i == 0) { n /= i; cout << i << " "; }
    if (n > 1) { cout << n << " "; }
    cout << "\n";
}
int main() { int n; while (cin >> n) { factorize(n); } }
```

Entrada	Saída
50	2 5 5
1000000007	1000000007
1000000009	1000000009
1000696969	1000696969
998244353	998244353
720720	2 2 2 2 3 3 5 7 11 13

Primos e coprimos

- Números primos são aqueles cuja fatoração prima é composta apenas por ele mesmo (ou alternativamente, só podem ser divididos por 1 ou por ele mesmo).
- 1 não é primo.
- São infinitos e são numerosos. A prova é deixada para matemáticos do século 3 BCE.
- Sua densidade pode ser aproximada por

$$\pi(n) \approx \frac{n}{\ln n}$$

- $\pi(10^6) = \frac{10^6}{\ln 10^6} \approx 72382$ (o número real é 78496)
- **Coprimos:** dois números com fatoração completamente distinta ($15 = 3 \cdot 5$ e $26 = 2 \cdot 13$, por exemplo, são coprimos)

Somas harmônicas e complexidades estranhas

Por conta da densidade dos primos, essa aula tem alguma complexidades "esquisitas". Em particular:

$$\lim_{n\to\infty}\left(\sum_{\substack{p \text{ primo}\\ p\leq n}}\frac{1}{p} - \ln(\ln n)\right) = M \approx 0{,}2614972\ldots$$

A constante de Meissel-Mertens acima nos deixa concluir que um laço sobre números primos é $\mathcal{O}(\lg\lg n)$.
Ao mesmo tempo, a constante de Euler-Mascheroni abaixo leva à ideia de que um laço com passos de tamanho incremental é $\mathcal{O}(\lg n)$:

$$\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln n\right) = \gamma \approx 0{,}5772156$$

Crivos e geradores de números

Crivos

- Vêm da ideia de, a partir de uma lista de números, filtrar os que não satisfizerem alguma condição.
- **Crivo de Eratóstenes:** obter primos em $\mathcal{O}(n \lg \lg n)$ e com menor uso de memória (suporta algo na magnitude de 10^9 números)
- **Crivo de Euler:** obter primos em $\mathcal{O}(n)$ (potencialmente mais lento que o crivo de Eratóstenes) sob um custo maior de memória (mas que permite uma fatoração mais rápida)
- **Crivos segmentados:** construir primos maiores usando menos memória (não veremos aqui)
- **Extensões do crivo:** `https://codeforces.com/blog/entry/22229` (veremos aqui)
- Mais coisas assim no capítulo 5 do livro dos Halim.

Crivo de Erastótenes

Código `sieve.cpp`

```
#include <bits/stdc++.h>
using namespace std;
vector<bool> sieve (1e7+15, true);
void eratosthenes /* O(n lg lg n) */ (int n) {
    for (int i = 2; i * i <= n; i++) if (sieve[i])
        for (int j = i * i; j <= n; j += i)
            sieve[j] = false;
}
int main() {
    int n; cin >> n;
    eratosthenes(n);
    for (int i = 2; i <= n; i++)
        if (sieve[i]) { cout << i << " "; }
    cout << "\n";
}
```

Entrada	Saída
30	2 3 5 7 11 13 17 19 23 29

Crivo de Euler

Código `linearsieve.cpp`

```
#include <bits/stdc++.h>
using namespace std;
vector<int> pr, lp (1e7+15);
void eulersieve /* O(n) */ (int n) {
    for (int i = 2; i <= n; i++) {
        if (lp[i] == 0) { lp[i] = i; pr.push_back(i); }
        for (int j = 0; j < int(pr.size())
                && pr[j] <= lp[i] && i*pr[j] <= n; j++)
            lp[i * pr[j]] = pr[j];
    }
}
int main() {
    int n; cin >> n; eulersieve(n);
    for (int p : pr) { cout << p << " "; } cout << "\n";
}
```

Entrada	Saída
30	2 3 5 7 11 13 17 19 23 29

Maior divisor primo em $\mathcal{O}(n \lg \lg n)$

Código `bigpsieve.cpp`

```
int big[n + 1] = {1, 1};
for (int i = 1; i*i <= n; ++i)
    if (big[i] == 1)
        for (int j = i; j <= n; j += i)
            big[j] = i;
```

Número de divisores

Sabendo a fatoração de um número inteiro, podemos calcular quantos divisores (primos e não primos) ele tem. Se

$$n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

Então n tem

$$d(n) = (e_1 + 1)(e_2 + 1) \dots (e_k + 1)$$

divisores.

Intuição: qualquer divisor de n vai ter em sua fatoração um subconjunto da fatoração de n, então o i-ésimo primo na fatoração de n vai ter expoente $0 \leq f \leq e_i$.

Implementação de um gerador de número de divisores

Código `divisors.cpp`

```
vector<int> divisors (n + 1);

void number_of_divisors /* O(n lg n) */ (int n) {
  for (int i = 1; i <= n; ++i)
    for (int j = i; j <= n; j += i)
      divisors[j]++;
}
```

Soma dos divisores

Numa linha de raciocínio similar, a soma dos divisores de n será somar todas as possibilidades para cada primo que divide n:

$$(p_1^0 + p_1^1 + p_1^2 + ... + p_1^{e_1})(p_2^0 + p_2^1 + p_2^2 + ... + p_2^{e_2})...$$

Da matemática discreta, sabemos que

$$\sum_{i=0}^{n} p_1^i = \frac{p_1^{e_1+1} - 1}{p_1 - 1}$$

Então, a soma dos divisores de n será

$$\prod_{i=0}^{d(n)-1} \frac{p_i^{e_i+1} - 1}{p_i - 1}$$

Implementação de um gerador de soma de divisores

Código `sumdivsieve.cpp`

```
void sumdiv /* O(n lg n) */ (int n) {
    int sumdiv[n+1];
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            sumdiv[j] += i;
}
```

Função totiente de Euler

- A função totiente de euler conta quantos números menores ou iguais a n que são coprimos com n.
- Ela tem propriedades multiplicativas: $\phi(mn) = \phi(m)\phi(n)$.
- Para um p primo, $\phi(p) = p - 1$
- Podemos implementá-la em $O(n \lg \lg n)$.

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

+	1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	4	2	6	4	6	4
10	10	4	12	6	8	8	16	6	18	8
20	12	10	22	8	20	12	18	12	28	8
30	30	16	20	16	24	12	36	18	24	16
40	40	12	42	20	24	22	46	16	42	20
50	32	24	52	18	40	24	36	28	58	16

Implementação da função totiente de Euler

Código totient.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 1e6+15;
vector<ll> phi (N);
void euler_totient /* O(n lg lg n) */ (int n) {
    for (int i = 1; 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;
}
int main() {
    euler_totient(1e6);
    int n; while (cin >> n) { cout << phi[n] << "\n"; }
}
```

Teorema de Euler (da teoria dos números)

Já vimos na Aula 2, o pequeno teorema de Fermat:

$$a^{p-1} = 1 \pmod{p},$$

onde p é **primo**.
Porém Euler o generalizou, sendo que

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

onde n é qualquer módulo, e a é coprimo com n, e $\phi(n)$ é a função totiente de Euler. É claro que $\phi(p) = p - 1$, onde p é primo.

MDC e MMC

Máximo Divisor Comum e Mínimo Múltiplo Comum

O **Máximo Divisor Comum** (MDC) entre dois números a e b é o maior inteiro g que divide tanto a quanto b.
Já o **Mínimo Múltiplo Comum** (MMC) entre a e b é o menor inteiro divisível por a e b.
Como obtê-los?

- **MDC:** Algoritmo de Euclides
- **MMC:** $\frac{ab}{\mathrm{MDC}(a,b)}$

Importante: Desde o C++17, `gcd` e `lcm` já são funções prontas do C++, mas conhecer a sua definição pode ajudar bastante!

Implementação do Algoritmo de Euclides

Código `euclides.cpp`

```
ll gcd /* O(lg min(a, b)) */ (ll a, ll b) {
        if (b == 0) { return a; }
        return gcd(b, a%b);
}

ll lcm /* O(lg min(a, b)) */ (ll a, ll b) {
        return a*(b/gcd(a, b));
}
```

Estendendo o Algoritmo de Euclides

- A complexidade do algoritmo de Euclides é $\mathcal{O}(\lg \min\{a, b\})$.
- Podemos calcular o inverso multiplicativo mais rápido do que $\mathcal{O}(\lg P)$ (exponenciação binária de primos), mas pra isso precisamos saber do x.

$$ax = g \pmod{b}, g = 1$$

$$\implies ax = 1 \pmod{b}$$

- Queremos resolver $ax + by = gcd(a, b)$. Faremos isso com o algoritmo de Euclides, salvando os valores de x e y.
- Quais valores x e y assumirão sabendo que na chamada subsequente obtivemos x_1 e y_1?

Estendendo o Algoritmo de Euclides (continuado)

Na chamada recursiva, obtivemos:

$$bx_1 + (a \bmod b)y_1 = g$$

Nesta chamada, estamos tratando de:

$$ax + by = g$$

Sabendo que $a \pmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$, manipulamos as equações para obter:

$$g = ay_1 + b\left(x_1 - y_1 \left\lfloor \frac{a}{b} \right\rfloor\right)$$

Isto é,

$$x = y_1$$

e

$$y = x_1 - y_1 \left\lfloor \frac{a}{b} \right\rfloor$$

Implementação do Euclides estendido

Código `extgcd.h`

```
ll extgcd /* O(lg min(a, b)) */ (ll a, ll b, ll& x, ll& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    ll g = extgcd(b, a%b, x, y);
    tie(x, y) = make_tuple(y, x - (a/b)*y);
    return g;
}

ll inv(ll a, ll m) {
    ll x, y; extgcd(a, m, x, y);
    return ((x % m) + m) % m;
}
```

Resolvendo equações diofantinas lineares

Para resolver uma equação do tipo $ax + by = c$, podemos encontrar o MDC entre a e b com o algoritmo de Euclides estendido para obter x_g, y_g tais que $ax_g + by_g = \gcd(a, b) = g$. Se g divide c, podemos achar uma solução (caso contrário, não). No caso de existirem soluções, uma delas é:

$$x = x_g \frac{c}{g}$$

$$y = y_g \frac{c}{g}$$

Podemos obter novas soluções com $x_{\text{novo}} = x + k\frac{b}{g}$ e $y_{\text{novo}} = y - k\frac{a}{g}$ (note os sinais diferentes!)

Implementação das equações diofantinas lineares

Código diophantine.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
#include "extgcd.h"
void solve(ll a, ll b, ll c, int i) {
    ll x, y, g = extgcd(a, b, x, y);
    if (c % g) { return; }
    x *= c/g; y *= c/g;
    while (i--) {
        cout << "(x=" << x << " y=" << y << ") ";
        x += b/g; y -= a/g;
    }
}
int main() {
    ll a, b, c; cin >> a >> b >> c; solve(a, b, c, 2); }
```

Entrada	Saída
39 15 12	(x=8 y=-20) (x=13 y=-33)

Teorema Chinês do Resto

É possível resolver

$$\begin{aligned} x &= a_1 \pmod{m_1} \\ x &= a_2 \pmod{m_2} \\ &\vdots \\ x &= a_n \pmod{m_n} \end{aligned}$$

Onde todos os $m_1, m_2, \ldots m_n$ são coprimos, e a única solução é congruente $\pmod{\prod m_i}$.

Resolvendo o Teorema Chinês do Resto

Suponha que queremos resolver

$$x = a_1 \pmod{m_1}$$
$$x = a_2 \pmod{m_2}$$

Sendo m_1 e m_2 coprimos.

- Seja m_2^{-1} um inteiro que satisfaça $m_2 m_2^{-1} = 1 \pmod{m_1}$
- Seja m_1^{-1} um inteiro que satisfaça $m_1 m_1^{-1} = 1 \pmod{m_2}$
- Então achamos $e_1 = m_2 m_2^{-1}$ de forma que $e_1 = 1 \pmod{m_1}$ e $e_1 = 0 \pmod{m_2}$.
- E achamos $e_2 = m_1 m_1^{-1}$ de forma que $e_2 = 1 \pmod{m_2}$ e $e_2 = 0 \pmod{m_1}$.
- Finalmente, $x = a_1 e_1 + a_2 e_2$.

Generalizando...

- Qual número é 0 (mod m)? Qualquer número múltiplo de m.
- Então multiplicamos cada número por $\frac{\prod m_i}{m}$, isso significa que o resultado será zero em todas os outros módulos exceto m.
- Em seguida queremos retirar esse número da nossa base m, fazemos isso pegando o seu inverso na base m e multiplicando.
- Multiplicamos isso pela constante que se quer igualar e somamos a resposta, concluindo o sistema m atual.

Implementação do Teorema Chinês do Resto

Código `crt.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using vll = vector<ll>;
#include "extgcd.h"
ll crt /* O(t lg m1*..*mt) */ (vll a, vll m, int t) {
    ll p = 1; for (ll& mi : m) { p *= mi; }
    ll ans = 0;
    for (int i = 0; i < t; i++) {
        ll pp = p / m[i];
        ans = (ans + ((a[i] * inv(pp, m[i])) % p * pp) % p) % p;
    }
    return ans;
}
int main() {
    int t; while (cin >> t) {
        vll a (t), m (t);
        for (int i = 0; i < t; i++) { cin >> a[i] >> m[i]; }
        cout << crt(a, m, t) << "\n";
    }
}
```

Implementação do Teorema Chinês do Resto (continuado)

Entrada	Saída
3 2 3 3 5 2 7 2 1 2 2 3 2 151 783 57 278	23 5 31471

Teorema Chinês do Resto generalizado

- A ideia do Teorema Chinês do Resto pode ser aplicada também a números que não são coprimos entre si.
- Transformamos as equações em várias equações que são coprimas entre si.
- Fazemos isso de forma inteligente, então isso não fica evidente, se ancorando na Identidade de Bézout.
- Atente-se que a solução pode não existir! (quando equações se contradizem)
- Se existir solução, existem infinitas soluções subtraindo ou somando o mínimo múltiplo comum entre todos os módulos.
- A implementação é mais robusta que o do Teorema Chinês do Resto visto anteriormente.

Implementação do Teorema Chinês do Resto generalizado

Código `noncoprimecrt.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using vll = vector<ll>;
#include "extgcd.h"
ll norm(ll a, ll b) { a %= b; return (a < 0) ? a + b : a; }
pair<ll, ll> crt_single(ll a, ll n, ll b, ll m) {
    ll x, y; ll g = extgcd(n, m, x, y);
    if ((a - b) % g) { return {-1, -1}; }
    ll lcm = (m/g) * n;
    return {norm(a + n*(x*(b-a)/g % (m/g)), lcm), lcm};
}
ll crt /* O(t lg m1*..*mt) */ (vll a, vll m, int t) {
    ll ans = a[0], lcm = m[0];
    for (int i = 1; i < t; ++i) {
        tie(ans, lcm) = crt_single(ans, lcm, a[i], m[i]);
        if (ans == -1) { return -1; }
    }
    return ans;
}
int main() {
    int t; while (cin >> t) {
        vll a (t), m (t);
        for (int i = 0; i < t; i++) { cin >> a[i] >> m[i]; }
        cout << crt(a, m, t) << "\n";
    }
}
```

Exponenciação de Matrizes

Você conhece essa matriz?

- Podemos fazer também exponenciação de matrizes!
- Assim podemos calcular rapidamente números de Fibonacci:

$$f(n) = f(n-1) + f(n-2)$$

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1\,1 \\ 1\,0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

$$= \begin{bmatrix} 1\,1 \\ 1\,0 \end{bmatrix}^n \begin{bmatrix} f(0) \\ f(-1) \end{bmatrix}$$

- Com matrizes é possível calcular toda função linear da forma:

$$f(n) = k_1 f(n-1) + k_2 f(n-2) + \ldots + k_m f(n-m)$$

  Esse tipo de coisa aparece em problemas de nível mundial!
- No livro do Atkin tem algumas outras aplicações, como descobrir o caminho mínimo em um grafo usando n arestas (quantidade de multiplicações).

Calculando Fibonacci com exponenciação binária

Código `fibonacci.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const int M = 1e9;
using ll = long long; using mat = vector<vector<ll>>;
void mat_mul(mat& res, mat a) { mat b = res;
    for (int i = 0; i < 2; i++) for (int j = 0; j < 2; j++)
        res[i][j] = ((a[i][0]*b[0][j]) % M
                   + (a[i][1]*b[1][j]) % M) % M;
}
void fast_pow(mat& res, mat a, int b) {
    if (b == 1) { res = a; return; }
    fast_pow(res, a, b/2);
    mat_mul(res, res); if (b % 2) { mat_mul(res, a); }
}
int main() {
    int n; while (cin >> n) {
        mat x {{ 1, 1 }, { 1, 0 }};
        fast_pow(x, x, n); cout << x[0][0] << "\n";
    }
}
```

Calculando Fibonacci com exponenciação binária (continuado)

Entrada	Saída
10	89
30	1346269
100	817084101
10000000	120937501
100000000	60937501

Outros resultados

Número de Catalan

- Frequentemente aparece em problemas de contagem.
- Número de jeitos de organizar parênteses em uma string.
- Número de árvores binárias cheias com $n + 1$ folhas.
- Muitos outros.

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Implementação do número de Catalan

Código catalan.cpp

```
ll catalan(int n) {
    if (n == 0) { return 1; }
    ll a = (2*(2*(n-1)+1)) % P;
    ll res = (((a*inv(n+1, P)) % P) * catalan(n-1)) % P;
    return res;
}
```

Teorema de Lucas

$$\binom{m}{n} = \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p}$$

onde

$$m = m_k p^k + m_{k-1} p^{k-1} + \ldots + m_1 p + m_0$$

e

$$n = n_k p^k + n_{k-1} p^{k-1} + \ldots + n_1 p + n_0$$

Mas o que isso significa na prática?

- $\binom{m}{n}$ é divisível pelo número primo p se pelo menos um dos dígitos de n na base p é maior que o dígito correspondente de m na base p.
- $\binom{m}{n}$ é ímpar apenas se a representação binária de n é um subconjunto da representação binária de m.

Testando primalidade

- Podemos testar se um número é primo mais rápido por meio de métodos aleatórios.
- Um deles é o do Miller-Rabin que nos permite testar a primalidade rápido o suficente (complexidade estimada é de $\mathcal{O}(k \lg^3 n)$ onde k é o número de testemunhas).

Miller-Rabin: Pow de 128 bits

Código `fastpow128.h`

```
ll fast_pow(ll a, ll e, ll m) {
    if (e == 0) { return 1; }
    if (e == 1) { return a; }
    ll res = fast_pow(a, e/2, m);
    res = (i128)res * res % m;
    if (e % 2) res = (i128)res * a % m;
    return res;
}
```

Implementação do Miller-Rabin

Código `millerrabin.h`

```
vector<int> witnesses = {
    // primes from 2 to 37, enough for 64 bits
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37
};
bool is_prime(ll n) {
    if (n < 2) { return false; }
    int s = __builtin_ctzll(n - 1);
    ll d = n >> s;
    for (int a : witnesses) {
        if (n == a) { return true; }
        ll p = fast_pow(a, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = (i128)p * p % n;
        if (p != n - 1 && i != s) { return false; }
    }
    return true;
}
```

Utilização do Miller-Rabin

Código `millerrabin.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using i128 = __int128_t;
#include "fastpow128.h"
#include "millerrabin.h"
int main() {
    ll n; while (cin >> n) if (is_prime(n))
        cout << n << "\n";
}
```

Entrada	Saída
120244006539835491	120244006539835489
120244006539835489	24316731626151113 3
24316731626151113 3	262890056873052241
24316731626151113 7	99999999999999989
262890056873052241	
99999999999999991	
99999999999999989	

Fatorando usando Miller-Rabin: Pollard's rho

Código `pollardrho.h`

```
ll pollard(ll n) {
    auto f = [n](ll x) {
        return (fast_pow(x, x, n) + 1) % n;
    };
    if (n % 2 == 0) return 2;
    for (ll i = 2; ; i++) {
        ll x = i, y = f(x), p;
        while ((p = gcd(n + y - x, n)) == 1)
            x = f(x), y = f(f(y));
        if (p != n) { return p; }
    }
}
void factorize /* O(n^(1/4)) */ (ll n) {
    if (n == 1) { return; }
    if (is_prime(n)) { cout << n << " "; return; }
    ll x = pollard(n);
    factorize(x); factorize(n/x);
}
```

Fatorando usando Miller-Rabin: Pollard's rho (continuado)

Código `pollardrho.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using i128 = __int128_t;
#include "fastpow128.h"
#include "millerrabin.h"
#include "pollardrho.h"
int main() {
    ll n; while (cin >> n) { factorize(n); cout << "\n"; }
}
```

Entrada	Saída
120244006539835491	3 4008133513278497
120244006539835489	120244006539835489
24316731626151113 3	24316731626151113 3
24316731626151113 7	3 1288877 62888679127
262890056873052241	262890056873052241
99999999999999991	23 71 307 2251 6257 141623
99999999999999989	99999999999999989

E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Tem muitos problemas interessantes de Teoria dos Números.
- No CSES tem alguns problemas legais.
- Nos artigos de Teoria dos Números no CP-Algorithms tem vários problemas referenciados para testar seus conhecimentos.
- Para entender mais sobre o algoritmo de Miller-Rabin: `https://www.youtube.com/watch?v=_MscGSN5J6o`.
- Bons estudos!