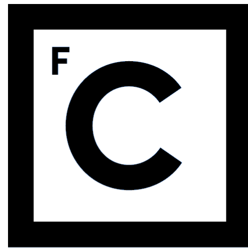
Introdução à Teoria dos Números
Aplicações Criptográficas I



Ciências
ULisboa

Faculdade
de Ciências
da Universidade
de Lisboa

Diogo Sousa · desousa [at] fc.ul.pt

20 de Março de 2019

Resumo

Neste guião abordamos as primitivas utilizadas na implementação de sistemas criptográficos cuja segurança e modo de operação se baseiam em resultados da Teoria dos Números.

Iremos comparar implementações *naif* com aquelas de bibliotecas estabelecidas ao nível da sua complexidade de implementação e eficiência.

Introdução

Os protocolos de criptografia simétrica como o RSA e a troca de chaves Diffie-Helman estão omnipresentes no nosso dia-a-dia conectado à Internet. São peças essenciais da infraestrutura que mantém as nossas comunicações seguras, estejamos a conversar com alguém no WhatsApp ou a partilhar fotos no Facebook.
Nesta aula vamos cobrir as primitivas necessárias à sua implementação, através de alguns exemplos e exercícios. Relembro aqui uma regra chave da criptografia:

`Never roll your own crypto.`

As implementações serão "toy implementations", não são robustas o suficiente para o mundo real. Se no futuro precisarem de utilizar funções criptográficas, usem bibliotecas validadas como `OpenSSL`, `libsodium` ou `NaCL`. Ao longo deste documento vou dar preferência à nomenclatura em Inglês.

Extended Euclidian Algorithm

O algoritmo de Euclides é utilizado para calcular o máximo divisor comum, referido como `mdc/gcd`. É um exemplo bastante comum de uma função recursiva:

Recursive GCD

```
def gcd(a, b):
    return a if b == 0 else gcd(b, a % b)

# Pythonista++
def gcd(a, b):
    return a if not b else gcd(b, a % b)
```

Aqui estamos a aproveitar que o zero (`0`) é avaliado como `False` em Python. O ganho em desempenho é insignificante, é apenas uma questão de estilo.
A verificação de `b == 0` leva a que o algoritmo tenha executar um mínimo de duas chamadas recursivas para resolver o melhor caso, `gcd(x,1)`.
O pior caso deste algoritmo ocorre quando a diferença entre `a` e `b` é maximizada a cada chamada recursiva. Isto corresponde a uma sequência bem conhecida que iremos abordar na aula prática.
Esta implementação, apesar de *naif*, é bastante rápida, tendo um limite assimptótico de $\mathcal{O}(\log(a+b))$. Se o número que estamos a analisar couber num registo (32 ou 64 bits), o tempo de execução é $\mathcal{O}(1)$ amortizado. No dia em que este guião era finalizado foi inclusive publicado um artigo com métodos para o seu cálculo em tempo constante. Ver Fast constant-time gcd computation and modular inversion.
Graças à forma transparente como o Python lida com números muito grandes (maiores que 2^{64}), conseguimos fazer estas operações com praticamente qualquer número que caiba na memória disponível. Contudo, as implementações reais são feitas em C/C++, devido à necessidade de desempenho. Nesses casos, como os números excedem largamente os 64 bits, utiliza-se o algoritmo de Stein. Este

implementa o máximo divisor comum através de operações binárias mais simples e que permite um ganho de desempenho em processadores com instruções especializadas. Contudo, apesar de ser cerca de 60% mais rápido, é mais difícil de implementar e a sua complexidade assimptótica é igual ao algoritmo que vimos acima, porque é uma função do número de dígitos de `a` e `b`.
Exemplo extraído da biblioteca openSSL (`https://github.com/openssl/openssl`):

Binary GCD

```
static BIGNUM *euclid(BIGNUM *a, BIGNUM *b) {
    BIGNUM *t;
    int shifts = 0;
    bn_check_top(a);
    bn_check_top(b);

    /* 0 <= b <= a */
    while (!BN_is_zero(b)) {
        /* 0 < b <= a */
        if (BN_is_odd(a)) {
            if (BN_is_odd(b)) {
                if (!BN_sub(a, a, b))
                    goto err;
                if (!BN_rshift1(a, a))
                    goto err;
                if (BN_cmp(a, b) < 0) {
                    t = a;
                    a = b;
                    b = t;
                }
            } else {            /* a odd - b even */
                if (!BN_rshift1(b, b))
                    goto err;
                if (BN_cmp(a, b) < 0) {
                    t = a;
                    a = b;
                    b = t;
                }
            }
/* Other branching code */
}
```

A ideia agora é expandir este algoritmo para calcular, utilizando as mesmas operações, os coeficientes de Bezout. Estes coeficientes (x, y) são a solução da seguinte equação diofantina. Seja $d = gcd(a, b)$, temos que:

$$ax + by = d$$

Isto pode ser reescrito como:

$$ax - 1 = (-y)\, b \leftrightarrow ax \equiv 1 \pmod{b}$$

Tendo esses valores, conseguimos calcular rapidamente o inverso multiplicativo de um dado número, uma operação essencial ao sistema RSA.
A ideia passa por acrescentar as operações necessárias ao algoritmo que vimos acima, mantendo a sua complexidade.
Uma solução, aproveitando que o Python permite dupla atribuição, é construir a função de forma iterativa e ir atualizando os valores de x e y a cada passo.

Iterative Bezout

```
def extended_gcd_I(a,b):
    s, old_s = 0,1
    t, old_t = 1,0
    r, old_r = b,a
    while r:
        q = old_r//r
        old_r,r = r, old_r - q*r
        old_s,s = s, old_s - q*s
        old_t,t = t, old_t - q*t
    return (old_r,old_s,old_t)
```

Outra opção é verificar que é possível resolver a equação $ax + by = gcd(a, b)$ em ordem a y dado $a, b, x, gcd(a, b)$. Isto pode parecer mais simples, mas tem pior performance porque estamos a recorrer a multiplicações e divisões, operações mais custosas.

Non-iterative Bezout

```
def extended_gcd(a, b):
    if not a:
        return (b, 0, 1)
    g, x, y = extended_gcd(b%a,a)
    return (g, y-(b//a)*x,x)
```

Multiplicative Inverse

Com estes elementos em mão, calcular o inverso multiplicativo é bastante rápido.

Using GCD to calculate modInv

```
def modinv(a, m):
    gcd, x, y = extended_gcd(a, m)
    return -1 if gcd != 1 else x % m
>>> extended_gcd(21,13)
>>> (1, 5, -8) # 5*21 - 8*13 = 1
>>> 21*(5%13) = 105
>>> 105%13 = 1
```

Fast Modular Exponentiation

O valor por omissão da chave pública (e) é 65537, o que corresponde a $2^{16}+1$. Este número tem um conjunto de propriedades úteis:
- É um primo de Fermat, garantindo que $gcd(\phi(n), e) = 1$.
- É grande o suficiente para garantir que o módulo acaba aplicado à mensagem, contornando as vulnerabilidades resultantes de um expoente público pequeno.
- A sua representação binária, $10\dots01$ implica que pode ser computado muito rapidamente através de *binary shifts*.

A questão agora coloca-se de como elevar rapidamente um número a este expoente. Existem vários métodos, de complexidades diferentes. O caso mais simples é iterar linearmente até ao expoente k, o que nos dá um limite de $\mathcal{O}(k)$ considerando que as operações de multiplicação são feitas em tempo constante, algo possível em CPUs modernos para números que cumpram certos requisitos. Se considerarmos a complexidade das operações de multiplicação $M(n)$, então estamos a olhar para $\mathcal{O}(M(n)\,k)$.

O método mais comum é o de exponenciação binária, visto ser trivial ter acesso à representação binária de um dado número. Este, contudo, não produz a sequência óptima de multiplicações.

Vamos aqui comparar diversos métodos de exponenciação:

- Exponenciação linear com módulo no final.
- Novamente linear, mas reduzindo modularmente a cada passo.
- Método de exponenciação binário. (*left-to-right*)
- Biblioteca base do Python.

Mais à frente vamos tentar encontrar o método ótimo para a exponenciação.

Fast Modular Exponentiation

```
def linear_mod_exp(message,k,modulus):
    r = 1
    for _ in range(k):
        r*=message
    return r%modulus

def linear_reducing_mod_exp(message,k,modulus):
    r = 1
    for _ in range(k):
        r = (r * message)%modulus
    return r

def bin_mod_exp(message,k,modulus):
    guide = bin(k)[2:]
    r = 1
    for bit in guide:
        r = (r*r)%modulus
        if bit == '1': # 1
            r =(r*message)%modulus
    return r

def lib_pow(message,k,modulus):
    return pow(message,k,modulus)
```

Running 100 rounds of each with m = 7, k = 65537, mod = 257
linear_mod_exp: `0.283061449` seconds
linear_reducing_mod_exp: `0.007414695000000009` seconds
bin_mod_exp: `3.5570000000006984e-06` seconds
lib_pow: `1.1669999999952551e-06` seconds

Miller-Rabin Primality Test

Quase toda a criptografia assimétrica assenta nos números primos. Como não podemos recorrer simplesmente a uma lista e "escolher um qualquer", é necessária uma forma de determinar se um dado número, escolhido de forma segura e aleatória, é primo*. Aqui é primo com um asterisco porque o que estamos a verificar realmente é se o número em questão é *provavelmente* primo, com uma precisão arbitrária. Aqui a ênfase não é que o teste seja 100% preciso mas sim **rápido**.
Existem testes deterministas, como o AKS, mas são muito "lentos". O seu tempo de execução assimptótico é de $\tilde{\mathcal{O}}((\log n)^6)$. O teste que vamos usar é o Miller-Rabin, cujo tempo de execução é $\tilde{\mathcal{O}}((\log n)^4)$.

Esta diferença de custo é relevante porque executar um teste determinista num número candidato que não é primo acabaria por comprometer o funcionamento dos algoritmos que desses dependessem. O algoritmo procura "testemunhas" que cumpram as seguintes condições:

$$a^d \not\equiv 1 \pmod{n}$$

$$a^{2^r d} \not\equiv -1 \pmod{n}$$

Como existe a possibilidade do número ser um "mentiroso", ou seja, composto e as equações se manterem corretas, executamos este processo por k rondas. Quanto maior o número de rondas, maior a confiança que o número em questão é primo. Para o RSA geralmente são executadas 64 rondas, para uma probabilidade de falso positivo de 2^{-128}.

Este algoritmo pode ser transformado numa versão determinista, bastando para tal testar todas as testemunhas possíveis até à raiz quadrada. Aliás, o conjunto de testemunhas necessário e suficiente já está estabelecido até certos limites. Isto permite tornar o teste determinista sacrificando algum desempenho e também acelerar a versão não-determinista.

Para calcular valores apropriados ao RSA, a versão determinista não terminaria em tempo útil para testar todos os valores $a < 2(\ln n)^2$ para um número de 2048 bits.

Witness List Fonte: Wikipedia

$n < 2047$ **:** $\{2\}$

$n < 1,373,653$ **:** $\{2, 3\}$

$n < 9,080,191$ **:** $\{31, 73\}$

$n < 25,326,001$ **:** $\{2, 3, 5\}$

$n < 3,215,031,751$ **:** $\{2, 3, 5, 7\}$

$n < 4,759,123,141$ **:** $\{2, 7, 61\}$

$n < 1,122,004,669,633$ **:** $\{2, 13, 23, 1662803\}$

$n < 2,152,302,898,747$ **:** $\{2, 3, 5, 7, 11\}$

$n < 3,474,749,660,383$ **:** $\{2, 3, 5, 7, 11, 13\}$

$n < 341,550,071,728,321$ **:** $\{2, 3, 5, 7, 11, 13, 17\}$

$n < 3,825,123,056,546,413,051$ **:** $\{2, 3, 5, 7, 11, 13, 17, 19, 23\}$

$n < 18,446,744,073,709,551,616$[a] **:** $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$

$n < 318,665,857,834,031,151,167,461$ **:** $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$

$n < 3,317,044,064,679,887,385,961,981$ **:** $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41\}$

[a] 2^{64}

Vamos então começar por uma implementação básica, que segue o algoritmo à risca e não aproveita estes resultados. Depois vamos comparar a sua performance com uma implementação otimizada.

Basic Miller-Rabin

```
def is_prime(n, k=16):
    import random as rnd # Local scope import
    if n == 2 or n == 3:
        return True
    if n < 3 or not n & 1: # Even check
        return False
    d = n - 1
    r = 0
    while d > 1 and not d % 2:
        d //= 2
        r += 1
    # This gives us n-1 as 2^r * d, with d odd.
    # k-iterations for confidence
    while k:
        a = rnd.randint(2, n - 2)
        x = pow(a, d, n)  # a^d mod n
        if x == 1 or x == (n - 1):
            k -= 1
            continue
        for _ in range(r):
            x = pow(x, 2, n)
            if x == (n - 1):
                break
        else: # Loop finished, composite
            return False
        k -= 1
    return True
```

Running 10 runs against optimized_rm for n in [1,1000000]

Normal: `13.941958300000001 seconds`

Optimized: `1.6327429499999995 seconds`

Exercício

Problem 122 - Efficient exponentiation

Efficient exponentiation

The most naive way of computing n^{15} requires fourteen multiplications:

$$n \times n \times \ldots \times n = n^{15}$$

But using a "binary" method you can compute it in six multiplications:

$$n \times n = n^2$$
$$n^2 \times n^2 = n^4$$
$$n^4 \times n^4 = n^8$$
$$n^8 \times n^4 = n^{12}$$
$$n^{12} \times n^2 = n^{14}$$
$$n^{14} \times n = n^{15}$$

However it is yet possible to compute it in only five multiplications:

$$n \times n = n^2$$
$$n^2 \times n = n^3$$
$$n^3 \times n^3 = n^6$$
$$n^6 \times n^6 = n^{12}$$
$$n^{12} \times n^3 = n^{15}$$

We shall define m(k) to be the minimum number of multiplications to compute n^k; for example m(15) = 5.

For $1 \leq k \leq 200$, find $\sum$ m(k).

Para uma versão mais simples, podem calcular os valores para $1 \leq k \leq 50$.