Clase 04: Number Theory

Miguel Miní

16-01-2020

Number Theory

En esta sección veremos herramientas de teoría de números indispensables, muchos de los problemas en programación competitiva necesitan el entendimiento de estos problemas:

Relación de congruencia

Para un entero positivo n, dos números a y b se dicen congruentes módulo n. Si la diferencia $a - b$ es un entero múltiplo de n. Y es considerada como:

$$a \equiv b \ (mod \ n)$$

Propiedades:

- Reflexibidad: $a \equiv a \ (mod \ n)$
- Simetría: $a \equiv b \ (mod \ n)$ si $b \equiv a \ (mod \ n)$, para todo a, b, n.
- Transitividad: si $a \equiv b \ (mod \ n)$ y $b \equiv c \ (mod \ n)$, entonces $a \equiv c \ (mod \ n)$

Si $a = pn + r$ y $b = qn + r$ (1) (https://en.wikipedia.org/wiki/Division_algorithm), entonces $a \sim r$, $b \sim r$ y $a \sim b$ al mínimo r positivo se le denomina el característico de la clase y denotamos

$$[a] = [b] = r$$

.

Aritmética modular

- Las relaciones de congruencia funcionan bien bajo operaciones aritméticas, esto quiere decir que $[a] + [b] = [a + b]$ y $[a].\,[b] = [a.\,b]$.

nota: esto quiere decir que si yo hago operaciones modulares sobre n, entonces yo puedo realizarlas en cualquier orden, además, si la respuesta es menor a n, la respuesta final será igual aplicando aritmética usual o aritmética modular.

Puede encontrar más información aquí:

- Modular Arithmetic for Beginners (https://codeforces.com/blog/entry/72527)

Máximo Común Divisor

Dados a y b, el máximo común divisor es el máximo entero g menor a min(a, b), tal que $a \equiv 0 \ (mod \ g)$ y $b \equiv 0 \ (mod \ g)$.

Si d divide a g, además, se cumple que $g = d \ gcd(a/d, b/d)$.

Exponenciación Rápida:

Este algoritmo solo necesita que b sea un entero en base 2, y que se aplique una función asociativa sobre a, b veces. Esto nos deja la puerta abierta para poder hacer ab, a^b, A^b donde A puede ser una matriz.

Descripción:

En este caso nosotros estamos implementando $a^b \ (mod\ m)$, para ello podemos pensar b como una secuencia de bits: $b_k, b_{k-1}, \dots, b0$, entonces por propiedad de la exponenciación: $a^b = (a^{b_k 2^k})(a^{b_{k-1} 2^{k-1}}) \dots (a^{b_0 2^0})$, esto nos dice que podemos hallar la secuencia $a^{2^0}, a^{2^1}, \dots, a^{2^k}$ y elegir cual conviene respecto a la secuencia de b.

```
int ex(int a, int b, int m) {
  int r = 1;
  while (b > 0) {
    if (b&1) r = r * 1ll * a % m; //aca aplicas la funcion
    a = a * 1ll * a % m; //aca aplicas la funcion
    b >>= 1;
  }
  return r;
}
```

Algoritmo de Euclides:

Euclides describe este algoritmo en sus Elementos, c. 300 BC, este es basado en usar recursivamente el algoritmo de la división.

Descripción:

hacemos $x_0 = a$, $x_1 = b$, entonces en cada paso i, tenemos:

$x_{i-1} = q_i x_i + x_{i+1}$ para $0 \leq i \leq k$

así hasta que $x_k = 0$, en ese caso x_{k-1} es la respueta a nuestro problema.

Propiedades del Algoritmo de Euclides:

para $0 < i \leq k$, $q_i >= 1$ luego $x_{i-1} >= x_i + x_{i+1}$ y $x_i > x_{i+1}$ con lo cuál $x_{i-1} > 2 * x_{i+1}$, por tanto, la secuencia x_i decrece para elementos consecutivos con la misma paridad en forma exponencial.

Por otro lado, si d divide a x_{i-1} y x_i, entonces d divide a x_{i+1}, por tanto, el máximo común divisor divide a x_{k-1}. En el mismo sentido, todos los x_i con $0 \leq i < k - 1$ son divisibles por x_{k-1}. Concluimos x_{k-1} es el máximo común divisor de a y b.

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

int gcd(int a, int b) {
  while (b != 0) {
    int r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

Identidad de Bézout

La Identidad de Bézout (2) (https://brilliant.org/wiki/bezouts-identity/) nos dice que dados a y b enteros positivos, existen x e y que cumplen:

$$ax + by = gcd(a, b)$$

No probaremos que la identidad se cumple, pero esta sale facilmente manipulando el algoritmo de $Euclides$

Sin embargo, la identidad junto con el algoritmo de euclides nos dicen como hallar dichos x e y.

$$ax + by = gcd(a, b)$$

$$bx + (a\%b)y = gcd(b, a\%b)$$

La última ecuación se puede reescribir como:

$$bx + (a - (a//b)b)y = gcd(b, a\%b)$$

Pero además sabemos que gcd(a, b) = gcd(b, $a\%b$). Si reacomodamos la segunda ecuación modificada tenemos:

$$ay + b(x - (a//b)y) = gcd(a, b)$$

Con lo cual tenemos una función recursiva para x e y:

$$(x_i, y_i) = (y_{i+1}, x_{i+1} - (a//b)y_{i+1})$$

Implementación:

```
int bezout(int a, int b, int& x_0, int& y_0) {
  if (b == 0) { //solucion base a 1 + b 0 = a
    x_0 = 1;
    y_0 = 0;
    return a;
  }
  int x_1, y_1;
  int g = bezout(b, a%b, x_1, y_1);
  x_0 = y_1;
  y_0 = x_1 - (a/b) * y_1;
  return g;
}
```

Como una nota final, luego de que yo ya halle una solución a este problmea, puedo hallar todas:

$$\frac{abt}{g} - \frac{abt}{g} + ax + by = g$$

$$a(x + \frac{bt}{g}) + b(y - \frac{at}{g}) = g$$

tenga presente que al agregar un valor a x, entonces y deberia variar, por otro lado el valor que agrego debe ser múltiplo de a y múltiplo de b, por tanto debería ser el mínimo común múltiplo.

Como sería la solución para:

$$ax + by = c$$

Pequeño Teorema de Fermat

El pequeño teorema de Fermat nos dice que cuando n es primo y $0 < a < n$, entonces:

$$a^{n-1} \equiv 1 \ (mod \ n)$$

Para llegar a esto primero debemos definir la inversa de a módulo n, como un entero b tal que:

$ab \equiv 1 \ (mod \ n)$

vemos fácilmente que para n primo, y $0 < a < n$ se cumple por el Teorema de Bézout:

$ax + ny = 1$

entonces, $ax \sim_n 1$

ahora, debido a que esto se cumple:

afirmación: $ai \equiv aj \ (mod \ n)$ si y solo si $[i] = [j]$.

para ver esto, solo debemos multiplicar por la inversa de a a ambos lados.

por tanto, esto quiere decir que $a(1, 2, \dots, n-1)$ es una permutación!

lo que implica que:

$$\prod_{i=1}^{n} ai \equiv \prod_{i=1}^{n} i$$

,

por tanto

$$a^{n-1} \prod_{i=1}^{n} i \equiv \prod_{i=1}^{n} i$$

Teorema de Euler

De la misma forma que en el caso anterior, podemos hacer el mismo proceso con los números copŕimos a n, dandonos la fórmula:

$$a^{\phi(n)} \equiv 1 \ (mod \ n)$$

Puede ver más información en: Euler's totient function (https://cp-algorithms.com/algebra/phi-function.html)

Teorema Chino del Resto

El teorema chino del resto se presenta en algunos problemas de teoría de números, donde debo resolver lo siguiente:

$$x = a_1 \ (mod \ b_1)$$

$$x = a_2 \ (mod \ b_2)$$

$$\vdots$$

$$x = a_n \ (mod \ b_n)$$

Para resolver este tipo de sistemas (que no necesariamente tiene solución) podemos reducirlo a resolver para dos ecuaciones.

$$x \equiv a_1 \ (mod \ b_1)$$

$$x \equiv a_2 \ (mod \ b_2)$$

cuyo sistema es equivalente al siguiente:

$$x = a_1 + k_1 b_1$$

$$x = a_2 + k_2 b_2$$

que es lo mismo:

$$a_2 + k_2 b_2 = a_1 + k_1 b_1$$

$$a_2 - a_1 = x^{'} b_1 - y^{'} b_2$$

ahora supongamos que hallamos $x^{'}$, entonces

$$x = a_1 + x^{'} b_1$$

Como sabemos que todas las solciones tiene la forma: $\frac{x^{'}(a_2 - a_1) + b_2 t}{g}$

entonces:

$$x = \frac{x^{'}(a_2 - a_1)b_1 + b_1 b_2 t}{g}$$

por tanto:

$$x \equiv \frac{x^{'}(a_2 - a_1)b_1}{g} \ (mod \ \frac{b_1 b_1}{g})$$

lo que quiere decir que hemos comprimido dos ecuaciones!

Puedes ver más en: (Tutorial) Chinese Remainder Theorem (https://codeforces.com/blog/entry/61290)

Hallando Divisores:

Para hallar los divisores de un número n, tenemos por ejemplo un enfoque sencillo:

```
for (int d = 1; d <= n; ++d) {
  if (n % d == 0) {
    cout << d << " es un divisor" << endl;
  }
}
```

Pero se puede hacer en mejor tiempo:

```
for (int d = 1; d*d <= n; ++d) {
  if (n % d == 0) {
    cout << d << " es un divisor" << endl;
    if (d != n/d) {
      cout << n/d << "es un divisor" << endl;
    }
  }
}
```

La gran diferencia es que el segundo corre en un tiempo $O(\sqrt{n})$

Esto se puede debido a la observación que si $ab = n$, entonces a o b es menor a $\sqrt{(n)}$.

Hallando Numeros Primos

Primero debemos darnos cuenta que un número es primo si realizando la solución anterior, el número solo debe ser divisible por 1, y por defecto por n.

```
if (n == 1) cout << "No es un numero primo" << endl;
else {
  bool is_prime = 1;
  for (int d = 2; d*d <= n; ++d) {
    if (n % d == 0) {
      is_prime = 0;
    }
  }
  if (is_prime) cout << "Es un numero primo" << endl;
  else cout << "No es un numero primo" << endl;
}
```

Lamentablemente si queremos verificar que los n primeros numeros sean primos, esto nos demoraria $O(n\sqrt{n})$

Criba de Eratostenes

La criba de Eratostenes nos presenta un metodo simple para saber si los primeros n números y corre en un tiempo $O(n \log \log n)$, la idea es eliminando de menor a mayor todos los números que son múltiplos de algún primo.

```
int n;
vector<char> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i <= n; i++) {
    if (is_prime[i] && (long long)i * i <= n) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

Puede ver más en: Sieve of Eratosthenes (https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html)

Además se puede reducir su tiempo a $O(n)$ con algunas más observaciones:

Puede ver más en: (Tutorial) Math note — linear sieve (https://codeforces.com/blog/entry/54090)

Con una criba además podemos verificar que un número es primo en $O(\frac{\sqrt{n}}{\log n})$.

Debido al teorema de los números primos (https://en.wikipedia.org/wiki/Prime_number_theorem) que nos asegura que en el rango 1 hasta m solo hay $O(\frac{m}{\log m})$ (por qué solo nos importan solo los primos? porque si un número d divide a n, entonces d debe tener un divisor primo menor o igual a el, que debe dividir a n también)

Hallando divisores primos más rapido

Según el proceso que describe Eratostenes, yo puedo construir la criba manteniendo cuál es el menor divisor primo que divide a cada número i.

criba modificada para hallar menor divisor primo de un número:

```
std::vector <int> prime;
bool is_composite[MAXN];
int min_prime[MAXN];

void sieve (int n) {
    std::fill (is_composite, is_composite + n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) {
          prime.push_back (i);
          min_prime[i] = i;
        }
        for (int j = 0; j < prime.size () && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            min_prime[i * prime[j]] = prime[j];
            if (i % prime[j] == 0) break;
        }
    }
}
```

función para hallar los divisores primos de un número:

```
vector<pair<int, int>> t; //aca guardare la descomposicion en primos y sus exponentes
while (n != 1) {
  int p = min_prime[n];
  int e = 0;
  while (n%p == 0) {
    e += 1;
    n /= p;
  }
  t.push_back({p, e});
}
```

Contest

El contest lo puedes encontrar aquí (https://vjudge.net/contest/352567).