Geração de Números Primos

INE5429 - Segurança em Computadores

Trabalho Individual #1

Quenio Cesar Machado dos Santos (14100868)

Florianópolis, 08/05/2016

Sumário

1 Introdução

Neste trabalho vamos investigar e implementar a geração de números primos por um computador.

A geração de números primos pode ser dividida em dois passos:

- A geração de números aleatórios.
- A verificação se os números gerados são primos ou não.

Na próxima seção, vamos investigar a geração de números aleatórios. Posteriormente, vamos abordar a verificação de *primalidade* dos números gerados.

2 Números Pseudo-Aleatórios

Uma sequência de números aleatórios pode ser gerada automaticamente por um computador. Existem algoritmos desenvolvidos para esta finalidade que são chamados de *geradores de números pseudo-aleatórios*.

As sequências resultantes destes geradores não são verdadeiramente aleatórias. Estas são geradas a partir de um grupo pequeno de valores iniciais - chamados sementes. Porém, alguns dos valores sementes podem ser verdadeiramente aleatórios. Por exemplo, eles podem originar-se do relógio do próprio computador ou ainda de ruídos da câmera de vídeo e do microfone.

O *período* dos algoritmos - o tamanho da sequência gerada sem repetição - também são normalmente muito longos. O que na prática permite a utilização destes algoritmos em aplicações que necessitam de uma fonte geradora de números aleatórios, tais como *criptografia*.

Existem vários métodos disponíveis para a implementação de geradores de números pseudo-aleatórios. Nas próximas seções, vamos utilizar um dos métodos para implementar um gerador na linguagem C. Esta linguagem de programação permite a operação de números de precisão variável - através da biblioteca *GMP* - e também gera código de máquina muito eficiente.

2.1 O Método `xorshift`

O método escolhido para a geração de números pseudo-aleatórios é `xorshift`. Este utiliza operações XOR e *bit-shift* para gerar novos números a partir de números anteriores.

Algoritmos implementados com `xorshift` executam muito rapidamente, pois utilizam operações implementadas por instruções do processador do computador que precisam de poucos *ciclos*. Entretanto, os parâmetros - valores sementes e constantes usados nas operações - precisam ser escolhidos com cuidado para permitir longos períodos sem repetição de números.

Na sua forma original, o método `xorshift` não é adequado para uso em criptografia, pois não gera períodos muito longos. Mas este pode facilmente ser melhorado com o uso de uma operação de soma ou multiplicação. Estas variações são denominadas respectivamente de `xorshift+` e `xorshift*`.

Na próxima seção, vamos listar e comentar o código-fonte da implementação de `xorshift+` desenvolvida neste trabalho.

2.2 A Implementação

O método `xorshift` exige uma sequência inicial de números - as sementes - a serem utilizadas para a geração dos números. Quanto maior o número de sementes, mais longo será o período do gerador.

Na implementação abaixo, utilizamos um vetor de 16 inteiros de 64 bits, que permite um período máximo de $2^{1024} - 1$:

```
static const size_t s_size = 16;
static const uint64_t seed[s_size] = {
    UINT64_C(3176816624292027912),
    UINT64_C(5825261852996363023),
    UINT64_C(2657966707352449287),
    UINT64_C(1350174793163142913),
```

```
    UINT64_C(3916890652982003621),
    UINT64_C(8441565361753670199),
    UINT64_C(4048287511883088063),
    UINT64_C(7692991216032604145),
    UINT64_C(9168990544929416867),
    UINT64_C(4210552781383542944),
    UINT64_C(1295553278952537540),
    UINT64_C(4600491088959978208),
    UINT64_C(1426091184381835091),
    UINT64_C(2349336915254555439),
    UINT64_C(2525507937445642803),
    UINT64_C(4828313259052508846)
};
```

Se utilizarmos as sementes tais quais elas foram definidas acima, nosso gerador irá sempre gerar a mesma sequência de números em cada nova execução do programa. Para tornar a sequência variável entre execuções, utilizados o relógio do computador para gerar novas sementes, como implementado abaixo:

```
void init_seed(uint64_t *s)
{
    for (int i = 0; i < s_size; i++)
    {
        s[i] = clock() + seed[i];
    }
}
```

Observe no código-fonte acima que o tempo em nanoseconds do relógio é somado ao valor inicial da semente para gerar uma nova semente, diferente em cada execução do programa. Estas novas sementes é que realmente serão utilizadas pelo algoritmo gerador, implementado a seguir:

```
// xorshift+ 1024-bit
uint64_t random_int64()
{
    // Posição atual no vetor de sementes:
    static size_t i = 0;

    // Inicializa sementes somente na primeira execução desta função:
    static uint64_t s[s_size];
    if (s[i] == 0) init_seed(s);

    // Estabelece os valores de semente a serem usados nesta geração:
    uint64_t s0 = s[i];
    uint64_t s1 = s[i = (i + 1) & (s_size - 1)];
```

```
    const uint64_t ps1 = s1;

    // Gera o novo número usando shifts e xor:
    s1 ^= s1 << 31;
    s[i] = s1 ^ s0 ^ (s1 >> 11) ^ (s0 >> 30);

    // Usa a soma para aumentar a aleatoriedade desta função:
    return s[i] + ps1;
}
```

Agora, a função implementada acima nos permite gerar sequências pseudo-aleatórias de inteiros de 64 bits. Para gerar números com maior número de bits, utilizamos o gerador e concatenamos seus números em números maiores:

```
static inline size_t div_ceiling(size_t op1, size_t op2)
{
    return 1 + ((op1 - 1) / op2);
}

void random_mpz(mpz_t rop, size_t bit_count)
{
    static const size_t byte_bit_count = 8;

    // Geração dos componentes do número:
    size_t size = div_ceiling(bit_count, byte_bit_count);
    size_t count = div_ceiling(size, sizeof(int64_t));
    uint64_t n[count];
    for (int i = 0; i < count; i++) n[i] = random_int64();

    // Se o número for maior que 64 bits, precisamos do vetor inteiro:
    static const size_t int64_bit_count = (sizeof(int64_t) * byte_bit_count);
    if (bit_count >= int64_bit_count)
    {
        mpz_init_set_int64(rop, count, n);
    }
    else
    {
        // Caso seja menos de 64 bits, capturamos apenas parte do número gerado:
        const size_t ignored_bits = (int64_bit_count - bit_count);
        mpz_init_set_ui(rop, n[0] >> ignored_bits);
    }
}
```

Observe então que a função gera números de precisão arbitrária, usando o gerador de 64 bits como base. A próxima mostra a execução do gerador.

2.3 A Execução

O código-fonte abaixo permite gerar números aleatórios de 128 bits, 256, 512, e assim sucessivamente, até 4096 bits:

```
int main()
{
    static const int base = 10;
    static const size_t max_bit_count = 4096;
    static const size_t bit_count_inc = 128;

    const clock_t start = clock();

    size_t bit_count = bit_count_inc;
    while (bit_count <= max_bit_count)
    {
        const clock_t s = clock();

        mpz_t n;
        random_mpz(n, bit_count);

        printf("%-4lu (t = %10.6lf)", bit_count, elapsed_secs(s));
        printf(" -> ");
        mpz_out_str(NULL, base, n);
        printf("\n");

        bit_count += bit_count_inc;
    }

    printf("\nTotal Time: %lf\n", elapsed_secs(start));

    return 0;
}
```

Abaixo, temos a saída do programa:

```
128  (t =   0.000019) -> 165257786787652276919289564940747825877
256  (t =   0.000002) -> 84865960580376472255847981412862837933171166611691443137137826
384  (t =   0.000000) -> 31852217344472305884457179334775136689440822191028097354529882
512  (t =   0.000001) -> 88772193534872596786423966191960800928580564811599898306359137
640  (t =   0.000001) -> 14915044336942455357227904669001306206046844049308262542119379
768  (t =   0.000001) -> 88095383357609257249468206304658577684967276347567133927456927
896  (t =   0.000000) -> 39758918994265387265620577662407340193974476868075949335506087
1024 (t =   0.000029) -> 12319185307086644259390212505945432768233324762841849335080426
1152 (t =   0.000003) -> 40424975074552161332138947969217701946379649662984038698333494
```

```
1280 (t =   0.000001) -> 72905425991366704227351303259658222443745888988782831614000867l
1408 (t =   0.000011) -> 57839225488180196849407150860650262290892562402691312732936819
1536 (t =   0.000002) -> 97912120165776148897339863372206591380114367482332189273014390
1664 (t =   0.000026) -> 76059186925355973902493946468927129450887022580763859836317053
1792 (t =   0.000005) -> 18669962223684692135130090574391367432028608443501263437307311
1920 (t =   0.000051) -> 89941163084660163416986362863848127388581928062768504325038488
2048 (t =   0.000004) -> 25363877649607935128678164449238574449818980565663387350943681
2176 (t =   0.000013) -> 18059700653186973633818434762516416089387220381023540052938173
2304 (t =   0.000009) -> 32210597600966517613419775059618159901909309622809678225929875
2432 (t =   0.000003) -> 47210658455954646649506082454614632610264956973995850885590199
2560 (t =   0.000011) -> 18409508837193091211733731520171362190502545015838177658791839
2688 (t =   0.000030) -> 92397954424001202345470140259223339800545152366026986495210038
2816 (t =   0.000010) -> 24502577402481463896501731384200002061795490268458940785996398
2944 (t =   0.000010) -> 37448113669427079342826773849804675823891441927115663795691206
3072 (t =   0.000002) -> 12990730192584808804007796670557646942353578334899602033885255
3200 (t =   0.000010) -> 19101726153992156855935937789857733591982359106245527252266739
3328 (t =   0.000011) -> 53222163215110356311534874485080714351683251590495470909523528
3456 (t =   0.000023) -> 40073372781920421131763670452666632047333169321351522343007195
3584 (t =   0.000010) -> 20109273353137259512678703323696440557065867689333481709104978
3712 (t =   0.000010) -> 11443070641775140691840362999472963012686760705347439017090515
3840 (t =   0.000009) -> 15743557374762690792053187269761159298295132915847420377728455
3968 (t =   0.000010) -> 69735494067894676415244965076417727779989930066729076893087786
4096 (t =   0.000011) -> 65830022507160160928898451947793702995720645962449132142396595

Total Time: 0.001847
```

Observe na saída acima que o tempo de geração varia muito de uma execução para outra. Acreditamos que isto ocorre porque existem vários fatores envolvidos na execução, além do número de bits gerados, tais como o alinhamento de bytes nos registradores do processador para diferentes comprimentos em *bits* e também o próprio cache de memória.

Apesar da variação, a magnitude não varia muito, o que nos faz concluir que o tamanho dos números não influencia muito o desempenho de geração de números com o `xorshift+`. Seria interessante observar se esta correlação se mantém com outros métodos.

3 Números Primos

Agora, que já sabemos como gerar os números aleatórios, vamos verificar se os números gerados são primos ou não.

Existem vários métodos disponíveis para verificar a *primalidade* de um número. Na próxima seção, vamos descrever e implementar o método de *Fermat*.

3.1 O Método de *Fermat*

Fermat é um método simples que vai permitir determinar quão provavelmente um número é primo. Ou seja, este método não nos permite afirmar com certeza a *primalidade* (a não ser no casos de números muito pequenos), mas que a probabilidade é alta.

O algoritmo básico é seguinte:

- Dado um número n a ser verificado.
- Repita k vezes:
  - Gere um número aleatório entre $[2, n-2]$, chamado a.
  - Se $a^{n-1} \not\equiv 1 \pmod{n}$, então n **não** é primo.
- Se $a^{n-1} \equiv 1 \pmod{n}$ para todos valores gerados de a, então n **provavelmente** é primo.

Observe dois pontos importantes com relação ao método de *Fermat*:

1. Não há certeza de que n é primo, mas sim uma probabilidade.
2. Quanto maior o valor de k, maior a certeza de que n é primo.

Na seção abaixo, implementaremos *Fermat*.

3.2 A Implementação

O código-fonte abaixo implementa a verificação de números primos usando *Fermat*:

```
// Gera um número aleatório dentro do intervalo determinado:
void random_mpz_interval(mpz_t rop, mpz_t min, mpz_t max)
{
    static const int base = 2;

    const size_t bit_count = mpz_sizeinbase(max, base);

    mpz_init(rop);
    do
    {
        mpz_clear(rop);
        random_mpz(rop, bit_count);
    }
    while ((mpz_cmp(rop, min) < 0) || (mpz_cmp(rop, max) > 0));
}

// Fermat
bool is_probably_prime(mpz_t op)
{
    // Define o valor de n:
    mpz_t n;
    mpz_init_set(n, op);
    mpz_abs(n, n);
```

```
    // Se n entre [1, 3], então número é primo:
    if (mpz_cmp_ui(n, 3) <= 0) return true;

    // Define o valor inferior do intervalo (min = 2):
    mpz_t min;
    mpz_init_set_ui(min, 2);

    // Define o valor superior do intervalo (max = n - 2):
    mpz_t max;
    mpz_init_set(max, n);
    mpz_sub_ui(max, max, 2);

    // Define valor do expoente:
    mpz_t exp;
    mpz_init_set(exp, n);
    mpz_sub_ui(exp, exp, 1);

    // Faz k = 5; repete 5 vezes:
    static const int k = 5;
    for (int i = 0; i < k; i++)
    {
        // Gera número dentro do intervalo pré-estabelecido:
        mpz_t a;
        random_mpz_interval(a, min, max);

        // Calcula o potência com mod:
        mpz_t r;
        mpz_init(r);
        mpz_powm(r, a, exp, n);

        // Se resultado != 1, então com certeza não é primo:
        if (mpz_cmp_ui(r, 1) != 0) return false;
    }

    // Após 5 testes, é provavelmente primo:
    return true;
}
```

Observe na listagem acima que usamos um valor de *k* baixo para minimizar o tempo de execução. Quando maior o valor de *k*, mais tempo leva a execução da função `is_probably_prime()`. Porém, um valor de *k* pequeno está aumentando as chances de um falso positivo. É necessário estabelecer uma boa relação entre desempenho e confiabilidade. Dependendo da aplicação, a confiabilidade pode ser mais importante que o desempenho, exigindo valores maiores de *k*.

Na função implementada abaixo, usamos a função `is_probably_prime()` para gerar números

primos:

```
void find_prime(mpz_t rop, size_t bit_count)
{
    do
    {
        random_mpz(rop, bit_count);
        if (mpz_even_p(rop)) mpz_sub_ui(rop, rop, 1); // Verifica somente números ímpare
    }
    while (!is_probably_prime(rop));
}
```

Observe que, dependendo do tamanho do número primo esperado (em *bits*), a execução da função `find_prime()` pode levar muito tempo executando até encontrar um número, dado que se está procurando números aleatoriamente. Para agilizar a procura, estamos procurando somente os números impares, visto que o único número par primo é o número 2.

3.3 A Execução

O código-fonte abaixo gera números primos de 128 bits, 256, 512, e assim sucessivamente, até 4096 bits:

```
int main()
{
    static const int base = 10;
    static const size_t max_bit_count = 4096;
    static const size_t bit_count_inc = 128;

    const clock_t start = clock();

    size_t bit_count = bit_count_inc;
    while (bit_count <= max_bit_count)
    {
        const clock_t s = clock();

        mpz_t n;
        find_prime(n, bit_count);

        printf("%-4lu (t = %10.6lf)", bit_count, elapsed_secs(s));
        printf(" -> ");
        mpz_out_str(NULL, base, n);
        printf("\n");

        bit_count += bit_count_inc;
    }
```

```
    printf("\nTotal Time: %lf\n", elapsed_secs(start));

    return 0;
}
```

Abaixo, temos a execução do código listado acima:

```
28   (t =   0.000683) -> 324865934876219014188817636179210243083
256  (t =   0.006194) -> 7927511354012793197660695243813133581635576780496490606542075413
384  (t =   0.002596) -> 3366317891664240476797202248281000733399218511415684470657800588
512  (t =   0.008433) -> 6978075481958970225533514553564411133610484390566161785769301444
640  (t =   0.146810) -> 4423755595021443879521191171814494528412136459701376771855280711
768  (t =   0.474838) -> 1269435556431714073618151512144716869435128127654473280607865566
896  (t =   0.100969) -> 1813672841847057827290190880450249169214767946142545976892166477
1024 (t =   0.527501) -> 9356253868058104617034723120930219066304736019943257434273098088
1152 (t =   0.841618) -> 3435780091398951399978226506833748954258328877737712604556261111
1280 (t =   0.113459) -> 1973273848898491869270388895444502523505005172553064860737317366
1408 (t =   1.488586) -> 3199131537936147905992178053204559320651232566917789756472360144
1536 (t =   0.776068) -> 5203668497361773421303366877367234957547540331838153044712149666
1664 (t =   1.109293) -> 6374163483298629165256636151025440123881317427077069201342661077
1792 (t =   9.877566) -> 1043074186669180636169482803696036345397746629495748711351133166
1920 (t =  18.682015) -> 8423100380605616451922009746688532373577631157912050786695258600
2048 (t =   1.032783) -> 8025978000412898115870959418055867596418656343015678706435958888
2176 (t =   8.196905) -> 2309339950814787343021962087793561432720051163166020129829007288
2304 (t =   8.954942) -> 7457721921248971175278894944772993278847301432384983822988331528
2432 (t =  20.020598) -> 1779115820526216355076157429334840469211601028897111167154284866
```

Observe na saída acima que o maior número de bits impresso foi 2432; não foi 4096. Isto ocorre porque quanto maior o número, menor a probablidade de se encontrar um número primo, portanto mais iterações são necessárias, o que leva mais tempo.

A função geradora de números aleatórios também tem um efeito no desempenho. Observe que alguns números menores foram mais custosos que outros números maiores. Acreditamos que isto ocorra porque o gerador de números não é tão aleatório, dependendo do número de bits, o que torna mais difícil achar primos em determinadas sequencias. Seria interessante verificar o desempenho de outros métodos de geração de números, ou variações do mesmo método.

3.4 *Miller-Rabin*

Miller-Rabin é uma extensão do método de *Fermat* que adiciona mais uma verificação:

$a^{2^r d} \equiv -1 \pmod{n}$ para todo r no intervalo $0 \leq r \leq s - 1$, onde s e r são positivos e r é impar.

Se n passa esta igualdade para todos os valores de r, então n é certamente um primo.

Devido a maior certeza do que no método de *Fermat*, este método é mais confiável para aplicações de criptografia.

4 Conclusão

Os algoritmos de geração de números primos exigem muito computacionalmente do processador. Quanto maior o número a ser gerado, maior a quantidade de ciclos necessários para se encontrar um número primo. A escolha adequada do método pode aumentar a confiabilidade do número encontrado e também diminuir o tempo de processamento necessário.

Uma continuação interessante deste trabalho seria comparar os diferentes métodos e verificar seu desempenho e confiabilidade. Por exemplo, com a implementação de *Miller-Rabin* e também com a implementação de outros métodos de geração de números aleatórios.