Primalidad y factorización

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Campamento Caribeño ACM-ICPC 2016

Contenidos

Contenidos

Definición

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.

Definición

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 $11 \equiv 18 \pmod 7$

Definición

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 $11 \equiv 18 \pmod 7$
- 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.

Propiedades

A los efectos de realizar sumas, restas y productos, la aritmética módular es análoga a la aritmética usual, manteniendo sus propiedades importantes.

- $a + b \equiv b + a \pmod{M}$
- $(a + b) + c \equiv a + (b + c) \pmod{M}$
- 0 es el neutro de la suma.
- Para todo a existe un único inverso aditivo modular $-a$, $a + (-a) \equiv 0 \pmod{M}$. $a - b \equiv a + (-b) \pmod{M}$
- $a \cdot b \equiv b \cdot a \pmod{M}$
- $(a \cdot b) \cdot c \equiv a \cdot (b \cdot c) \pmod{M}$
- 1 es el neutro del producto.
- $(a + b) \cdot c \equiv a \cdot c + b \cdot c \pmod{M}$

Forma operacional en el código

Supongamos que se tiene que computar una suma de los números enteros `a[0]` hasta `a[N-1]`, pero solamente nos importan los últimos 4 dígitos (equivale a trabajar módulo 10000).

```
int result = 0;
for (int i = 0; i < N; i++)
    result = (result + a[i]) %10000;
```

Como decíamos antes, a nivel de operaciones trabajar con aritmética modular equivale a simplemente tomar módulo luego de cada operación aritmética básica.

Problema ante números negativos

Sin embargo, la implementación anterior puede resultar problemática al trabajar con números **negativos**.

- Si por ejemplo fuera $N = 2$, `a[0]=123` y `a[1]=-200`, el código anterior produce -77, que puede no ser lo deseado.
- Incluso si no hay números negativos en el problema, es muy común que **restemos** números en nuestra solución.
- Estos resultados con valores negativos ocurren porque el resultado de la división entera se redondea hacia cero en los lenguajes y plataformas más populares.

Problema ante números negativos

Sin embargo, la implementación anterior puede resultar problemática al trabajar con números **negativos**.

- Si por ejemplo fuera $N = 2$, `a[0]=123` y `a[1]=-200`, el código anterior produce -77, que puede no ser lo deseado.
- Incluso si no hay números negativos en el problema, es muy común que **restemos** números en nuestra solución.
- Estos resultados con valores negativos ocurren porque el resultado de la división entera se redondea hacia cero en los lenguajes y plataformas más populares.
- Solución:

```
int MOD(int x, int M){return ((x%M)+M)%M;}
```

Cuidado con el overflow

- Otro problema al que es especialmente común enfrentarse al trabajar con aritmética modular es el peligro de tener overflow en las operaciones.
- Por esto es que tomamos módulo luego de cada operación, y no solamente al final de todo el programa.
- Truquito en C++: Tener en cuenta el tipo `__int128`, entero de 128 bits. No está presente en todo judge, pero puede ser muy útil cuando está disponible.

¿Qué pasa con la división?

- ¿Podemos operar modularmente con la división tal cual lo hacemos con sumas, restas y productos?

¿Qué pasa con la división?

- ¿Podemos operar modularmente con la división tal cual lo hacemos con sumas, restas y productos?
- **NO**. Por ejemplo:
  $\frac{10}{2} \equiv 5 \pmod 8$, pero $10 \equiv 2 \pmod 8$ y $\frac{2}{2} \equiv 1 \not\equiv 5 \pmod 8$
- Podemos garantizar que este "truco" funciona cuando el módulo es un número **primo.**
  $\frac{27}{3} \equiv 2 \pmod 7$, $27 \equiv 6 \pmod 7$ y $\frac{6}{3} \equiv 2 \pmod 7$
  Pero solo si el divisor **no es cero** (módulo p)
  $\frac{140}{14} \equiv 3 \pmod 7$, pero $140 \equiv 14 \equiv 0 \pmod 7$ y $\frac{0}{0} \equiv ? \pmod 7$

¿Qué pasa con la división?

- ¿Podemos operar modularmente con la división tal cual lo hacemos con sumas, restas y productos?
- **NO**. Por ejemplo:
  $\frac{10}{2} \equiv 5 \pmod 8$, pero $10 \equiv 2 \pmod 8$ y $\frac{2}{2} \equiv 1 \not\equiv 5 \pmod 8$
- Podemos garantizar que este "truco" funciona cuando el módulo es un número **primo.**
  $\frac{27}{3} \equiv 2 \pmod 7$, $27 \equiv 6 \pmod 7$ y $\frac{6}{3} \equiv 2 \pmod 7$
  Pero solo si el divisor **no es cero** (módulo p)
  $\frac{140}{14} \equiv 3 \pmod 7$, pero $140 \equiv 14 \equiv 0 \pmod 7$ y $\frac{0}{0} \equiv ? \pmod 7$
- ¿Pero y si aún con un módulo primo, la división modular no resulta una división entera?
  $\frac{12}{3} \equiv 4 \pmod 7$, pero $12 \equiv 5 \pmod 7$ y $\frac{5}{3} \equiv ? \pmod 7$

Inversos modulares

Definición

Decimos que b es inverso de a módulo M si $a \cdot b \equiv 1 \pmod{M}$.

Notar que solo un $a \not\equiv 0 \pmod{M}$ podría tener un inverso, y de existir el inverso es único, y a su vez a resulta ser el inverso de b.

Inversos modulares

Definición

Decimos que b es inverso de a módulo M si $a \cdot b \equiv 1 \pmod{M}$.

Notar que solo un $a \not\equiv 0 \pmod{M}$ podría tener un inverso, y de existir el inverso es único, y a su vez a resulta ser el inverso de b.

Teorema

Si p es un número primo, entonces todo número $a \not\equiv 0 \pmod{p}$ tiene un inverso módulo p.

Inversos modulares: utilidad

- Recordemos que para realizar $3/2 = 1{,}5$, en realidad podríamos multiplicar directamente por el inverso de 2, es decir $3 \cdot 0{,}5 = 1{,}5$
- Lo mismo podemos hacer modularmente. Por ejemplo, $5 \cdot 3 \equiv 1 \pmod 7$, así que $inv(3) = 5$. Y entonces recordando el ejemplo anterior:
  $\frac{12}{3} \equiv 4 \pmod 7$,
  $12 \equiv 5 \pmod 7$ y
  "$\frac{5}{3}$" $\equiv 5 \cdot inv(3) \equiv 5 \cdot 5 \equiv 4 \pmod 7$
- De esta forma, ya podemos dividir modularmente al trabajar con un número primo (excepto cuando el divisor se hace 0 módulo *p*).
- ¿Pero cómo calculamos los inversos?

Contenidos

Pequeño teorema de Fermat

Teorema

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

- Por ejemplo $6^{30} = 7131416765184947029025 \cdot 31 + 1$
- ¿Para qué puede servir este teorema?

Aplicación 1: Cálculo de inversos

- Recordemos que dado $a \not\equiv 0$, si encontramos algún número x tal que $a \cdot x \equiv 1 \pmod{p}$, x será automáticamente el inverso de a.
- Si tomamos $x = a^{p-2}$, ¿Cuánto vale $a \cdot x$?

Aplicación 1: Cálculo de inversos

- Recordemos que dado $a \not\equiv 0$, si encontramos algún número x tal que $a \cdot x \equiv 1 \pmod{p}$, x será automáticamente el inverso de a.
- Si tomamos $x = a^{p-2}$, ¿Cuánto vale $a \cdot x$?
- Tenemos $a \cdot x = a^{p-1} \equiv 1 \pmod{p}$ por el Pequeño Teorema de Fermat.
- Luego para cada a su inverso será simplemente a^{p-2}.

Aplicación 2: Testeo de residuo cuadrático

Definición

Un resto r se dice un *residuo cuadrático* módulo p si existe x tal que $x^2 \equiv r \pmod{p}$

Por ejemplo los residuos cuadráticos módulo 5 son $0, 1, 4$. Notar que 0 siempre es residuo cuadrático módulo p.

- Si $r \not\equiv 0$ es residuo cuadrático, ¿Cuánto vale $r^{\frac{p-1}{2}}$?

Aplicación 2: Testeo de residuo cuadrático

Definición

Un resto r se dice un *residuo cuadrático* módulo p si existe x tal que $x^2 \equiv r \pmod{p}$

Por ejemplo los residuos cuadráticos módulo 5 son $0, 1, 4$. Notar que 0 siempre es residuo cuadrático módulo p.

- Si $r \not\equiv 0$ es residuo cuadrático, ¿Cuánto vale $r^{\frac{p-1}{2}}$?
- $r \equiv x^2$ para algún $x \not\equiv 0$, y entonces $r^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv 1 \pmod{p}$
- Se puede verificar que además si para algún r vale $r^{\frac{p-1}{2}} \equiv 1 \pmod{p}$, r es residuo cuadrático módulo p.

Contenidos

Potenciación logarítmica

- En los ejemplos anteriores hemos reducido algunos problemas a calcular a^b módulo M, para enteros no negativos a, b, M.
- ¿Cómo hacemos esto más eficientemente que realizando $b - 1$ multiplicaciones?

Potenciación logarítmica (idea)

- Una buena idea es pensar en ir elevando al cuadrado sucesivamente, lo cual permite que el exponente crezca rápidamente.
- Pensado de manera recursiva, si llamamos $f(a,n) = a^n$:

$$f(a,n) = \begin{cases} 1 & \text{si } n = 0 \\ f\left(a^2, \frac{n}{2}\right) & \text{si } n \text{ es par} \\ a \cdot f\left(a^2, \left\lfloor \frac{n}{2} \right\rfloor\right) & \text{si } n \text{ es impar} \end{cases}$$

Potenciación logarítmica (código)

```
typedef long long tint;
tint potlog(tint a, tint b, const tint M)
{
    tint res = 1;
    while (b > 0)
    {
        if (b % 2 != 0)
            res = MOD(res*a, M);
        a = MOD(a*a, M);
        b /= 2;
    }
    return res;
}
```

Invariante de ciclo:
La respuesta que deseamos es $res \cdot (a^b$ módulo $M)$
Este método realiza solamente $O(\lg b)$ multiplicaciones.

Contenidos

Problema

- Una necesidad muy usual al trabajar con números primos es la de calcular *todos* los primos desde 1 hasta N para un cierto N.
- La idea es realizarlo de manera más eficiente que verificando la primalidad de cada número por separado.

Idea 1

- Si verificásemos cada número por separado, lo que haríamos sería recorrer sus posibles **divisores** para ver si es primo.
- ¿Que pasaría si en lugar de probar los divisores de cada número, descartásemos sus **múltiplos**?

Idea 1

- Si verificásemos cada número por separado, lo que haríamos sería recorrer sus posibles **divisores** para ver si es primo.
- ¿Que pasaría si en lugar de probar los divisores de cada número, descartásemos sus **múltiplos**?
- Hay $\frac{N}{1}$ múltiplos de 1, $\frac{N}{2}$ múltiplos de 2, $\cdots$ $\frac{N}{N}$ múltiplos de N. Luego si recorremos todo el costo total es

$$\sum_{i=1}^{N} \frac{N}{i} = N \sum_{i=1}^{N} \frac{1}{i} = \Theta(N \lg N)$$

Idea 1 (cont.)

- Esta idea es verdaderamente muy sencilla, y como vimos ya alcanza una eficiencia aceptable.
- Además, al recorrer los múltiplos de todos los números, se encuentran todos los divisores propios de todos los números.
- De esta forma es extremadamente fácil modificar esta versión para computar fácilmente sumas de divisores, cantidades, y otras funciones similares.

```
for(int i = 0; i < MAX; i++) p[i] = true;
p[0] = p[1] = false;
for (int i = 2; i < MAX; i++)
    for (int j = 2*i; j < MAX; j += i) p[j] = false;
```

Idea 2

- Todo número compuesto tiene un divisor **primo**.
- Por lo tanto, alcanza con descartar los múltiplos de los números primos que vamos encontrando.
- La cantidad de operaciones a realizar en este caso se reduce a $\Theta(N \lg \lg N)$, que es "casi lineal".

```
for(int i = 0; i < MAX; i++) p[i] = true;
p[0] = p[1] = false;
for (int i = 2; i < MAX; i++)
if (p[i])
    for (int j = 2*i; j < MAX; j += i) p[j] = false;
```

Idea 3

- Todo número compuesto tiene un divisor primo p, con $p \leq \sqrt{N}$.
- Podemos parar el proceso de descarte de múltiplos en $\sqrt{N}$.
- La cantidad de operaciones sigue siendo $\Theta(N \lg \lg N)$.

```
for(int i = 0; i < MAX; i++) p[i] = true;
p[0] = p[1] = false;
for (int i = 2; i*i < MAX; i++)
if (p[i])
    for (int j = 2*i; j < MAX; j += i) p[j] = false;
```

Idea 4

- De manera similar al caso anterior, si $N < p^2$ es compuesto, tiene un divisor primo menor que p, y ya habrá sido descartado.
- Podemos comenzar el descarte de los múltiplos de i por i^2.
- Aún con esta optimización, la cantidad de operaciones sigue siendo $\Theta(N \lg \lg N)$.

```
for(int i = 0; i < MAX; i++) p[i] = true;
p[0] = p[1] = false;
for (int i = 2; i*i < MAX; i++)
if (p[i])
    for (int j = i*i; j < MAX; j += i) p[j] = false;
```

Tiempos

Para `MAX=300M`:

`RecorrerMultiplos`	39.1 s
`CribaBasica`	5.6 s
`CribaHastaRaiz`	3.2 s
`CribaHastaRaizDesdeICuadrado`	3.0 s

Contenidos

Contenidos

Algoritmo ingenuo

- Como ya hemos mencionado, un número compuesto N tendrá un divisor primo menor o igual a $\sqrt{N}$
- Un algoritmo simple $O(\sqrt{N})$ consistirá entonces de un chequeo de todos los números enteros en el rango $[2, \sqrt{N}]$, en busca de divisores de N.

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.
- Observemos que si $p > 2$ es primo y $x^2 = 1 \pmod{p}$, x solo puede ser 1 o -1 módulo p.
- Luego 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 $a^{2^{\alpha-2} k} \equiv 1 \text{ o } -1 \pmod{p}$
- 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:

$$
\begin{array}{rcccccccl}
\mathbf{j} & & & & & & & & \\
\alpha & 1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-2 & ? & -1 & \cdots & 1 & 1 & 1 & 1 & \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\
2 & ? & ? & \cdots & -1 & 1 & 1 & 1 & \\
1 & ? & ? & \cdots & ? & -1 & 1 & 1 & \boxed{a^{2k} = (a^k)^2 \equiv -1} \\
0 & ? & ? & \cdots & ? & ? & -1 & 1 &
\end{array}
$$

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:

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:

$$
\begin{array}{r|cccccccl}
\textbf{j} & & & & & & & & \\
\alpha & 1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-1 & -1 & 1 & \cdots & 1 & 1 & 1 & 1 & \\
\alpha-2 & ? & -1 & \cdots & 1 & 1 & 1 & 1 & \boxed{a^{2^{\alpha-2}k} \equiv -1} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\
2 & ? & ? & \cdots & -1 & 1 & 1 & 1 & \\
1 & ? & ? & \cdots & ? & -1 & 1 & 1 & \\
0 & ? & ? & \cdots & ? & ? & -1 & 1 &
\end{array}
$$

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:

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 es compuesto, 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 < 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

Contenidos

Factorización logarítmica

- Buena alternativa si nos interesa poder factorizar rápidamente cualquier número hasta N, y aceptamos el costo de una criba hasta N.
- Si en lugar de solamente guardar si un número es primo o no, guardamos un primo que lo divida mientras hacemos la criba, luego podemos saber un divisor primo de cualquier número en $O(1)$. Esto permite factorizar cualquier número en $O(\lg N)$.

```
for(int i = 0; i < MAX; i++) p[i] = i;
p[0] = p[1] = 1;
for (int i = 2; i*i < MAX; i++)
if (p[i] == i)
    for (int j = i*i; j < MAX; j += i) p[j] = i;
```

Contenidos

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

Introducción

Problema

Supongamos que tenemos una sucesión $x_1, x_2, x_3, \cdots$
Queremos ir leyéndola hasta encontrar la primera repetición (es decir, los menores i, j tales que $x_i = x_j$ con $i < j$).

- ¿Cómo podemos resolver esta tarea?

Introducción

Problema

Supongamos que tenemos una sucesión $x_1, x_2, x_3, \cdots$
Queremos ir leyéndola hasta encontrar la primera repetición (es decir, los menores i, j tales que $x_i = x_j$ con $i < j$).

- ¿Cómo podemos resolver esta tarea?
- Árbol binario de búsqueda
- Tabla hash.
- Lista de valores

Introducción (cont.)

- Un árbol binario de búsqueda (`set` de C++, `TreeSet` de Java) es una estructura eficiente que lo resuelve en $O(j \lg j)$.
- Requiere $O(j)$ memoria.
- Requiere un operador $<$ para los valores.

Introducción (cont.)

- Una tabla hash (`unordered_set` de C++, `HashSet` de Java) es una estructura eficiente que lo resuelve en $O(j)$ (**asumiendo una buena función de Hash**).
- Requiere $O(j)$ memoria.
- Requiere que se pueda computar una función de hash sobre cada valor.

Introducción (cont.)

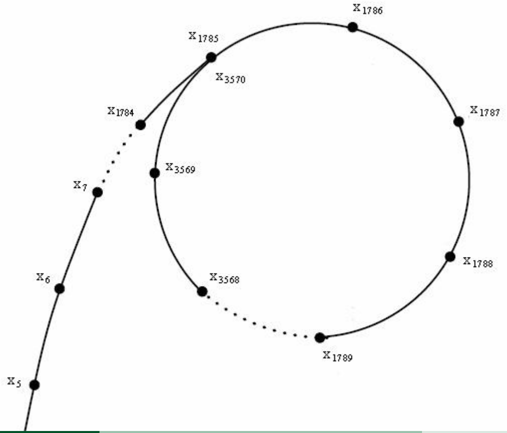
- Una simple lista de valores (`vector` de C++, `ArrayList` de Java) es una estructura que lo resuelve en $O(j^2)$.
- Requiere $O(j)$ memoria.
- Únicamente requiere un operador de igualdad ($=$) sobre los elementos.

Introducción (cont.)

- En el caso general (si leemos los x_i de la entrada) es difícil mejorar estas estructuras.
- Sin embargo, un caso muy común se da cuando la sucesión se obtiene por aplicación reiterada de alguna función f:
  $x_1, f(x_1), f(f(x_1)), f(f(f(x_1))), \cdots$
- Veremos un algoritmo para dicho caso particular, que logrará lo mejor entre todos ellos y más:
  - O(1) memoria
  - O(j) tiempo (u $O(j)$ aplicaciones de f si el costo de f no es $O(1)$)
  - **Únicamente requiere un operador de igualdad ($=$) sobre los elementos**

Estructura de ρ

- En este caso, cuando aparezca la primera repetición $x_i = x_j$, necesariamente será $x_{i+1} = f(x_i) = f(x_j) = x_{j+1}$ y la secuencia entra en un ciclo de período $j - i$ que comienza en i.
- Gráficamente ($i = 1785$, $j = 3570$):



Caso aún más particular

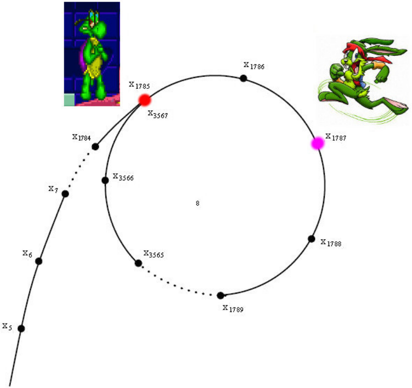
- Cuando la f es inversible, no es difícil ver que la primera repetición será de la forma $i = 1$, $x_1 = x_j$.
- En efecto, si f es inversible, la ρ debe degenerar a un ciclo simple, pues de lo contrario x_i tendría dos antecesores por f, y eso no puede ocurrir.
- El algoritmo para este caso sencillo es entonces aplicar f sucesivas veces hasta encontrar un elemento tal que $x_j = x_1$.

Algoritmo de la liebre y la tortuga de Floyd

- La idea en este caso es tener dos punteros, $a = x_2$ que será la tortuga y $b = x_3$ que será la liebre.
- a avanzará de a un elemento, recorriendo toda la secuencia.
- b en cambio avanzará de a **dos** elementos.
- En cada paso verificamos si $x_a = x_b$ y continuamos hasta que así sea.
- Notar que luego de i pasos (no conocemos i), la tortuga y la liebre estarán ambos en el ciclo (digamos en $a_0 = i$ y b_0).
- Luego de eso, en a lo más $j - i = T$ pasos más coincidirán (su diferencia se incrementa en 1 cada paso).
- Notar que si solamente nos interesa encontrar **alguna** coincidencia, podemos terminar aquí.

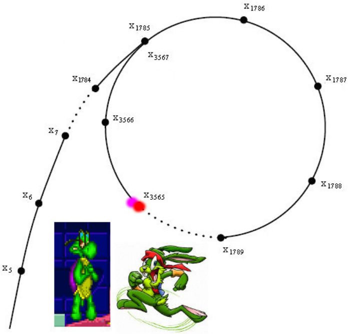
Algoritmo de la liebre y la tortuga de Floyd (cont)

- Luego de $i = 1784$ pasos. $a_0 = 1785$ y $b_0 = 1787$. Llamemos $d = b_0 - a_0 = 2$.



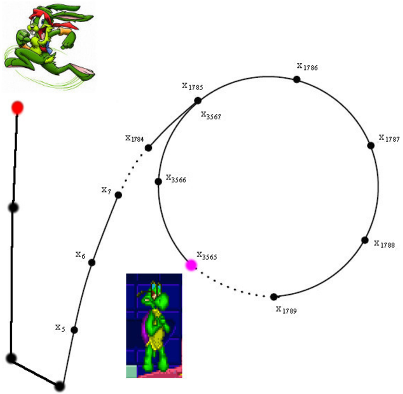
Algoritmo de la liebre y la tortuga de Floyd (cont)

- Como en cada paso adicional la liebre se acerca en 1 a la tortuga, la alcanza luego de $T - d = 1780$ pasos más. Notar que se encuentran a d del comienzo del ciclo.



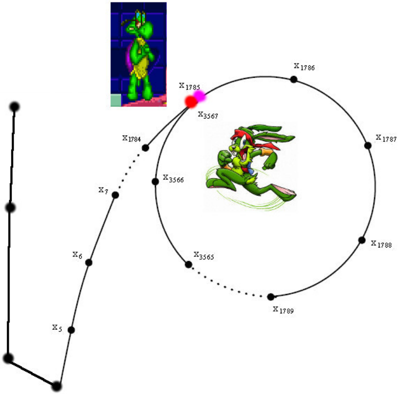
Algoritmo de la liebre y la tortuga de Floyd (cont)

- Para completar el algoritmo, una vez que se encuentran ambos, reinicializamos la liebre (o la tortuga, da igual) al origen, y además, ahora hacemos moverse a la liebre de a un solo paso por vez, igual que la tortuga.



Algoritmo de la liebre y la tortuga de Floyd (cont)

- *i* pasos más tarde, ambos estarán en x_i, que será el nuevo punto de encuentro. Una vez allí, es fácil dejar uno fijo y dar una vuelta al ciclo con el otro para determinar su longitud. La cantidad total de pasos es como mucho $2j$.



Alternativa

- Una alternativa menos mencionada en la literatura, pero con mejores constantes (pero iguales complejidades asintóticas) es el algoritmo de Brent.
- Este se basa en ir duplicando la longitud de ciclo candidata en cada paso. Se puede leer sobre él en wikipedia (*Brent's Cycle Detection*).

Contenidos

Algoritmo de la ρ de Pollard

- Más tarea.