Aritmética y Primalidad

Pablo Zimmermann

Universidad Nacional de Rosario

11th Caribbean Camp

Contenidos

Contenidos

Aritmética modular

Aritmética módulo $M (\mathbb{Z}_M)$

La aritmética módulo M consiste en una modificación de la aritmética usual de números enteros, en la cual trabajamos únicamente con el resto de los números al ser divididos por un cierto entero fijo M > 0, ignorando "todo lo demás" de los números involucrados.

- Así, $11 = 18$ si estamos trabajando módulo 7, pues ambos dejan un resto de 4 en la división por 7. Esto se suele notar $11 \equiv 18 \pmod 7$ o $11 \equiv_7 18$
- Una forma operacional de ver esta aritmética es suponer que todo el tiempo tenemos los números reducidos al rango de enteros en [0, M), y tomamos el resto de la división por M para devolverlos a ese rango luego de cada operación.

Aritmética modular

De esta forma, las operaciones de suma, resta y multiplicación se extienden trivialmente y mantienen sus propiedades conocidas

$$a \pm b = c \quad \Longrightarrow \quad a \pm b \equiv_m c$$

$$a.b = c \quad \Longrightarrow \quad a.b \equiv_m c$$

Aritmética modular

De esta forma, las operaciones de suma, resta y multiplicación se extienden trivialmente y mantienen sus propiedades conocidas

$$a \pm b = c \quad \Longrightarrow \quad a \pm b \equiv_m c$$

$$a.b = c \quad \Longrightarrow \quad a.b \equiv_m c$$

La división no es tan simple y se define como la multiplicación por el inverso, de modo que

$$a/b \quad \Longrightarrow \quad a.b^{-1} \qquad \text{con} \qquad b.b^{-1} = 1$$

¿Siempre existe un inverso módulo m? ¿Cómo lo calculamos?

Pequeño teorema de Fermat

Teorema

Si p es primo y $a \not\equiv 0 \pmod{p}$, entonces $a^{p-1} \equiv 1 \pmod{p}$

Pequeño teorema de Fermat

Teorema

Si p es primo y $a \not\equiv 0 \pmod{p}$, entonces $a^{p-1} \equiv 1 \pmod{p}$

- Para cada $a \not\equiv 0 \pmod{p}$ su inverso será a^{p-2}.
  Pues $a \cdot a^{p-2} \equiv a^{p-1} \equiv 1 \pmod{p}$

Pequeño teorema de Fermat

Teorema

Si p es primo y $a \not\equiv 0 \pmod{p}$, entonces $a^{p-1} \equiv 1 \pmod{p}$

- Para cada $a \not\equiv 0 \pmod{p}$ su inverso será a^{p-2}.
  Pues $a \cdot a^{p-2} \equiv a^{p-1} \equiv 1 \pmod{p}$
- Entonces para calcular un inverso (y poder dividir) necesitamos calcular a^{p-2}

Elevar modulo p

Queremos calcular a^n.
El algoritmo obvio es hacer a · a · a · . . . · a, n veces. El tiempo de ejecución es $O(n)$.

Queremos algo mejor. Notemos que si n es par, n = 2k tenemos:

$$a^n = a^{2k} = (a^k)^2 = a^k * a^k$$

Y si n es impar, con n = 2k + 1.

$$a^n = a^{2k+1} = a * (a^k)^2 = a * a^k * a^k$$

En ambos casos reducimos el problema a la mitad con O(1) multiplicaciones, por lo que podemos elevar en O(log(n)).

Expmod

Agregandole lo aprendido sobre modulo, nos queda:

```
typedef long long ll;
ll expmod (ll b, ll e, ll m){ //O(log e)
    if(!e) return 1;
    ll q= expmod(b,e/2,m); q=(q*q)%m;
    return e%2? (b * q)%m : q;
}
```

Expmod de Caloventor en Dos

Contenidos

GCD

El máximo común divisor de dos números a y b es el d más grande tal que $d|a$ y $d|b$. Observamos que

$$a = q.b + r \implies \gcd(a,b) = \gcd(b,r)$$

$$a = b \implies \gcd(a,b) = \gcd(a,0) = a$$

lo cual nos lleva al siguiente algoritmo

```
ll gcd(ll a, ll b){return b?gcd(b, a%b):a;}
```

GCD

El máximo común divisor de dos números a y b es el d más grande tal que $d|a$ y $d|b$. Observamos que

$$a = q.b + r \implies \gcd(a,b) = \gcd(b,r)$$

$$a = b \implies \gcd(a,b) = \gcd(a,0) = a$$

lo cual nos lleva al siguiente algoritmo

```
ll gcd(ll a, ll b){return b?gcd(b, a%b):a;}
```

Se puede mostrar que $\gcd(F_{n+1}, F_n)$ requiere exactamente n operaciones (donde F_n son los números de Fibonacci). Como F_n crece exponencialmente y es la peor entrada posible para este algoritmo, el tiempo de ejecución es $\mathcal{O}(\log n)$.

GCD extendido

Se puede probar que

$$\gcd(a, m) = 1 \iff 1 = a.x + m.y$$

GCD extendido

Se puede probar que

$$\gcd(a, m) = 1 \quad \iff \quad 1 = a.x + m.y$$

Luego $x \equiv_m a^{-1}$, de modo que a tiene inverso mod m si y sólo si $\gcd(a, m) = 1$.

GCD extendido

Se puede probar que

$$\gcd(a, m) = 1 \iff 1 = a.x + m.y$$

Luego $x \equiv_m a^{-1}$, de modo que a tiene inverso mod m si y sólo si $\gcd(a, m) = 1$. Para encontrar x e y, los rastreamos a través del algoritmo de Euclides:

```
ll x, y;
void extendedEuclid (ll a, ll b){ //a*x + b*y = d
  if (!b) { x = 1; y = 0; d = a; return;}
  extendedEuclid (b, a%b);
  ll x1 = y;
  ll y1 = x - (a/b) * y;
  x = x1; y = y1;
}
```

ExtendedEuclid de Caloventor en Dos

Contenidos

Números naturales

Recordamos que

$p \in \mathbb{N}$ es primo $\iff$ 1 y p son los únicos divisores de p en $\mathbb{N}$

Dado $n \in \mathbb{N}$, podemos factorizarlo en forma única como

$$n = p_1^{e_1} \ldots p_k^{e_k}$$

Encontrar números primos y/o la factorización de un dado número será útil para resolver problemas que involucran:

- funciones multiplicativas
- divisores de un número
- números primos y factorizaciones en general :-)

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor N (*e.g.* para factorizar m necesitamos todos los primos hasta $\sqrt{m}$).

Algoritmos para encontrar números primos

Queremos encontrar todos los números primos hasta un dado valor N (*e.g.* para factorizar m necesitamos todos los primos hasta $\sqrt{m}$).

- Un algoritmo ingenuo: para cada $n \in [2, N)$, controlamos si n es divisible por algún primo menor o igual que $\sqrt{n}$ (todos ellos ya han sido encontrados). Con algunas optimizaciones:

```
p[0] = 2; P = 1;
for (i=3; i<N; i+=2) {
  bool isp = true;
  for (j=1; isp && j<P && p[j]*p[j]<=i; j++)
    if (i%p[j] == 0) isp = false;
  if (isp) p[P++] = i;
}
```

Primer algoritmo para encontrar números primos

Cada número considerado puede requerir hasta $\pi(\sqrt{n}) = \mathcal{O}\left(\sqrt{n}/\ln n\right)$ operaciones $\implies$ este algoritmo es supra-lineal.

Contenidos

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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	51

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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~~	51

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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~~	~~51~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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~~	~~51~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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~~	~~51~~

Algoritmos para encontrar números primos (cont.)

- Un algoritmo muy antiguo: iteramos sobre una tabla con los números en el intervalo $[2, N)$: cada número sin tachar que encontramos es primo, de modo que podemos tachar todos sus múltiplos en la tabla

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~~	~~51~~

Algoritmos para encontrar números primos (cont.)

El código correspondiente es

```
memset(isp, true, sizeof(isp));
for (i=2; i<N; i++)
  if (isp[i]) for (j=2*i; j<N; j+=i) isp[j] = false;
```

Criba de Eratóstenes

Algoritmos para encontrar números primos (cont.)

El código correspondiente es

```
memset(isp, true, sizeof(isp));
for (i=2; i<N; i++)
  if (isp[i]) for (j=2*i; j<N; j+=i) isp[j] = false;
```

Criba de Eratóstenes

Podemos hacer que la criba tenga más información (0 si es primo, p el primo que lo divide en caso contrario)

```
memset(criba, 0, sizeof(criba));
for (int p=2; p<N; p++)
  if (!criba[p]) for (ll j=(ll)p*p; j<N; j+=i) criba[j] = p;
```

Criba de Eratóstenes con Información

Factorización usando la criba

El algoritmo anterior es $\mathcal{O}(N \log \log N)$, pero puede llevarse a $\mathcal{O}(N)$ con algunas optimizaciones.

```
memset(criba, 0, sizeof(criba));
for (i=4; i<N; i+=2) criba[i] = 2; criba[2] = 0;
for (int p=3; p<N; p+=2)
  if (!criba[p]) for (ll j=(ll)p*p; j<N; j+=i) criba[j] = p;
```

Criba de Eratóstenes, optimizada y extendida

Factorización usando la criba

El algoritmo anterior es $\mathcal{O}(N \log \log N)$, pero puede llevarse a $\mathcal{O}(N)$ con algunas optimizaciones.

```
memset(criba, 0, sizeof(criba));
for (i=4; i<N; i+=2) criba[i] = 2; criba[2] = 0;
for (int p=3; p<N; p+=2)
  if (!criba[p]) for (ll j=(ll)p*p; j<N; j+=i) criba[j] = p;
```

Criba de Eratóstenes, optimizada y extendida

```
#define MAXP 100000 //no necesariamente primo
int criba[MAXP+1];
void crearcriba(){
  int w[] = {4,2,4,2,4,6,2,6};
  for(int p=25;p<=MAXP;p+=10) criba[p]=5;
  for(int p=9;p<=MAXP;p+=6) criba[p]=3;
  for(int p=4;p<=MAXP;p+=2) criba[p]=2;
  for(int p=7,cur=0;p*p<=MAXP;p+=w[cur++&7]) if (!criba[p])
    for(int j=p*p;j<=MAXP;j+=(p<<1)) if(!criba[j]) criba[j]=p;
}
```

Criba usada por Caloventor en Dos

Contenidos

Test de Rabin - Miller (Introducción)

- El test de Rabin-Miller es un algoritmo **probabilístico**, muy eficiente para verificar si un número es primo.
- Se basa en su antecesor, el *test de Fermat*.
- Recordemos: $a \not\equiv 0 \pmod{p} \Rightarrow a^{p-1} \equiv 1 \pmod{p}$

Test de Fermat

- El test de Fermat es un test probabilístico para verificar si un número candidato N es primo.
- Se selecciona para ello un entero al azar $a \in [1, N)$.
- Si N es primo necesariamente será $a^{N-1} \equiv 1 \pmod{N}$, así que si esto no ocurre descartamos al número como primo.
- Si esto ocurre, el número pasó el test de Fermat con a como testigo. El test puede repetirse con varios valores de a para aumentar la confianza.

Test de Fermat: problema

- El test de Fermat es eficiente, pero tiene un problema: existen ejemplos de números que pasan el test de Fermat para todo valor de a coprimo con N, pero que son compuestos.
- Estos números extremos son raros y se denominan de *Carmichael*. Los primeros son 561, 1105, 1729, 2465, 2821, 6601, 8911.
- Con estos números, el test solamente los detecta como compuestos si a es múltiplo de uno de los primos que dividen a N, y por lo tanto el test es prácticamente una búsqueda de divisores aleatoria.

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

- Si $p > 2$ es primo y $x^2 = 1 \pmod{p}$, x solo puede ser 1 o -1 módulo p (porque Z_p es un cuerpo...)

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

- Si $p > 2$ es primo y $x^2 = 1 \pmod p$, x solo puede ser 1 o -1 módulo p (porque Z_p es un cuerpo...)
- Por otro lado si $p - 1 = 2^{\alpha}k$, con k impar y $\alpha \geq 1$, tenemos que para cualquier $a \not\equiv 0 \pmod p$ debe ser $a^{2^{\alpha}k} \equiv 1 \pmod p$.

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

- Si $p > 2$ es primo y $x^2 = 1 \pmod{p}$, x solo puede ser 1 o -1 módulo p (porque Z_p es un cuerpo...)
- Por otro lado si $p - 1 = 2^\alpha k$, con k impar y $\alpha \geq 1$, tenemos que para cualquier $a \not\equiv 0 \pmod{p}$ debe ser $a^{2^\alpha k} \equiv 1 \pmod{p}$.
- Pero entonces $a^{2^{\alpha-1} k} \equiv 1$ o $-1 \pmod{p}$

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

- Si $p > 2$ es primo y $x^2 = 1 \pmod p$, x solo puede ser 1 o -1 módulo p (porque Z_p es un cuerpo...)
- Por otro lado si $p - 1 = 2^\alpha k$, con k impar y $\alpha \geq 1$, tenemos que para cualquier $a \not\equiv 0 \pmod p$ debe ser $a^{2^\alpha k} \equiv 1 \pmod p$.
- Pero entonces $a^{2^{\alpha-1}k} \equiv 1 \text{ o } -1 \pmod p$
- Y si fuera 1, entonces nuevamente $a^{2^{\alpha-2}k} \equiv 1 \text{ o } -1 \pmod p$

Test de Rabin - Miller (idea)

El test de Rabin-Miller elimina este problema verificando una condición más fuerte.

- Si $p > 2$ es primo y $x^2 = 1 \pmod p$, x solo puede ser 1 o -1 módulo p (porque Z_p es un cuerpo...)
- Por otro lado si $p - 1 = 2^{\alpha}k$, con k impar y $\alpha \geq 1$, tenemos que para cualquier $a \not\equiv 0 \pmod p$ debe ser $a^{2^{\alpha}k} \equiv 1 \pmod p$.
- Pero entonces $a^{2^{\alpha-1}k} \equiv 1$ o $-1 \pmod p$
- Y si fuera 1, entonces nuevamente $a^{2^{\alpha-2}k} \equiv 1$ o $-1 \pmod p$
- Y así podemos repetir el razonamiento hasta que $a^k \equiv 1$ o bien $a^{2^j k} \equiv -1$ para algún $0 \leq j < \alpha$

Test de Rabin - Miller (idea cont.)

Tenemos entonces las siguientes posibilidades para el valor de $a^{2^j k}$ (una por columna):

$$
\begin{array}{cccccccc}
\mathbf{j} & & & & & & & \\
\alpha & 1 & 1 & \cdots & 1 & 1 & 1 & 1 \\
\alpha-1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 \\
\alpha-2 & ? & -1 & \cdots & 1 & 1 & 1 & 1 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
2 & ? & ? & \cdots & -1 & 1 & 1 & 1 \\
1 & ? & ? & \cdots & ? & -1 & 1 & 1 \\
0 & ? & ? & \cdots & ? & ? & -1 & 1
\end{array}
$$

Test de Rabin - Miller (implementación)

En general estas son chequeadas desde abajo hacia arriba, de forma que cada valor necesario es el cuadrado del que se necesitó en el paso anterior:

$$
\begin{array}{rcccccccl}
\mathbf{j} & & & & & & & & \\
\alpha & 1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-2 & ? & -1 & \cdots & 1 & 1 & 1 & 1 & \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\
2 & ? & ? & \cdots & -1 & 1 & 1 & 1 & \\
1 & ? & ? & \cdots & ? & -1 & 1 & 1 & \\
0 & ? & ? & \cdots & ? & ? & -1 & 1 & a^k \equiv 1 \text{ o } -1
\end{array}
$$

Test de Rabin - Miller (implementación)

En general estas son chequeadas desde abajo hacia arriba, de forma que cada valor necesario es el cuadrado del que se necesitó en el paso anterior:

$$
\begin{array}{rccccccc}
\mathbf{j} & & & & & & & \\
\alpha & 1 & 1 & \cdots & 1 & 1 & 1 & 1 \\
\alpha-1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 \\
\alpha-2 & ? & -1 & \cdots & 1 & 1 & 1 & 1 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
2 & ? & ? & \cdots & -1 & 1 & 1 & 1 \\
1 & ? & ? & \cdots & ? & -1 & 1 & 1 \\
0 & ? & ? & \cdots & ? & ? & -1 & 1
\end{array}
$$

$a^{2k} = (a^k)^2 \equiv -1$

Test de Rabin - Miller (implementación)

En general estas son chequeadas desde abajo hacia arriba, de forma que cada valor necesario es el cuadrado del que se necesitó en el paso anterior:

j							
α	1	1	$\cdots$	1	1	1	1
$\alpha-1$	-1	1	$\cdots$	1	1	1	1
$\alpha-2$	?	-1	$\cdots$	1	1	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
2	?	?	$\cdots$	-1	1	1	1
1	?	?	$\cdots$	?	-1	1	1
0	?	?	$\cdots$	?	?	-1	1

$$a^{2^2 k} = (a^{2k})^2 \equiv -1$$

Test de Rabin - Miller (implementación)

En general estas son chequeadas desde abajo hacia arriba, de forma que cada valor necesario es el cuadrado del que se necesitó en el paso anterior:

j								
α	1	1	$\cdots$	1	1	1	1	
$\alpha - 1$	-1	1	$\cdots$	1	1	1	1	
$\alpha - 2$	?	-1	$\cdots$	1	1	1	1	$a^{2^{\alpha-2}k} \equiv -1$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	
2	?	?	$\cdots$	-1	1	1	1	
1	?	?	$\cdots$	?	-1	1	1	
0	?	?	$\cdots$	?	?	-1	1	

Test de Rabin - Miller (implementación)

En general estas son chequeadas desde abajo hacia arriba, de forma que cada valor necesario es el cuadrado del que se necesitó en el paso anterior:

j								
α	1	1	$\cdots$	1	1	1	1	
$\alpha-1$	-1	1	$\cdots$	1	1	1	1	$a^{2^{\alpha-1}k} \equiv -1$
$\alpha-2$	?	-1	$\cdots$	1	1	1	1	
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	
2	?	?	$\cdots$	-1	1	1	1	
1	?	?	$\cdots$	?	-1	1	1	
0	?	?	$\cdots$	?	?	-1	1	

Test de Rabin - Miller (conclusión)

- Si ninguno de los casos anteriores se da, concluímos que definitivamente el número no es primo.
- Si alguno funciona, ese valor de *a* funciona y el número parece ser primo.
- Al igual que en el test de Fermat, conviene utilizar varios valores de *a* para aumentar la confianza.
- En el caso del test de Rabin-Miller, tenemos la garantía de que si $N > 2$ es compuesto impar, al menos el 75 % de los posibles restos *a* no nulos módulo *N* lo demostrarán usando el test.
- Por lo tanto si repetimos el test *k* veces sobre un número compuesto, eligiendo números de manera aleatoria, uniforme e independiente, la probabilidad de error es como máximo $\frac{1}{4^k}$.
- Los números primos siempre pasan el test, y son reportados como tales.

Test de Rabin - Miller (bonus)

- Si los números a verificar no son demasiado grandes, se conocen versiones deterministas del test probando con un conjunto específico de valores de *a*.
- Por ejemplo wikipedia menciona:
  - if n < 4,759,123,141 $> 2^{32}$, it is enough to test:
    a = 2, 7, and 61;
  - if n < 3,825,123,056,546,413,051, it is enough to test
    a = 2, 3, 5, 7, 11, 13, 17, 19, and 23.
  - if n < 18,446,744,073,709,551,616 = 2^{64}, it is enough to test:
    a = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37.
- Los artículos citados son:
  - Jaeschke, Gerhard (1993), “On strong pseudoprimes to several bases”, Mathematics of Computation 61 (204): 915-926
  - Jiang, Yupeng; Deng, Yingpu (2014). “Strong pseudoprimes to the first eight prime bases”. Mathematics of Computation 83 (290): 2915-2924. doi:10.1090/S0025-5718-2014-02830-5

Test de Rabin - Miller (código)

```
bool es_primo_prob (ll n, int a)
{
  if (n == a) return true;
  ll s = 0,d = n-1;
  while (d %2 == 0) s++,d/=2;

  ll x = expmod(a,d,n);
  if ((x == 1) || (x+1 == n)) return true;

  forn (i, s-1){
    x = mulmod(x, x, n);
    if (x == 1) return false;
    if (x+1 == n) return true;
  }
  return false;
}

bool rabin (ll n){ //devuelve true si n es primo
  if (n == 1) return false;
  const int ar[] = {2,3,5,7,11,13,17,19,23};
  forn (j,9)
    if (!es_primo_prob(n,ar[j]))
      return false;
  return true;
}
```

Rabin-Miller de Caloventor en Dos

Contenidos

Algoritmo ingenuo

- Sabemos que si no hay ningún factor primo hasta $\sqrt{N}$, N debe ser primo.
- En virtud de esto, es natural dar un algoritmo de factorización que pruebe todos los posibles factores hasta ese valor.
- Notar que podemos cortar en la raíz de la parte de N que falta factorizar, acelerando el proceso cuando hay bastantes factores chicos y uno grande.
- El peor caso sigue siendo $\Theta(\sqrt{N})$

```
for(int i = 2; i*i <= N; i++)
while (N % i == 0)
{
    N /= i;
    reportarFactorPrimo(i);
}
if (N > 1)
    reportarFactorPrimo(N);
```

Contenidos

Usando la Criba (1)

- El algoritmo ingenuo pierde la batalla cuando queremos factorizar un conjunto de números porque no guarda ninguna información.
- Una solución para esto es utilizar la criba
- La criba la corremos una vez $O(MAXP)$ (lo cuál podemos considerar para ciertos problemas una cantidad ínfima de tiempo).
- Y luego factorizamos utilizando la criba con un tiempo máximo de $O(lgn)$ por número.

Usando la Criba (1)

```
//factoriza bien numeros hasta MAXP
map<ll,ll> fact2(ll n){ //O (lg n)
  map<ll,ll> ret;
  while (criba[n]){
    ret[criba[n]]++;
    n/=criba[n];
  }
  if(n>1) ret[n]++;
  return ret;
}
```

Fact2 - Factoriza en lg(n)

Usando la Criba (2)

Fact2 tiene un problema, que la iniciación es muy cara para ciertos primos $\geq 10^8$. Pero podemos hacer un punto medio.

Usando la Criba (2)

Fact2 tiene un problema, que la iniciación es muy cara para ciertos primos $\geq 10^8$. Pero podemos hacer un punto medio.

- Usamos la criba para calcular solo el conjunto de primos.
- Usamos el algoritmo ingenuo pero solo aplicando al conjunto de primos
- Entonces solo vamos a necesitar calcular la criba hasta la raiz cuadrada de nuestro máximo número a buscar
- Como se puede aproximar la cantidad de primos a $O(N/lg(N))$, la complejidad nos quedará:
  - $O(\sqrt{MAXN})$ para incializar
  - $O(\sqrt{MAXN/lg(MAXN)})$ por cálculo a realizar.

Usando la Criba (2)

```
vector<int> primos;
void buscarprimos(){
  crearcriba();
  forr (i,2,MAXP+1) if (!criba[i]) primos.push_back(i);
}

//factoriza bien numeros hasta MAXP^2
map<ll,ll> fact(ll n){ //O (cant primos)
  map<ll,ll> ret;
  forall(p, primos){
    while(!(n%*p)){
      ret[*p]++;//divisor found
      n/=*p;
    }
  }
  if(n>1) ret[n]++;
  return ret;
}
```

Fact - Factoriza en O(cant-primos)

Ciertos Números

- Imaginemos que tenemos $N \leq 10^{14}$
- El algoritmo ingenuo hace 10^7 comparaciones máximo
- Hay 664580 primos para comparar con la optimización de Fact (15 veces menos)
- $lg(10^{14}) = 46{,}5$ (Complejidad de Fact2 si pudiésemos hacer la criba completa)

Contenidos

Paradoja de los cumpleaños

- ¿Cuál es la probabilidad de que dos personas cumplan años el mismo día, en una sala con 23 personas?

Paradoja de los cumpleaños

- ¿Cuál es la probabilidad de que dos personas cumplan años el mismo día, en una sala con 23 personas?
- 50.7 %

Paradoja de los cumpleaños

- ¿Cuál es la probabilidad de que dos personas cumplan años el mismo día, en una sala con 23 personas?
- 50.7 %
- En general, dado un universo de n objetos, la cantidad de elementos que hay que sacar al azar hasta que la probabilidad de que dos sean iguales sea al menos 50 % es $\left\lceil \sqrt{2n \ln 2} \right\rceil + \epsilon$, donde $\epsilon \in \{0, 1\}$
- Similarmente, la cantidad esperada de elementos que hay que sacar al azar hasta que aparezca una primera repetición es $\sqrt{\frac{\pi n}{2}} + \frac{2}{3} + \epsilon$, donde $|\epsilon| \leq 1$ (Ramanujan, Watson y Knuth).

Paradoja de los cumpleaños

- ¿Cuál es la probabilidad de que dos personas cumplan años el mismo día, en una sala con 23 personas?
- 50.7 %
- En general, dado un universo de n objetos, la cantidad de elementos que hay que sacar al azar hasta que la probabilidad de que dos sean iguales sea al menos 50 % es $\left\lceil \sqrt{2n \ln 2} \right\rceil + \epsilon$, donde $\epsilon \in \{0, 1\}$
- Similarmente, la cantidad esperada de elementos que hay que sacar al azar hasta que aparezca una primera repetición es $\sqrt{\frac{\pi n}{2}} + \frac{2}{3} + \epsilon$, donde $|\epsilon| \leq 1$ (Ramanujan, Watson y Knuth).
- **En resumen**, son $O(\sqrt{n})$ pasos hasta la primera repetición.

Algoritmo de la ρ de Pollard

- Asumimos que N es compuesto (podemos comenzar verificando su primalidad con algún test rápido como Rabin-Miller).
- La idea es aprovechar la paradoja de los cumpleaños para encontrar un factor propio de N rápidamente.
- Una vez que encontramos un factor de N, basta repetir el procedimiento recursivamente hasta descomponer a N en primos.

Algoritmo de la ρ de Pollard (cont.)

- Supongamos que $p \leq \sqrt{N}$ es un primo que divide a N.
- Si vamos generando números entre 0 y $N - 1$ al azar, sus restos módulo p también serán aleatorios.
- La cantidad de pasos esperados hasta que se repita un valor módulo N es $\Theta(\sqrt{N})$.
- Pero la cantidad de pasos esperados hasta que se repita un valor módulo p es $\Theta(\sqrt{p}) = O(\sqrt[4]{N})$
- Luego esperamos que exista una repetición módulo p rápidamente, mucho antes de que haya una repetición módulo N.

Algoritmo de la ρ de Pollard (cont.)

- Supongamos que $p \leq \sqrt{N}$ es un primo que divide a N.
- Si vamos generando números entre 0 y $N-1$ al azar, sus restos módulo p también serán aleatorios.
- La cantidad de pasos esperados hasta que se repita un valor módulo N es $\Theta(\sqrt{N})$.
- Pero la cantidad de pasos esperados hasta que se repita un valor módulo p es $\Theta(\sqrt{p}) = O(\sqrt[4]{N})$
- Luego esperamos que exista una repetición módulo p rápidamente, mucho antes de que haya una repetición módulo N.
- ¿Pero cómo detectamos esta repetición, **si no conocemos** p **a priori**?

Algoritmo de la ρ de Pollard (cont.)

- Si x e y son dos valores de nuestra secuencia que coinciden módulo p, $x - y \equiv 0 \pmod{p}$
- Entonces $p | GCD(|x - y|, N)$
- Este GCD puede calcularse con el algoritmo de Euclides sin conocer p.
  - Si da $1 < GCD < N$, hemos encontrado un factor de N.
  - Si da $GCD = N$, hemos tenido una repetición en la secuencia módulo N.
  - Si da $GCD = 1$, no hemos detectado ninguna repetición módulo p.

Algoritmo de la ρ de Pollard (cont.)

- Notar que con este truco podemos verificar si x e y dados son coincidentes módulo algún p.
- Es decir, a la hora de buscar repeticiones en nuestra secuencia, **solamente tenemos un operador de igualdad**.
- La mejor estructura para buscar repeticiones en general con solamente ese operador tomaba $O(j^2)$, lo cual nos devolvería a la complejidad $O(\sqrt{N})$

Algoritmo de la ρ de Pollard (cont.)

- Notar que con este truco podemos verificar si x e y dados son coincidentes módulo algún p.
- Es decir, a la hora de buscar repeticiones en nuestra secuencia, **solamente tenemos un operador de igualdad**.
- La mejor estructura para buscar repeticiones en general con solamente ese operador tomaba $O(j^2)$, lo cual nos devolvería a la complejidad $O(\sqrt{N})$
- Solución: Utilizar una secuencia **pseudoaleatoria**, "en lugar de" generar números verdaderamente al azar.
- Una función pseudoaleatoria módulo N que funciona bien es $f(X) = X^2 + AX + B$, con $1 \leq A, B < N$ elegidos al azar.
- Con esto la secuencia será x_1, $f(x_1)$, $f(f(x_1))$

Algoritmo de la ρ de Pollard (implementación)

- Solo falta algo que encuentre repetición modulo N.
- Podemos utilizar el algoritmo de la liebre y la tortuga (que encuentra la primera repetición en una sucesión).
- O el algoritmo de Brent (una mejora)

```
int factor(int N) {
    A = elegir al azar;
    B = elegir al azar;
    // f es X*(X+A) + B modulo N
    int x = 2, y = 2, d;
    do {
        x = f(x);
        y = f(f(y));
        d = gcd(abs(x-y), N);
    } while (d == 1);
    return d;
}
```

Algoritmo de la ρ de Pollard (conclusiones)

- Si tenemos mala suerte y factor retorna N, repetimos la llamada hasta que los valores de A y B funcionen.
- El evento anterior normalmente no ocurre, ya que la secuencia se repite módulo p antes que módulo N.
- Como dijimos, la complejidad esperada es $O(\sqrt{p})$ hasta extraer un factor, siendo p un primo que divida a N.
- La complejidad total esperada del algoritmo resulta ser entonces $O\left(\sqrt[4]{N}\right)$ operaciones aritméticas y cálculos de GCD.
- Más precisamente, $O\left(\sum_{p \mid \frac{N}{p_{max}}} \sqrt{p}\right)$, considerados con multiplicidad según el exponente en la factorización de $\frac{N}{p_{max}}$.
- Notar que aunque N sea muy grande, si los primos que dividen a N son pequeños, salvo a lo sumo un único primo grande con exponente 1, el algoritmo es extremadamente rápido.

Pollard-rho (Código)

```
ll rho(ll n){
    if( (n & 1) == 0 ) return 2;
    ll x = 2 , y = 2 , d = 1;
    ll c = rand() %n + 1;
    while( d == 1 ){
        x = (mulmod( x , x , n ) + c)%n;
        y = (mulmod( y , y , n ) + c)%n;
        y = (mulmod( y , y , n ) + c)%n;
        if( x - y >= 0 ) d = gcd( x - y , n );
        else d = gcd( y - x , n );
    }
    return d==n? rho(n):d;
}

map<ll,ll> prim;
void factRho (ll n){ //O (lg n)^3. un solo numero
  if (n == 1) return;
  if (rabin(n)){
    prim[n]++;
    return;
  }
  ll factor = rho(n);
  factRho(factor);
  factRho(n/factor);
}
```

Rolando de Caloventor en Dos

Funciones Multiplicativas

Las funciones multiplicativas son aquellas funciones de naturales en naturales tal que si m y n son coprimos $f(n * m) = f(n) * f(m)$.

Dado $n = p_1^{e_1} \dots p_k^{e_k}$, algunas de estas funciones son:

- $d(n) = \prod_{i=1}^{k}(e_i + 1)$, la cantidad de divisores de un número.
- $\sigma(n) = \prod_{i=1}^{k}(\frac{p^{e_i+1}}{p-1})$, la suma de los divisores de un número.
- $\phi(n) = \prod_{i=1}^{k}(p^e - p^{e-1})$, la cantidad de números menores que n coprimos con n.

Todas estas funciones se pueden calcular en $O(log(n))$ luego de realizar la criba.

Códigos

```
ll sumDiv (ll n){
  ll rta = 1;
  map<ll,ll> f=fact(n);
  forall(it, f) {
  ll pot = 1, aux = 0;
  forn(i, it->snd+1) aux += pot, pot *= it->fst;
  rta*=aux;
  }
  return rta;
}
```

Ejemplo de una Función que calcula la suma de los divisores

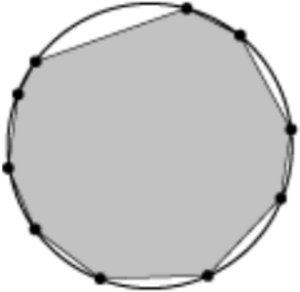
```
//Usar asi:  divisores(fac, divs, fac.begin()); NO ESTA ORDENADO
void divisores(const map<ll,ll> &f, vector<ll> &divs, map<ll,ll>::
    iterator it, ll n=1){
    if(it==f.begin()) divs.clear();
    if(it==f.end()) { divs.pb(n);  return; }
    ll p=it->fst, k=it->snd; ++it;
    forn(_, k+1) divisores(f, divs, it, n), n*=p;
}
```

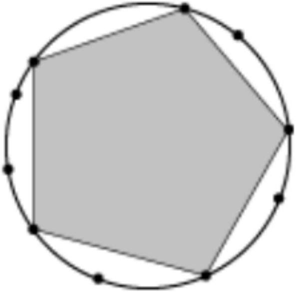
Ejemplo de una Función que calcula todos los divisores

Problema Ejemplo

Shrinking Polygons

Dado $N \leq 10^4$ puntos en una circurferencia y la distancia entre cada par consecutivo de ellos $d_i \leq 10^3$, encontrar el polígono regular más grande que esté formado solo por los puntos.





Cantidad de Movimientos de Torre

Problema

Dado un tablero de N x N ($N = 8$), tenemos una torre de ajedrez en una casilla inicial $H2$. Contar de cuántas maneras distintas puede ir a una casilla final $B4$ en $T \leq 100$ movimientos.

Cantidad de Movimientos de Torre

Problema

Dado un tablero de N x N ($N = 8$), tenemos una torre de ajedrez en una casilla inicial $H2$. Contar de cuántas maneras distintas puede ir a una casilla final $B4$ en $T \leq 100$ movimientos.

- Y si fuera $N \leq 10^6$

Una jugada

- Volvamos al tablero de $8x8$
- Tenemos que llegar a b4.
- Pensemos, en cada casilla, de cuántas formas podemos realizarlo si tuviéramos una sola jugada

0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0

Dos jugadas

- Si seguimos la DP para 2 movimientos:

2	6	2	2	2	2	2	2
2	6	2	2	2	2	2	2
2	6	2	2	2	2	2	2
2	6	2	2	2	2	2	2
6	14	6	6	6	6	6	6
2	6	2	2	2	2	2	2
2	6	2	2	2	2	2	2
2	6	2	2	2	2	2	2

Notación

- Llamemos a b4 como casilla tipo 1.
- Llamemos a las casillas de la misma fila o misma columna, tipo 2.
- Llamemos a las otras, casillas tipo 3. Por ejemplo, h2 es de tipo 3. Aquí tenemos cada casilla con su tipo:

3	2	3	3	3	3	3	3
3	2	3	3	3	3	3	3
3	2	3	3	3	3	3	3
3	2	3	3	3	3	3	3
2	1	2	2	2	2	2	2
3	2	3	3	3	3	3	3
3	2	3	3	3	3	3	3
3	2	3	3	3	3	3	3

Observaciones

- ¿Y Cómo nos va a servir esto para un tablero más grande $N \leq 10^6$?
- Se puede ver que desde cada tipo vamos a tener las mismas posibilidades, por lo tanto podemos analizar cada tipo por separado. En particular, veamos como funciona el tablero de $N = 8$
- Desde la casilla tipo 1, podemos ir solo a casillas tipo 2.
- Desde una casilla tipo 2, podemos ir a la casilla tipo 1, a 6 casillas tipo 2 manteniendo fila o columna y a 7 del tipo 3.
- Desde una casilla tipo 3, podemos ir 2 casillas del tipo 2 y el resto (12) son del tipo 3.

En resumen:

Tipo	$Ct1$	$Ct2$	$Ct3$
1	0	14	0
2	1	6	7
3	0	2	12

Tablita para muchas jugadas

Y dado el tablero de 1 jugada, ya resolvimos el problema en 2 jugadas:

Tipo	1*jug*	2*jug*
1	0	$14 = 14 * 1$
2	1	$6 = 1 * 0 + 6 * 1 + 7 * 0$
3	0	$2 = 2 * 1 + 12 * 0$

Tablita para muchas jugadas

Y dado el tablero de 2 jugada, podemos resolver el problema en 3 jugadas:

Tipo	$1jug$	$2jug$	$3jug$
1	0	14	$84 = 14 * 6$
2	1	6	$64 = 1 * 14 + 6 * 6 + 7 * 2$
3	0	2	$36 = 2 * 6 + 12 * 2$

Tablita para muchas jugadas

Y podemos seguir hasta el número de jugadas que necesitemos y el tipo de casilla inicial que tengamos... :

Tipo	1*jug*	2*jug*	3*jug*	4*jug*	5*jug*	...
1	0	14	84	896	10080	...
2	1	6	64	720	9136	...
3	0	2	36	560	8160	...

- Se puede hacer algo similar para cualquier $N \leq 10^6$

Tipo	$Ct1$	$Ct2$	$Ct3$
1	0	$2 * N - 2$	0
2	1	$N - 2$	$N - 1$
3	0	2	$2 * N - 4$

Cantidad de Movimientos de Torre

Problema

Dado un tablero de N x N ($N = 8$), tenemos una torre de ajedrez en una casilla inicial $H2$. Contar de cuántas maneras distintas puede ir a una casilla final $B4$ en $T \leq 100$ movimientos.

- Y si $N \leq 10^6$
- Y si $T \leq 10^9$

Exponenciación de Matrices

```
struct mnum{
  static const ll mod=1000000009;
  ll v;
  mnum(ll v=0): v(v%mod) {}
  mnum operator+(mnum b) const {return v+b.v;}
  mnum operator*(mnum b) const {return v*b.v;}
};

struct Matrix{
  mnum m[4][4];
  Matrix operator*(const Matrix &p) const {
    Matrix r;
    forn(i,4) forn(j,4) { r.m[i][j]=0;
      forn(k,4) r.m[i][j] = r.m[i][j] + m[i][k]*p.m[k][j];
    }
    return r;
  }
};
```

Códigos de mnum y Matrix

Exponenciación de Matrices

```
Matrix eye() {
  Matrix r;
  forn(i,4) forn(j,4) r.m[i][j] = (i==j);
  return r;
}

Matrix operator^(const Matrix &p, int n){
  if(!n) return eye();//identidad
  Matrix q=p^(n/2); q=q*q;
  return n%2? p * q : q;}
```

Exponenciación de Matrix

Exponenciación de Matrices

```
ll M,N,A,B,C,D;
int T;

int main() {
    cin >> M >> N >> T >> A >> B >> C >> D;
    Matrix X,Y,R;
    Y.m[0][0]= A==C && B==D; // Misma Casilla
    Y.m[1][0]= A!=C && B==D; // Misma Columna
    Y.m[2][0]= A==C && B!=D; // Misma Fila
    Y.m[3][0]= A!=C && B!=D; // No coincide ninguna
    X.m[0][0]=0; X.m[0][1]=1; X.m[0][2]=1; X.m[0][3]=0;
    X.m[1][0]=M-1; X.m[1][1]=M-2; X.m[1][2]=0; X.m[1][3]=1;
    X.m[2][0]=N-1; X.m[2][1]=0; X.m[2][2]=N-2; X.m[2][3]=1;
    X.m[3][0]=0; X.m[3][1]=N-1; X.m[3][2]=M-1; X.m[3][3]=M+N-4;

    cout << ((X^T)*Y).m[0][0].v << endl;
  }
    return 0;
}
```

Solución Final

Cosas que faltarían y están en eldiego

- Polinomios (evaluación, operaciones, propiedades, etc.).
- Integración Numérica (Simpson)
- Karatsuba o FFT (Multiplicación rápida de Vectores)
- Gauss-Jordan para resolver matrices

Referencias

- Clase de Fidel Schaposnik: “Schaposnik-2015”
- Clase de Pablo Blanc: “Blanc-2017”
- Clase de Ariel Zylber: “Zylber-2014”
- El Diego: https://github.com/mvpossum/eldiego
- *Introduction to Algorithms, 2nd Edition*. MIT Press.
  31 Number-Theoretic Algorithms
  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein