Universidade Estadual da Paraíba
Centro de Ciências e Tecnologia
Departamento de Computação

**Disciplina : Técnica E Análises de Algoritmos**

Professor: Thiago Santana Batista

# Teoria dos Números

# Introdução

- A teoria dos números é provavelmente a mais interessante e bela área da matemática.

- A prova de Euclides de que existe um número infinito de números primos continua tão clara e elegante hoje como há mais de 2 mil anos atrás.

- Questões simples como checar se $a^n + b^n = c^n$ tem soluções para valores inteiros de a, b e c. Na verdade, isso é o teorema de Fermat.

- Teoria dos números é extremamente importante não só na vida real como para computadores. Veremos algumas teorias interessantes.

## Números Primos

- Um número primo é um inteiro p > 1 que é divisível por 1 e por ele mesmo.

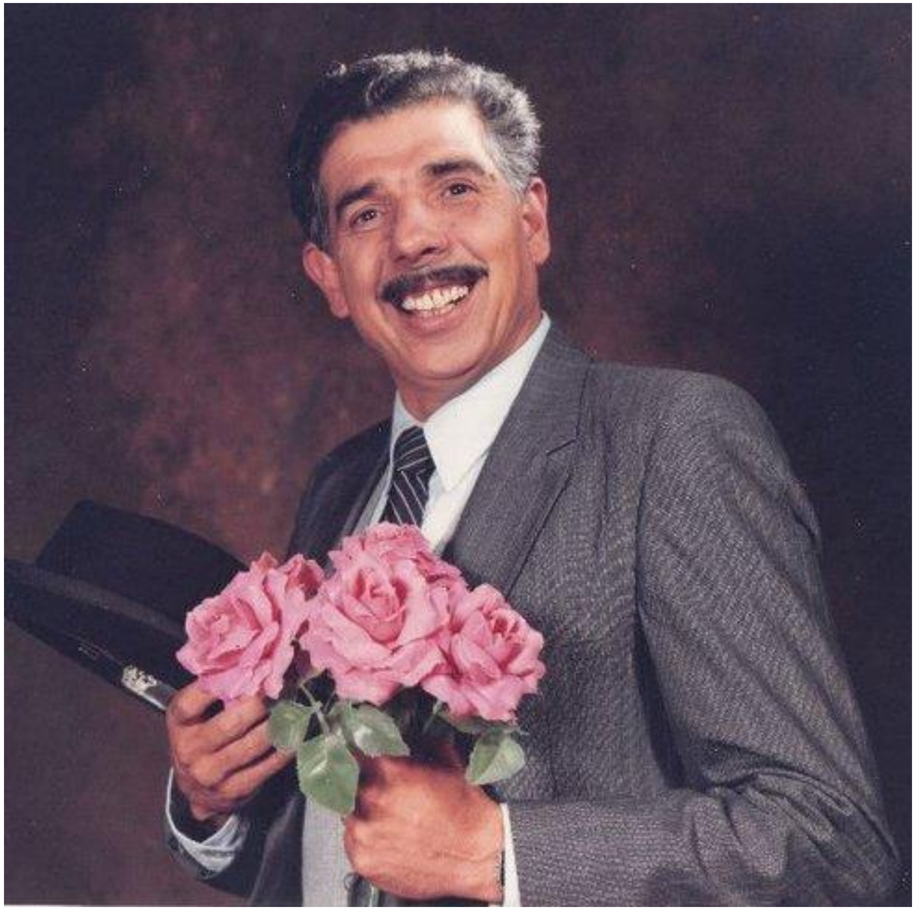
- Uma outra forma de representar:
  - Se p é primo, então p = a.b para inteiros onde a <= b o que implica que a = 1 e b = p.

- Os 10 primeiros números primos:
  - 2, 3, 5, 7, 11, 13, 17, 19, 23 e 27

- Os números primos são extremamente importantes por causa da: teoria fundamental da aritmética.

## Números Primos

- <u>teoria fundamental da aritmética</u>



## Números Primos

- A teoria fundamental da aritmética declara que qualquer número inteiro pode ser expresso apenas de uma maneira como o produto de primos.

- Exemplo:
  - 105 = unicamente representado por 2*5*7.
  - 32 = unicamente representado por 2*2*2*2*2.

- Esse conjunto único de números multiplicados até n é chamado de **fatorização prima** de n.

- Dizemos que um número primo é fator de X se ele aparece na fatorização prima. Qualquer outro número que não é primo é chamado de composto (composite).

# Números Primos - Encontrando

- A forma força bruta de encontrar um número primo:
  - Sucessão de divisões encontrando os divisores do número.

- Como 2 é o único primo par, uma vez verificado que x não é par, basta verificar os candidatos ímpares como fatores candidatos.

- Adicionalmente, podemos dizer que n é primo se for mostrado que não há fatores primos não-triviais abaixo de $\sqrt{n}$. Por quê?
  - Suponha que x é composto mas é o menor fator primo não trivial p que é maior que $\sqrt{n}$.
  - Então x/p deve dividir também x, e deve ser maior que p, caso contrário, teríamos o encontrado antes.
  - Mas o produto de 2 números maior que $\sqrt{n}$ tem que ser maior que n, o que é uma contradição.

# Números Primos - Encontrando

- A computação da fatorização prima envolve não apenas encontrar o primeiro fator primo, mas mostrar todas as ocorrências desse fator no produto restante:

```
prime_factorization(long x)
{
        long i;                 /* counter */
        long c;                 /* remaining product to factor */

        c = x;
        while ((c % 2) == 0) {
                printf("%ld\n",2);
                c = c / 2;
        }

        i = 3;
        while (i <= (sqrt(c)+1)) {
                if ((c % i) == 0) {
                        printf("%ld\n",i);
                        c = c / i;
                }
                else
                        i = i + 2;
        }

        if (c > 1) printf("%ld\n",c);
}
```

## Números Primos - Encontrando

- A condição de parada i > $\sqrt{c}$ é um pouco problemático, pois ***sqrt()*** é uma função numérica com precisão imperfeita.
  - Por segurança, foi acrescentando uma iteração extra.

- Uma outra forma seria evitar a computação do ponto flutuante e terminar quando i*i > c.
  - Entretanto a multiplicação pode causar overflow quando trabalhado com grandes inteiros.

- A multiplicação pode ser evitada observando que $(i+1)^2 = i^2 + 2i + 1$.
  - Adicionando i + i + 1 a $i^2$ nos leva a $(i+1)^2$.

# Números Primos - Encontrando

- Para uma melhor performance, podemos mover ***sqrt(c)*** para fora do laço principal e apenas atualizá-lo quando o valor de c mudar.

- Entretanto, o programa mostrado anteriormente responde quase que instantaneamente para o primo 2.147.483.647.

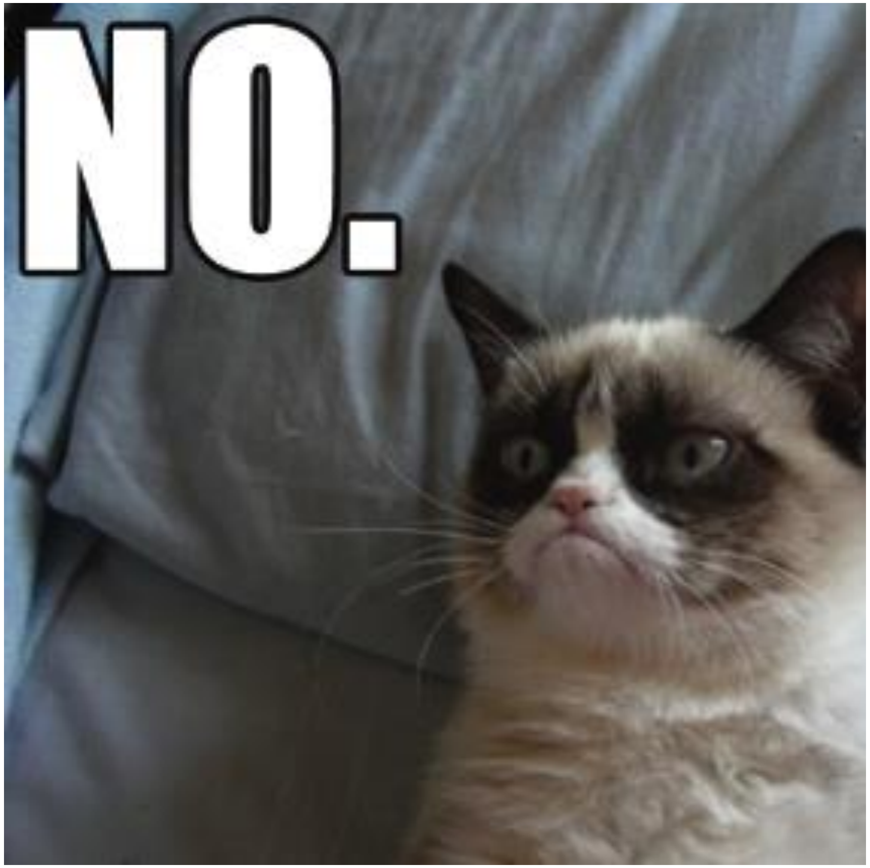
# Números Primos – Contando Primos

- Quantos números primos existem?

- É fácil perceber que encontrar um primo é algo raro a medida que o número aumenta. Será que em um dado momento ele deixa de existir?

- A resposta é:

# Números Primos – Contando Primos



# Números Primos – Contando Primos

- A prova de Euclides mostra que existe um número infinito de primos.
  - Ele prova por contradição.
  - Saber desse prova não é estritamente necessário para maratonas de programação, mas um sinal de uma pessoa com educação.

- Por isso, iremos revisá-la.

## Números Primos – Contando Primos

- Vamos assumir que existem apenas um número finito de primos, p1, p2 ... Pn.
- Logo: m = 1 + $\prod_{i=1}^{n} pi$ é o produto de todos os primos mais 1.
- Como isso será maior que qualquer primo da nossa lista, m deve ser um composto (composite). Logo, algum primo deve dividi-lo.
- Mas qual primo?

# Números Primos – Contando Primos

- Sabemos que m não é divisível por p1, pois deixa um restante de 1.

- Logo, m não é divisível por p2, pois também deixa um restante de 1.

- M deixa um restante de 1 quando dividido por qualquer primo pi, para i <= i <= n.

- De fato, p1,p2, ..., pn não pode conter a lista completa de primos, porque senão m também seria um primo.

- Como isso contradiz a assertiva, não existe uma lista completa com todos os primos, logo é infinito.

## Números Primos – Contando Primos

- Não apenas há um número infinito de primos, como de fato eles são relativamente comuns.

- Existem aproximadamente x/ln(x) primos menor ou igual a x.
  - Ou seja, 1 a cada ln(x) número é primo.

# Divisibilidade

- Dizemos que b divide a se a = bk.
  - Logo, b é um divisor de a.

- Como consequência dessa definição, o menor número divisor natural de cada inteiro (que não 0) é 1.
  - Por que? Não existe um inteiro k tal que a = 0.k

- Como encontramos todos os divisores de um dado inteiro?

- Vimos que um número é representado unicamente como o produto de fatores primos.

- Cada divisor é o produto de algum subconjunto desses fatores primos.

# Divisibilidade

- Esses subconjuntos podem ser encontrados utilizando técnicas de backtracking.

- Entretanto, deve-se tomar cuidado com fatores primos duplicados.

- Por exemplo: a fatorização prima de 12 tem 3 termos (2, 2 e 3), mas 12 tem apenas 6 divisores (1, 2, 3, 4, 6, 12)

## Divisibilidade – O maior divisor comum (GCD)

- Como 1 divide qualquer inteiro, o menor divisor comum de cada par inteiros a,b é 1.

- O maior divisor comum (Greatest Common Divisor - GCD) é o maior divisor compartilhado para um dado par de inteiros.

- Considere a fração x/y, por exemplo: 24/36.
  - A forma reduzida dessa fração é alcançada quando dividimos tanto o numerador quanto o denominador por gcd(x,y), que nesse caso seria 12.
  - Dizemos que 2 inteiros são relativamente primos se o GCD deles for 1.

# Divisibilidade – O maior divisor comum (GCD)

- O algoritmo de Euclides para encontrar GCD é considerado o primeiro algoritmo elaborado da história.

- A forma bruta de encontrar o GCD seria testar todos os divisores do primeiro número no segundo.
  - Ou encontrar a fatorização prima de ambos os inteiros e verificar o produto de todos os fatores em comum.

- Essas duas formas envolvem muitas operações.

## Divisibilidade – O maior divisor comum (GCD)

- O algoritmo de Euclides parte de 2 observações.

- Primeiro:
  - Se b/a então GCD(a,b) = b
  - Isso é bastante claro, se b divide a então a = bk para algum inteiro k, logo: GCD(bk,b) = b.

- Segundo:
  - Se a = bt + r para inteiros t e r, então GCD(a,b) = GCD(b,r)
  - Por que? Por definição GCD(a,b) = GCD(bt + r, b)
  - Qualquer divisor comum de a e b deve funcionar com r, pois bt claramente deve ser divisível por qualquer divisor de b.

## Divisibilidade – O maior divisor comum (GCD)

- O algoritmo de Euclides é recursivo, onde em cada repetição ele substitui o inteiro grande pelo resto do módulo do número menor.

- Logo, isso geralmente corta um dos argumentos pela metade e com um número de iterações logarítmicas consegue-se chegar em um caso base.

# Divisibilidade – O maior divisor comum (GCD)

- Peguemos o exemplo, onde a = 34398 e b = 2132.

$$
\begin{aligned}
gcd(34398, 2132) &= gcd(34398 \bmod 2132, 2132) = gcd(2132, 286) \\
gcd(2132, 286) &= gcd(2132 \bmod 286, 286) = gcd(286, 130) \\
gcd(286, 130) &= gcd(286 \bmod 130, 130) = gcd(130, 26) \\
gcd(130, 26) &= gcd(130 \bmod 26, 26) = gcd(26, 0)
\end{aligned}
$$

- Logo, GCD(34398, 2132) = 26

## Divisibilidade – O maior divisor comum (GCD)

- Entretanto, o algoritmo de Euclides pode nos dar mais do que apenas GCD(a,b).

- Ele pode encontrar inteiros x e y tal que:

$$a*x + b*y = GCD(a,b)$$

- Isso nos ajudará bastante para resolver congruências lineares.

- Sabemos que GCD(a,b) = GCD(b, a') onde a' = a – b[a/b].

- Assumindo que conhecemos os inteiros x' e y' tal que:

$$b*x' + a'*y' = GCD(a,b)$$

- Substituindo a':

$$b*x' + (a-b[a/b])*y' = GCD(a,b)$$

# Divisibilidade – O maior divisor comum (GCD)

- Basta executar de forma recursiva utilizando o seguinte caso base:

$$a*1 + 0*0 = GCD(a,0)$$

- Para o exemplo anterior teríamos que:

$$34398 * 15 + 2132*-242 = 26$$

# Divisibilidade – O maior divisor comum (GCD)

```
/*      Find the gcd(p,q) and x,y such that p*x + q*y = gcd(p,q)      */

long gcd(long p, long q, long *x, long *y)
{
        long x1,y1;                          /* previous coefficients */
        long g;                              /* value of gcd(p,q) */

        if (q > p) return(gcd(q,p,y,x));

        if (q == 0) {
                *x = 1;
                *y = 0;
                return(p);
        }

        g = gcd(q, p%q, &x1, &y1);

        *x = y1;
        *y = (x1 - floor(p/q)*y1);

        return(g);
}
```

## Divisibilidade – O menor múltiplo comum (LCM)

- O menor múltiplo comum é o menor inteiro que pode ser dividido por dois inteiros quaisquer.
- Por exemplo o LCM de 24 e 36 é 72.
- LCM nos serve quando queremos computar simultaneamente a periodicidade de 2 eventos distintos.
- Quando será o próximo ano (após 2000) que a eleição presidencial (que ocorre a cada 4 anos) coincidirá com o censo (que ocorre a cada 10 anos)?
  - Os eventos coincidem a cada 20 anos pois LCM(4,10) = 20

# Divisibilidade – O menor múltiplo comum (LCM)

- É evidente que LCM(x,y) >= max(x,y). Similarmente, já que x*y é um múltiplo tanto de x quanto de y, LCM(x,y) <= x*y.

- A única forma que pode existir um menor múltiplo comum é se há um fator não trivial compartilhado entre x e y.

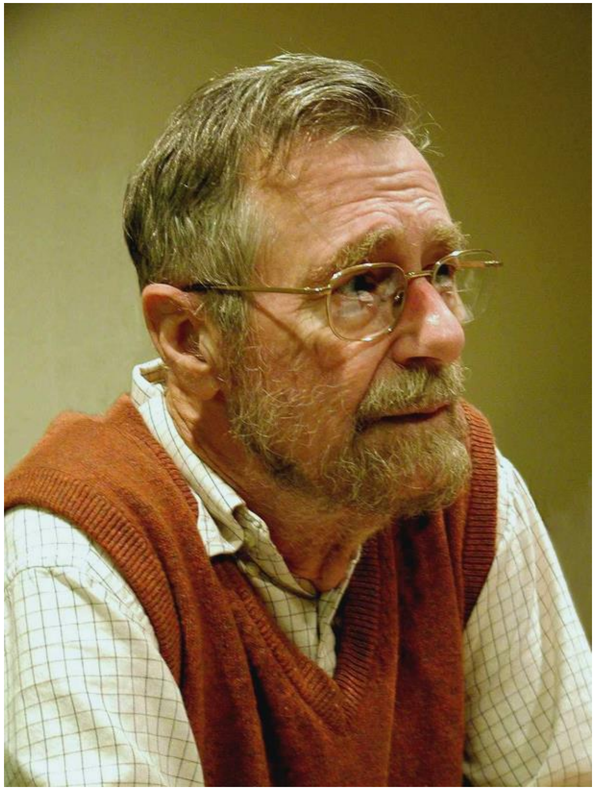
- Essa observação, juntamente com o teorema de Euclides, nos dá uma eficiente forma de computar LCM:

  $$LCM(x,y) = x*y/GCD(x,y)$$

- Dijikstra desenvolveu um algoritmo que evita a multiplicação o que acaba com a possibilidade de overflow.

# Divisibilidade – O menor múltiplo comum (LCM)

- Ah, estes são Euclides e Dijikstra, respectivamente



# Aritmética Modular

- Em alguns cenários, o resto de uma divisão é suficiente para resolvermos nossos problemas.

- Exemplo: Seu aniversário caiu na quarta esse ano. Em que dia da semana irá cair no ano que vem?
  - Tudo o que você precisa saber é que o resto do número de dias entre agora e o próximo ano (que pode ser 365 ou 366) dividido por 7 (dias da semana).
  - Logo, poderia ser quinta ou sexta (caso seja bissexto)

- A chave para algumas computações eficientes é utilizando o conceito de aritmética modular.
  - Claro que poderíamos simplesmente calcular o número por completo e depois encontrar o resto. Mas para números inteiros é muito mais simples utilizar este método.

## Aritmética Modular

- O número pelo qual estamos dividindo se chama módulo e o resto chamamos de resíduo.

- A chave para a aritmética modular é entender como as 4 operações básicas da aritmética funcionam:

- <u>Adição</u> – O que é (x + y) mod n? Podemos simplificar para:

  $$((x \bmod n) + (y \bmod n)) \bmod n$$

  - Isso para evitar a adição de grandes números.
  - Exemplo: Qual é o menor troco que terei se receber R$123,45 da minha mãe e R$94,67 do meu pai?

(12,345 mod 100) + (9,467 mod 100) = (45 + 67) mod 100 = 12 mod 100

# Aritmética Modular

- Subtração – Subtração é adição com números negativos.
  - Quanto seria o menor troco após gastos R$52,53?
  - (12 mod 100) – (53 mod 100) = -41 mod 100 = 59 mod 100
  - Basta converter o número negativo n em positivo adicionando um múltiplo de n a ele.

- Multiplicação – Como multiplicação é apenas repetição de adições:

$$xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$$

  - Exemplo: Quanto de troco terei se ganhar R$17,28 por hora em um período de 2.143 horas?

(1,728 * 2,143) mod 100 = (28 mod 100)*(43 mod 100) = 4 mod 100

# Aritmética Modular - Aplicações

- Encontrando o menor dígito:
  - Qual é o menor dígito de $2^{100}$ ?
  - Podemos utilizar força bruta, calcular o número e decidir quem é o mesmo.
  - Entretanto, precisamos apenas saber o resultado de: $2^{100}$ mod 10.
  - Basta fazer algo do tipo:

$$
\begin{aligned}
2^3 \bmod 10 &= 8 \\
2^6 \bmod 10 &= 8 \times 8 \bmod 10 \rightarrow 4 \\
2^{12} \bmod 10 &= 4 \times 4 \bmod 10 \rightarrow 6 \\
2^{24} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\
2^{48} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\
2^{96} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\
2^{100} \bmod 10 &= 2^{96} \times 2^3 \times 2^1 \bmod 10 \rightarrow 6
\end{aligned}
$$

# Aritmética Modular - Aplicações

- RSA algoritmo de criptografia:
  - A mensagem encriptada, possui um inteiro m que será elevada a potência k (que é a chave pública recebida).
  - Após isso pega o resultado mod n.

- Cálculos de calendários:
  - Conforme mostramos alguns exemplos, a modulação aritmética ajuda bastante nesse tipo de cálculo.

# Congruências

- Congruência é uma notação alternativa para representar a aritmética modular.

- Dizemos que $a \equiv b \pmod{m}$ se $m|(a-b)$.
    - Por definição se a mod m for b, então $a \equiv b \pmod{m}$.

- Essa notação é importante para cenários do tipo: qual cenário satisfaz x para a congruência $x \equiv 3 \pmod{9}$?
    - Claramente, $x = 3$.
    - Adicionar ou deletar módulos (9 nessa instância) nos dá outra solução.
    - O conjunto de soluções de todos os inteiros na forma $9y + 3$, onde y é qualquer inteiro.

# Congruências

- E para cenários como $2x \equiv 3 \pmod 9$ e $2x \equiv 3 \pmod 4$?
  - Tentativa e erro deve convencê-lo que os inteiros na forma $9y + 6$ satisfaz o primeiro
  - O segundo não tem solução.

## Operações com congruências

- Congruências funcionam com adição, subtração e multiplicação e uma forma limitada de divisão:

- Adição e Subtração – Suponha $a \equiv b \pmod{n}$ e $c \equiv d \pmod{n}$.
  - $a + c \equiv b + d \pmod{n}$.
  - Exemplo: suponha que $4x \equiv 7 \pmod{9}$ e $3x \equiv 3 \pmod{9}$. Logo:
  - $4x - 3x \equiv 7 - 3 \pmod{9} \Rightarrow x \equiv 4 \pmod{9}$

- Multiplicação – É lógico que $a \equiv b \pmod{n}$ implica que $a*d \equiv b*d \pmod{n}$ adicionando a congruência reduzida em si mesmo d vezes. Logo, $a \equiv b \pmod{n}$ e $c \equiv d \pmod{n}$ que implica em:

$$ac \equiv bd \pmod{n}$$

## Operações com congruências

- Congruências funcionam com adição, subtração e multiplicação e uma forma limitada de divisão:

- <u>Divisão</u> – Não podemos utilizar o cancelamento de fatores comuns com congruências.
  - Note que $6*2 \equiv 6*1 \pmod{3}$, mas claramente $2 \not\equiv 1 \pmod{3}$.
  - Para ver qual é o problema note que podemos redefinir a divisão por uma multiplicação inversa, logo a/b é o mesmo que $ab^{-1}$
  - Logo, podemos calcular $a/b \pmod{n}$ se podemos encontrar o inverso $b^{-1}$ tal como $bb^{-1} \equiv 1 \pmod{n}$.
  - Entretanto, nem sempre o inverso existe, tente achar uma solução para $2x \equiv 1 \pmod{4}$

## Operações com congruências

- Podemos simplificar a congruência $a*d \equiv b*d \pmod{d*n}$ para $a \equiv b \pmod{n}$, então podemos dividir os 3 termos por um fator mutualmente comum se existir.

- Logo, $170 \equiv 30 \pmod{140}$ implica que $17 \equiv 3 \pmod{14}$.

- Entretanto, a congruência $a \equiv b \pmod{n}$ tem que ser falsa (ou seja, não tem solução) se GCD(a,n) não divide b.

## Resolvendo congruências lineares

- Uma congruência linear é uma equação na forma $a*x \equiv b \pmod{n}$.
- Lembrando que nem toda equação tem solução, como por exemplo, $a*x \equiv 1 \pmod{n}$ não tem solução.
- De fato, $a*x \equiv 1 \pmod{n}$ tem uma solução se e somente se o módulo e o multiplicador são relativamente primos (GCD(a,n)=1).
- Podemos utilizar o algoritmo de Euclides para encontrar o inverso através da solução para $a*x' + n*y' = GCD(a,n) = 1$. Logo:

$$a*x \equiv 1 \pmod{n} \Rightarrow a*x \equiv a*x' + n*y' \pmod{n}$$

# Resolvendo congruências lineares

- $n*y' \equiv 0 \pmod{n}$, então o inverso é simplesmente x' do algoritmo de Euclides.

- Em geral, existem 3 casos, dependendo do relacionamento entre a, b e n:
  - GCD(a, b, n) > 1 – Então podemos dividir os 3 termos pelo seu divisor para recuperar o seu congruente equivalente. Isso nos dá uma solução simples mod a nova base, ou o equivalente GCD(a, b, n) soluções (mod n).
  - GCD(a,n) não divide b – Então, a congruência não terá solução
  - GCD(a,n) = 1 – Então há uma solução (mod n). $x = a^{-1}*b$ funciona, já que $a*a^{-1}*b \equiv b \pmod{n}$. Como mostrado acima, o inverso existe e pode ser encontrado pelo algoritmo de Euclides.

# Resolvendo congruências lineares

- Existe o Teorema do Resto Chinês que nos dá uma ferramenta para trabalhar com sistemas de congruências sobre diferentes módulos.

- Deem uma lida para identificar como ele funciona.

# Bibliotecas de Teoria de Números

- A classe BigInteger de Java inclui vários métodos úteis da teoria dos números.
  - GCD – BigInteger GCD(BigInteger val) – retorna um BigInteger onde o valor é o GCD de abs(this) e abs(val).

  - Exponenciação modular – BigInteger modPow(BigInteger exp, BigInteger m) – retorna um BigInteger que é o resultado de $this^{exp}$ mod m.

  - Módulo inverso – BigInteger modInverse(BigInteger m) – retorna um BigInteger com o valor $this^{-1}$ (mod m). Isto resolve a congruência y*this ≡ 1 (mod n) retornando y se existir.

# Bibliotecas de Teoria de Números

- A classe BigInteger de Java inclui vários métodos úteis da teoria dos números.
  - Teste de Primo – boolean isProbablePrime(int segurança) – retorna true se for um primo provável e false se for um composto (composite). Se retorna true, a probabilidade de ser primo é >= $1 - 1/2^{segurança}$

# Bibliografia


- Livro Programming Challenges, Steven S. Skiena, Miguel A. Revilla