Aritmética modular y factorización

(o "¿Qué de lo que vi en Discreta 1 me sirve para las competencias?")

June 22, 2018

Números primos - repaso

- Un número $p \in \mathbb{N}$ es *primo* si sus únicos divisores positivos son 1 y p.
- Dado $n \in \mathbb{N}$, podemos factorizarlo de forma única como

$$n = p_1^{e_1} ... p_k^{e_k}$$

  (donde los p_i son primos y los $e_i \in \mathbb{N}$)
- Encontrar la factorización de un número aparece muy seguido en problemas que involucran divisibilidad.

Factorización - algoritmo $O(\sqrt{n})$

- Para factorizar, podemos utilizar el hecho de que un número es primo si y sólo si no es divisible por ningún primo menor o igual que $\sqrt{n}$
- Algoritmo: Probamos dividir el número por los naturales a partir de 2 en orden creciente. Mientras podemos, dividimos n por ese número, y cortamos cuando el número que estamos probando es $> \sqrt{n}$

Factorización - algoritmo $O(\sqrt{n})$

- Para factorizar, podemos utilizar el hecho de que un número es primo si y sólo si no es divisible por ningún primo menor o igual que $\sqrt{n}$
- Algoritmo: Probamos dividir el número por los naturales a partir de 2 en orden creciente. Mientras podemos, dividimos n por ese número, y cortamos cuando el número que estamos probando es $> \sqrt{n}$

```
map<long long,int> fact(long long n){ // O(sqrt(n))
  // devuelve mapeo primo -> exponente
  map<long long,int> res;
  for(long long i = 2; i*i <= n; ++i){
    while(n % i == 0){
      res[i]++;
      n /= i;
    }
  }
  if(n > 1)
    res[n]++;
  return res;
}
```

Factorización $O(log(n))$ con criba

- El algoritmo anterior funciona bastante bien si queremos factorizar un sólo número. Pero podemos hacerlo mejor si necesitamos factorizar varios.
- Podemos usar la criba de Eratóstenes, pero en vez de indicar si un número es primo o no, indicamos un primo que lo divida.
- Una vez que tenemos la criba, podemos saber en $O(1)$ un primo que divide al número. Vamos dividiéndolo por un primo que lo divide hasta llegar a 1.
- Este algoritmo realiza tantos pasos como factores primos tenga el número. Por lo tanto es $O(log(n))$.

Factorización con criba - código

```cpp
int p[MAXN]; // mapea numero -> factor primo

void init_criba(){ // O(MAXN log(log(MAXN)))
  fill(p, p + MAXN, -1); // para saber si encontramos primo
  for(int i = 2; i < MAXN; ++i)
    if(p[i] < 0) // i es primo
      for(int j = i; j < MAXN; j += i)
        p[j] = i;
}

map<int,int> fact(int n){ // O(log(n))
                          // (dado que inicializamos p)
  // devuelve mapeo primo->exponente
  map<int,int> res;
  while(n > 1){
    res[p[n]]++;
    n /= p[n];
  }
  return res;
}
```

Factorización - ¿qué algoritmo usar?

- Para factorizar pocos números hasta $10^{12} \rightarrow$ primer algoritmo.
- Para factorizar muchos números hasta $10^{7} \rightarrow$ segundo algoritmo.
- Hay algoritmos más eficientes que funcionan para números hasta 10^{18}, pero los omitimos por ser más complejos y porque no suelen ser necesarios.

Generar divisores - algoritmo $O(\sqrt{n})$

- Para encontrar todos los divisores de un número, podemos usar que si $a \cdot b = n$, entonces $a \leq \sqrt{n}$ ó $b \leq \sqrt{n}$.
- Entonces, si iteramos para todos los números i entre 1 y $\sqrt{n}$ y cuando $i \mid n$ agregamos i y $\frac{n}{i}$ a la lista de divisores, de esta forma encontramos todos los divisores.

Generar divisores - algoritmo $O(\sqrt{n})$

- Para encontrar todos los divisores de un número, podemos usar que si $a \cdot b = n$, entonces $a \leq \sqrt{n}$ ó $b \leq \sqrt{n}$.
- Entonces, si iteramos para todos los números i entre 1 y $\sqrt{n}$ y cuando $i \mid n$ agregamos i y $\frac{n}{i}$ a la lista de divisores, de esta forma encontramos todos los divisores.

```cpp
vector<long long> divs(long long n){ // O(sqrt(n))
  vector<long long> res;
  for(long long i = 2; i*i <= n; i++){
    if(n % i == 0){
      res.push_back(i);
      if(i*i != n)
        res.push_back(n/i);
    }
  }
  return res;
}
```

Generar los divisores de muchos números chicos

- Si queremos los divisores de muchos números chicos (hasta 10^6) podemos simplemente iterar para cada número hasta MAXN, y agregarlo a la lista de divisores de todos sus múltiplos hasta MAXN.

Generar los divisores de muchos números chicos

- Si queremos los divisores de muchos números chicos (hasta 10^6) podemos simplemente iterar para cada número hasta MAXN, y agregarlo a la lista de divisores de todos sus múltiplos hasta MAXN.

```
vector<int> divs[M];

void init_divs(){
  for(int i = 1; i < M; ++i)
    for(int j = i; j < M; j+=i)
      divs[j].push_back(i);
}
```

Generar los divisores de muchos números chicos

- Si queremos los divisores de muchos números chicos (hasta 10^6) podemos simplemente iterar para cada número hasta MAXN, y agregarlo a la lista de divisores de todos sus múltiplos hasta MAXN.

```
vector<int> divs[M];

void init_divs(){
  for(int i = 1; i < M; ++i)
    for(int j = i; j < M; j+=i)
      divs[j].push_back(i);
}
```

- La complejidad de este algoritmo es

$$O(\sum_{i=1}^{M} \frac{M}{i}) = O(M \sum_{i=1}^{M} \frac{1}{i}) = O(M\ log(M))$$

Aritmética modular - Introducción

- Muchos problemas cuya respuesta es un número muy grande no piden que se de el resultado explícitamente, sino el resto de dividir el resultado por algún número M (es decir, el resultado *módulo* M).
- Para ese tipo de problemas, tenemos que usar algunas propiedades del módulo.

Aritmética modular - Propiedades

- Decimos que $a \equiv_M b$ (a "congruente" a b módulo M) cuando a y b tienen el mismo resto en la división por M (o sea , cuando $M \mid b - a$).
- Propiedad:

$$\text{Si } a \equiv_M a' \text{ y } b \equiv_M b', \text{ entonces} \quad \begin{aligned} a + b &\equiv_M a' + b' \\ a - b &\equiv_M a' - b' \\ a \cdot b &\equiv_M a' \cdot b' \end{aligned}$$

- Esta propiedad nos permite ir sacando módulo después de cada operación, evitando el overflow.
- Esta propiedad **no** vale para la división.

Aritmética modular - Cont.

```cpp
const long long M = 1e9 + 7;

long long suma(long long a, long long b){
  return (a + b) % M;
}

long long resta(long long a, long long b){
  return (a + M - b) % M;
}

long long mult(long long a, long long b){
  return (a * b) % M;
}
```

- En C++, se debe tener cuidado con los números negativos al sacar módulo, ya que `(-5) % 3` es −2, y no 1 (que sería lo más razonable). Se resuelve sumando M cuando el resultado es negativo.

Pequeño teorema de Fermat

- Para poder dividir con módulo, introducimos el concepto de inverso modular: a^{-1} es tal que $a \cdot a^{-1} \equiv_M 1$ y lo llamamos "inverso de a módulo M".
- Si tenemos inverso modular, entonces $a/b \equiv_M a \cdot b^{-1}$.
- El inverso modular no siempre existe, pero si M es primo podemos usar lo siguiente:

Pequeño teorema de Fermat

Si M es primo y $a \not\equiv_M 0$, entonces $a^{M-1} \equiv_M 1$.

- Luego, si M es primo y $a \not\equiv_M 0$, entonces su inverso es a^{M-2} *mod* M

Exponenciación logarítmica

- ¿Cómo podemos hacer para evaluar $a^b \ mod \ M$ eficientemente?
- Podemos usar que $a^b = (a^{b/2})^2$ para obtener un algoritmo $O(log(b))$.

Exponenciación logarítmica

- ¿Cómo podemos hacer para evaluar $a^b \ mod \ M$ eficientemente?
- Podemos usar que $a^b = (a^{b/2})^2$ para obtener un algoritmo $O(log(b))$.

```
long long modexp(long long a, long long b, long long M){
  if(b == 0)
    return 1;
  if(b % 2 == 1)
    return (a * modexp(a, b-1, M)) % M;
  long long t = modexp(a, b/2, M);
  return (t*t) % M;
}
```

Exponenciación logarítmica - código iterativo

- Alternativamente, podemos hacer una implementación iterativa (más rápida) fijándonos en la descomposición binaria de b.

Exponenciación logarítmica - código iterativo

- Alternativamente, podemos hacer una implementación iterativa (más rápida) fijándonos en la descomposición binaria de *b*.

```
long long modexp(long long a, long long b, long long M){
  long long res = 1;
  while(b){
    if(b & 1)
      res = (res * a) % M;
    b >>= 1;
    a = (a * a) % M;
  }
  return res;
}
```

Aplicación - números combinatorios

- Recordemos que $\binom{n}{k}$ es la cantidad de formas de elegir k elementos distintos de un conjunto de n.
- Para $0 \leq k \leq n$, tenemos $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$.
- Si M es primo (y más grande que el máximo n), podemos calcular eficientemente $\binom{n}{k}$ $mod\ M$, si antes precomputamos los factoriales $mod\ M$.

Números combinatorios - código

```
const long long M = 1e9 + 7;
long long fact[MAXN];

void init_facts(){
  fact[0] = 1;
  for(int i = 1; i < MAXN; ++i)
    fact[i] = (fact[i-1] * i) % M;
}

long long comb(int n, int k){
  if(k < 0 || k > n)
    return 0;
  long long a = fact[n]; // n!
  long long b = modexp(fact[k], M-2); // (k!)^(-1)
  long long c = modexp(fact[n-k], M-2); // ((n-k)!)^(-1)
  return (((a*b) % M)*c) % M;
}
```

Greatest common divisor (GCD)

El máximo común divisor (o gcd por sus siglas en inglés) de a y b es el mayor número que es divisor tanto de a como de b.

Algunas propiedades:

- Si $d \mid a$ y $d \mid b$ entonces $d \mid gcd(a, b)$.
- El gcd es conmutativo, asociativo y la multiplicación distribuye sobre el gcd: $gcd(a \cdot d, b \cdot d) = gcd(a, b) \cdot d$.

GCD (cont.)

- *Propiedad:* $a = q \cdot b + r \implies gcd(a, b) = gcd(b, r)$
- Esta propiedad nos da un algoritmo para calcular gcd (algoritmo de Euclides):

GCD (cont.)

- *Propiedad:* $a = q \cdot b + r \implies gcd(a, b) = gcd(b, r)$
- Esta propiedad nos da un algoritmo para calcular gcd (algoritmo de Euclides):

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

GCD (cont.)

- *Propiedad:* $a = q \cdot b + r \implies gcd(a, b) = gcd(b, r)$
- Esta propiedad nos da un algoritmo para calcular gcd (algoritmo de Euclides):

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

- La complejidad es $O(log(max(a, b)))$.
- g++ provee una función `__gcd`. Recomendamos **no** usarla porque es más lenta y tiene problemas con los ceros.

Least common multiple (LCM)

- Una función muy relacionada al gcd es el mínimo común múltiplo (lcm).
- $lcm(a, b)$ es el menor número positivo que es múltiplo de a y de b.
- Se puede calcular haciendo $lcm(a, b) = \frac{a}{gcd(a,b)} \cdot b$.
- Ojo que algunos problemas parece que salen haciendo lcm, pero hay que tener en cuenta la posibilidad de overflow.

Otras cosas

Cosas relacionadas que no incluimos (por tiempo) pero que también sirven, aunque no son tan comunes.

- Algoritmo de Euclides extendido y ecuaciones diofánticas.
- Función φ de Euler y sus propiedades.
- Teorema chino del resto.