Teoria dos Números na Maratona

Fernando Monteiro Kiotheka

Escola de Inverno da Maratona de Programação 2023

XLIII Congresso da Sociedade Brasileira de Computação
CSBC 2023 João Pessoa, PB

8 de agosto de 2023

Introdução

Quem sou eu

- Bacharel em Ciência da Computação, fazendo mestrado em Sistemas Distribuídos, uma área de Redes de Computadores (vim para o CSBC apresentar no CTIC!).
- Atual *coach* dos **16 times** da Universidade Federal do Paraná.
- Ministrei por três vezes a optativa de maratona na UFPR, a disciplina de *Desafios de Programação* em parceria com Raul Gomes, Victor Alflen e Vinicius Tikara.
- Ex-competidor: estourei minha quota de 5 subregionais e 2 finais brasileiras (uma no gramado de casa...).
- Nosso time "Jump and Link to the Past" obteve a 25ª colocação em Campo Grande.
- Não sou superestrela, nem tenho muito rating no Codeforces, mas tenho fé que a minha motivação vai trazer a UFPR de volta aos holofotes! (conhecem um tal Bruno Ribas?)

Mas o esencial...

- Gosto de dar aula.
- Gosto de estruturas de dados e algoritmos, nisso que atuo na minha pesquisa e isso que precisa para maratona.
- Gosto de falar sobre o que eu não sou especialista.

O que é teoria dos números?

Uma área da matemática pura.

- Estudo dos números inteiros e funções aritméticas.
- Originalmente chamada de **aritmética**.

Por que cientistas da computação gostam de números inteiros?

- Representar números inteiros em computadores é moleza.
- Números reais... não.

Humanos estudando teoria dos números

Figura: Tablete (de argila) em Larsa, Mesopotamia, ca. 1800 BCE.

- É uma lista de triplas pitagóricas ($a^2 + b^2 = c^2$).
- São muitas e muito grandes para ser obtidas por força bruta.
- Suspeita-se que foi utilizado a identidade:

$$\left(\frac{1}{2}\left(x - \frac{1}{x}\right)\right)^2 + 1 = \left(\frac{1}{2}\left(x + \frac{1}{x}\right)\right)^2.$$

Problemas famosos da teoria dos números

- **Último Teorema de Fermat**: não existem inteiros positivos a, b e c que satisfaçam $a^n + b^n = c^n$ para $n > 2$.
  - Pierre de Fermat escreveu em por volta de 1637 na margem de uma cópia do livro Arithmetica que ele tinha a prova, mas não cabia na margem.
  - Pra passar problemas rápido, as vezes temos que fazer a mesma coisa. A prova fica para o leitor.
  - Este teorema foi provado, só que em 1994 pelo matemático Andrew Wiles. Podemos afirmar que a prova não é trivial.
- **Hipótese de Riemann**: não vamos explicar aqui mas é famoso e importante.

Outros problemas famosos da teoria dos números

- **Conjectura de Goldbach**: todo número primo maior que dois pode ser representado pela soma de dois números primos.
  - Funciona até inteiros menores que $4 \cdot 10^{18}$.
- **Conjectura dos primos gêmeos**: existem infinitos primos gêmeos (da forma p e $p + 2$, onde os dois são primos).
  - Maiores já encontrados: $2996863034895 \cdot 2^{1290000} \pm 1$.
  - 808 675 888 577 436 primos gêmeos menores que 10^{18}.
- **Teorema dos números primos**: a distribuição da quantidade de primos até N asintoticamente aproxima $\frac{N}{\log N}$.

O que não é teoria dos números?

Existem algumas outras áreas da matemática que compartilham o mesmo "nível" de hierarquia que teoria dos números como:

- **Geometria**: propriedades do espaço.
- **Álgebra**: manipular equações e fórmulas.
- **Cálculo e Análise**: estudo da mudança contínua, gostam de números reais e complexos.
- **Matemática discreta**: contáveis, combinatória e grafos.
- **Lógica e teoria dos conjuntos**: abstrato, computação.
- **Estatística**: amostras de dados e probabilidade.
- **Matemática computacional**: aproximações, discretização e computação aplicada a matemática.

Mas tudo tem intersecção e pra resolver problemas, vamos por vezes se aprofundar em áreas como álgebra e matemática discreta.

Decompondo e Formando Números

Expressando números com outros números

Através de **operadores matemáticos**, podemos combinar números. Assim, podemos representar um determinado número como sendo o resultado da operação de outros números.

$$1 + 2 + 3 = 6$$
$$1 \cdot 2 \cdot 3 = 6$$

E a álgebra permite fazer essas **equações** de forma mais genérica:

$$a + b + c = 6$$
$$abc = 6$$

- Observe que no caso da soma, se considerarmos os números reais, existem infinitas soluções. Se considerarmos apenas os inteiros positivos, temos uma quantidade finita de soluções.
- Porém no caso da multiplicação, ao considerar os inteiros positivos, ainda há uma quantidade infinita de soluções.

Equações com restrições

- Equações onde as variáveis tem que ser inteiras são chamadas de **equações diofantinas**.
- Vimos a pouco o estudo das triplas pitagóricas ($a^2 + b^2 = c^2$), que é um exemplo de uma equação diofantina não-linear.
- O Último Teorema de Fermat também descreve uma equação diofantina: $a^n + b^n = c^n$.
- A conjectura de Goldbach é um exemplo de equação diofantina da forma $p + q = n$ com a restrição de que p e q são primos, e a ideia é que eles existam pra todo n.
- Nós veremos como resolver qualquer equação diofantina da forma $ax + by = c$ em breve.
- Mas primeiro, vamos parar para pensar na equação diofantina que descreve a representação dos nossos números?

Sistemas de numeração

É um sistema que permite que um conjunto de números seja representado por numerais de uma forma consistente. Exemplos incluem:

- Números romanos: I, II, III, IV, V, VI, VII, ...
- Números em português: um, dois, três, quatro, cinco, seis, ...
- Ou... marcas de contagem:

Figura: Marcas de contagem usadas na França, Portugal, Espanha, e suas ex-colônias, incluindo a América Latina.

Mas o mais comum são os sistemas de numeração **posicionais**.

Sistemas de numeração posicionais

Um sistema de numeração posicional se baseia em uma **base** b. Além disso existem b símbolos representando os números em $[0, b-1]$ e a sua posição diz qual a potência da base b será multiplicada pelo número que o símbolo representa.
Uma definição genérica que inclui os números racionais pode ser:

$$(a_n a_{n-1} \cdots a_1 a_0, c_1 c_2 c_3 \cdots)_b = \sum_{k=0}^{n} a_k b^k + \sum_{k=1}^{\infty} c_k b^{-k}.$$

Então por exemplo, na base 2, temos que:

$$(11{,}001)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

Note que falta, por exemplo, o sinal de positivo e negativo, que é algo que podemos adicionar facilmente.

Vantagens e desvantagens

Vantanges:

- Podemos representar todo número racional com no máximo $log_b n + 1$ símbolos na parte inteira.
- Operações aritméticas podem ser decompostas por posição (como fazemos multiplicação e adição no papel)
- Todo número tem uma representação única (ou pelo menos uma representação padrão).

Desvantagens:

- Não é bijetivo; `01` e `1` ambos codificam o mesmo número.

Sistemas de numeração bijetivos

Se fizermos com que o 0 seja representado por uma string vazia, podemos utilizar os b símbolos pra representar os números de $[1, b]$ em cada posição. Assim, em um sistema decimal bijetivo:

- Decimal 10 seria representado por um símbolo como A.
- Decimal 11 seria 11, 20 vira 1A, etc.

Um exemplo de um sistema bijetivo assim é o usado em planilhas:

- A, B, C, D, ... Y, Z, AA, AB, AC, ..., ZX, ZZ, AAA, ...

Mas e a maratona?

É muito comum os problemas utilizarem as propriedades dos sistemas de numeração. Exemplos incluem:

- Ler um número de muitos dígitos (por exemplo 10^5) fazendo processamento dígito a dígito. Exemplo: Tire o módulo 3 de um número grande e imprima o resultado.
- Sequências onde é necessário levar em conta o tamanho da representação de cada um dos números. O fato do decimal não ser bijetivo causa problemas aqui que devem ser tratados. Exemplo: Ache o dígito X da sequëncia $1234567891011\ldots$
- Conte quantas vezes aparece determinado dígito respeitada algumas restrições. Muitas vezes dá pra combinar com programação dinâmica, veja PDs de dígito.
- Lidar com números grandes em geral (representar em vetor, fazer operações de forma rápida).

Bases negativas

A base pode ser negativa, e o que acontece é que em metade das posições, os números subtream ao invés de somar. Por exemplo no sistema negadecimal, o intervalo $[-5,5]$ é representado por

$$15, 16, 17, 18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 190, 191, 192, 193, 194, 195.$$

Por exemplo o número 15 é interpretado como:

$$1 \cdot (-10)^1 + 5 \cdot (-10)^0 = -10 + 5 = -5.$$

Todo número pode ser representado de forma única, e o algoritmo é o mesmo de outras bases, por meio da aplicação contínua de módulo e divisão. A vantagem é que não é necessário um sinal.

Implementação de base negativa

Código baseneg.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    cin.tie(0); ios_base::sync_with_stdio(0);
    ll n; cin >> n;
    string ans;
    while (n) {
        int r = n % -10;
        n /= -10;
        if (r < 0) { r += 10; n += 1; }
        ans = char(r+'0') + ans;
    }
    cout << ans << "\n";
}
```

Base fibonacci

- Todo inteiro pode ser representado **unicamente** pela soma de números de Fibonacci onde quaisquer dois não podem ser consecutivos na sequência. Assim, podemos fazer uma sequência de 0s e 1s para representar um inteiro.
- O inteiro 12 pode ser representado por 10101, ou seja, a soma de $F(2) + F(4) + F(6)$, ou $1 + 3 + 8$. Podemos representar números negativos se utilizarmos o sistema negafibonacci.
- Isso pode ser feito de forma gananciosa, isto é, pegando o maior número de Fibonacci menor que o número e ir subtraíndo.

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`

```
#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 Funções Fundamentais

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

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

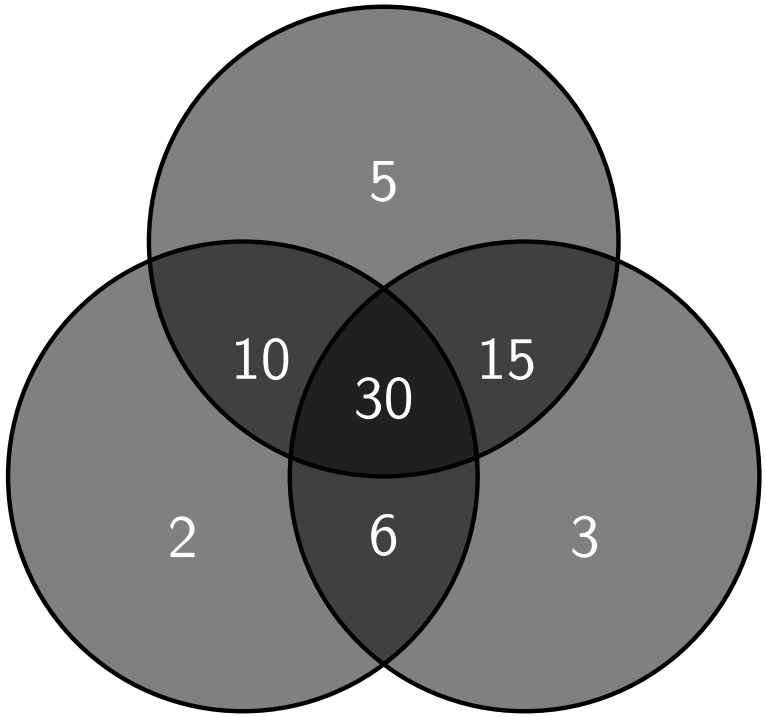
Implementação da função totiente de Euler

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"; }
}
```

Princípio da inclusão-exclusão

Dado um problema onde queremos achar quantos números de um intervalo $[1, n]$ são múltiplos de 2, 3 ou 5, como faz? Temos que imaginar os múltiplos do intervalo $[1, n]$ como um diagrama:



Assim, podemos concluir que a quantidade é dada por:

$$C_{2,3,5}(N) = C_2 + C_3 + C_5 - C_{10} - C_{15} - C_6 + C_{30}$$

Onde C_k é o número de múltiplos de k em $[1, n]$, ou seja, $\lfloor n/k \rfloor$.

Princípio da inclusão-exclusão generalizado

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{i<j} |A_i \cap A_j| + \ldots + (-1)^n \sum_{i<j} |A_1 \cap \ldots \cap A_n|$$

- Como implementar computacionalmente?
- Essa conta se resume a:
  - Pegar todos os subconjuntos.
  - Verificar o tamanho do subconjunto atual e determinar o sinal.
  - Obter o resultado da intersecção nos elementos.
  - Multiplicar o resultado pelo sinal.
  - Somar no total.
- Implementação é deixada ao leitor.

Aritmética Modular

Quero contar, mas não cabe no meu inteiro!

- Muitos problemas pedem contagens, mas geralmente o número pode ser gigantesco e não cabe no inteiro.
- Para resolver esse problema, quem cria os problemas geralmente usa aritmética modular usando um módulo **primo**.
- Isso significa que a resposta serve somente para provar que você sabe contar, mas não é o número exato.

Invariantes do módulo

Invariante é uma propriedade de um objeto matemático que permanece inalterada depois de operações ou transformações de algum tipo. Dentro da aritmética modular, temos várias operações onde as classes dos números são invariantes:

- $a + m \pmod{m} = a \pmod{m}$.
- $a + b \pmod{m} = (a \pmod{m}) + (b \pmod{m}) \pmod{m}$.
- $a * b \pmod{m} = (a \pmod{m}) * (b \pmod{m}) \pmod{m}$.
- $a - b \pmod{m} = (a \pmod{m}) - (b \pmod{m}) + m \pmod{m}$.
- $a^b \pmod{m} = (a \pmod{m})^b \pmod{m}$.

Cuidado com a subtração!

No C e no C++, o módulo de um número negativo com módulo positivo é negativo, então some o módulo para garantir números positivos. O Python não compartilha do mesmo comportamento.

Código `negmod.cpp`

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int a = ((5 - 8) % 10 - 14) % 10;
    cout << a << " " << (a + 10) % 10 << "\n";
}
```

Entrada	Saída
	-7 3

Código `negmod.py`

```python
print(a := ((5 - 8) % 10 - 14) % 10, a + 10 % 10)
```

Entrada	Saída
	3 3

Inverso multiplicativo

Resultado do pequeno teorema de Fermat:

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

onde p é **primo**.
Isso é bem útil para podermos dividir em módulo. Afinal,

$$\begin{aligned}
aa^{-1} &= 1 && \pmod{p} \\
aa^{-1} &= a^{p-1} && \pmod{p} \\
a^{-1} &= a^{p-1}a^{-1} && \pmod{p} \\
a^{-1} &= a^{p-2}. && \pmod{p}
\end{aligned}$$

Então usando a exponenciação, conseguimos o inverso multiplicativo em módulo (para p primo).

Teorema de Euler (da teoria dos números)

Euler generalizou o pequeno teorema de Fermat, e afirma 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.

Exponenciação binária

- Multiplicar pode ser caro: a^b é feito em $\mathcal{O}(b)$, e em problemas de contagem b pode ser grande.
- Podemos fazer exponenciação em $\lg b$ se aproveitando do fato que $a^b = (a^{\frac{b}{2}})^2$.
- Para o caso discreto:
  - $a^b = (a^{b/2})^2$ se b é par.
  - $a^b = a(a^{\lfloor b/2 \rfloor})^2$ caso contrário.
- Podemos fazer a conta $\pmod n$.

Implementação da exponenciação binária

Código pow.cpp

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

Entrada	Saída
2 100000	883109376
3 100000	522000001

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

Equações com Módulo

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!

Entendendo o MMC e o MDC com a fatoração prima

Frequentemente, é útil olhar para essas operações como operações nos (multi-)conjuntos de fatorações primas.
No caso do MMC, pegamos o maior subconjunto comum:

$$\mathrm{MDC}(60, 18) = \mathrm{MDC}(2^2 \cdot 3 \cdot 5, 2 \cdot 3^2) = 2 \cdot 3 = 6$$

No caso do MMC, para cada primo, escolhemos a maior potência dele em um dos números:

$$\mathrm{MMC}(60, 18) = 2^2 \cdot 3^2 \cdot 5 = 180.$$

Usando essas definições, muitos problemas de teoria de números podem ser resolvidos com programação dinâmica, através de definições recursivas.

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) usando ele, mas pra isso precisamos saber qual o valor 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";
    }
}
```

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.

Teste de Primalidade

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	24316731626151133
24316731626151133	262890056873052241
24316731626151137	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
26289005687305224 1	26289005687305224 1
9999999999999991	23 71 307 2251 6257 141623
9999999999999989	9999999999999989