Teoria dos Números

Números primos

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Números primos

Números primos

Definição de número primo

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 $n > 1$ 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, ...
- 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

Identificação de números primos

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

Identificação de primos com complexidade $O(n)$

```
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
- A divisão contrasta com a adição e a multiplicação as quais podem, em geral, ser realizadas em um ou dois ciclos do processador
- Também são realizadas muitas operações desnecessárias
- Por exemplo, se n for ímpar, qualquer tentativa de se encontrar um divisor par de n 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 por possíveis divisores d tais que $d \leq \sqrt{n}$
- 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

Verificação de primalidade em $O(\sqrt{n})$

```
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;
}
```

Nova redução na complexidade da verificação de primalidade

- 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 $\sqrt{n}$
- Suponha que a listagem P de todos os números primos seja conhecida
- Um algoritmo que utiliza apenas os elementos de P como candidatos a divisores tem complexidade $O(\pi(\sqrt{n}))$

Função $\pi(n)$

Definição da função $\pi(n)$

Seja n um inteiro positivo. A função $\pi(n)$ retorna o número de primos menores ou iguais a n.

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

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

O crivo de Erastótenes

Listagem dos N primeiros primos

- Na prática, para se verificar se um ou poucos números são primos, `is_prime3()` é suficiente
- Para verificar um conjunto de inteiros n, pode ser útil gerar uma lista de primos de antemão, a qual permitirá a identificação imediata de números presentes nesta listagem
- Uma maneira de se listar os N primeiros primos seria iterar sobre os inteiros do intervalo $[1, N]$ e, para cada um deles, invocar a rotina `is_prime3()`
- A complexidade deste algoritmo seria $O(N \times \sqrt{N})$.

Listagem dos N primeiros primos em $O(N^{3/2})$

```
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: o Crivo de Erastótenes
- A ideia do crivo é eliminar os números compostos, os quais 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 $N = 50$

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 $N = 50$

$\boxed{1}$	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 $N = 50$

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 $N = 50$

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

Exemplo do crivo de Erastótenes para $N = 50$

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 $N = 50$

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

Exemplo do crivo de Erastótenes para $N = 50$

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 $N = 50$

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 $N = 50$

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 $N = 50$

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

Exemplo do crivo de Erastótenes para $N = 50$

- Como o próximo número primo, a saber 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 do crivo de Erastótenes em C++ pode o vetor de *bits* `sieve` para marcar os números
- Zero ou falso indica que o número não é primo

Implementação do crivo em C++

```cpp
vector<int> primes2(int N) {
    vector<int> ps;
    bitset<MAX> sieve;              // MAX deve ser maior do que N
    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,

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

- 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 do que 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
- Estas duas observações reduzem novamente a constante de complexidade
- Para evitar problemas de *overflow* na 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 do que N
    vector<long long> 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 partir de 3, primeiro há um múltiplo de 3, depois um número cujo resto da divisão por 3 é 2, por fim um número cuja divisão por 3 é 1, e o ciclo se reinicia
- Os múltiplos de 3, portanto, podem ser ignorados

Crivo com múltiplos de 2 e 3 tratados à parte

```cpp
vector<long long> primes5(long long N)
{
    bitset<MAX> sieve;              // MAX deve ser maior do que N
    vector<long long> 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 saída para as rotinas de teste 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 propostos

1. AtCoder Beginner Contest 096D – Five, Five Everywhere
2. AtCoder Beginner Contest 149C – Next Prime
3. Codeforces 327B – Hungry Sequence
4. OJ 543 – Goldbach
5. OJ 11752 – The Super Powers

Referências

1. The PrimesPage. How Many Primes Are There?. Acesso em 08/11/2017.
2. Wikipédia. Harmonic Number. Acesso em 08/11/2017.
3. Wikipédia. Mertens' theorems. Acesso em 08/11/2017.

Soluções dos problemas propostos

AtCoder Beginner Contest 096D – Five, Five Everywhere

Versão resumida do problema: dado um inteiro n, determine uma sequência de inteiros $a_1, a_2, \ldots, a_N$ tais que

- $a_i < 55555$ é primo
- Todos os elementos da sequência são distintos
- A soma de quaisquer 5 elementos da sequência resulta em um número composto

Restrição: $5 \leq N \leq 55$

Solução com complexidade $O(M \log \log M)$

- A solução consiste em identificar um subconjuntos de primos que atendam a primeira e a terceira condições
- Os $\pi(55555) = 5637$ primos podem se gerados pelo crivo de Erastótenes
- Devem ser selecionados dentre os primos aqueles cujo resto da divisão por 5 seja k, com $k > 0$
- Qualquer k no intervalo $[1, 4]$ gera uma lista de mais de 1.400 primos
- Assim, após o filtro todos os elementos selecionados serão da forma $5m + k$, e daí

$$(5m_1 + k) + (5m_2 + k) + \ldots + (5m_5 + k) = 5(m_1 + m_2 + \ldots + m_5 + k)$$

- Ou seja, a soma de quaisquer N elementos dentre os filtrados é divisível por 5, e portanto é um número composto
- A complexidade será igual a complexidade do crivo usado para gerar os primos até M, onde $M = 555555$

Solução com complexidade $O(M \log \log M)$

```
vector<int> solve(int N)
{
    auto ps = primes5(55555);

    vector<int> qs;
    int k = 2;          // Escolha arbitrária: qualquer valor em [1, 4] é válido

    for (auto p : ps)
        if (p % k == 0)
            qs.push_back(p);

    vector<int> ans(qs.begin(), qs.begin() + N);

    return ans;
}
```

Codeforces 327B – Hungry Sequence

Versão resumida do problema: dado um inteiro N, determine uma sequência de inteiros $a_1, a_2, \ldots, a_N$ tais que

- $a_i < a_j$ se $i < j$
- a_i não divide a_j, para qualquer $i < j$

Restrição: $N \leq 10^5$

Solução com complexidade $O(M \log \log M)$

- Suponha que você deseje iniciar uma sequência com estas características em $a_1 = k$
- Devido ao segundo critério, nenhum dos elementos subjacentes da sequência pode ser múltiplo de k
- Ou seja, incluir k na sequência "criva" todos seus múltiplos
- Desta maneira, iniciando com $a_1 = 2$ (pois 1 divide qualquer número) e aplicando o crivo de Erastótenes, os candidatos a demais elementos são todos primos
- Como $\pi(10^7) = 664579 > 10^5$, basta imprimir na saída os N primeiros primos

Solução com complexidade $O(M \log \log M)$

```
vector<int> solve(int N)
{
    auto ps = primes5(10000000);

    vector<int> ans(ps.begin(), ps.begin() + N);

    return ans;
}
```

OJ 11752 – The Super Powers

Versão resumida do problema: liste todos os inteiros $n < 2^{64}$ tais que $n = a^r = b^s$, com $a \neq b$ e $r, s > 1$.

Solução

- O principal ponto a ser observado é que n tem que ser um número da forma m^c, onde c é um número composto
- Isto porque se c é composto, ele pode ser escrito como $c = rs$, com $r, s > 1$
- Daí

$$n = m^c = m^{rs} = (m^r)^s = (m^s)^r$$

- Como 4 é o menor número composto e $n^4 > 2^{64}$ para todos $n \geq 2^{16}$, a listagem dos números desejados pode ser obtida elevando-se todos os inteiros positivos no intervalo $[1, 2^{16})$ a todos os números compostos c no intervalo $[1, 64)$

Solução

- Os compostos menores ou iguais a n podem ser obtidos por meio de uma variante da função que determina se o número é ou não primo
- As possíveis repetições podem ser eliminadas se os resultados forem armazenados em um conjunto
- Para evitar o *overflow* no cálculo de n^c, é preciso saber se o resultado é ou não menor do que 2^{64}
- Isto pode ser verificado por meio de logaritmos, pois $n^c < 2^{64}$ se

$$c \log_2 n < 64 \log_2 2 = 64$$

Solução

```
vector<int> composite(int m)
{
    vector<int> cs;

    for (int n = 2; n < m; ++n)
        for (int d = 2; d * d <= n; ++d)
            if (n % d == 0)
            {
                cs.push_back(n);
                break;
            }

    return cs;
}
```

Solução

```
unsigned long long power(int a, int n)
{
    unsigned long long res = 1;

    while (n--)
        res *= a;

    return res;
}
```

Solução

```cpp
set<unsigned long long> solve()
{
    auto cs = composite(64);
    set<unsigned long long> ans;

    for (int n = 1; n < (1 << 16); ++n)
    {
        for (auto c : cs)
            if (c*log2(n) < 64)
                ans.insert(power(n, c));
    }

    return ans;
}
```