Primalidad, Factorización y más

Pablo Blanc
(con diapos robadas a Agustín Santiago Gutiérrez)

Buen Kilo de Pan Flauta

Training Camp 2017

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

Pequeño teorema de Fermat

Teorema

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

Aplicación 1: Cálculo de inversos

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

Aplicación 1: Cálculo de inversos

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

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

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

MCD

Dados a y b podemos calcular su máximo común divisor $d = (a : b)$.

MCD

Dados a y b podemos calcular su máximo común divisor $d = (a : b)$.

Sabemos que existen x e y tal que $ax + by = d$.

MCD

Dados a y b podemos calcular su máximo común divisor $d = (a : b)$.

Sabemos que existen x e y tal que $ax + by = d$.

Si a y b son coprimos tenemos $ax + by = 1$, entonces x es el inverso de a módulo b.

Algortimo de Euclides

Podemos hallar x e y mediantes el algoritmo de Euclides.

```
struct dxy {tint d,x,y;};
dxy mcde(tint a, tint b) {
   dxy r, t;
   if (b == 0) {
      r.d = a; r.x = 1; r.y = 0;
   } else {
      t = mcde(b,a %b);
      r.d = t.d; r.x = t.y;
      r.y = t.x - a/b*t.y;
   }
   return r;
}
```

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

Teorema Chino del Resto

Supongamos que $n_1, n_2, \ldots, n_k$ son enteros positivos coprimos dos a dos. Entonces, para enteros dados $a_1, a_2, \ldots, a_k$, existe un entero x que resuelve el sistema de congruencias simultáneas

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

y este x es único módulo $N = n_1 n_2 \ldots n_k$.

Teorema Chino del Resto

```
tint modq(x, q) { return (x % q + q) % q ; }
tint tcr(tint* r, tint* m, int n) {
   tint p=0, q=1;
   forn(i, n) {
      p = modq(p-r[i], q);
      dxy w = mcde(m[i], q);
      if (p % w.d) return -1; // sistema incompaible
      q = q / w.d * m[i];
      p = modq(r[i] + m[i] * p / w.d * w.x, q);
   }
   return p; // x \equiv p (q)
}
```

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

Criba

La criba de Eratóstenes nos permite hallar todos los números primos menores que un número natural dado N.

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

Su complejidad es $O(N \cdot ln(ln(N)))$.

Factorización logarítmica

Si nos interesa poder factorizar rápidamente cualquier número hasta N, 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 $O(\lg N)$.

```
1    for(int i = 0; i < N; i++) p[i] = i;
2    p[0] = p[1] = 1;
3    for (int i = 2; i*i < N; i++)
4    if (p[i] == i)
5        for (int j = i*i; j < N; j += i) p[j] = i;
```

Contenidos

1 Preliminares
- Fermat
- MCD
- Teorema Chino del Resto
- Criba

2 Primalidad
- Verificación directa
- Test de Rabin - Miller

3 Factorización
- Factorización directa
- Algoritmo de la liebre y la tortuga de Floyd
- Factorización rápida

4 Multiplicación Rápida
- Karatsuba
- Fast Fourier transform

Algoritmo ingenuo

- 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

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

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):

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

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	$a^k \equiv 1$ o -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

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

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
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
2	?	?	$\cdots$	-1	1	1	1
1	?	?	$\cdots$	?	-1	1	1
0	?	?	$\cdots$	?	?	-1	1

$a^{2^{\alpha-2}k} \equiv -1$

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	$a^{2^{\alpha-1}k} \equiv -1$
$\alpha - 2$	?	-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 < 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

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

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 $\Theta(\sqrt{N})$

```
1   for(int i = 2; i*i <= N; i++)
2   while (N %i == 0)
3   {
4       N /= i;
5       reportarFactorPrimo(i);
6   }
7   if (N > 1)
8       reportarFactorPrimo(N);
```

Contenidos

1 Preliminares
- Fermat
- MCD
- Teorema Chino del Resto
- Criba

2 Primalidad
- Verificación directa
- Test de Rabin - Miller

3 Factorización
- Factorización directa
- Algoritmo de la liebre y la tortuga de Floyd
- Factorización rápida

4 Multiplicación Rápida
- Karatsuba
- Fast Fourier transform

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.)

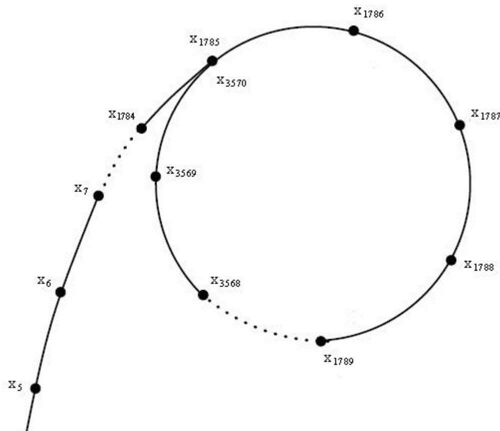
- Una simple lista de valores (`vector` de C++, `ArrayList` de Java) es una estructura que lo resuelve en $O(j^2)$.
- Requiere $O(j)$ memoria.
- Únicamente requiere un operador de igualdad ($=$) sobre los elementos.

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

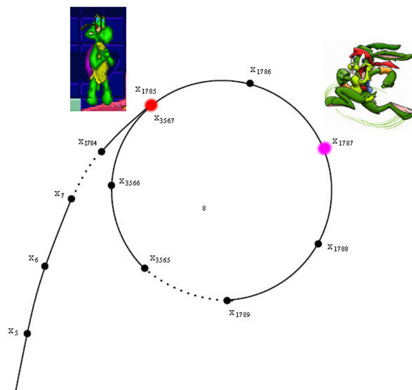
- Cuando la f es inversible, no es difícil ver que la primera repetición será de la forma $i = 1$, $x_1 = x_j$.
- En efecto, si f es inversible, la ρ debe degenerar a un ciclo simple, pues de lo contrario x_i tendría dos antecesores por f, y eso no puede ocurrir.
- El algoritmo para este caso sencillo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

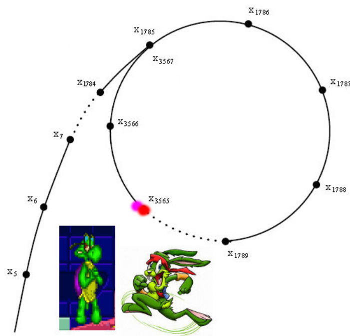
Algoritmo de la liebre y la tortuga de Floyd (cont)

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



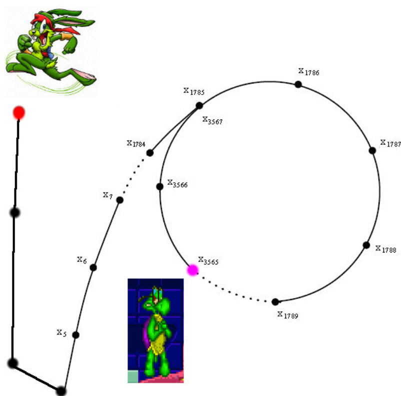
Algoritmo de la liebre y la tortuga de Floyd (cont)

- 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.



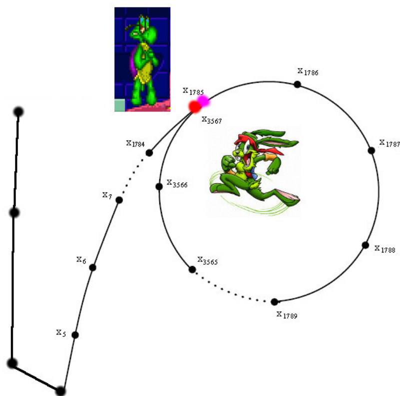
Algoritmo de la liebre y la tortuga de Floyd (cont)

- 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 factores constantes (aunque idénticas 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

1 Preliminares
- Fermat
- MCD
- Teorema Chino del Resto
- Criba

2 Primalidad
- Verificación directa
- Test de Rabin - Miller

3 Factorización
- Factorización directa
- Algoritmo de la liebre y la tortuga de Floyd
- Factorización rápida

4 Multiplicación Rápida
- Karatsuba
- Fast Fourier transform

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 $p | MCD(|x - y|, N)$
- Este MCD puede calcularse con el algoritmo de Euclides sin conocer p.
  - Si da $1 < MCD < N$, hemos encontrado un factor de N.
  - Si da $MCD = N$, hemos tenido una repetición en la secuencia módulo N.
  - Si da $MCD = 1$, no hemos detectado ninguna repetición módulo p.

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

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.
- Con esto la secuencia será x_1, $f(x_1)$, $f(f(x_1))$ y podemos utilizar el algoritmo de la liebre y la tortuga.

Algoritmo de la ρ de Pollard (implementación)

- Una función pseudoaleatoria módulo N que funciona bien es $f(X) = X^2 + AX + B$, con $1 \leq A, B < N$ elegidos al azar.

```
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 = mcd(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 MCD.
- 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.

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

Karatsuba

- Supongamos que queremos multiplicar x e y, dos números de N digitos. (vamos a pensar en base 10 pero podria ser en una base grande o podrian tratarse de polinomios).
- Para hacerlo de manera directa necesitamos hacer $O(N^2)$ multiplicaciones (nos interesa contar la cantidad de multiplicaciones que hacemos y no las sumas que son mucho más baratas).

Karatsuba

La idea del algoritmo de Karatsuba es escribir

$$x = a10^{N/2} + b$$
$$y = c10^{N/2} + d$$
$$xy = ac10^{N} + (ad + cb)10^{N/2} + bd$$

y para calcular ac, $ad + cb$ y bd hacemos tres multiplicaciones. Calculamos ac, bd y $(a + b)(c + d)$. Luego

$$ad + cb = (a + b)(c + d) - ac - bd.$$

Karatsuba

La idea del algoritmo de Karatsuba es escribir

$$
\begin{aligned}
x &= a10^{N/2} + b \\
y &= c10^{N/2} + d \\
xy &= ac10^{N} + (ad + cb)10^{N/2} + bd
\end{aligned}
$$

y para calcular ac, $ad + cb$ y bd hacemos tres multiplicaciones.
Calculamos ac, bd y $(a + b)(c + d)$. Luego

$$ad + cb = (a + b)(c + d) - ac - bd.$$

Con esta optimización logramos una complejidad de $O(N^{\log_2 3})$.

Karatsuba

```
karatsuba(x,y)
   if x < 10 or y < 10
      return xy
   m =(max(log10(x), log10(y)) + 1) / 2
   pot = 10^m
   b=x%pot, a=x/pot
   d=y%pot, c=y/pot
   z0 = karatsuba(a, c)
   z1 = karatsuba(a + b, c + d)
   z2 = karatsuba(b, d)
   devolver z0  pot^2 + (z1 - z0 - z2)  pot + z2
```

Contenidos

1. Preliminares
   - Fermat
   - MCD
   - Teorema Chino del Resto
   - Criba
2. Primalidad
   - Verificación directa
   - Test de Rabin - Miller
3. Factorización
   - Factorización directa
   - Algoritmo de la liebre y la tortuga de Floyd
   - Factorización rápida
4. Multiplicación Rápida
   - Karatsuba
   - Fast Fourier transform

FFT

Dados $p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ y $q(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}$ queremos calcular $c(x) = p(x)q(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2}$.

- Plan 1: nos sale en $O(n^2)$.
- Plan 2: Karatsuba nos sale en $O(n^{\log_2 3})$.
- Plan 3: Cooley–Tukey FFT, nos va a salir en $O(n \ln(n))$.

(Lo de FFT lo saque de http://web.cs.iastate.edu/ cs577/handouts/polymultiply.pdf)

Idea

- Elegir m puntos $x_0, x_1, ..., x_{2n-2}$.
- Evaluar p en estos puntos.
- Evaluar q en estos puntos.
- Calcular $c(x_0), \ldots, c(x_{2n-2})$, con la formula $c(x) = p(x)q(x)$.
- Interpolar para hallar los coeficientes c_i.

Discrete Fourier Transform

Sea w_n una raíz primitiva n-esima de la unidad (podemos trabajar en $\mathbb{C}$ o en $\mathbb{Z}_p$ para cierto p de forma tal que exista una raiz primitiva n-esima, $n|p-1$).
Sea $p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ (asumimos n potencia de 2).
Queremos evaluar p en $1, w_n, \ldots, w_n^{n-1}$.

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w_n & w_n^2 & \ldots & w_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \ldots & w_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Discrete Fourier Transform

Definimos

$$p_0(x) = a_0 + a_2 x + a_{n-2} x^{\frac{n}{2}-1}$$

$$p_1(x) = a_1 + a_3 x + a_{n-1} x^{\frac{n}{2}-1}$$

Luego

$$p(x) = p_0(x^2) + x p_1(x^2)$$

Entonces necesitamos evaluar p_0 y p_1 en $1, w_n^2, \ldots, (w_n^{n-1})^2$.
Debemos notar que esta lista de tiene solo $n/2$ números.
Esto nos va a permitir hacer un algoritmo del tipo divide and conquer que va a tomar $O(n \ln(n))$.

DFT

```
DFT(a,n):
    si n = 1
        devolver a
    a0=[a[0],a[2],...,a[n-2]]
    a1=[a[1],a[3],...,a[n-1]]
    y0 = DFT(a0, n/2)
    y1 = DFT(a1, n/2)
    w=1
    for k=0 to n/2-1
       y_k=y0_k+w y1_k
       y_{k+n/2}=y0_k-w y1_k
       w=w w_n
    return y
```

Inverse Discrete Fourier Transform

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)^2} \end{pmatrix}^{-1} =$$

$$\frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n^{-1} & w_n^{-2} & \dots & w_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & w_n^{-(n-1)} & w_n^{-2(n-1)} & \dots & w_n^{-(n-1)^2} \end{pmatrix}$$

Entonces recuperar los coeficientes es analogo a lo que ya hicimos.

FFT

- Evaluamos p y q en $1, w_{2n}, w_{2n}^2, \ldots, w_{2n}^{2n-1}$.
- Calculamos $c(1), c(w_{2n}), c(w_{2n}^2), \ldots, c(w_{2n}^{2n-1})$, con la formula $c(x) = p(x)q(x)$.
- Interpolamos los coeficientes c_i.

Referencias

- *Introduction to Algorithms, 2nd Edition*. MIT Press.
  31 Number-Theoretic Algorithms
  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein