Desafios de Programação

Prof. Eduardo Theodoro

Universidade Federal de Mato Grosso do Sul (UFMS)

Teoria dos Números

Primo

Um número é primo se é divisível apenas por 1 e por ele mesmo.

Como verificar se um número é primo?

Seja n o número que desejamos verificar se é primo.
Se existir algum divisor de n entre 2 e $\frac{n}{2}$ então n não é primo! Caso contrário n é primo. A complexidade do algoritmo então é $O(\frac{n}{2})$.

Checar se um número é primo

Podemos melhorar? SIM!

Melhoria 1

Seja n o número que desejamos verificar se é primo.
Se existir algum divisor de n entre 2 e $\lceil\sqrt{n}\rceil$ então n não é primo! Caso contrário n é primo. **Motivo:** Se n for divisível por p, então $n = p \times q$. Se q fosse menor do que p, então q ou um fator primo de q teriam dividido n anteriormente. A complexidade do algoritmo então é $O(\sqrt(n))$.

Checar se um número é primo

Podemos melhorar? SIM!

Melhoria 1

Seja n o número que desejamos verificar se é primo.
Se existir algum divisor de n entre 2 e $\lceil\sqrt{n}\rceil$ então n não é primo! Caso contrário n é primo. **Motivo:** Se n for divisível por p, então $n = p \times q$. Se q fosse menor do que p, então q ou um fator primo de q teriam dividido n anteriormente. A complexidade do algoritmo então é $O(\sqrt(n))$.

Checar se um número é primo

Podemos melhorar novamente? SIM!

Melhoria 2

Testar se n é divisível pelos divisores $\in [3, 5, 7, \ldots, \sqrt(n)]$. Podemos desconsiderar os números pares visto que se um número par fosse um divisor de n, então n já teria sido dividido anteriormente por 2. Devemos checar então o valor 2 em um caso separado! A complexidade do algoritmo então é $O(\frac{\sqrt(n)}{2})$.

Checar se um número é primo

Podemos melhorar novamente? SIM!

Melhoria 2

Testar se n é divisível pelos divisores $\in [3, 5, 7, \ldots, \sqrt(n)]$. Podemos desconsiderar os números pares visto que se um número par fosse um divisor de n, então n já teria sido dividido anteriormente por 2. Devemos checar então o valor 2 em um caso separado! A complexidade do algoritmo então é $O(\frac{\sqrt(n)}{2})$.

Checar se um número é primo

Podemos melhorar ainda mais? SIM!!!

Melhoria 3

Testar se n é divisível pelos divisores primos $\leq \sqrt(n)$. Essa ideia gera um algoritmo mais rápido que $O(\sqrt(n))$, de modo que a complexidade do algoritmo é $O(|primes \leq sqrt(n)|)$.

Exemplo: existem **500** números ímpares entre $[1 \ldots \sqrt(n^6)]$, contudo, existem apenas **168** números primos nesse intervalo.

O **teorema dos números primos** nos diz que o número de primos menores ou iguais a M é limitado superiormente por $O(\frac{M}{ln(M)-1})$, logo, a complexidade da função para teste números primos é $O(\frac{\sqrt(n)}{ln(\sqrt(n))})$.

Checar se um número é primo

Podemos melhorar ainda mais? SIM!!!

Melhoria 3

Testar se n é divisível pelos divisores primos $\leq \sqrt(n)$. Essa ideia gera um algoritmo mais rápido que $O(\sqrt(n))$, de modo que a complexidade do algoritmo é $O(|primes \leq sqrt(n)|)$.

Exemplo: existem **500** números ímpares entre $[1 \ldots \sqrt(n^6)]$, contudo, existem apenas **168** números primos nesse intervalo.

O **teorema dos números primos** nos diz que o número de primos menores ou iguais a M é limitado superiormente por $O(\frac{M}{ln(M)-1})$, logo, a complexidade da função para teste números primos é $O(\frac{\sqrt(n)}{ln(\sqrt(n))})$.

Crivo de Eratóstenes

Qual o problema da Melhoria 3

Necessita que sejam gerados previamente todos os números primos até $\sqrt(n)$.!

Como calcular todos os números primos ate $\sqrt(n)$ de uma maneira rápida?

Algoritmo do Crivo de Eratóstenes (*Siege de Eratóstenes*)

Crivo de Eratóstenes

Propósito

Gerar uma lista de números primos no intervalo de $[0, n]$.

Ideia

Utiliza a estrutura **bitset**.
Primeiramente, seta todos os números no intervalo como *possiveis primos*, com exceção dos valores 0 e 1 (**cuidado!!** ao vezes o número 1 é considerado primo). Após isso, pegue o número 2 e sete todos os múltiplos de 2 (a partir de 4) como sendo um número não primo. Após isso, pegue o número 3 e marque todos os seus múltiplos (a partir de 6) como não primos. Depois, pegue o número 5 e marque todos os seus múltiplos (a partir de 10) como não primos (e assim por diante).
Após isso, todos os números ainda marcados no intervalo $[0, n]$ **são primos**. A complexidade desse algoritmo é de $O(nloglogn)$.

Crivo de Eratóstenes

```
ll _sieve_size;                    // ll is defined as: typedef long long ll;
bitset<10000010> bs;                  // 10^7 should be enough for most cases
vi primes;                   // compact list of primes in form of vector<int>

void sieve(ll upperbound) {          // create list of primes in [0..upperbound]
  _sieve_size = upperbound + 1;                  // add 1 to include upperbound
  bs.set();                                                // set all bits to 1
  bs[0] = bs[1] = 0;                                     // except index 0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);  // also add this vector containing list of primes
} }                                           // call this method in main method
```

Crivo de Eratóstenes

```
bool isPrime(ll N) {                  // a good enough deterministic prime tester
  if (N <= _sieve_size) return bs[N];                     // O(1) for small primes
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;                    // it takes longer time if N is a large prime!
}                      // note: only work for N <= (last prime in vi "primes")^2

// inside int main()
  sieve(10000000);                         // can go up to 10^7 (need few seconds)
  printf("%d\n", isPrime(2147483647));                           // 10-digits prime
  printf("%d\n", isPrime(136117223861LL));        // not a prime, 104729*1299709
```

Encontrando Fatores Primos - Fatorando um número

Ideia

Um inteiro N (*não primo*) pode ser expresso como $N = PF \times N'$, em que PF é um fator primo e $N' = N/PF$ - ou seja, podemos reduzir o valor de N através da remoção de seu fator primo. Este processo pode ser repetido até que $N = 1$.
Como sabemos, todos os fatores primos de N são menores ou iguais a $sqrt(n)$. Logo, podemos utilizar o Crivo de Eratóstenes para gerar todos os primos de $[0, \sqrt{}(n)]$ e verificarmos quais desses números primos são fatores primos de N.
A complexidade desse algoritmo é de $O(\sqrt{}(n)/ln(\sqrt{}(n)))$.

Encontrando Fatores Primos - Fatorando um número

```
vi primeFactors(ll N) {          // remember: vi is vector<int>, ll is long long
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx]; // using PF = 2, then 3,5,7,... is also ok
  while (N != 1 && (PF * PF <= N)) {   // stop at sqrt(N), but N can get smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }    // remove this PF
    PF = primes[++PF_idx];                                 // only consider primes!
  }
  if (N != 1) factors.push_back(N);     // special case if N is actually a prime
  return factors;   // if N does not fit in 32-bit integer and is a prime number
}                         // then 'factors' will have to be changed to vector<ll>

// inside int main(), assuming sieve(1000000) has been called before
  vi res = primeFactors(2147483647);            // slowest, 2147483647 is a prime
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("> %d\n", *i);

  res = primeFactors(136117223861LL);   // slow, 2 large pfactors 104729*1299709
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("# %d\n", *i);
```

Funções envolvendo Fatores Primos

- Contar o número de fatores primos de N
- Contar o número de diferentes fatores primos de N
- Somar os fatores primos de N
- Contar o número de divisores de N
- Somar os divisores de N

Contar o número de fatores primos de N

```
lli numPF(lli n){

   lli indice(0), resp(0);
   lli pf = primos(indice);

   while(n!=1 && (pf*pf<=n)){
     while(n%pf==0){ n/=pf; resp++; }
     pf = primos[++indice];
   }

   if(n!=1) return resp+1; /* no caso de 'n' ser um número primo */
   return resp; /* caso 'n' não seja primo */

}
```

Contar o número de diferentes fatores primos de N

```
lli numDiffPF(lli n){

   lli indice(0), cont(0);
   lli pf = primos(indice);

   while(n!=1 && (pf*pf<=n)){
     while(n%pf==0) n/=pf;
     pf = primos[++indice];
     resp++;
   }

   if(n!=1) return resp+1; /* no caso de 'n' ser um número primo */
   return resp; /* caso 'n' não seja primo */

}
```

Somar os fatores primos de N

```
lli sumPF(lli n){

   lli indice(0), soma(0);
   lli pf = primos(indice);

   while(n!=1 && (pf*pf<=n)){
     while(n%pf==0){ n/=pf; soma+=pf; }
     pf = primos[++indice];
   }

   if(n!=1) return n; /* no caso de 'n' ser um número primo */
   return soma; /* caso 'n' não seja primo */

}
```

Contar o número de divisores N

Se $N = a^i \times b^j \times \ldots \times c^k$ então N possui $(i+1) \times (j+1) \times \ldots \times (k+1)$ divisores.

```
lli numDiv(lli n){

   lli indice(0), resp(1);
   lli pf = primos(indice);

   while(n!=1 && (pf*pf<=n)){
     int aux = 0;
     while(n%pf==0){ n/=pf; soma+=pf; aux++; }
     resp*= (aux+1);
     pf = primos[++indice];
   }

   if(n!=1) resp*= 2; /* no caso de 'n' ser um número primo */
   return resp; /* caso 'n' não seja primo */

}
```

Somar o número de divisores N

Se $N = a^i \times b^j \times \ldots \times c^k$ então a soma dos divisores de N é $\frac{a^{i+1}-1}{a-1} \times \frac{b^{j+1}-1}{b-1} \times \ldots \times \frac{c^{k+1}-1}{c-1}$.

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;   // start from ans = 1
  while (PF * PF <= N) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);  // last
  return ans;
}
```

Crivo Modificado - Número de fatores primos de uma range de valores

Se o número de diferentes fatores primos precisar ser determinado para um intervalo de valores inteiros então existe um algoritmo melhor do que chamar o método numDiffPF(n) repetivas vezes.
A ideia consiste em realizar uma pequena modificação no algoritmo do crivo, de modo que no lugar de encontrar os fatores primos, nós começamos pelos números primos e modificamos os valores de seus múltiplos.

```
memset(numDiffPF, 0, sizeof numDiffPF);
for (int i = 2; i < MAX_N; i++)
  if (numDiffPF[i] == 0)                          // i is a prime number
    for (int j = i; j < MAX_N; j += i)
      numDiffPF[j]++;              // increase the values of multiples of i
```

Máximo Divisor Comum (GCD) e Mínimo Múltiplo Comum (LCM)

Definição

O GDC de dois inteiros (a,b) detonado por gcd(a,b) é definido como o maior inteiro positivo d tal que $d|a$ e $d|b$, ou seja, d divide a e d divide b.

$$gcd(20, 12) = 4 \tag{1}$$

Uma aplicação prática para GCD é a de simplificar frações:
$\frac{4}{8} = \frac{4/gcd(4,8)}{8/gcd(4,8)} = \frac{1}{2}$

Máximo Divisor Comum (GCD) e Mínimo Múltiplo Comum (LCM)

Solução - Algoritmo de Euclides

```
int gcd(int a, int b) { return (b == 0 ? a : gcd(b, a%b)); }
```

Máximo Divisor Comum (GCD) e Mínimo Múltiplo Comum (LCM)

O LCM de dois inteiros (a,b), detonado por $lcm(a, b)$ é definido como o menor inteiro positivo l tal que $a|l$ e $b|l$.

$$lcm(20, 12) = 60 \tag{2}$$

É sabido que $a \times b = gcd(a, b) \times lcm(a, b)$. Ou seja, calculando o $gcd(a, b)$ através do algoritmo de Euclides, podemos obter o $lcm(a, b)$.

Algoritmo de Euclides Estendido

Além de computar o gcd(a,b), calcula os coeficientes x e y de modo que:

$$ax + by = gcd(a, b) \tag{3}$$

Exemplo - MDC(120, 23)

(1) $120/23 = 5$ resta 5
(2) $23/5 = 4$ resta 3
(3) $5/3 = 1$ resta 2
(4) $3/2 = 1$ resta 1
(5) $2/1 = 2$ resta 0

- 5 = 1*120 - 5*23
- 3 = 1*23 - 4*5 (Substituindo o 5 temos)
  3 = 1*23 - 4*(1*120 - 5*23)
  3 = -4*120 + 21*23
- 2 = 1*5 - 1*3 (Substituindo o valor de 5 e 3 temos)
  2 = 1(1*120 - 5*23) - 1(-4*120 + 21*23)
  2 = 5*120 - 26*23
- 1 = 1*3 - 1*2 (Novamente substituindo 3 e 2)
  1 = 1(-4*120 + 21*23) - 1(5*120 - 26*23)
  1 = -9*120 + 47*23

Algoritmo de Euclides Estendido

```
void euclidianoEstendido(int a, int b, int& alpha, int& beta, int& mdc) {
    int x[2] = {1, 0};
    int y[2] = {0, 1};

    /* Enquanto o resto da divisão de a por b não for zero, eu continuo o algoritmo. */
    while (a % b != 0) {
        int quociente = a / b;

        /* Atualizando os valores de a e b. */
        int temp = a;
        a = b;
        b = temp % b;

        /* Atualizando os valores de x e y. */
        int X = x[0] - (x[1] * quociente);
        int Y = y[0] - (y[1] * quociente);

        x[0] = x[1];
        x[1] = X;
        y[0] = y[1];
        y[1] = Y;


    }

    mdc = b;
    alpha = x[1];
    beta = y[1];
}
```

Definição

Equação polinomial que permite duas ou mais variáveis assumirem apenas valores **inteiros**. Uma **Equação Linear Diofantina** é uma equação da forma: $ax + by = c$

$$25x + 18y = 839 \tag{4}$$

Equações Lineares Diofantinas

Resolvendo Equações Lineares Diofantinas

Seja a e b inteiros com $d = gcd(a, b)$. A equação $ax + by = c$ não possui solução inteira se $d|c$ não é verdade. Mas, se $d|c$, então existem infinitas soluções inteiras.
Pelo algoritmo de Euclides Estendido, podemos encontrar inteiros s e t de modo que $as + bt = gcd(a, b)$. Uma vez encontrados s e t, como estamos assumindo que $gcd(a, b)|c$, então existe um inteiro k tal que:

$$a(s * k) + b(t * k) = gcd(a, b) * k \tag{5}$$

Ou seja, $x = sk$ e $y = tk$ são uma solução para a equação.