UNIVERSIDADE DE SÃO PAULO

TRABALHO DE FORMATURA

Teoria dos Números e Computação: Uma abordagem utilizando problemas de competições de programação

Autor:
Antonio R. de Campos Junior

Supervisor:
Dr. Carlos Eduardo Ferreira

Tese apresentada em cumprimento dos requisitos para o curso Bacharel em Ciência da Computação

Instituto de Matemática e Estatística

25 de janeiro de 2016

"To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science."

Albert Einstein

Resumo

Teoria do Números é um vasto ramo da matemática que estuda números inteiros. Números primos, fatorização de números inteiros, funções aritméticas, são alguns dos tópicos mais estudados e também importantes para resolução de problemas computacionais.

Hoje em dia a importância da Teoria do Números na Computação é inquestionável, e desse modo, esse trabalho vem ilustrar como a teoria pode ser aplicada na criação de algoritmos para resolução de problemas computacionais, em especial problemas de competições de programação.

Equações diofantinas, Congruência Modular, Números de Fibonacci, são alguns dos assuntos que serão abordados nesse trabalho. Após a devida demostração da teoria serão exibidos alguns problemas de competições de programação que aplicam essa teoria, seguido da implementação e análise do algoritmo que resolve o problema abordado.

Agradecimentos

Gostaria de agradecer ao doutorando *Renzo Gonzalo Gomez Diaz* pelo auxílio na revisão dos textos e na seleção dos problemas. E um agradecimento especial ao Professor Doutor *Carlos Eduardo Ferreira* pelo suporte durante todo o desenvolvimento desse trabalho.

Sumário

Capítulo 1

Divisibilidade

1.1 Introdução

A noção de divisibilidade dos números inteiros é fundamental na **Teoria dos Números**. Nesta seção vamos descrever algumas definições e propriedades que serão utilizadas ao longo desse trabalho.

Definição 1 *A notação $d|n$ ("d **divide** n"), significa que existe um inteiro q, tal que, $n = dq$. Se $d|n$ dizemos que n é múltiplo de d. Caso n não seja múltiplo de d (ou seja, d não divide n), escrevemos $d \nmid n$.*

Definição 2 *A notação $d \bmod n$ ("d **módulo** n"), significa o resta da divisão de d por n.*

Proposição 1 $d|n$, $d|m \Rightarrow d|(n + m)$

Demonstração: Se $d|n$ e $d|m$, então existem inteiros q e k, tal que, $n = qd$ e $m = kd$. Desse modo temos:
$(n + m) = qd + kd = (q + k)d \Rightarrow d|(n + m)$ □

Proposição 2 $d|(\frac{n}{m}) \Rightarrow dm|n$

Demonstração:
$d|(\frac{n}{m}) \Rightarrow \exists q \in \mathbb{Z} \mid \frac{n}{m} = qd$
$d|(\frac{n}{m}) \Rightarrow n = q(dm) \Rightarrow dm|n$ □

Proposição 3 *Dado um subconjunto dos inteiros $S = \{S_1, S_2, S_3, ..., S_n\}$ ordenado crescentemente, e um número inteiro d, tal que, $d|(S_i - S_{i-1})$, $2 \leq i \leq n$, temos que:*
$d|(S_i - S_j), \forall S_i, S_j \in S$.

Demonstração: Tome $S_i, S_j \in S$ quaisquer, e sem perda de generalidade assuma que $S_i \geq S_j$ (ie, $i \geq j$, pois S está ordenado crescentemente).
Como $i \geq j$, tome $r \in \mathbb{N}$ como sendo a diferença entre i e $j : i = j + r$.
Vamos agora provar por indução que $d|(S_{j+r} - S_j)$.
Para $r = 0$ ou $r = 1$ a demostração segue trivialmente.
Assuma que o corolário funciona para $(r - 1)$, ie, $d|(S_{j+r-1} - S_j)$.
Temos então que:
$d|(S_{j+r} - S_{j+r-1}) \Rightarrow d|(S_{j+r} - S_{j+r-1}) + (S_{j+r-1} - S_j)$ (▷ **Proposição 1**)
$d|(S_{j+r} - S_{j+r-1}) \Rightarrow d|(S_{j+r} - S_j)$ □

Proposição 4 *A **Proposição** 3 funciona mesmo se o conjunto S não estiver ordenado.*

Demonstração: Deixaremos a demostração a cargo do leitor.

Definição 3 *A operação **módulo** de dois inteiros a e b, dada por "$a \bmod b$", representa o resto da divisão de a por b.*

Teorema 1 (Teorema da Divisão) *Para todo número inteiro a e qualquer número inteiro positivo n, existem inteiros únicos q e r, tal que:*

$a = qn + r, 0 \leq r < n$

*O valor q ($q = \lfloor \frac{a}{n} \rfloor$) é chamado de **quociente** da divisão, e o valor r ($r = a \bmod n$) é chamado de **resto** (ou **resíduo**) da divisão.*

Demonstração: Suponha que q e r não sejam únicos, ie, que exista q^* e r^* tal que: $a = q^*n + r^*, 0 \leq r^* < n$.

$a = qn + r = q^*n + r^* \Rightarrow (r - r^*) = (q^* - q)n \Rightarrow n|(r - r^*)$ já que $n|(q^* - q)n$.

Porém, como $r \neq r^*$, e tanto r quanto r^* são menores que n, temos que:

$(r \bmod n) \neq (r^* \bmod n) \Rightarrow n \nmid (r - r^*)$.

Chegando numa contradição, e assim q e r são únicos. $\square$

Corolário 1 $d|n$, $d|m \Rightarrow d|(n \bmod m)$

Demonstração:

$d|n \Rightarrow n = k_1 d, k_1 \in \mathbb{Z}$

$d|m \Rightarrow m = k_2 d, k_2 \in \mathbb{Z}$

$n = qm + (n \bmod m) \Rightarrow (n \bmod m) = n - qm$ ($\triangleright$ Teorema 1)

$(n \bmod m) = k_1 d - qk_2 d = (k_1 - qk_2)d \Rightarrow d|(n \bmod m)$ $\square$

Corolário 2 $d|m$, $d|(n \bmod m) \Rightarrow d|n$

Demonstração:

$d|m \Rightarrow m = k_1 d, k_1 \in \mathbb{Z}$

$d|(n \bmod m) \Rightarrow (n \bmod m) = k_2 d, k_2 \in \mathbb{Z}$

$n = qm + (n \bmod m) \Rightarrow n = qk_1 d + k_2 d$ ($\triangleright$ Teorema 1)

$n = (qk_1 + k_2)d \Rightarrow d|n$ $\square$

1.1.1 Divisores

Nessa subseção mostraremos um algoritmo simples para calcular todos os divisores de um dado número inteiro positivo qualquer.

Teorema 2 *O número de divisores de $n \in \mathbb{Z}^+$ é da ordem de $O(\sqrt{n})$.*

Demonstração: Tome um divisor d de n qualquer, com $d > \sqrt{n}$. Dessa forma sabemos que existe um inteiro q, tal que $n = qd$ (observe que q também é divisor de n). Como $d > \sqrt{n}$ então $q < \sqrt{n}$. Assim, para qualquer divisor d de n maior que $\sqrt{n}$, existe exatamente um divisor q de n menor que $\sqrt{n}$ correspondente ao mesmo. O que implica que só existem no máximo $\sqrt{n}$ divisores maiores que $\sqrt{n}$. Por outro lado, claramente só existem $\sqrt{n}$ divisores menores que $\sqrt{n}$. Concluímos então que o número total de divisores de n é da ordem de $O(\sqrt{n})$. $\square$

Pseudocódigo:

Algorithm 1 Encontra todos os divisores de N

```
1:  procedure FINDDIVISORS (N)
2:      D ← ∅                              ▷ Conjunto D contém os divisores de N
3:      for (d = 1; d² ≤ N; d + +) do
4:          if d ∤ N then
5:              continue
6:          D ← D ∪ {d}
7:          q ← N/d
8:          if q ≠ d then
9:              D ← D ∪ {q}
10:     return D
```

Análise: O laço da linha 3 consome tempo $O(\sqrt{N})$, testando se os números menores que $\sqrt{N}$ são divisores. Na linha 7 são calculados os divisores correspondentes maiores que $\sqrt{N}$. E a condição da linha 8 garante que se N for quadrado perfeito, então é inserido $\sqrt{N}$ somente uma vez no conjunto D. Assim a complexidade total do algoritmo é $O(\sqrt{N})$.

1.2 Números Primos

Definição 4 *Todo número inteiro n (n > 1) que têm apenas dois divisores distintos (1 e n) é chamado de número primo. Se n (n > 1) não for primo, dizemos que n é número composto.*

Teorema 3 (Fatoração Única) *Um número natural qualquer $n > 1$, pode ser escrito unicamente como um produto da forma: $n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$, onde os p_i são números primos, $p_1 < p_2 < ... < p_k$, e os números a_i são inteiros positivos.*

Demonstração: Primeiro vamos mostrar que existe tal fatoração para um inteiro n qualquer. Se n for primo, então n já está fatorado. Se n for composto, então existe um primo p que divide n, ie, $n = pq$. Se q for primo então n estará fatorado, ou $n = p^1 q^1$ ou $n = p^2$ (se $p = q$). Caso q não seja primo, repetimos o mesmo processo para q, de modo que teremos uma fatoração de q no final do processo, e por consequência uma fatoração de n.

Para mostrar a unicidade, tome as duas fatorações de n, $p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$ e $q_1^{b_1} q_2^{b_2} ... q_r^{b_r}$. Por definição temos, $p_1 < p_2 < ... < p_k$ e $q_1 < q_2 < ... < q_r$.

Se $k < r$ então existe um primo em $\{q_1, q_2, ..., q_r\}$ que divide n, mas não aparece na fatoração $p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$, chegando numa contradição. Analogamente, podemos provar que se $k > r$ chegaremos em outra contradição. E assim, concluímos que $k = r$.

Por outro lado, todo primo que aparece na primeira fatoração deve também aparecer segunda fatoração (e vice versa). Pelo fato das fatorações estarem ordenadas, teremos que $p_i = q_i$, $1 \leq i \leq k$.

Como a primeira fatoração divide a segunda fatoração e vice versa, temos também que $a_i = b_i$, $1 \leq i \leq k$. $\square$

1.3 Máximo Divisor Comum

Definição 5 *O Máximo Divisor Comum de dois inteiros quaisquer a e b (com a ou b diferente de zero), denotado por $MDC(a, b)$, é o maior inteiro que divide ambos a e b. Se $MDC(a, b) = 1$ dizemos que a e b são primos entre si.*

Por definição, temos também que $MDC(a, 0) = a$.

Corolário 3 *Para números inteiros quaisquer a e b, $MDC(a, b) = MDC(b, a \bmod b)$*

Demonstração: Pelo **Corolário** 1 e 2, temos:

$d|a, d|b \Leftrightarrow d|b, d|(a \bmod b)$

Assim, qualquer divisor de a e b é também divisor de b e $(a \bmod b)$ (e vise versa), implicando que o **Máximo Divisor Comum** de a e b é igual ao **Máximo Divisor Comum** de b e $(a \bmod b)$. □

Proposição 5 $MDC(a, b) = d \Rightarrow MDC(\frac{a}{d}, \frac{b}{d}) = 1$

Demonstração: Para $d = 1$, a prova é trivial.

Suponha que $MDC(\frac{a}{d}, \frac{b}{d}) = r > 1$. Assim temos:

$r|\frac{a}{d} \Rightarrow dr|a$ (▷ **Proposição** 2)

$r|\frac{b}{d} \Rightarrow dr|b$ (▷ **Proposição** 2)

$r > 1 \Rightarrow dr > d \Rightarrow dr > MDC(a, b)$

Chegamos então numa contradição, pois dr é divisor comum de a e b, e dr é maior que o **Máximo Divisor Comum** de a e b. □

Proposição 6 *Para números inteiros quaisquer a e b, $MDC(a, b) = MDC(a, a \pm b)$*

Demonstração: A prova dessa expressão vem do fato de que qualquer divisor de a e b, é também divisor de $(a \pm b)$.

Proposição 7 *Para números inteiros quaisquer a e b, temos:*

$MDC(a, b) = 1 \Rightarrow MDC(a, bk) = MDC(a, k)$, com $k \in \mathbb{Z}$

Demonstração: A prova dessa expressão vem do fato de que qualquer divisor d de a e bk, é também divisor de k, pois d não divide b ($MDC(a, b) = 1$).

Corolário 4 *$MDC((p - 1)!, p) = 1$, para p primo qualquer.*

Demonstração: Tome um inteiro positivo $d < p$. Como p é primo, sabemos que $MDC(p, d) = 1$, ie, d não tem nenhum fator primo em comum com p. Assim o produto $(p - 1)!$ de todos os inteiros positivos menores que p também não terá nenhum fator primo em comum com p. □

1.3.1 Algoritmo de Euclides

A ideia principal do **Algoritmo de Euclides** é calcular recursivamente o **Máximo Divisor Comum** de dois números baseando-se no **Corolário** 3.

Pseudocódigo:

Algorithm 2 Algoritmo de Euclides

```
procedure MDC(a, b)
    if b = 0 then
        return a
    return MDC(b, a mod b)
```

Análise: Primeiro provaremos a corretude do algoritmo fazendo uma indução sobre o número de chamadas recursivas N.

Se $N = 0$, então $b = 0$, e desse modo $MDC(a, 0) = a$, ie, para esse caso base o algoritmo retorna o valor correto.

Agora assuma que $MDC(a, b)$ faça $N > 0$ chamadas recursivas. Pela hipótese de indução a chamada $MDC(b, a \bmod b)$ retorna o valor correto. Pelo **Corolário** 3 sabemos que $MDC(a, b) = MDC(b, a \bmod b)$, provando assim a corretude do algoritmo.

Agora faremos uma análise da complexidade do algoritmo.

O *Algoritmo de Euclides* consome tempo proporcional à $O(\log b)$. Para provar a análise do custo do algoritmo é preciso conhecer algumas propriedades da *Sequência de Fibonacci*, e desse modo, a demostração será feita no **Capítulo 3**, **Subseção 3.2.1**.

1.3.2 Teorema de Bézout

Proposição 8 *Seja o conjunto de combinações lineares positivas* $S := \{x \in \mathbb{Z}, x > 0 \mid x = ma + nb, m, n \in \mathbb{Z}\}$*, onde os números* a *e* b *são inteiros, e pelo menos um desses números é diferente de zero. Temos então que* $S \neq \emptyset$.

Demonstração: As combinações possíveis para a e b são:

$a > 0 \Rightarrow |a| = 1.a + 0.b$

$a < 0 \Rightarrow |a| = (-1).a + 0.b$

$b > 0 \Rightarrow |b| = 0.a + 1.b$

$b < 0 \Rightarrow |b| = 0.a + (-1).b$

Como não temos ambos a e b iguais à zero, então S deve conter pelo menos $|a|$ ou $|b|$, e assim $S \neq \emptyset$ □

Corolário 5 *Seja o conjunto de combinações lineares positivas* $S := \{x \in \mathbb{Z}, x > 0 \mid x = ma + nb, m, n \in \mathbb{Z}\}$*, onde os números* a *e* b *são inteiros, e pelo menos um desses números é diferente de zero. Temos então que o menor número* $d \in S$ *divide todos os elementos de* S.

Demonstração: Como $d \in S$, $\exists m, n \in \mathbb{Z} \mid d = ma + nb$.

Tome $x \in S$ qualquer. Pelo Teorema 1 $x = qd + r$, $0 \leq r < d$.

Suponha que $d \nmid x$, ie, $x \neq qd$ e $0 < r$. Como $x \in S$, $\exists m^*, n^* \in \mathbb{Z} \mid x = m^*a + n^*b$, e assim:

$x = m^*a + n^*b, x = qd + r \Rightarrow r = m^*a + n^*b - qd = m^*a + n^*b - q(ma + nb)$

$\Rightarrow r = (m^* - qm)a + (n^* - qn)b \Rightarrow r \in S$, pois $r > 0$

Chegamos numa contradição, pois $r \in S$, $d \in S$, $r < d$ e d é o menor elemento em S.

Desse modo, temos que d divide todos os elementos de S. □

Teorema 4 (Teorema de Bézout) $\forall\, a, b \in \mathbb{Z}$ *(com pelo menos um dos dois números diferente de zero),* $\exists\, x, y \in \mathbb{Z} \mid ax + by = mdc(a, b)$.

Demonstração: Tome o conjunto das combinações lineares de a e b:

$S := \{x \in \mathbb{Z}, x > 0 \mid x = ma + nb, m, n \in \mathbb{Z}\}$.

Pelo **Proposição** 8 sabemos que $S \neq \emptyset$. Como S contém somente números positivos e não é vazio, S está limitado inferiormente por *zero* e assim, S tem um elemento mínimo que chamaremos de d.

Como $d \in S$, então existe $u, v \in \mathbb{Z}$, tal que, $d = ua + vb$. Pelo **Corolário** 5, sabemos que d divide todos elementos em S, em particular:

d divide $|a|$ e $|b| \Rightarrow d|MDC(a, b) \Rightarrow 0 < d \leq MDC(a, b)$

Por outro lado, $MDC(a, b)$ também divide a e b:

$MDC(a, b)|a$ e $MDC(a, b)|b \Rightarrow MDC(a, b)|(ua + vb)$

$\Rightarrow MDC(a, b)|d \Rightarrow MDC(a, b) \leq d$

$MDC(a, b) \leq d$ e $d \leq MDC(a, b) \Rightarrow MDC(a, b) = d \Rightarrow MDC(a, b) \in S$ $\square$

1.4 Crivo de Erastóteles

O *Crivo de Erastótenes* é um algoritmo criado pelo matemático **Erastótenes** (a.C. 285-194 a.C.) para o cálculo de números primos até um certo valor limite N. O algoritmo mantém uma tabela com N elementos, e para cada primo, começando pelo número 2, marca na tabelo os números compostos múltiplos desses primos. Desse modo, ao final do algoritmo, os elementos não marcados são números primos.

Pseudocódigo:

Algorithm 3 Crivo de Erastótenes paro o cálculo de números primos

1: **procedure** CRIVOERASTÓTENES (N)
2: $\quad isPrime[] \leftarrow$ new Array$[N]$ $\qquad \triangleright$ $isPrime[]$ é um vetor booleano
3: $\quad isPrime[1] \leftarrow false$
4:
5: $\quad$ **for** $(p = 2; p \leq N; p++)$ **do**
6: $\qquad isPrime[p] \leftarrow true$
7:
8: $\quad$ **for** $(p = 2; p^2 \leq N; p++)$ **do**
9: $\qquad$ **if** $isPrime[p] = false$ **then**
10: $\qquad\quad$ **continue**
11: $\qquad$ **for** $(n = p^2; n \leq N; n = n + p)$ **do**
12: $\qquad\quad isPrime[n] \leftarrow false$
13:
14: $\quad$ **return** $isPrime[]$

Análise: Primeiro provaremos a corretude do algoritmo. Para isso provaremos que ao final do algoritmo teremos a seguinte implicação: $isPrime[n] = false \Leftrightarrow n$ é composto.

Todos os valores que n assume no laço da linha 11 são múltiplos de p, e assim temos que: $isPrime[n] = false \Rightarrow n$ é composto.

Agora tome um n composto qualquer menor que N. Sabemos que existe um primo $q \leq \sqrt{n} \leq \sqrt{N}$ que divide n. E desse modo, em uma das iterações do laço da linha 8 teremos $p = q$. Como q é primo então $isPrime[q] = true$, e desse modo o laço da linha 11 será executado, iterando sobre os múltiplos de q menores que N, ie, em alguma iteração será executada a operação $isPrime[n] \leftarrow false$, implicando assim em: $isPrime[n] = false \Leftarrow n$ é composto.

Agora faremos uma análise da complexidade do algoritmo.

O laço da linha 4 itera sobre todos os valores de 2 até N e assim consome tempo $O(N)$. Já o laço mais interno na linha 9 itera sobre todos os números entre p^2 e N que são múltiplos de p, e assim consome tempo $O(N/p)$.

Desse modo a complexidade conjunta dos laços das linhas 6 e 9, será $O(\sum_{p \text{ primo } \mid \ p \leq sqrt(N)} \frac{1}{p})$ (observe que o laço da linha 6 itera sobre os primos menores que $sqrt(N)$).

Como foi sugerido por *Leonhard Euler* no século XVIII, temos que $\sum_{p \text{ primo } \mid \ p \leq sqrt(N)} \frac{1}{p} = O(N \log\log N)$. E assim a complexidade final do algoritmo é $O(N \log\log N)$.

1.5 Equações Diofantinas

Equações Diofantinas são equações polinomiais com variáveis inteiras. Alguns exemplos são mostrados a seguir, sendo x, y, z incógnitas, e a, b, n constantes inteiras:

$ax + by = n$ ($\triangleright$ *Equação Diofantina Linear*)

$ax + by = MDC(a, b)$ ($\triangleright$ *Identidade de Bézout*)

$x^n + y^n = z^n$ ($\triangleright$ Equação base do *Último Teorema de Fermat*)

$x^2 - ny^2 = \pm 1$ ($\triangleright$ *Equação de Pell*)

Nesse trabalho abordaremos somente *Equações Diofantinas Lineares*, da forma:

$\sum_{i=1}^{k} a_i x_i = c$

onde c e a_i são constantes, e x_i variáveis inteiras.

Teorema 5 *Dados inteiros a, b, c , temos que:*

$MDC(a, b)|c \Leftrightarrow$ a Equação Diofantina $ax + by = c$, tem solução inteira.

Demonstração: Provaremos primeiro a ida da implicação.

Pelo Teorema 4 sabemos que existem inteiros x^* e y^*, tal que, $ax^* + by^* = MDC(a, b)$. Logo:

$MDC(a, b)|c \Rightarrow \exists! q \in \mathbb{Z} \mid c = MDC(a, b)q \Rightarrow a(x^*q) + b(y^*q) = MDC(a, b)q = c$

Provaremos agora a volta da implicação.

Sabemos que existem inteiros u e v, tal que, $a = MDC(a, b)u$ e $b = MDC(a, b)v$. Logo:

$ax + by = c$, tem solução inteira $\Rightarrow MDC(a, b)ux + MDC(a, b)vy = c$

$\Rightarrow MDC(a, b)(ux + vy) = c \Rightarrow MDC(a, b)|c$. $\square$

1.5.1 Algoritmo de Euclides Estendido

Algoritmo de Euclides Estendido é uma extensão do *Algoritmo de Euclides* que calcula não só o $MDC(a, b)$, para dados a e b, mas também encontra uma solução para a *Identidade de Bézout* $ax + by = MDC(a, b)$.

Pseudocódigo:

Algorithm 4 Algoritmo de Euclides Estendido

```
1: procedure ExtendedMDC(a, b)
2:     if a = 0 then
3:         return [b, 0, 1]
4:
5:     [d, x, y] ← ExtendedMDC(b mod a, a)
6:     x* ← y − ⌊b/a⌋x
7:     y* ← x
8:     return [d, x*, y*]
```

Análise: Primeiro mostraremos a corretude do algoritmo.

Em vez de retornar um inteiro, $ExtendedMDC(a, b)$ retorna uma tupla $[d, x, y]$, onde $d = MDC(a, b)$ e x,y são a solução da equação $ax+by = d = MDC(a, b)$. Sabemos que $[d, x, y]$ na linha 5 satisfaz a equação, $(b \bmod a)x + ay = d$, assim:

$(b \bmod a)x + ay = d \Rightarrow (b - \lfloor \frac{b}{a} \rfloor a)x + ay = d$

$\Rightarrow a(y - \lfloor \frac{b}{a} \rfloor x) + b(x) = ax^* + by^* = d = MDC(a, b)$

Observe que o *Algoritmo de Euclides Estendido* é baseado no *Algoritmo de Euclides*, tendo a mesma complexidade $O(\log(a + b))$.

Corolário 6 *Tome* $[d, x_0, y_0]$ *como sendo a tupla retornada pelo* $ExtendedMDC(a, b)$*, com* a*,* b*,* c *inteiros e* $MDC(a, b)|c$*. Então temos que todas as soluções da equação* $ax + by = c$ *são da forma:* $x = (x_0\frac{c}{d} + \frac{bq}{d})$*,* $y = (y_0\frac{c}{d} - \frac{aq}{d})$*, em que* $q \in \mathbb{Z}$*.*

Demonstração: Pelo *Algoritmo de Euclides Estendido* sabemos que $ax_0 + by_0 = d$. Sabemos também que $d|c$ por definição, ie, existe $k \in \mathbb{Z}$ tal que $c = kd$. Assim temos que $x = x_0k$ e $y = y_0k$ é solução, já que $a(x_0k) + b(y_0k) = kd = c$.

Agora tome a solução x^*, y^* qualquer $(ax^* + by^* = c)$. Subtraindo as últimas duas equações temos:

$a(x_0k - x^*) + b(y_0k - y^*) = 0 \Rightarrow a(x_0k - x^*) = b(y^* - y_0k) \Rightarrow a|[b(y^* - y_0k)]$

$\Rightarrow \frac{a}{MDC(a,b)}|(y^* - y_0k) \Rightarrow \exists! q \in \mathbb{Z}$, tal que $\frac{a}{MDC(a,b)}q = (y^* - y_0k)$

$\Rightarrow y^* = y_0k + \frac{a}{MDC(a,b)}q \Rightarrow y^* = y_0\frac{c}{d} - \frac{aq}{d}$

Analogamente, temos: $x^* = x_0\frac{c}{d} + \frac{bq}{d}$

Assim todas as soluções de $ax + by = c$ são da forma: $x = x_0\frac{c}{d} + \frac{bq}{d}$, $y = y_0\frac{c}{d} - \frac{aq}{d}$ $\square$

1.6 Problemas Propostos

1.6.1 UVA-543

543 - Goldbach's Conjecture

Resumo: É dado um número inteiro n $(6 \leq n < 10^6)$. O problema consiste em verificar se n pode ser escrito como a soma de dois números primos ímpares. E em caso positivo dizer quais são esses primos.

Solução: Para resolver esse problema basta rodar o **Algoritmo** 3 para $N = n$, e fazer uma varredura linear no vetor $isPrime[]$. Se existir um índice a ($6 \leq a \leq n$) tal que $isPrime[a]$ é $true$ e $isPrime[n-a]$ também é $true$, então o problema acima tem solução.

Pseudocódigo:

Algorithm 5 Sum of odd primes

```
1: procedure SumOfPrimes(n)
2:     isPrime[] ← CrivoErastotenes(n)
3:
4:     for i := 6 to n do
5:         if isPrime[i] e isPrime[n − i] then
6:             return [i, n − i]
7:
8:     return "No Solution"
```

Análise: O *Crivo de Erastótenes* consome tempo proporcional a $O(n \log \log n)$, e o laço na linha 4 consome tempo $O(n)$. Assim a complexidade do algoritmo $SumOfPrimes(n)$ é $O(n \log \log n)$.

1.6.2 CodeChef-GOC203

GOC203 - Fight for Attendence

Resumo: São dados inteiros a, b, c ($1 \leq a, b, c \leq 10^6$) e a equação $ax + by = c$. O problema consiste em determinar quando tal equação tem solução inteira.

Solução: Solução é decorrente do Teorema 5, bastando checar se $MDC(a, b)|c$.

Pseudocódigo:

Algorithm 6 Fight for Attendence

```
procedure EquationSolution(a, b, c)
    if MDC(a, b)|c then
        return true
    return false
```

Análise: O algoritmo tem a mesma complexidado do *Algoritmo de Euclides*, $O(\log{(a + b)})$.

1.6.3 UVA-10407

10407 - Simple Division

Resumo: Tome $P(S) := \{x \in \mathbb{Z} \mid \forall a, b \in S, a \equiv b (mod\ x)\}$ em que $S \subset \mathbb{Z}$.
O problema consiste em encontrar o valor máximo de $P(S)$ dado um conjunto S.

Solução: Seja $S = \{S_1, S_2, S_3, ..., S_n\}$, com $n = |S|$, o conjunto dado pelo problema (assumiremos que os valores de S estão ordenados crescentemente).

Tome um número qualquer $d \in P(S)$. Por definição temos que $\forall S_i, S_j \in S$, $S_i \equiv S_j (mod\ d) \Rightarrow (S_i - S_j) \equiv 0 (mod\ d) \Rightarrow d \mid (S_i - S_j)$.

Pelo **Proposição** 3 sabemos que:

$d | (S_i - S_{i-1}), \forall i \in \mathbb{N}, 2 \leq i \leq n \Rightarrow d | (S_i - S_j), \forall S_i, S_j \in S \Rightarrow d \in P(S)$.

E desse modo, para calcular o valor máximo de $P(S)$ só precisamos calcular o Máximo Divisor Comum das diferenças $(S_i - S_{i-1})$ com i variando de 2 a n $\square$.

Pseudocódigo:

Algorithm 7 Simple Division

```
procedure GETMAXIMUMVALUE (S)
    S ← sort(S)                        ▷ sort(X) retorna o conjunto X ordenado.
    maxValue ← 0
    for i := 2 to |S| do
        maxValue ← MDC(maxValue, S_i − S_{i−1})
    return maxValue
```

Análise: Pela **Proposição** 4 sabemos que não é preciso o conjunto S ser ordenado. Assim a complexidade do algoritmo final será $O(|S| \log(\max(S_i - S_{i-1})))$.

1.6.4 CodeChef-MAANDI

MAANDI - Maxim and Dividers

Resumo: Calcular quantos divisores de um número inteiro n $(1 \leq n \leq 10^9)$, contém os dígitos 4 e 7 na forma decimal. Por exemplo, para $n = 94$ os únicos divisores que contém tais dígitos são: $47, 94$.

Solução: Para esse problema, basta calcular todos os divisores de n, com o **Algoritmo** 1, e depois verificar quais deles contêm os dígitos 4 ou 7.

Pseudocódigo:

Algorithm 8 Maxim and Dividers

```
1:  procedure FINDOVERLUCKIDIVISORS (N)
2:      D ← FindDivisors(n)                              ▷ Algoritmo 1
3:      count ← 0
4:
5:      for each d ∈ D do
6:          hasDigits ← false
7:
8:          while d > 0 do
9:              resto ← d mod 10
10:             if resto = 4 || resto = 7 then
11:                 hasDigits ← true
12:             d ← d/10
13:
14:         if hasDigits then
15:             count ← count + 1
16:
17:     return count
```

Análise: O laço da linha 5 roda em tempo $O(\sqrt{n})$, já que o número de elementos em D é da ordem de $O(\sqrt{N})$. O número de dígitos de cada divisor d é da ordem de $O(\log_{10} d)$, ou melhor $O(\log n)$. Assim o laço da linha 8 consome tempo proporcional à $O(\log n)$ e a complexidade total do algoritmo é $O(\sqrt{n} \log n)$.

1.6.5 UVA-10090

10090 - Marbles

Resumo: É dado um inteiro n $(1 < n \leq 2 * 10^9)$ que corresponde ao número de bolinhas disponíveis que serão colocadas em caixas. São dados também dois tipos de caixas: A caixa do tipo 1 que pode armazenar n_1 bolinhas e custa c_1; E a caixa do tipo 2 que pode armazenar n_2 bolinhas e custa c_2.

O problema consiste em encontrar o custo mínimo para comprar caixas de forma que todas as caixas compradas estejam totalmente preenchidas e todas as n bolinhas estejam dentro de alguma caixa.

Solução: Imagine que compramos x caixas do tipo 1 e y caixas do tipo 2. Desse modo, o problema se resume em encontrar os valores x e y, tal que:

$n_1x + n_2y = n$, com $x, y \geq 0$, e que $c_1x + c_2y$ seja mínimo.

Como no enunciado do problema é garantido que existe uma solução, então pelo Teorema 5 sabemos que $MDC(n_1, n_2)|n$.

Pelo **Corolário** 6 sabemos que toda solução x e y são da forma:
$x = (x_0\frac{n}{d} + \frac{n_2q}{d})$ e $y = (y_0\frac{n}{d} - \frac{n_1q}{d})$, onde $d = MDC(n_1, n_2)$ e q é um inteiro qualquer.
Desse modo temos:

$x \geq 0 \Rightarrow (x_0\frac{n}{d} + \frac{n_2q}{d}) \geq 0 \Rightarrow q \geq \lceil -\frac{x_0n}{n_2} \rceil$

$y \geq 0 \Rightarrow (y_0\frac{n}{d} - \frac{n_1q}{d}) \geq 0 \Rightarrow q \leq \lfloor \frac{y_0n}{n_1} \rfloor$

$\lceil -\frac{x_0n}{n_2} \rceil \leq q \leq \lfloor \frac{y_0n}{n_1} \rfloor$

Queremos encontrar o valor $min\{c_1x + c_2y\}$, onde $min\{f\}$ é o valor mínimo que a expressão f assume. Portanto:

$min\{c_1x + c_2y\} = min\big\{c_1(x_0\frac{n}{d} + \frac{n_2q}{d})\big\}$

$min\{c_1x + c_2y\} = min\big\{q(\frac{n_2c_1 - n_1c_2}{d}) + n(\frac{c_1x_0 + c_2y_0}{d})\big\}$

$min\{c_1x + c_2y\} = min\big\{q(\frac{n_2c_1 - n_1c_2}{d})\big\} + n(\frac{c_1x_0 + c_2y_0}{d})$, já que $n(\frac{c_1x_0 + c_2y_0}{d})$ é constante.

Se $(\frac{n_2c_1 - n_1c_2}{d})$ for positivo, então $q(\frac{n_2c_1 - n_1c_2}{d})$ assume valor mínimo quando q é mínimo, ie, $q = \lceil -\frac{x_0n}{n_2} \rceil$.

Por outro lado, se $(\frac{n_2c_1 - n_1c_2}{d})$ for negativo, então $q(\frac{n_2c_1 - n_1c_2}{d})$ assume valor mínimo quando q é máximo, ie, $q = \lfloor \frac{y_0n}{n_1} \rfloor$.

Pseudocódigo:

Algorithm 9 Marbles

```
1: procedure FindMinimumPrice(n, c_1, n_1, c_2, n_2)
2:     [d, x_0, y_0] ← ExtendedMDC(n_1, n_2)
3:
4:     if ((n_2c_1 − n_1c_2)/d) ≥ 0 then
5:         return ⌈−(x_0n)/n_2⌉((n_2c_1 − n_1c_2)/d)
6:
7:     else
8:         return ⌊(y_0n)/n_1⌋((n_2c_1 − n_1c_2)/d)
```

Análise: O único trecho do algoritmo que não roda em tempo constante é a chamada $ExtendedMDC(n_1, n_2)$ na linha 2, logo a complexidade total do algoritmo é $O(\log(n_1 + n_2))$.

1.6.6 UVA-718

718 - Skyscraper Floors

Resumo: É dado um prédio com F andares (numerados de 0 até $F - 1$) e E elevadores. Cada elevador i tem uma posição inicial Y_i ($Y_i \geq 0$) e uma constante X_i ($X_i > 0$), de tal forma que os únicos andares que esse elevador consegue chegar são da forma, $Y_i + X_it$, com t inteiro. Cada elevador i não consegue atingir andares menores que Y_i e maiores ou iguais a F, ie, $Y_i \leq Y_i + X_it \leq F - 1$, ou melhor, $0 \leq t \leq \frac{F-1-Y_i}{X_i}$. Dado os valores F, E, e as constantes Y_i, X_i para cada elevador, o problema consiste em verificar se é possível ir do andar A até o andar B ($0 \leq A, B < F$) usando os E elevadores.

Solução: Primeiro imagine que temos um grafo bidirecionado com E vértices, onde cada vértice representa um elevador e cada aresta (u, v) nos indica que os elevadores u e v conseguem chegar em algum andar em comum. Sabemos quais elevadores atingem o andar A, basta verificar se $Y_i + X_i t = A$ tem solução t inteira. Analogamente sabemos quais elevadores atingem o andar B. Então só precisaríamos fazer uma busca (BFS ou DFS) nesse grafo e verificar se há um caminho de um elevador que atinge o andar A até algum elevador que atinge o andar B.

Porém, para esse problema, não entraremos em detalhe nos algoritmos envolvendo grafos. Nos focaremos na parte matemática do problema, que envolve descobrir quando dois elevadores conseguem chegar em algum andar em comum, nos possibilitando assim, construir o grafo e resolver o problema.

Dois elevadores u e v atingem um andar em comum, se existe inteiros t_u ($0 \leq t_u \leq \frac{F-1-Y_u}{X_u}$) e t_v ($0 \leq t_v \leq \frac{F-1-Y_v}{X_v}$), tal que $Y_u + X_u t_u = Y_v + X_v t_v$, o que nos dá a *Equação Diofantina Linear* $X_u t_u + (-X_v)tv = (Y_v - Y_u)$.

Vamos mostrar agora um método para calcular t_u e t_v, tal que $at_u + bt_v = c$, com $a = X_u$, $b = -X_v$ e c=$(Y_v - Y_u)$. Pelo Teorema 5, sabemos que essa equação tem solução, se e somente se, $MDC(a, b)|c$. Observe também que se $Y_u = Y_v$ os elevadores estarão conectados. Checaremos essas restrições no começo do algoritmo, e daqui para frente assumiremos que $MDC(a, b)|c$ e $Y_u \neq Y_v$.

Tome d, t_1, t_2 como sendo os valores retornados por $ExtendedMDC(a, b)$, temos então pelo **Corolário** 6 que todas as soluções da equação $at_u + bt_v = c$, são da forma $t_u = (t_1 \frac{c}{d} + \frac{bq}{d})$ e $t_v = (t_2 \frac{c}{d} - \frac{aq}{d})$, com $q \in \mathbb{Z}$. Logo:

$$t_u = (t_1 \tfrac{c}{d} + \tfrac{bq}{d}) \Rightarrow \tfrac{-t_1 c}{b} \leq q \leq \big[(\tfrac{F-1-Y_u}{X_u})d - t_1 c\big]\tfrac{1}{b}, \text{ já que } 0 \leq t_u \leq \tfrac{F-1-Y_u}{X_u}$$

Analogamente temos:

$$t_v = (t_2 \tfrac{c}{d} - \tfrac{aq}{d}) \Rightarrow \tfrac{t_2 c}{a} \geq q \geq \big[t_2 c - (\tfrac{F-1-Y_v}{X_v})d\big]\tfrac{1}{a}, \text{ já que } 0 \leq t_u \leq \tfrac{F-1-Y_u}{X_u}$$

Das duas inequações acima, temos:

$$max\Big(\tfrac{-t_1 c}{b}, \big[t_2 c - (\tfrac{F-1-Y_v}{X_v})d\big]\tfrac{1}{a}\Big) \leq q \leq min\Big(\tfrac{t_2 c}{a}, \big[(\tfrac{F-1-Y_u}{X_u})d - t_1 c\big]\tfrac{1}{b}\Big)$$

Portanto, se a inequação acima tiver solução inteira q, os elevadores u e v serão conectados pelo andar $Y_u + X_u t_u = Y_u + X_u\big[(t_1 + \frac{bq}{d})\frac{c}{d}\big]$.

Pseudocódigo:

Algorithm 10 Verifica se os elevadores u e v estão conexos.

1: **procedure** $ElevatorsConected(X_u, Y_u, X_v, Y_y, F)$
2: $\quad a \leftarrow X_u$
3: $\quad b \leftarrow -X_v$
4: $\quad c \leftarrow Y_v - Y_u$
5: $\quad [d, t_1, t_2] \leftarrow ExtendedMDC(a, b)$
6:
7: $\quad$ **if** $Y_u = Y_v$ **then**
8: $\quad\quad$ **return** $true$
9:
10: $\quad$ **if** $d \nmid c$ **then**
11: $\quad\quad$ **return** $false$
12: $\quad\quad\quad$ ▷ A partir desse ponto temos: $c \neq 0$ e $d|c$
13:
14: $\quad letf \leftarrow max\left(\frac{-t_1c}{b}, \left[t_2c - (\frac{F-1-Y_v}{X_v})d\right]\frac{1}{a}\right)$
15: $\quad right \leftarrow min\left(\frac{t_2c}{a}, \left[(\frac{F-1-Y_u}{X_u})d - t_1c\right]\frac{1}{b}\right)$
16:
17: $\quad$ **if** $\lceil left \rceil \leq \lfloor right \rfloor$ **then**
18: $\quad\quad$ **return** $true$
19:
20: $\quad$ **return** $false$

Análise: O único trecho do algoritmo que não roda em tempo constante é a chamada $ExtendedMDC(a, b)$ na linha 5, logo a complexidade total do algoritmo é $O(\log(a+b)) = O(\log(X_u + X_v))$.

Capítulo 2

Aritmética Modular

2.1 Congruência

Definição 6 *Para a e b inteiros, dizemos que a é congruente à b módulo m ($a \equiv b(mod\ m), m > 0$) se a e b produzem o mesmo resto na divisão por m (ie, $m|(a-b)$). Caso contrário ($m \nmid (a-b)$), dizemos que a não é congruente à b módulo m ($a \not\equiv b(mod\ m)$).*

Definição 7 *Dizemos que o conjunto de inteiros $S = \{s_1, s_2, ..., s_k\}$ é um sistema completo de resíduos modulo n se: $\forall a \in \mathbb{Z}, \exists! s_i \in S \mid a \equiv s_i(mod\ n)$*

Proposição 9 *Dados inteiros a, b, c, d com $MDC(c, d) = 1$, temos que:*
$ac \equiv bc(mod\ d) \Rightarrow a \equiv b(mod\ d)$.

Demonstração:
$ac \equiv bc(mod\ d) \Rightarrow d|(ac - bc) \Rightarrow d|c(a - b) \Rightarrow d|(a - b)$, já que $MDC(c, d) = 1$
$\Rightarrow a \equiv b(mod\ d)$. $\square$

Proposição 10 *O conjunto $R = \{0, 1, 2, 3, ..., n - 1\}$, é um sistema completo de resíduos módulo n.*

Demonstração: Pelo Teorema 1 sabemos que para qualquer inteiro a, existe q, r tal que, $a = qn + r, 0 \le r < n$. Assim, $a \equiv r(mod\ n)$, com $r \in R$. $\square$

Teorema 6 *Se o conjunto $S = \{s_0, s_1, s_2, ..., s_{k-1}\}$ é um sistema completo de resíduos módulo n, então $k = n$.*

Demonstração: Tome o conjunto $R = \{0, 1, 2, 3, ..., n - 1\}$. Pela **Proposição** 10 sabemos que R é um sistema completo de resíduos módulo n.

Podemos concluir então, que cada elemento s_i de S é congruente a exatamente um dos elementos r_i em R, o que nos garante $|S| \le |R|$. Por outro lado, o conjunto S é por definição um sistema completo de resíduos módulo n, e desse modo cada elemento r_i de R é congruente a exatamente um dos elementos s_i em S, o que nos garante $|R| \le |S|$. Assim, como $|S| \le |R|$ e $|R| \le |S|$, temos que $|R| = n = k = |S|$. $\square$

2.2 Congruência Linear

Definição 8 *Congruências da forma* $ax \equiv b(mod\ m)$, *onde* a, b *e* m *são inteiros e* x *é uma incógnita, são chamadas de Congruências Lineares.*

Definição 9 *Um inteiro* q *é chamado de* ***Resíduo Quadrático*** *módulo* n *se é congruente a um quadrado perfeito módulo* n*, ie, se existe* x*, tal que:* $x^2 \equiv q\ (mod\ n)$.

Proposição 11 $ax \equiv b(mod\ m)$ *tem solução* $\Rightarrow MDC(a, m)|b$

Demonstração: Suponha que $\exists x \in \mathbb{Z}$, tal que $ax \equiv b(mod\ m)$, assim temos:
$ax \equiv b(mod\ m) \Rightarrow m|(b - ax) \Rightarrow \exists! r \in \mathbb{Z}$ tal que $(b - ax) = mr$
$\Rightarrow b = mr + ax \Rightarrow MDC(a, m)|b$, pois $MDC(a, m)$ divide tanto a como m. $\square$

Proposição 12 $ax \equiv b(mod\ m)$ *tem solução* $\Leftarrow MDC(a, m)|b$

Demonstração:
$MDC(a, m)|b \Rightarrow (ax + my)|b$ ($\triangleright$ Teorema 4)
$\Rightarrow \exists! r \in \mathbb{Z}$ tal que, $b = (ax + my)r \Rightarrow b = (xr)a + (yr)m$
$\Rightarrow b \equiv xra + yrm\ (mod\ m) \Rightarrow b \equiv xra\ (mod\ m)$
E assim, a **Congruência Linear** $az \equiv b(mod\ m)$, tem solução $z = xr$. $\square$

Corolário 7 $ax \equiv b(mod\ m)$ *tem solução* $\Leftrightarrow MDC(a, m)|b$

Demonstração: Segue trivialmente das **Proposições** 11 e 12.

Teorema 7 *O conjunto* $S = \{s_0, s_1, s_2, ..., s_{n-1}\}$ *com* $s_i = im+p$, $m, n, p \in \mathbb{Z}$, *e* $MDC(m, n) = 1$ *é um sistema completo de resíduos módulo* n.

Demonstração: Primeiro provaremos que $\forall a \in \mathbb{Z}, \exists! s_i \in S \mid a \equiv s_i(mod\ n)$.
Tome um inteiro a qualquer e a Congruência Linear: $(a - p) \equiv xm(mod\ n)$. Pelo **Corolário** 7 sabemos que essa Congruência Linear tem solução, já que $MDC(m, n) = 1$. Assim:
$(a - p) \equiv xm(mod\ n) \Rightarrow a \equiv xm + p(mod\ n) \Rightarrow a \equiv im + p(mod\ n), i = x \bmod n$
$\Rightarrow a \equiv s_i(mod\ n)$
Agora provaremos que $s_i \not\equiv s_j(mod\ n)$ para $i \neq j$.
Tome s_i e s_j em S com $i \neq j$, $0 < |i - j| < n$. Claramente $(i - j) \not\equiv 0(mod\ n)$, já que que i e j são distintos e $0 \leq i, j < n$. Portanto $(i - j)m \not\equiv 0(mod\ n)$, já que $MDC(m, n) = 1$, e assim:
$(i - j)m \not\equiv 0(mod\ n) \Rightarrow im \not\equiv jm(mod\ n) \Rightarrow im + p \not\equiv jm + p(mod\ n)$
$\Rightarrow s_i \not\equiv s_j(mod\ n)$.
Disso segue que S é um sistema completo de resíduos módulo n. $\square$

2.3 Teoremas de Fermat e de Wilson

2.3.1 Teorema de Fermat

Teorema 8 (Pequeno Teorema de Fermat) *Dado um número primo qualquer p, temos que:* $a^{p-1} \equiv 1(\text{mod } p), \forall a \in \mathbb{Z} \mid MDC(a,p) = 1$

Demonstração: Tome os conjuntos $S = \{s_0, s_1, s_2, ..., s_{p-1}\}$, com $s_i = ai$, e $T = \{0, 1, 2, ..., p-1\}$. Claramente o conjunto T é um *Sistema completo de resíduos módulo* p. Pelo Teorema 7 sabemos que S também é um *Sistema completo de resíduos módulo* p, e assim, $\forall s_i \in S$, $\exists! t_j \in T, 0 \le t_j \le (p-1)$, tal que $s_i \equiv t_j (mod\, p)$. Dessa informação podemos derivar a seguinte congruência modular:

$s_1.s_2.s_3...s_{p-1} \equiv t_1.t_2.t_3...t_{p-1} (mod\, p)$ ($\triangleright$ Observe que o correspondente a s_0 é t_0)

$\Rightarrow a.2a.3a....(p-1)a \equiv 1.2.3....(p-1)(mod\, p) \Rightarrow a^{p-1}(p-1)! \equiv (p-1)!(mod\, p)$

E aplicando os **Corolários** 4 e **Proposição** 9, temos:

$a^{p-1} \equiv 1(mod\, p)$. $\square$

Teorema 9 *Dados os inteiros a e b quaisquer e um número primo p, com $MDC(a,p) = 1$, temos que:*

$a^b \equiv a^{b \text{ mod } (p-1)}(\text{mod } p)$

Demonstração: Pelo Teorema 1 podemos escrever $b = q(p-1)+r$, onde $r = b \text{ mod } (p-1)$. Assim temos:

$a^b = a^{q(p-1)+r} = a^{q(p-1)}a^r = (a^{p-1})^q a^r \Rightarrow a^b \equiv (a^{p-1})^q a^r (mod\, p)$

Pelo Teorema 8 temos $a^{p-1} \equiv 1(\text{mod } p)$. Logo:

$a^b \equiv (1)^q a^r \equiv a^r \equiv a^{b \text{ mod } (p-1)} (mod\, p)$. $\square$

2.3.2 Inverso Multiplicativo Modular

Definição 10 *Um inteiro x é chamado de Inverso Multiplicativo de a módulo m, se $ax \equiv 1$ $(mod\, m)$, para a e m inteiros.*

Teorema 10 *Se p é primo e a é primo entre si com p, então: $a^{p-2} \equiv a^{-1}$ $(mod\, p)$, ou seja, a^{p-2} é inverso multiplicativo de a módulo p.*

Demonstração: Pelo Teorema 8 sabemos que $a^{p-1} \equiv 1$ $(mod\, p)$. Sabemos também que existe um inverso multiplicativo a^{-1} de a módulo p, já que a e p são primos entre si. Desse modo, temos que:

$a^{p-1} \equiv 1$ $(mod\, p) \Rightarrow a^{p-1}a^{-1} \equiv 1.a^{-1}$ $(mod\, p) \Rightarrow a^{p-2} \equiv a^{-1}$ $(mod\, p)$.$\square$

2.3.3 Teorema de Wilson

Proposição 13 *Se p é um número primo, e a um inteiro tal que $1 \le a \le p-2$, então:*

$a^2 \equiv 1$ $(mod\, p) \Leftrightarrow a = \pm 1$

Demonstração: Claramente se $a = \pm 1$ então $a^2 \equiv 1$ $(mod\, p)$. Provaremos então somente a ida da implicação.

Sabemos que $a^2 \equiv 1$ $(mod\, p)$, desse modo temos que:

$a^2 \equiv 1\ (mod\ p) \Rightarrow (a^2 - 1) \equiv 0\ (mod\ p) \Rightarrow (a-1)(a+1) \equiv 0\ (mod\ p)$
$\Rightarrow (a-1) \equiv 0\ (mod\ p)$ ou $(a+1) \equiv 0\ (mod\ p)$
$\Rightarrow a \equiv 1\ (mod\ p)$ ou $a \equiv -1\ (mod\ p) \Rightarrow a = 1$ ou $a = -1$. □

Teorema 11 (Teorema de Wilson) *Se p é um número primo, então $(p-1)! \equiv -1\ (mod\ p)$*

Demonstração: Para $p = 2$ o resultado é trivial, então assumiremos que $p \geq 3$.

Pela **Proposição 13** sabemos que $a^2 \equiv 1\ (mod\ p) \Leftrightarrow a = \pm 1$ para $1 \leq a \leq p-2$, e assim o inverso multiplicativo de um número b em $2 \leq b \leq p-2$ é diferente de b. Podemos então agrupar os $\frac{p-3}{2}$ pares de inversos multiplicativos no intervalo $[2, p-2]$ de modo que teremos: $2.3....(p-2) \equiv 1\ (mod\ p)$.

Desse mode, temos que:
$(p-1)! \equiv 2.3.4...(p-2)(p-1) \equiv (p-1) \equiv -1\ (mod\ p)$. □

2.4 Exponenciação

2.4.1 Exponenciação Binária

Exponenciação Binária é um algoritmo muito usado em *Ciência da Computação* principalmente no campo da *Criptografia*.

O algoritmo recebe inteiros a e b, e calcula a^b, usando divisão e conquista sobre a seguinte equação:

$$a^b = \begin{cases} 1 & \text{se } b = 0 \\ (a^{\lfloor \frac{b}{2} \rfloor})^2 & \text{se } b \text{ for par} \\ a(a^{\lfloor \frac{b}{2} \rfloor})^2 & \text{se } b \text{ for ímpar} \end{cases}$$

Pseudocódigo:

Algorithm 11 Exponenciação Binária

```
procedure EXPBIN(a, b)
    if b = 0 then
        return 1

    pot ← EXPBIN(a, ⌊b/2⌋)
    pot ← pot^2

    if b ≡ 0(mod 2) then
        return pot
    else
        return a(pot)
```

Análise: O tempo $T(a,b)$ que o algoritmo consome é dado por: $T(a,b) = T(a, b/2) + O(1)$, em que $T(a, 0) = O(1)$.

Expandindo essa recursão é fácil perceber que a complexidade do algoritmo é $O(\log b)$.

2.4.2 Exponenciação Binária Modular

Exponenciação Binária Modular é uma variação do algoritmo *Exponenciação Binária* que calcula $a^b \bmod m$, para dodos inteiros a, b, e m. Em geral *Exponenciação Binária Modular* é mais usado que *Exponenciação Binária*, pelo fato da expressão a^b crescer rapidamente e causar *overflow*.

Pseudocódigo:

Algorithm 12 Exponenciação Modular

procedure $EXPMOD(a, b, m)$
  if $b = 0$ **then**
    return 1

  $pot \leftarrow EXPMOD(a, \lfloor \frac{b}{2} \rfloor, m)$
  $pot \leftarrow pot^2 \mod m$

  if $b \equiv 0 (mod\ 2)$ **then**
    return pot
  else
    return $a(pot) \mod m$

Análise: O tempo $T(a, b, m)$ que o algoritmo consome é dado por: $T(a, b, m) = T(a, b/2, m) + O(1)$, em que $T(a, 0, m) = O(1)$.

Assim esse algoritmo tem a mesma complexidade $O(\log b)$ da *Exponenciação Binária*.

2.4.3 Exponenciação de Matriz

Exponenciação de Matriz é uma outra variação do algoritmo *Exponenciação Binária* que calcula $A^b \mod m$, para dados inteiros b e m e a matriz quadrada $A_{n,n}$.

Definição 11 *A expressão "$A \mod m$", em que A é uma matriz e m um inteiro qualquer, representa uma matriz B com as mesmas dimensões que A, tal que: $B_{ij} = A_{ij} \mod m$, $\forall B_{ij} \in B$.*

Pseudocódigo:

Algorithm 13 Exponenciação de Matriz

procedure $EXPMAT(A, b, m)$
  if $b = 0$ **then**
    return I ▷ I representa a matriz identidade de dimensão n

  $P \leftarrow EXPMAT(A, \lfloor \frac{b}{2} \rfloor, m)$
  $P \leftarrow P^2 \mod m$

  if $b \equiv 0 (mod\ 2)$ **then**
    return P
  else
    return $A(P) \mod m$

Análise: A recursão desse algoritmo é similar a recursão da *Exponenciação Binária*, exceto pela operação de produto, que nesse caso são produtos de matrizes. O tempo gasto para multiplicar duas matrizes quadradas de dimensão n é $O(n^3)$. Assim, a complexidade total do algoritmo é $O(n^3 \log b)$.

2.5 Problemas Propostos

2.5.1 SPOJ-DCEPC11B

DCEPC11B - Boring Factorials

Resumo: São dados inteiros N $(1 \leq N \leq 2.10^9)$ e o número primo P $(1 < P \leq 2.10^9)$, de tal forma que a diferença entre N e P é pequena. O problema consiste em calcular $N! \bmod P$.

Solução: Pelo **Teorema de Wilson** sabemos que $(P-1)! \equiv -1(mod\ P)$.
Temos que o problema pode ser dividido em dois casos:

Caso 1: $N \geq P$
Nesse caso, teremos que:
$N! \equiv (P-1)! \prod_{i=P}^{N} i \equiv (-1) \prod_{i=P}^{N} i\ (mod\ P)$.

Caso 2: $N < P$
Nesse caso, teremos que:
$N! \equiv (P-1)! \prod_{i=N+1}^{P-1} i^{-1} \equiv (-1) \prod_{i=N+1}^{P-1} i^{-1}\ (mod\ P)$.

Para resolver o **caso 1** só precisamos aplicar o **Teorema de Wilson** e construir o produto $\prod_{i=P}^{N} i$ iterativamente. Para o **caso 2**, podemos usar o **Pequeno Teorema de Fermat** para calcular o inverso multiplicativo dos números no produto $\prod_{i=N+1}^{P-1} i^{-1}$.

Pseudocódigo:

Algorithm 14 Boring Factorials

```
1:  procedure BoringFact (N, P)
2:      solution ← −1
3:
4:      if N ≥ P then
5:          for (i = P; i ≤ N; i++) do
6:              solution ← (solution.i) mod P
7:      else
8:          for (i = N + 1; i ≤ P − 1; i++) do
9:              inverse ← MODEXP(i, P − 2, P)                ▷ Teorema 10
10:             solution ← (solution . inverse) mod P
11:
12:     return solution
```

Análise: A complexidade do laço da linha 5 é dado por $O(N-P)$. Já o laço da linha 8 consome tempo proporcional a $O((P-N)\log P)$, já que a complexidade do algoritmo $MODEXP()$ na linha 9 é $O(\log P)$.
Assim, a complexidade total do algoritmo é $O(|N-P|\log P)$.

2.5.2 CodeChef-IITK2P10

IITK2P10 - Chef and Pattern

Resumo: Tome a seguinte função $f_K : \mathbb{N}^* \longmapsto \mathbb{N}$:

$$f_K(x) = \begin{cases} 1 & \text{se } x = 1 \\ K & \text{se } x = 2 \\ \prod_{i=1}^{x-1} f_K(i) & \text{se } x \geq 3 \end{cases}$$

São dados dois números inteiros N, K ($1 \leq N \leq 10^9$, $1 \leq K \leq 10^5$). O problema consiste em calcular a expressão: $f_K(N) \bmod p$, em que $p = (10^9 + 7)$.

Solução: Escrevendo os valores dos primeiros termos que a função assume, temos: $f_K(1) = 1$, $f_K(2) = K$, $f_K(3) = K$, $f_K(4) = K^2$, $f_K(5) = K^4$, $f_K(6) = K^8$, $f_K(7) = K^{16}$.

Provaremos, por indução, que $f_K(N) = K^{2^{N-3}}$, $N \geq 3$.

Para os primeiros termos essa expressão é trivialmente verificada.

Assuma que a expressão funciona para algum número natural qualquer $(R-1) \geq 3$ ($f_K(R-1) = K^{2^{R-4}}$).

Nessas condições temos que:

$f_K(R) = \prod_{i=1}^{R-1} f_K(i) = 1.K.\prod_{i=3}^{R-1} f_K(i) = K\prod_{i=3}^{R-1} K^{2^{i-3}} = K\prod_{j=0}^{R-4} K^{2^j}$

$f_K(R) = KK^{\sum_{j=0}^{R-4} 2^j} = KK^{2^{R-3}-1} = K^{2^{R-3}}$ $\square$

Para calcular o valor de $f_K(N) \bmod p$, podemos aplicar o Teorema 9, já que p é um número primo e $MDC(p, K) = 1$:

$f_K(N) \bmod p = K^{2^{N-3}} \bmod p = K^{2^{N-3} \bmod (p-1)} \bmod p$

Reduzindo o problema, dessa maneira, em calcular: $K^{2^{N-3}} \bmod (10^9 + 7)$.

Pseudocódigo:

Algorithm 15 Chef and Pattern

```
1: procedure F (N, K)
2:     p ← (10^9 + 7)
3:     exp ← EXPMOD(2, N − 3, p − 1)                ▷ Algoritmo 12
4:     solution ← EXPMOD(K, exp, p)       ▷ solution = K^(2^(N−3) mod (p−1)) mod p
5:     return solution
```

Análise: Como vimos anteriormente, as linhas 3 e 4 do algoritmo consomem tempo proporcional à $O(n \log n)$, e assim a complexidade total é $O(n \log n)$.

2.5.3 CodeChef-CSUMD

SUMD - My Fair Coins

Resumo: Existe um número infinito de moedas de dois tipos: as de 1 centavo, e as de 2 centavos. Todas as moedas tem dois lados (*cara* e *coroa*) que representam o valor da moeda.

É dado um inteiro N ($1 \leq N \leq 10^9$). O problema consiste em calcular o número de arranjos lineares dessas moedas, de modo que a soma das moedas é igual à N. A única restrição é que a primeira moeda de cada arranjo seja *cara*.

Como esse número pode ser muito grande, a resposta tem que ser dada módulo m ($m = 10^7 + 9$).

Solução: Tome $f_1(N)$ como sendo o número de arranjos lineares com a primeira moeda sendo *cara*. Analogamente, tome $f_2(N)$ como sendo o número de arranjos lineares com a primeira moeda sendo *coroa*. Por último, tome $f(N) = f_1(N) + f_2(N)$, como sendo o número total de arranjos cuja a soma das moedas é N.

Nessas condições temos as seguintes **Proposições**:

Proposição 1: $f_1(N) = f_2(N)$
Prova: Para cada arranjo em f_1, se trocarmos a primeira moeda de *cara* para *coroa*, teremos um arranjo correspondente em f_2. E assim, há uma bijeção de f_1 para f_2.

Proposição 2: $f_1(N) = f_1(N-1) + f_2(N-1) + f_1(N-2) + f_2(N-2)$
Prova: Tome um arranjo A qualque em f_1. A primeira moeda de A pode ser tanto uma moeda *cara* de 1 centavo, como uma moeda *cara* de 2 centavos. O número de arranjos que começam com a moeda de 1 centavo é $f(N-1)$ e o número de arranjos que começam com a moeda de 2 centavos é $f(N-2)$. Assim temos:
$f_1(N) = f(N-1) + f(N-2) = f_1(N-1) + f_2(N-1) + f_1(N-2) + f_2(N-2)$.

Proposição 3: $f_1(N) = 2(f_1(N-1) + f_1(N-2))$
Prova: Decorrente das **Proposições 1** e **2**.

Claramente $f_1(1) = 1$, e $f_1(2) = 3$. Mostraremos um modo elegante de calcular $f_1(N)$ usando *Exponenciação de Matrizes*. Tome a matriz A com as seguintes entradas:

$$A_{2,2} = \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$$

Teremos então que:

$$\begin{bmatrix} f_1(N) & f_1(N-1) \end{bmatrix} = \begin{bmatrix} f_1(N-1) & f_1(N-2) \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} f_1(N) & f_1(N-1) \end{bmatrix} = \left(\begin{bmatrix} f_1(N-2) & f_1(N-3) \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix} \right) \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} f_1(N) & f_1(N-1) \end{bmatrix} = \begin{bmatrix} f_1(N-2) & f_1(N-3) \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}^2$$

Expandindo essa recursão até a base, teremos:

$$\begin{bmatrix} f_1(N) & f_1(N-1) \end{bmatrix} = \begin{bmatrix} f_1(2) & f_1(1) \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}^{N-2}$$

$$\begin{bmatrix} f_1(N) & f_1(N-1) \end{bmatrix} = \begin{bmatrix} 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}^{N-2}$$

Pseudocódigo:

Algorithm 16 My Fair Coins

```
procedure f_1(N, m)
    if N = 1 then
        return 1

    if N = 2 then
        return 3

    A ← [[2, 1], [2, 0]]

    B ← EXPMAT(A, N − 2, m)                ▷ Algoritmo 13

    return 3B_{0,0} + 1B_{1,0}
```

Análise: Esse algoritmo tem a mesma complexidade do **Algoritmo** 13, ou seja, $O(n^3 \log N)$ (onde n é a dimensão da matriz). Como nesse problema a matriz é sempre quadrada de dimensão $n = 2$, temos que a complexidade final do algoritmo será $O(\log N)$.

Capítulo 3

Funções Aritméticas

3.1 Φ de Euler

Definição 12 *A Função Totiente de Euler, denotada por $\Phi(n)$, é a função aritmética que conta o número de inteiros positivos menores ou iguais a n que são primos entre si com n (ou **coprimos**).*

$\Phi(n) := |\{x \in \mathbb{N}^* \mid MDC(x,n) = 1\}|$

Teorema 12 *$\Phi(n)$ é função multiplicativa, ie, $\Phi(mn) = \Phi(m)\Phi(n)$ para $MDC(m,n) = 1$.*

Demonstração: Primeiro vamos dispor os números de 1 à nm da seguinte maneira:

$$\begin{array}{lllll}
1 & (m+1) & (2m+1) & \dots & (n-1)m+1 \\
2 & (m+2) & (2m+2) & \dots & (n-1)m+2 \\
3 & (m+3) & (2m+3) & \dots & (n-1)m+3 \\
\vdots & & & & \\
m & 2m & 3m & \dots & nm
\end{array}$$

Tome a linha q, com os elementos $q, m+q, 2m+q, ..., (n-1)m+q$. Se o $MDC(m,q) = d > 1$, então nenhum termo dessa linha será primo com mn, já que:

$$MDC(m,q) \neq 1 \Rightarrow MDC(xm+q,q) \neq 1 \Rightarrow MDC(mn,q) \neq 1.$$

Logo, as únicas linhas da tabela que nos interessam são aquelas cujo primeiro termo é primo com m (observe que temos $\Phi(m)$ linhas que satisfazem essa condição).

Agora, para cada uma dessas $\Phi(m)$ linhas, queremos saber quantos termos são primos com n, já que todos elementos na linha são primos com m. Como $MDC(m,n) = 1$, pelo Teorema 7 sabemos que os termos $r, m+r, 2m+r, ..., (n-1)m+r$ formam um sistema completo de resíduos módulo n. Assim, cada uma dessas $\Phi(m)$ linhas contém $\Phi(n)$ elementos primos com n, e como eles são primos com m, são também primos com mn. E isto nos garante que $\Phi(m) = \Phi(m)\Phi(n)$. $\square$

Teorema 13 *$\Phi(p^k) = (p^k - p^{k-1})$, para p primo e k inteiro positivo.*

Demonstração: Como p é um número primo, para qualquer inteiro n, os únicos valores possíveis para $MDC(p^k, n)$ são: $1, p, p^2, ..., p^k$, e desse modo, se $MDC(p^k, n) \neq 1$ temos que $p|n$ (n é múltiplo de p). Assim, a quantidade de números não-primos e menores do que p^k é p^{k-1}.

Logo, temos que: $\Phi(p^k) = p^k - p^{k-1}$ $\square$

Teorema 14 (Fórmula Produto de Euler) $\Phi(n) = n\prod_{p|n}(1 - \frac{1}{p}) = n\prod_{p|n}(\frac{p-1}{p})$

Demonstração:

$\Phi(n) = \Phi(p_1^{a_1} p_2^{a_2} ... p_k^{a_k})$ ($\triangleright$ Teorema 3)
$\Phi(n) = \Phi(p_1^{a_1})\Phi(p_2^{a_2})...\Phi(p_k^{a_k})$ ($\triangleright$ Teorema 12)
$\Phi(n) = (p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1})...(p_k^{a_k} - p_k^{a_k-1})$ ($\triangleright$ Teorema 13)
$\Phi(n) = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}(1 - 1/p_1)(1 - 1/p_2)...(1 - 1/p_k)$
$\Phi(n) = n \prod_{p|n}(1 - \frac{1}{p})$ $\square$

3.1.1 Algoritmo Φ de Euler

Nessa subseção mostraremos um algoritmo para o cálculo de Φ **de Euler** para os primeiros N números naturais. O algoritmo baseia-se na idéia do **Crivo de Erastótenes** e no Teorema 14.

Pseudocódigo:

Algorithm 17 Calcula os primeiros N termos da função Φ

```
1:  procedure PHI(N)
2:      Φ[] ← newArray[N]
3:
4:      for (p = 1; p ≤ N; p++) do
5:          Φ[p] ← p
6:
7:      for (p = 2; p ≤ N; p++) do
8:          if Φ[p] ≠ p then                    ▷ Φ[p] ≠ p ⇔ p não é primo
9:              continue
10:
11:         for (n = p; n ≤ N; n = n + p) do
12:             Φ[n] ← Φ[n]((p−1)/p)
13:
14:     return Φ[]
```

Análise: Deixaremos a demonstração a cargo do leitor.
Dica: O laço mais interno, na linha 11 só é executado quando p é primo, e assim ele executa uma quantidade de vezes proporcional à: $E = \sum_{p \text{ primo}|p \leq N} \frac{N}{p}$.
Observe também que: $E \leq \sum_{p=2}^{N} \frac{N}{p} = O(N \log N)$.

Proposição 14 $\Phi(n^k) = n^{k-1}\Phi(n)$, *para inteiros positivos n e k.*

Demonstração: Pelo Teorema 14 sabemos que $\Phi(n) = n \prod_{p|n}(1 - \frac{1}{p}) = n \prod_{p|n}(\frac{p-1}{p})$. E assim, $\Phi(n^k) = n^k \prod_{p|n^k}(\frac{p-1}{p})$. Porém, um primo p divide n, se e somente se, p divide n^k, implicando em, $\prod_{p|n^k}(\frac{p-1}{p}) = \prod_{p|n}(\frac{p-1}{p})$.
Portanto, $\Phi(n^k) = n^k \prod_{p|n^k}(\frac{p-1}{p}) = n^k \prod_{p|n}(\frac{p-1}{p}) = n^{k-1}\Phi(n)$. $\square$

3.1.2 Teorema de Euler

Teorema 15 (Teorema de Euler) *Dados números inteiros a e n primos entre si, temos que: $a^{\Phi(n)} \equiv 1 (\text{mod } n)$. Observe que esse teorema é uma generalização do Teorema 8.*

Demonstração: Tome o conjunto $P = \{p_1, p_2, ..., p_{\Phi(n)}\}$, onde os elementos de P são os números distintos primos com n e menores que n. Claramente $p_1p_2...p_{\Phi(n)} \equiv 1(\text{mod } n)$.

Agora tome o conjunto $R = \{ap_1, ap_2, ..., ap_{\Phi(n)}\}$, em que $MDC(a, n) = 1$. Como a e n são primos entre si, então todos os elementos de R são também primos com n, e assim, teremos que $(ap_1)(ap_2)...(ap_{\Phi(n)}) \equiv 1(\text{mod } n)$.

Das congruências acima podemos concluir que:

$p_1p_2...p_{\Phi(n)} \equiv (ap_1)(ap_2)...(ap_{\Phi(n)})(\text{mod } n)$
$p_1p_2...p_{\Phi(n)} \equiv a^{\Phi}(n)p_1p_2...p_{\Phi(n)}(\text{mod } n)$

E assim, concluímos que $1 \equiv a^{\Phi}(n)(\text{mod } n)$, já que $p_1p_2...p_{\Phi(n)} \equiv 1(\text{mod } n)$. □

3.2 Sequência de Fibonacci

Definição 13 *A sequência de Fibonacci Fib_n é uma sequência de números inteiros positivos em que cada termo subsequente corresponde a soma dos dois termos anteriores.*

$$Fib_n := \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib_{n-1} + Fib_{n-2} & \text{se } n \geq 2 \end{cases}$$

Proposição 15 $MDC(Fib_n, Fib_{n-1}) = 1$, *para* $n \geq 2$

Demonstração: Tome os primeiros termos da sequência de Fibonacci: $1, 1, 2, 3, 5, 8, ...$ Claramente a expressão acima funciona para os primeiros termos. Assuma que a expressão funciona para um inteiro qualquer $(k-1) > 2$, ou seja, $MDC(Fib_{k-1}, Fib_{k-2}) = 1$.

Provaremos por indução que a expressão sempre funciona.
$MDC(Fib_k, Fib_{k-1}) = MDC(Fib_{k-1} + Fib_{k-2}, Fib_{k-1})$
$MDC(Fib_{k-1} + Fib_{k-2}, Fib_{k-1}) = MDC(Fib_{k-2}, Fib_{k-1})$ (▷ Pelo **Proposição** 6)
Logo, temos que:
$MDC(Fib_k, Fib_{k-1}) = MDC(Fib_{k-2}, Fib_{k-1}) = 1$ □

Proposição 16 $Fib_{m+n} = Fib_mFib_{n+1} + Fib_{m-1}Fib_n$

Demonstração: Provaremos esse corolário por indução no índice n.
A base da indução será, $n = 2$:
$Fib_{m+2} = Fib_m + Fim_{m+1} = Fib_m + Fib_m + Fib_{m-1}$
$Fib_{m+2} = 2Fib_m + 1Fib_{m-1} = Fib_mFib_3 + Fib_{m-1}Fib_2$
Assumindo que a expressão é válida para todos os valores menores que n, temos:
$Fib_{m+n} = Fib_{m+n-2} + Fib_{m+n-1}$
$Fib_{m+n} = (Fib_mFib_{n-1} + Fib_{m-1}Fib_{n-2}) + (Fib_mFib_n + Fib_{m-1}Fib_{n-1})$
$Fib_{m+n} = Fib_m(Fib_{n-1} + Fib_n) + Fib_{m-1}(Fib_{n-2} + Fib_{n-1})$
$Fib_{m+n} = Fib_mFib_{n+1} + Fib_{m-1}Fib_n$ □

Teorema 16 $MDC(Fib_m, Fib_n) = Fib_{MDC(m,n)}, \forall m, n \in \mathbb{Z}$

Demonstração:

$MDC(Fib_m, Fib_n) = MDC(Fib_m, Fib_{qm+r})$ ($\triangleright$ Teorema 1, $n = qm + r, 0 \leq r < n$)
$MDC(Fib_m, Fib_n) = MDC(Fib_m, Fib_{qm}Fib_{r+1} + Fib_{qm-1}Fib_r)$ ($\triangleright$ **Proposição** 16).
$MDC(Fib_m, Fib_n) = MDC(Fib_m, Fib_{qm-1}Fib_r)$

Pela **Proposição** 7 e sabendo que $MDC(Fib_m, Fib_{qm-1}) = 1$, temos:
$MDC(Fib_m, Fib_n) = MDC(Fib_m, Fib_r)$
$MDC(Fib_m, Fib_n) = MDC(Fib_m, Fib_{n \bmod m})$

Se tirarmos o símbolo funcional Fib, a última equação forma um passo do **Algoritmo de Euclides** ($MDC(m, n) = MDC(m, n \bmod m)$).

Podemos continuar esse processo até que o resto r se torne 0. O último resto não-nulo será exatamente o Máximo Divisor Comum do dois números originais.

Desse modo, aplicar o **Algoritmo de Euclides** em Fib_m e Fib_n funciona da mesma maneira que aplicar aos índices m e n. E assim, ao chegarmos na base da recursão, $MDC(m, n) = MDC(s, 0) = s$, teremos também: $MDC(Fib_m, Fib_n) = MDC(Fib_s, 0) = Fib_s = Fib_{MDC(m,n)}$ $\square$.

Teorema 17 $Fib_n = \frac{\sqrt{5}}{5}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$

Demonstração:

$Fib_{n+1} = Fib_n + Fib_{n-1}$
$Fib_{n+1} - kFib_n = Fib_n + Fib_{n-1} - kFib_n$
$Fib_{n+1} - kFib_n = Fib_n + Fib_{n-1} - kFib_n + (kFib_{n-1} - kFib_{n-1}) + (k^2Fib_{n-1} - k^2Fib_{n-1})$
$Fib_{n+1} - kFib_n = (1-k)(Fib_n - kFib_{n-1}) + (1 + k - k^2)Fib_{n-1}$

Se denotarmos as raízes de $k^2 - k - 1 = 0$ por k_1 e k_2, teremos que $k_1 = \frac{1-\sqrt{5}}{2}$ e $k_2 = \frac{1+\sqrt{5}}{2}$.
$Fib_{n+1} - k_1Fib_b = k_2(Fib_n - k_1Fib_{n-1})$
$Fib_{n+1} - k_2Fib_b = k_1(Fib_n - k_2Fib_{n-1})$

Por iterações sucessivas dessas duas equações teremos que:
$Fib_{n+1} - k_1Fib_b = k_2^n(Fib_1 - k_1Fib_0) = k_2^n$
$Fib_{n+1} - k_2Fib_b = k_1^n(Fib_1 - k_2Fib_0) = k_1^n$

Subtraindo membro à membro temos:
$Fib_n(k_2 - k_1) = k_2^n - k_1^n$
$Fib_n = \frac{k_2^n - k_1^n}{k_2 - k_1}$
$Fib_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{(\frac{1+\sqrt{5}}{2}) - (\frac{1-\sqrt{5}}{2})}$
$Fib_n = \frac{\sqrt{5}}{5}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$ $\square$

3.2.1 Análise do Algoritmo de Euclides

No **Algoritmo** 2, exceto a chamada recursiva, todas as operações são feitas em tempo constante. Assim, a complexidade do algoritmo $MDC(a, b)$ será proporcional ao número de chamadas recursivas que o algoritmo faz. Observe também que se $a \leq b$, então o algoritmo $MDC(a, b)$ fará uma chamada a mais do que se $a > b$ (na primeira iteração os parâmetros a e b seriam trocados de posição). Desse modo, sem perda de generalidade, assumiremos que $a > b$.

Proposição 17 *Se o algoritmo $MDC(a, b)$ faz N $(1 < N)$ chamadas recursivas, então: $a \geq Fib_{N+1}$ e $b \geq Fib_N$.*

Demonstração: Provaremos essa proposição por indução. A base da indução é $N = 1$. Como temos 1 chamada recursiva, então $b \neq 0$, ie, $b \geq 1 = Fib_1$. Como $a > b$, então $a \geq 1 = Fib_2$, provando assim a base da indução.

Agora assuma que $N > 1$, e que o algoritmo $MDC(a, b)$ faz N chamadas recursivas. Pelo Teorema 1 sabemos que existe q e r tal que $a = qb + r$, e $0 \leq r < b$. Assim, temos que $MDC(b, r)$ faz $N - 1$ chamadas recursivas.

Pela hipótese de indução, temos que $b \geq Fib_N$ e $r \geq Fib_{N-1}$. Além disso, como $a > b$, temos que $q > 1$.

Concluindo assim que, $a \geq b + r \geq Fib_N + Fib_{N-1} = Fib_{N+1}$. $\square$

Proposição 18 $Fib_N \geq \Phi^{N-1}$, *em que* $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ *é a proporção áurea.*

Demonstração: Provaremos essa proposição por indução. Para $N = 1$, temos: $Fib_1 = 1 \geq \Phi^0 = 1$.

Agora assuma que $N > 1$. Primeiro observe que $\Phi^2 = (\frac{1+\sqrt{5}}{2})^2 = (\frac{3+\sqrt{5}}{2}) = \Phi + 1$.

Pela hipótese de indução, temos que $Fib_N = Fib_{N-1} + Fib_{N-2} \geq \Phi^{N-2} + \Phi^{N-3} = \Phi^{N-3}(\Phi + 1) = \Phi^{N-1}$.$\square$

Pelas **Proposições** 17 e 18, temos que se $MDC(a, b)$ faz N chamadas recursivas, então $a \geq Fib_{N+1}$ e $b \geq Fib_N \geq \Phi^{N-1}$. Podemos concluir então que, $\log_\Phi b \geq N - 1$, ie, N é $O(\log b)$.

3.3 Problemas Propostos

3.3.1 UVA-11424

11424 - GCD - Extreme (I)

Resumo: É dado um inteiro positivo N ($1 < N < 200001$). O problema consiste em calcular o mais rápido possível a expressão:
$G(N) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} MDC(i, j)$.

Solução: Trivialmente a expressão acima pode ser calculada em tempo proporcional a $O(n^2 log(N))$, porém essa solução consome muito tempo e não será aceita no Judge Online. Vamos então mostrar uma solução mais eficiente.

Primeiramente reescrevemos a expressão acima da seguinte maneira:
$G(N) = \sum_{j=2}^{N} \sum_{i=1}^{j-1} MDC(i, j)$ ($\triangleright$ Observe que as expressões são equivalentes).
Tome agora a função $F(M) = \sum_{i=1}^{M-1} MDC(i, M)$. Com isso, $G(N) = \sum_{j=2}^{N} F(j)$.
Sabemos que todos os valores resultantes do método $MDC(i, M)$ calculados em $F(M)$ são divisores de M. Desse modo, podemos reescrever $F(M)$ da seguinte maneira:
$F(M) = \sum_{i=1}^{M-1} MDC(i, M) = \sum_{l=1}^{n} \lambda_l d_l$, em que, $d_1, d_2, ..., d_n$ são os divisores de M, λ_l é o número de vezes que o divisor d_l aparece na somatória $\sum_{i=1}^{M-1} MDC(i, N)$, e n é o número de divisores de M.

Pela **Proposição** 5 temos que: $MDC(i, M) = d_l \Rightarrow MDC(i/d_l, M/d_l) = 1$. Logo o número de vezes que o divisor d_l aparece na somatória será igual ao número de primos entre si com (M/d_l), ie, $\lambda_l = \Phi(M/d_l)$.

Reescrevendo novamente $F(M)$, temos:
$F(M) = \sum_{i=1}^{M-1} MDC(i, M) = \sum_{l=1}^{n} \lambda_l d_l = \sum_{l=1}^{n} \Phi(M/d_l)d_l$.
$G(N) = \sum_{j=2}^{N} \sum_{l=1}^{n} \Phi(j/d_l)d_l$ $\square$.

Pseudocódigo:

Algorithm 18 GCD - Extreme(I)

1: **procedure** G (N)
2: $\Phi[] \leftarrow PHI(N)$
3: $solution \leftarrow 0$
4: **for** j := 2 to N **do**
5: **for each** divisor d de j **do**
6: $solution \leftarrow solution + \Phi[j/d]d$
7: **return** $solution$

Análise: O método $PHI(N)$ na linha 2 consome tempo proporcional à $O(N\sqrt{N})$.
O número de divisores de j é proporcional à $O(\sqrt{N})$, já que $j \leq N$.
Assim a complexidade das linhas 4, 5, 6 do algoritmo é $O(N\sqrt{N})$.
Complexidade final do algoritmo: $O(N\sqrt{N})$.
OBS.: Para resolver o problema no Judge Online será preciso armazenar as soluções usando Programação Dinâmica.

3.3.2 TJU-3506

3506 - Euler Function

Resumo: São dados três números positivos n, m ($1 < n < 10^7$, $1 < m < 10^9$) e $d = 201004$. O problema consiste em calcular a expressão: $\Phi(n^m) \bmod d$.

Solução: Pela **Proposição** 14, temos:
$\Phi(n^m) \bmod d = (n^{m-1}\Phi(n)) \bmod d$
$\Phi(n^m) \bmod d = ((n^{m-1} \bmod d)(\Phi(n)) \bmod d) \bmod d)$
Desse modo, podemos calcular o primeiro fator do produto ($n^{m-1} \bmod d$) usando $EXPMOD()$ e o segundo fator com o **Algoritmo** 17.

Pseudocódigo:

Algorithm 19 Euler Functions

1: **procedure** $PhiEulerPotential(n, m, d)$
2: $\Phi[] \leftarrow PHI(n)$
3: $exp \leftarrow EXPMOD(n, m-1, d)$
4: $solution \leftarrow (exp\ \Phi[n]) \bmod d$
5: **return** $solution$

Análise: As linhas 3 e 4 do algoritmo consomem tempo proporcional a $O(\log m)$ e $O(1)$ respectivamente. Se precalcularmos o vetor $\Phi[]$, temos que a complexidade total para calcular cada instância do problema será: $O(\log m)$

3.3.3 CodeChef-IITK2P05

IITK2P05 - Factorization

Resumo: São dados um inteiro N ($2 \leq N \leq 10^{18}$) e o valor de $\Phi(N)$.
O problema consiste em fatorar N.

Solução: A solução trivial para fatorar N consome tempo proporcional a $O(\sqrt{N})$, porém para uma entrada na ordem de 10^{18} precisamos de um algoritmo mais eficiente.

Se N for primo, ie, $\Phi(N) = N - 1$, já temos a solução.

Assuma então que N é um número composto. Primeiro vamos iterar nos primeiros $\sqrt[3]{N}$ inteiros e remover todos os fatores primos de N nesse intervalo. Tome M como sendo o valor resultante.

Imagine que M tenha três ou mais fatores primos. Sabemos que M não tem nenhum fator primo menor que $\sqrt[3]{N}$, temos que: $M > (\sqrt[3]{N})^3 \Rightarrow M > N$ (contradição). Desse modo, temos que M tem no máximo dois fatores primos, e esses valores são maiores que $\sqrt[3]{N}$.

- **Caso 1:** M tem só um fator primo, ie, M é primo: Basta checar se $\Phi(M) = M - 1$
- **Caso 2:** M tem dois fatores primos iguais, $M = p^2$: Basta verificar se M é um quadrado perfeito. Pode ser feito facilmente com busca binária.
- **Caso 3:** M tem dois fatores primos distintos, $M = pq$: Se $M = pq$ então $\Phi(M) = (p-1)(q-1)$. Temos então um sistema com duas equações e duas incógnitas. Se resolvermos o sistema encontraremos a fatoração de M e assim a fatoração de N.

O único problema agora é calcular $\Phi(M)$ a partir de $\Phi(N)$. Assuma que $N = p_1^{a_1} p_2^{a_2} ... p_k^{a_k} M$, com $k \geq 0$ e p_i os fatores primos distintos de N removidos na primeira etapa do algoritmo. Temos então:

$N = p_1^{a_1} p_2^{a_2} ... p_k^{a_k} M \Rightarrow \Phi(N) = \Phi(p_1^{a_1} p_2^{a_2} ... p_k^{a_k} M)$

$N = p_1^{a_1} p_2^{a_2} ... p_k^{a_k} M \Rightarrow \Phi(N) = \Phi(p_1^{a_1})\Phi(p_2^{a_2})...\Phi(p_k^{a_k})\Phi(M)$ ($\triangleright$ Teorema 12)

$\Rightarrow \Phi(N) = (p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1})...(p_k^{a_k} - p_k^{a_k-1})\Phi(M)$ ($\triangleright$ Teorema 13)

$\Rightarrow \Phi(M) = \frac{\Phi(N)}{(p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1})...(p_k^{a_k} - p_k^{a_k-1})}$

Pseudocódigo:

Algorithm 20 Fatoração de N

1: **procedure** $Factorization(N, \Phi_N)$
2: $\quad S \leftarrow \emptyset$ $\triangleright$ S contém os fatores primos de N
3: $\quad M \leftarrow N$
4: $\quad \Phi_M \leftarrow \Phi_N$
5:
6: $\quad$ **if** $\Phi_N = N - 1$ **then** $\triangleright$ Se N for primo
7: $\qquad S \leftarrow S \cup \{N\}$
8: $\qquad$ **return** S
9:
10: $\quad$ **for each** p primo menor igual à $\sqrt[3]{N}$ **do**
11: $\qquad$ **while** $M \equiv 0(mod\ p)$ **do**
12: $\qquad\quad M \leftarrow \frac{M}{p}$
13: $\qquad\quad S \leftarrow S \cup \{p\}$
14:
15: $\quad$ **for each** $p \in S$ **do**
16: $\qquad \Phi_M \leftarrow \frac{\Phi_m}{p^a - p^{a-1}}$ $\triangleright$ $a :=$ número de vezes que o primo p é inserido em S
17:
18: $\quad$ **if** $\Phi_M = M - 1$ **then** $\triangleright$ Se M for primo
19: $\qquad S \leftarrow S \cup \{M\}$
20: $\qquad$ **return** S
21:
22: $\quad (p, q) \leftarrow System(M, \Phi_M)$ $\triangleright$ Resolve o sistema de 2 equações e 2 incógnitas
23: $\quad S \leftarrow S \cup \{p, q\}$
24: $\quad$ **return** S

Análise: O laço da linha 10 consome tempo proporcional a $O(\sqrt[3]{N})$. Já o laço da linha 11 consome tempo proporcional a $O(\log_p N)$, pois tem no máximo a iterações (a é o número de vezes que o fator primo p aparece em N) e assim, $p^a < N \Rightarrow a < \log_p N$.

As linhas 15-16 rodam em $O(log_p N)$, já que o número máximo de elementos distintos em S é $log_p N$ (p é o menor primo que divide N). E as linhas 18-23 rodam em $O(1)$.

Assim, o algoritmo total consome tempo proporcional a $O(\sqrt[3]{N} \log N)$. Observe que esse algoritmo é bem mais eficiente que o algoritmo trivial para fatoração $O(\sqrt{N})$.

3.3.4 CodeChef-PUPPYGCD

PUPPYGCD - Puppy and GCD

Resumo: São dados inteiros positivos N e D. O problema consiste em calcular o número de pares não-ordenados $\{A, B\}$, tal que $1 \leq A, B \leq N$ e $MDC(A, B) = D$.

Solução: Claramente se D for maior que N, não há nenhum par que satisfaz as condições. Logo, assumiremos que $D \leq N$.

Pela **Proposição** 5 sabemos que se $MDC(A, B) = D$ então $MDC(\frac{A}{D}, \frac{B}{D}) = 1$. Assim, podemos reduzir o problema da seguinte forma: calcular o número de pares não-ordenados $\{A, B\}$, tal que $1 \leq A, B \leq \frac{N}{D}$ e $MDC(A, B) = 1$. Como $\Phi(r)$ nos dá a quantidade de números menores ou igual a r coprimos com respeito a ele, temos que o número total de pares será igual a somatória de $\Phi(r)$, com $1 \leq r \leq \frac{N}{D}$. Observe que essa somatória nos dá somente a metade do número de pares, já que o problema consiste em calcular pares não-ordenados.

Pseudocódigo:

Algorithm 21 Puppy and GCD

```
1:  procedure CalculatePairs(N, P)
2:      if D > N then
3:          return 0
4:
5:      Φ[] ← PHI(N/D)
6:      count ← 0
7:
8:      for (A = 1; A ≤ N/D; A++) do
9:          count ← count + Φ[A]
10:
11:     count ← 2 count − 1
12:     return count
```

Análise: Se precalcularmos $\Phi[]$ teremos que o trecho mais custoso do algoritmo será o laço da linha 8, e assim a complexidade do algoritmo será $O(\frac{N}{D})$.

Obs.: Quando dobramos o valor de *count* na linha 11, tem um único $\{1, 1\}$ que é contado duas vezes, e por isso precisamos subtrair 1 do valor final. Esse par corresponde ao par $\{D, D\}$ do problema original.

3.3.5 CodeChef-MOREFB

71544 - Another Fibonacci

Resumo: São dados dois números inteiros N, K ($1 \leq N \leq 50000, 1 \leq K \leq N$) e um conjunto $S \subset \mathbb{N}$ com N elementos, tal que, $\forall s \in S, 1 \leq s \leq 10^9$.

Tome a seguinte função:

$F(S) = \sum_{A \subset S \text{ e } |A|=K} Fib(sum(A))$, onde $sum(A) = \sum_{a \in A} a$.

O problema consiste em calcular a expressão: $F(S) \bmod m$, onde $m = 99991$.

Solução: Pelo Teorema 17 temos que $Fib_n = \frac{\sqrt{5}}{5}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$.

Primeiro verificamos que 5 é *Resíduo Quadrático* módulo m, já que $5 \equiv 10104^2 (\bmod m)$ (esse valor pode ser encontrado iterando sobre todos valores menores que m), logo: $\sqrt{5} \equiv 10104 (\bmod m)$.

Verificamos também que 5 tem *Inverso Multiplicativo* módulo m, já que $MDC(5, m) = 1$. Assim, iterando sobre os valores até m, podemos encontrar 79993, que é *Inverso Multiplicativo* de 5, ie, $5 * 79993 \equiv 1 (\bmod m)$, ou simplesmente, $5^{-1} \equiv 79993 (\bmod m)$. Analogamente podemos calcular o *Inverso Multiplicativo* do 2 módulo n, $2^{-1} \equiv 49996 (\bmod m)$.

Substituindo esses valores na equação acima, temos:

$$Fib_n \equiv 10104 * 79993\big((10105 * 49996)^n - (-10103 * 49996)^n (\bmod m)$$

Como $10104 * 79993 \equiv 22019 (\bmod m)$, $10105 * 49996 \equiv 55048 (\bmod m)$ e $(-10103 * 49996) \equiv 44944 (\bmod m)$, temos que:

$Fib_n \equiv 22019(55048^n - 44944^n)(\bmod m)$

Para facilitar a demonstração usaremos as constantes $a = 22019$, $b = 55048$ e $c = 44944$, ie, $Fib_n \equiv a(b^n - c^n)(\bmod m)$. Vamos calcular $F(S) \bmod m$ em função de a, b e c.

Suponha que $N = 3$ e $S = \{s_1, s_2, s_3\}$.
Para $K = 1$, teremos:
$F(S) = Fib_{s_1} + Fib_{s_2} + Fib_{s_3} = a((b^{s_1} + b^{s_2} + b^{s_3}) - (c^{s_1} + c^{s_2} + c^{s_3}))$.
Para $K = 2$, teremos:
$F(S) = Fib_{s_1+s_2} + Fib_{s_1+s_3} + Fib_{s_2+s_3} = a((b^{s_1+s_2} + b^{s_1+s_2} + b^{s_2+s_3}) - (c^{s_1+s_2} + c^{s_1+s_3} + c^{s_2+s_3}))$.
Para $K = 3$, teremos:
$F(S) = Fib_{s_1+s_2+s_3} = a(b^{s_1+s_2+s_3} - c^{s_1+s_2+s_3})$.

Tome agora o polinômio:
$B(x) = (x - b^{s_1})(x - b^{s_2})(x - b^{s_3})$
$B(x) = x^3 + x^2(b^{s_1} + b^{s_2} + b^{s_3}) + x^1(b^{s_1+s_2} + b^{s_1+s_2} + b^{s_2+s_3}) + x^0(b^{s_1+s_2+s_3})$.
Percebemos, que a parte em $F(S)$ que contém b, é exatamente igual ao coeficiente do monômio de grau $N - K$ em $B(x)$.
Analogamente, podemos calcular a parte em $F(S)$ que contém c, com o polinômio $C(x) = (x - c^{s_1})(x - c^{s_2})(x - c^{s_3})$.

Assim, para resolver o problema de caso geral, basta criar os polinômios:
$B(x) = (x - b^{s_1})(x - b^{s_2})...(x - b^{s_N})$
$C(x) = (x - c^{s_1})(x - c^{s_2})...(x - c^{s_N})$

A solução trivial para construir os polinômios $B(x)$ e $C(x)$ consome tempo proporcional a $O(N^2)$, porém usando **Fast Fourier Transform**, podemos calcular tais polinômios em $O(N \log N)$.

Para mais informações sobre **Fast Fourier Transform** acesse:
http://mathworld.wolfram.com/FastFourierTransform.html.

Pseudocódigo:

Algorithm 22 Another Fibonacci

```
procedure F (S, N, K)
    m ← 99991
    a ← 22019
    b ← 55048
    c ← 44944

    [Bcoef, Ccoef] ← FastFourierTransform(S, N, K, m, a, b, c)

    return a(Bcoef − Coef) mod m
```

Análise: Na linha 7 é aplicado o algoritmo **Fast Fourier Transform** para calcular os coeficientes do monômio de grau $N - K$ dos polinômios $B(x)$ e $C(x)$ respectivamente. O complexidade final do algoritmo será igual a complexidade do algoritmo *Fast Fourier Transform* que é $O(N \log N)$.

3.3.6 Codeforces-227E

227E - Anniversary

Resumo: São dados quatro inteiros m, l, r, k ($1 \leq m \leq 10^9$, $1 \leq l \leq r \leq 10^{12}$, $2 \leq k \leq r-l+1$), e um conjunto $A = \{l, l+1, l+2, ..., r\}$. Dado um conjunto $S \subseteq A$ com k elementos, ie, $S = \{s_1, s_2, ..., s_k\}$, seja a função $F(S) = MDC(Fib_{s_1}, Fib_{s_2}, ..., Fib_{s_k})$.

O problema consiste em encontrar tal subconjunto S, de tal forma que $F(S)$ seja máxima. No final do algoritmo basta imprimir $F(S) \bmod m$.

Solução: Pelo Teorema 16 sabemos que, $MDC(Fib_{s_1}, Fib_{s_2}, ..., Fib_{s_k}) = Fib_{MDC(s_1, s_2, ..., s_k)}$. Desse modo, para maximizar $F(S)$, basta maximizar $MDC(s_1, s_2, ..., s_k)$, já que Fib_x é uma sequência crescente.

Suponha que o valor máximo para $MDC(s_1, s_2, ..., s_k)$ seja q. Como q divide pelo memos k inteiros $(s_1, s_2, ..., s_k)$ no conjunto A, então $\lfloor \frac{r}{q} \rfloor - \lceil \frac{l}{q} \rceil + 1 \geq k$, ie, o número de múltiplos de q entre l e r é maior ou igual à k.

Todos os valores que $\lfloor \frac{r}{q} \rfloor$ assume são da forma:

$i \in \{1, 2, 3, ..., \lfloor \sqrt{r} \rfloor\}$, se $\lfloor \frac{r}{q} \rfloor \leq \sqrt{r}$
$\lfloor \frac{r}{i} \rfloor$, com $i \in \{1, 2, 3, ..., \lfloor \sqrt{r} \rfloor\}$, se $\lfloor \frac{r}{q} \rfloor > \sqrt{r}$

Analogamente, podemos calcular todos os valores que $\lceil \frac{l}{q} \rceil$ assume.

Tendo todos esses valores armazenados em um conjunto Q, é possível calcular todos os valores distintos que $\lfloor \frac{r}{q} \rfloor - \lceil \frac{l}{q} \rceil + 1$ assume, e assim basta escolher o maior $q \in Q$, tal que $\lfloor \frac{r}{q} \rfloor - \lceil \frac{l}{q} \rceil + 1 \geq k$.

A partir do momento que calculamos q, ie, encontramos o valor máximo de $MDC(s_1, s_2, ..., s_k)$, precisamos de uma maneira eficiente de calcular $Fib_q \bmod m$. E para isso, usaremos **Exponenciação de Matriz** com a seguinte matriz:

$$A_{2,2} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Observe que: $\begin{bmatrix} Fib_q & Fib_{q-1} \end{bmatrix} = \begin{bmatrix} Fib_{q-1} & Fib_{q-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

E assim: $\begin{bmatrix} Fib_q & Fib_{q-1} \end{bmatrix} = \begin{bmatrix} Fib_2 & Fib_1 \end{bmatrix} A^{q-2} = \begin{bmatrix} 1 & 1 \end{bmatrix} A^{q-2}$

Pseudocódigo:

Algorithm 23 Anniversary

1: **procedure** $FindMaximumValue(m, r, l, k)$
2: $Q \leftarrow \emptyset$
3: $q \leftarrow 0$
4:
5: **for** $(i = 1; i \leq \sqrt{r}; i++)$ **do**
6: $Q \leftarrow Q \cup \{i\}$
7:
8: **for** $(i = 1; i \leq \sqrt{r}; i++)$ **do**
9: $Q \leftarrow Q \cup \{\lfloor \frac{r}{i} \rfloor\}$
10:
11: **for** $(i = 1; i \leq \sqrt{l}; i++)$ **do**
12: $Q \leftarrow Q \cup \{\lceil \frac{l}{i} \rceil\}$
13:
14: **for all** $q^* \in Q$ **do**
15: **if** $\lfloor \frac{r}{q^*} \rfloor - \lceil \frac{l}{q^*} \rceil + 1 \geq k$ **then**
16: $q \leftarrow max(q, q^*)$
17:
18: $A \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
19:
20: $B \leftarrow EXPMAT(A, q-2, m)$
21:
22: **return** $(B_{0,0} + B_{1,0}) \bmod m$

Análise: Os laços das linhas 5 e 8 consomem tempo proporcional a $O(\sqrt{r})$. O laço da linha 11 consome tempo proporcional a $O(\sqrt{l})$, que por sua vez é $O(\sqrt{r})$, já que $l \leq r$. Como é inserido um elemento em Q para cada iteração desses três primeiros laços, temos que no final do algoritmo o número de elementos em Q é da ordem de $O(\sqrt{r})$.

O laço da linha 14 consome tempo proporcional a $O(|Q|)$, ie, $O(\sqrt{r})$.

O algoritmo *EXPMAT()* roda em $O(\log q)$, ou simplesmente, $O(\log r)$, já que $q \leq r$.

Assim, a complexidade total do algoritmo é $O(\sqrt{r} + \log r)$.

Capítulo 4

Conclusão

Esse trabalho vem preencher um pouco da falta de um bom material didático sobre Teoria do Número aplicada em Competições de Programação.

Pelo fato de ser bem modular, é possível incrementar o conteúdo do mesmo com problemas e teorias de maneira coerente.

Assim, espero que esse trabalho ajude outros alunos da mesma forma que teria me ajudado no começo dos meus estudos.

Apêndice A

Curiosidades da ACM-ICPC

ACM-ICPC (International Collegiate Programming Contest) é uma competição de programação de várias etapas e baseada em equipe. O principal objetivo é encontrar algoritmos eficientes, que resolvem os problemas abordados pela competição, o mais rápido possível.

Nos últimos anos a ACM-ICPC teve um crescimento significativo. Se compararmos o número de competidores, temos que de 1997 (ano em que começou o patrocínio da IBM) até 2014 houve um aumento maior que 1500%, totalizando 38160 competidores de 2534 universidades em 101 países ao redor do mundo.

Para mais informações sobre as competições passadas acesse icpc.baylor.edu.

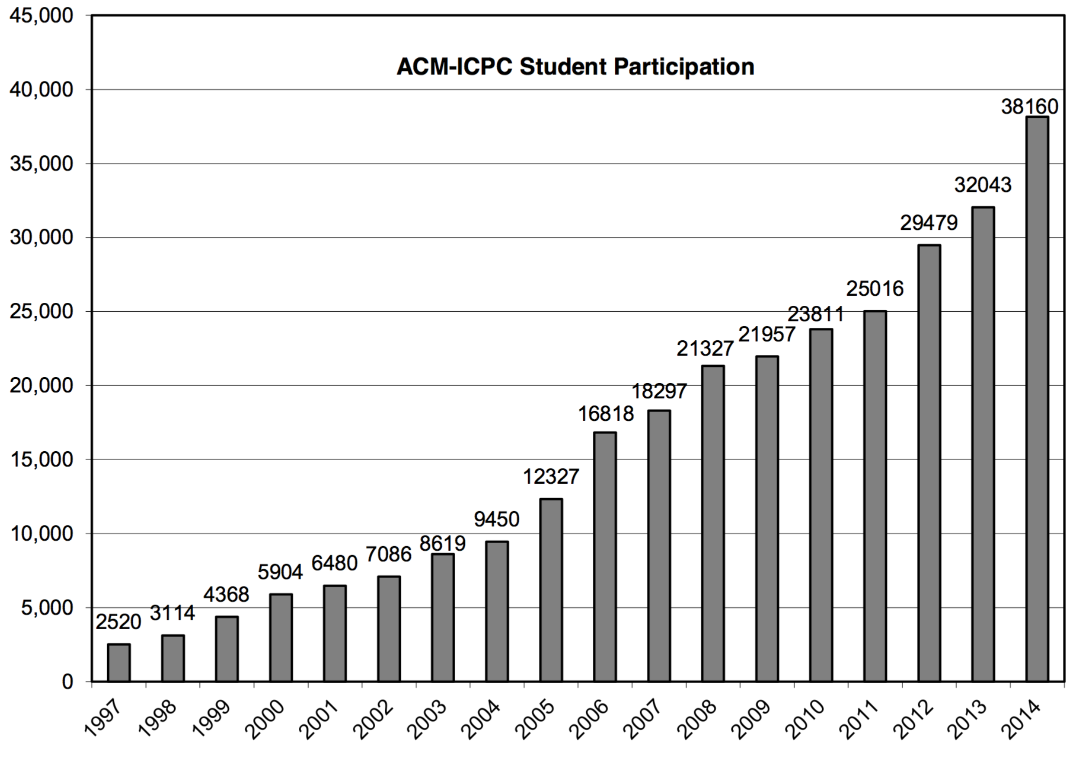

FIGURA A.1: Crescimento do número de participantes por ano.

Apêndice B

Juízes Online (Online Judges)

Online Judges são plataformas online que contam com um banco de dados com diversos tipos de problemas de competições de programação, e com um sistema de correção online.

Para comprovar que seu programa resolve o problema dado, basta enviar o código fonte da sua solução (em geral escrito em C++ ou JAVA) para uma dessas plataformas.

Alguns desses Online Judges são citados em seguida.

B.1 Ahmed-Aly

Juíz online que permite criar e participar de competições de programação online com problemas de outros *Online Judges*.

Também possibilita o gerenciamento das suas competições de programação e de seus amigos.

Para mais problemas de *Teoria dos Números* acesse http://a2oj.com/Category.jsp?ID=41.

Site: http://a2oj.com/

B.2 UVa

Criado em 1995 pelo matemático Miguel Ángel Revilla, é atualmente um dos Online Judges mais famoso entre os participantes da ACM-ICPC.

É hospedado pela Universidade de Valhadolide e conta com mais de 100000 usuários registrados.

Site: https://uva.onlinejudge.org/

B.3 URI

Projeto desenvolvido pelo Departamento de Ciência da Computação da Universidade Regional Integrada. Contando com um enorme repositório com problemas de competições de programação, o principal objetivo desse projeto é proporcionar uma plataforma para a prática de programação e compartilhamento de conhecimentos.

Site: https://www.urionlinejudge.com.br/

B.4 Topcoder

Empresa que administra competições de programação nas linguagens Java, C++ e C#.

É responsável também por aplicar competições de design e desenvolvimento de software.

Site: https://www.topcoder.com/

B.5 Codeforces

Site Russo dedicado às competições de programação.

Em 2013, Codeforces superou Topcoder com relação ao número de usuários ativos, apesar de ter sido criado quase 10 anos depois.

O estilo de problemas que esse site aplica é similar aos problemas encontrados na ACM-ICPC.

Site: http://codeforces.com/

B.6 CodeChef

Iniciativa educacional sem fins lucrativos lançada em 2009 pela Direct.

É uma plataforma de progamação competitiva que suporta mais de 35 linguagens de programação.

Site: https://www.codechef.com/

Bibliografia

[1] OLIVEIRA SANTOS, José Plínio *Introdução à Teoria dos Números* IMPA, 1998.
72-85 pg

[2] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford *Algoritmo Teoria e Prática* CAMPUS, 2002.