Matemática

Números Primos

Prof. Edson Alves
Faculdade UnB Gama

Definição de números primos

Seja p um número inteiro positivo. Dizemos que p é **primo** se ele possui exatamente dois divisores positivos: o próprio p e o número 1.

Um número natural $m > 1$ que não é primo é denominado número **composto**.

Consequências da definição de primos

- O número 1 não é primo, pois possui um único divisor positivo
- O menor número primo, e o único que é par, é o número 2
- Os próximos números primos são, a saber, $3, 5, 7, 11, 13, 17, 19, 23, \ldots$
- Se p e q são primos e p divide q, então $p = q$
- Se p é primo e p divide o produto ab, então p divide a ou p divide b (aqui o **ou** é inclusivo: pode acontecer que p divida ambos).

Identificação de números primos

Para se determinar se um inteiro positivo n é ou não primo pode-se recorrer diretamente à definição de primos e fazer uma busca completa dos possíveis divisores. Caso seja encontrado um divisor de n que seja diferente de 1 ou do próprio n, então n será composto

```
bool is_prime(int n)
{
    if (n < 2)
        return false;

    for (int i = 2; i < n; ++i)
        if (n % i == 0)
            return false;

    return true;
}
```

Complexidade da identificação de números primos

- A rotina `is_prime()`, embora seja de fácil entendimento e codificação, tem complexidade $O(n)$
- Há ainda o agravante que a principal operação realizada no laço é a divisão inteira, a qual é computacionalmente exigente (ao contrário da adição e multiplicação que, em geral, podem ser realizadas em um ciclo do processador)
- Também são realizadas muitas operações desnecessárias: por exemplo, se o número for ímpar, qualquer tentativa de se encontrar um divisor par é infrutífera

Eliminação de operações desnecessárias

```
bool is_prime2(int n)
{
    if (n < 2)
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    for (int i = 3; i < n; i += 2)
        if (n % i == 0)
            return false;

    return true;
}
```

Redução na complexidade da identificação de primos

- Embora a rotina `is_prime2()` reduza a quantidade de operações em relação à rotina `is_prime()`, a complexidade não foi reduzida, permanecendo em $O(n)$
- Para reduzir a complexidade, é preciso observar que deve-se procurar um possível divisor até apenas $\sqrt{n}$
- Isto se deve ao fato de que se d divide n, então $n = dk$, e ou d ou k deve ser menor ou igual à raiz quadrada de n
- Se ambos fossem maiores o produto dk seria maior do que n, uma contradição

Implementação com complexidade reduzida

```
bool is_prime3(int n)
{
    if (n < 2)
        return false;

    if (n == 2)
        return true;

    if (n % 2 == 0)
        return false;

    for (int i = 3; i * i <= n; i += 2)
        if (n % i == 0)
            return false;

    return true;
}
```

Função $\pi(n)$

- A rotina `is_prime3()` tem complexidade $O(\sqrt{n})$
- Observe que o teste do laço não utiliza a rotina `sqrt()`, para evitar erros de precisão e melhorar o tempo de execução
- É possível reduzir a complexidade uma vez mais, uma vez que os candidatos à divisores de `is_prime3()` são os ímpares entre 3 e a $\sqrt{n}$
- Se forem precomputados os primos menores ou iguais a $\sqrt{n}$ e eles forem utilizados como candidatos a divisores, a complexidade se torna $O(\pi(n))$, onde a função $\pi(n)$ retorna o número de primos menores ou iguais a n

Aproximação para $\pi(n)$

- O cálculo de $\pi(n)$ não é trivial, mas este valor pode ser aproximado:

$$\pi(n) \approx \frac{n}{\log n}$$

- Na prática, para se identificar se um ou poucos números são primos, `is_prime3()` é suficiente
- Para identificar vários inteiros n, pode ser útil gerar uma lista de primos de antemão, a qual permitirá a identificação imediata de números dentro do seu intervalo

Enumeração dos N primeiros primos

Uma maneira de se listar os N primeiros primos seria iterar sobre estes inteiros, e para cada um deles invocar a rotina `is_prime3()`. A complexidade deste algoritmo seria $O(N \times \pi(N))$.

```
vector<int> primes(int N)
{
    vector<int> ps;

    for (int i = 2; i <= N; ++i)
        if (is_prime3(i))
            ps.push_back(i);

    return ps;
}
```

O Crivo de Erastótenes

- Contudo, há uma forma mais eficiente de gerar esta lista, usando o Crivo de Erastótenes
- A ideia do crivo é eliminar os números compostos, que podem ser identificados imediatamente como múltiplos de um primo
- Para isto, são listados os N primeiros naturais
- A cada iteração do crivo, é identificado o próximo número primo e todos seus múltiplos são eliminados da lista
- Ao final do algoritmo a lista conterá apenas números primos

Exemplo do crivo de Erastótenes para $N = 50$

- Para ilustrar a ideia, considere $N = 50$
- Primeiramente liste todos os números positivos menores ou iguais a N:

```
  1    2    3    4    5    6    7    8    9   10
 11   12   13   14   15   16   17   18   19   20
 21   22   23   24   25   26   27   28   29   30
 31   32   33   34   35   36   37   38   39   40
 41   42   43   44   45   46   47   48   49   50
```

Exemplo do crivo de Erastótenes para $N = 50$

- O número 1 pode ser eliminado, pois não é primo:

```
 x   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30
31  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49  50
```

- O próximo número da sequência (2) é primo
- Deve-se, portanto, crivar todos os seus múltiplos, os quais são números compostos
- O número 2, porém, não deve ser crivado

Exemplo do crivo de Erastótenes para $N = 50$

```
 x   2   3   x   5   x   7   x   9   x
11   x  13   x  15   x  17   x  19   x
21   x  23   x  25   x  27   x  29   x
31   x  33   x  35   x  37   x  39   x
41   x  43   x  45   x  47   x  49   x
```

- O próximo número da sequência não crivado será primo (neste caso, o número 3)
- Os múltiplos de 3 ainda não crivados devem ser eliminados:

```
 x   2   3   x   5   x   7   x   x   x
11   x  13   x   x   x  17   x  19   x
 x   x  23   x  25   x   x   x  29   x
31   x   x   x  35   x  37   x   x   x
41   x  43   x   x   x  47   x  49   x
```

Exemplo do crivo de Erastótenes para $N = 50$

- O próximo número não crivado é 5, e seus múltiplos devem ser removidos:

```
 x   2   3   x   5   x   7   x   x   x
11   x  13   x   x   x  17   x  19   x
 x   x  23   x   x   x   x   x  29   x
31   x   x   x   x   x  37   x   x   x
41   x  43   x   x   x  47   x  49   x
```

- Seguindo com o número 7, a listagem se torna

```
 x   2   3   x   5   x   7   x   x   x
11   x  13   x   x   x  17   x  19   x
 x   x  23   x   x   x   x   x  29   x
31   x   x   x   x   x  37   x   x   x
41   x  43   x   x   x  47   x   x   x
```

Exemplo do crivo de Erastótenes para $N = 50$

- Como o próximo número, 11, é maior do que a raiz quadrada de 50, o processo pode ser interrompido
- Os números não crivados formam a relação de todos os primos menores ou iguais a N
- Uma implementação implementação do crivo em C++ utiliza o vetor de *bits* `sieve` para marcar os números
- Zero ou falso indica que o número é composto

Implementação do Crivo de Erastótenes

```
vector<int> primes2(int N)
{
    bitset<MAX> sieve;      // MAX deve ser maior ou igual a N
    vector<int> ps;

    sieve.set();            // Todos são "potencialmente" primos
    sieve[1] = false;       // 1 não é primo

    for (int i = 2; i <= N; ++i) {
        if (sieve[i]) {     // i é primo
            ps.push_back(i);

            for (int j = 2 * i; j <= N; j += i)
                sieve[j] = false;
        }
    }

    return ps;
}
```

Aproximação para a complexidade do crivo

- Na rotina `primes2()`, para cada i são crivados N/i números
- Portanto o número total $T(N)$ de operações é aproximadamente N vezes o N-ésimo número harmônico H_N, isto é,

$$T(N) \approx N + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{N} = NH_N = N\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}\right) \leq N \log N$$

- A desigualdade vale porque

$$\log N = \int_1^N \frac{1}{x} dx \geq \sum_{i=1}^{N} \frac{1}{i}$$

Complexidade do Crivo de Erastótenes

- Na aproximação para $T(N)$ a variável i assume todos os naturais no intervalo $[1, N]$, inclusive números compostos
- Porém na implementação i assume apenas valores primos
- Uma melhor aproximação seria, pelo Teorema de Merten,

$$\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \frac{N}{7} + \frac{N}{11} + \ldots \leq N \log \log N$$

- Logo a complexidade do crivo é $O(N \log \log N)$

Redução da constante de complexidade

- A implementação de `primes2()` é simples e direta
- É possível, contudo, diminuir a constante de complexidade e obter um melhor tempo de execução
- Primeiramente, os números pares podem ser tratados à parte: 2 é o único primo par, e os demais pares compostos não contribuem para o crivo
- O número 1 pode ser desprezado, uma vez que a saída da rotina é uma lista de primos

Crivo com tratamento diferente para pares

```
vector<int> primes3(int N)
{
    bitset<MAX> sieve;                       // MAX deve ser maior ou igual a N
    vector<int> ps { 2 };                    // Os pares são tratados à parte

    sieve.set();                             // Todos são "potencialmente" primos

    for (int i = 3; i <= N; i += 2) {        // Apenas ímpares são verificados agora
        if (sieve[i]) {                      // i é primo
            ps.push_back(i);

            for (int j = 2 * i; j <= N; j += i)
                sieve[j] = false;
        }
    }

    return ps;
}
```

Nova redução da constante de complexidade

- Embora o laço externo de `primes3()` só considere números ímpares, o laço interno itera por pares desnecessariamente
- Outra observação importante: o crivo deve começar no quadrado de `i`, pois quaisquer múltiplos de i menores do que i^2 já foram crivados ou ignorados
- Estas duas observações reduzem novamente a constante de complexidade
- Para evitar problemas de *overflow* com a condição do laço interno, o tipo de dado foi alterado de `int` para `long long`

Crivo sem pares nos laços

```
vector<long long> primes4(long long N)
{
    bitset<MAX> sieve;                                   // MAX deve ser maior ou igual a N
    vector<int> ps { 2 };                                // Os pares são tratados à parte

    sieve.set();                                         // Todos são "potencialmente" primos

    for (long long i = 3; i <= N; i += 2) {              // Apenas ímpares são verificados agora
        if (sieve[i]) {                                  // i é primo
            ps.push_back(i);

            for (long long j = i * i; j <= N; j += 2*i)  // Múltiplos ímpares >= i*i
                sieve[j] = false;
        }
    }

    return ps;
}
```

Tratamento de múltiplos de 3 à parte

- Assim como foi feito para os pares, os múltiplos de 3 também podem ser tratados à parte
- A inclusão prévia do 3 na lista de primos é a parte trivial, difícil é evitar os múltiplos de 3 no laço externo
- Isto pode ser feito observando que, seguindo a sequência dos ímpares a parte de 3, primeiro há um múltiplo de 3, depois um número cujo resto da divisão por 3 é 2, e por fim um número cuja divisão por 3 é 1, e o ciclo se reinicia
- Os múltiplos de 3, portanto, devem ser saltados

Crivo com múltiplos de 2 e 3 tratados à parte

```
vector<long long> primes5(long long N)
{
    bitset<MAX> sieve;          // MAX deve ser maior ou igual a N
    vector<int> ps { 2, 3 };    // Pares e múltiplos de 3 são tratados à parte

    sieve.set();                // Todos são "potencialmente" primos

    // O incremento alterna entre saltos de 2 ou 4, evitando os múltiplos de 3
    for (long long i = 5, step = 2; i <= N; i += step, step = 6 - step) {
        if (sieve[i]) {                                     // i é primo
            ps.push_back(i);

            for (long long j = i * i; j <= N; j += 2*i)    // Múltiplos ímpares >= i*i
                sieve[j] = false;
        }
    }

    return ps;
}
```

Verificação das implementações do crivo

Uma maneira de verificar rapidamente se o crivo está produzindo os primos corretamente é checar o número de primos gerados, segundo a tabela abaixo:

n	$\pi(n)$
10	4
100	25
1000	168
10000	1229
100000	9592
1000000	78498
10000000	664579

Possível *benchmark* para as rotinas de primalidade

```
==== Testes de primalidade:

is_prime(999983)  = 1 (0.010074394000000 ms)
is_prime2(999983) = 1 (0.005721907000000 ms)
is_prime3(999983) = 1 (0.000006486000000 ms)


==== Geração de primos até N:

primes(10000000)  = 664579 (2.428975496000000 ms)
primes2(10000000) = 664579 (0.172493493000000 ms)
primes3(10000000) = 664579 (0.136014180000000 ms)
primes4(10000000) = 664579 (0.067260405000000 ms)
primes5(10000000) = 664579 (0.059135050000000 ms)
```

Problemas

Referências

1. The PrimesPage. How Many Primes Are There?. Acesso eme 08/11/2017.

2. Wikipédia. Harmonic Number. Acesso em 08/11/2017.

3. Wikipédia. Mertens' theorems. Acesso eme 08/11/2017.