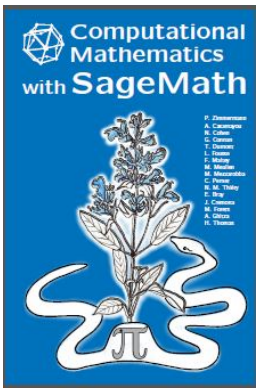
Notas_algebra_Abstracta



July 8, 2021

notas de rjcg: Algebra Abstracta - **Campos finitos y Teoría de Números elemental**. Resumen de: Computational Mathematics with SageMath/Paul Zimmermann

0.0.1 6.1 Los Campos Finitos y Anillos.

Son objetos básicos, en teoría de número y álgebra computacional. Muchos algoritmos en álgebra computarizada envuelven cómputos sobre campos finitos.

> **nota:** Un conjunto no vacío R es un anillo si tiene dos operaciones binarias (adición y multiplicación; que no debes ser interpretada como en su contexto aritmético), que satisfacen las siguientes condiciones.

6.1.1 El anillo de los enteros módulo n. Los anillos son estructuras matemáticas que cumplem dos leyes de composición interna binarias (distributiva y asociativa). En Sage, el anillo $\mathbb{Z}/n\mathbb{Z}$ de los enteros módulo n, usan el constructor **IntegerModRing** (o simplemente **Integer**). Todos los objetos construidos con este constructor y los derivados de ellos se reducen sistemáticamente módulo n, por lo que tienen una forma canónica (o normal): es decir, dos variables que representan el mismo valor módulo n también tienen la misma representación interna.

> **nota:** Los enteros forman un anillo; siendo $\mathbb{Z}$ un dominio integral (ya que posee la unidad o identidad: Existe un elemento $1 \in R$ tal que $1 \neq 0$ y $1a = a1 = a$ para cada elemento $a \in R$). De hecho si $ab = 0$ para dos enteros a y b, ya sea: $a = 0$ o $b = 0$. Por otro lado, Bajo las operaciones usuales de adición y multiplicación, todos los sistemas de números familiares son anillos: los racionales, $\mathbb{Q}$; los números reales, $\mathbb{R}$; y los números complejos, $\mathbb{C}$. Cada uno de estos anillos es un cuerpo. Sin embargo, $\mathbb{Z}$ no es un cuerpo. No hay un entero que sea el inverso multiplicativo de 2, pues $1/2$ no es un entero. Los únicos enteros con inverso multiplicativo son 1 y 1.

```
In [2]: a = IntegerModRing(15)(3); b = IntegerModRing(17)(3); a, b
```

```
Out[2]: (3, 3)
```

```
In [3]: a == b
```

```
Out[3]: False
```

Se puede recopilar información acerca de n, usando el método **base_ring** o **parent**, y del valor de n usando el método **characteristic**:

```
In [4]: R = a.parent(); R

Out[4]: Ring of integers modulo 15

In [5]: R.characteristic()

Out[5]: 15
```

Las operaciones básicas (adición, substracción y multiplicación) están sobrecargadas para los enteros módulo n, y llama a la función apropiada; también, los enteros son convertidos automáticamente cuando uno de los operandos es un entero módulo n:

```
In [6]: a + a, a - 17, a * a + 1, a^3

Out[6]: (6, 1, 10, 12)
```

Para la inversión, $1/a \bmod (n)$, o para la división, $b/a \bmod (n)$, Sage ejecuta las operaciones si es posible; por ejemplo, cuando a y n tienen factores comunes no triviales, un error de **ZeroDivisionError** es generado:

```
In [7]: 1/(a+1)

Out[7]: 4
```

> ***Nota:*** El inverso multiplicativo de un número entero n módulo p es otro entero $m = \bmod(p)$ tal que el producto mn es congruente con 1 mod (p). Esto significa que tal número m es el inverso multiplicativo en el anillo de los enteros z mod (p), es decir, $n - 1 = m \bmod (p)$. El inverso multiplicativo de n mod (p) existe si y sólo si n y p son coprimos, es decir, si $mcd(n, p) = 1$ (en Sage **gcd**). Si existe el inverso multiplicativo de un número n módulo p, entonces se puede definir la operación de división de cualquier otro número entre n módulo p, mediante la multiplicación de ese número por el inverso $n - 1$. Si p es un número primo, entonces todos los números excepto el cero (y sus congruentes —los múltiplos de p) son invertibles, lo que convierte al anillo de los enteros módulo p en un cuerpo.

```
In [8]: #  1/a # esto resultará en un ZeroDivisionError
```

Para obtener el valor de a, como un entero de este residuo $a \bmod (n)$, uno puede usar el método **lift** o incluso **ZZ**:

```
In [9]: z = a.lift(); y = ZZ(a); y, type(y), y == z

Out[9]: (3, <type 'sage.rings.integer.Integer'>, True)
```

El orden aditivo de $a \bmod (n)$, es el más pequeño entero $k > 0$ tal que $ka \equiv 0 \bmod (n)$. Esto es igual a $k = n/g$, donde g es el máximo común divisor; $g = gcd(a, n)$. El orden aditivo es dato por el método **additive_order** (nota: también se puede usar **Mod** o **mod** para definir los enteros módulo n):

```
In [10]: [[x,Mod(x,15).additive_order()]
             for x in range(0,8)]          # los ocho primeros valores
```

```
Out[10]: [[0, 1], [1, 15], [2, 15], [3, 5], [4, 15], [5, 3], [6, 5], [7, 15]]
```

EL orden multiplicativo de a módulo n, para un coprimo (primo relativo) a n, es el más pequeño entero $k > 0$ tal que $z^k = 1 \bmod (n)$; donde $\equiv$ denota la congruencia. Si se tiene un divisor común p con n, entonces $a^k \bmod (n)$ debería ser un múltiplo de p para todo k. El orden multiplicativo en Sagé, es dato por el método **multiplicative_order**. Si el orden multiplicativo es igual a $\varphi(n)$, el cual es el orden del grupo multiplicativo módulo n, uno dice que a es un generador de este grupo. También para $n = 15$, no existen generadores, ya que el orden máximal es $4 < 8 = \varphi(15)$:

```
In [11]: [[x, Mod(x,15).multiplicative_order()]
             for x in range(1,15) if gcd(x,15) == 1]

Out[11]: [[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14, 2]]
```

Un ejemplo con $n = p$ primo, donde 3 es un generador:

```
In [12]: p = 10^20 + 39; mod(2,p).multiplicative_order()

Out[12]: 50000000000000000019

In [13]: mod(3,p).multiplicative_order()

Out[13]: 100000000000000000038
```

Una operación importante sobre $\mathbb{Z}/n\mathbb{Z}$ es la exponenciación modular, la cual permite calcular $a^e \bmod (n)$, el algoritmo más eficiente requiere de el orden de $log(e)$ multiplicado por el cuadrado del módulo n. Esto es crucial para reducir todos los cálculos módulo n sistemáticamente, y no computar a^e, como si fuera un entero, como en el siguiente ejemplo:

```
In [14]: n = 3^100000; a = n-1; e = 100

In [15]: %timeit (a^e) % n

1 loop, best of 3: 1.11 s per loop


In [16]: %timeit power_mod(a,e,n)

10 loops, best of 3: 18.5 ms per loop
```

0.0.2 6.1.2 Campos Finitos

Los campos finitos se definen usando el constructor **FiniteField**, o simplemente **GF**. O también como el constructor primo **GF(p)**, com p primo. Se puede construir un campo finito no-primo $GF(q)$, con $q = pk$, donde p es primo y $k > 1$ un entero.

> **nota:** Los campos finitos con q elementos, son denotados como $\mathbb{F}_q$, o GF(q); donde "GF" es la abreviación de "Galois Field". Aquí usamos $\mathbb{F}_q$ para los objetos matemáticos, y los denotamos en código Sage, como: **GF(q)**.

Como con los anillos, los objetos creados como campo tienen una representación canónica,una reducción es llevada a cabo, en cada operación aritmética. Los campos finitos tienen las mismas propiedades que los anillos; con una adicional, la propiedad de invertir cada elemento no-cero:

> **nota:** El orden de un campo finito es el número de elementos en el campo. Existe un campo finito de orden q si y solo si q es la potencia de un número primo. Si q es la potencia de un primo, existe esencialmente un solo campo finito de orden q al cual denotaremos como $GF(q)$. Si $q = p^m$ donde p es un primo y m un entero positivo, entonces p es denominado la característica del campo $GF(q)$ y m es denominado el grado de extensión de $GF(q)$.

```
In [17]: # Hay que tener absoluto cuidado al definir un anillo (o un campo)...
         # R = IntegerModRing(33, is_field=True)
         # R in Fields()             # cierto
         # R is Fields()              # falso
         # R.is_field(proof= true)  # error
```

```
ValueError: THIS SAGE SESSION MIGHT BE SERIOUSLY COMPROMISED!
The order 33 is not prime, but this ring has been put
into the category of fields. This may already have consequences
in other parts of Sage. Either it was a mistake of the user,
or a probabilistic primality test has failed.
In the latter case, please inform the developers.
```

```
In [18]: R = GF(17); [1/R(x) for x in range(1,17)]
```

```
Out[18]: [1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

```
In [19]: [R(x) for x in range(1,17)]
```

```
Out[19]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

> **nota:** 15 es la carácteristica del anillo $\mathbb{Z}/n\mathbb{Z}$ y 3 el número a representar. Si el argumento opcional proof=True se proporciona, se prueba la primalidad y se informa la asignación de categoría errónea; si no es correcta, Sage responderá con una advertencia:

```
In [20]: R.is_field(proof=True)
```

```
Out[20]: True
```

Un campo finito no-primo $\mathbb{F}_p k$, con p primo y $k > 1$, es isomorfo al anillo del cociente de polinomios en $\mathbb{F}_p$, módulo un polinomio mónico irreductible f de grado k. En este caso, Sage proverá un nombre para el generador de campo, es decir la variable x; o el usuario puede proporcionar un nombre:

```
In [21]: R = GF(9,name='x'); R
```

```
Out[21]: Finite Field in x of size 3^2
```

Aquí, Sage automáticamente seleccionó el polinomio f:

```
In [22]: R.polynomial()

Out[22]: x^2 + 2*x + 2
```

Por tanto, los elementos de campo se representan mediante polinomios en el generador x, $a_{k1}x^{k1} + + a_1x + a_0$, con coeficientes a_i los cuales son elementos de $\mathbb{F}_p$:

Uno puede hacer que Sage use un específico polinomio irreductible f:

```
In [23]: Q.<x> = PolynomialRing(GF(3))
         R2 = GF(9, name='x', modulus=x^2+1); R2

Out[23]: Finite Field in x of size 3^2
```

Tenga cuidado: aunque los dos campos R y R2 creados anteriormente son ambos isomorfo a $\mathbb{F}_9$, Sage no proporciona isomorfismo entre ellos automáticamente:

```
In [24]: # p = R(x+1); R2(p)

Traceback (most recent call last):
...
TypeError: unable to coerce from a finite field other than the prime subfield
```

0.0.3 6.1.3 Reconstrucción racional

El problema de la reconstrucción racional es una aplicación útil de los métodos modulares. Dado un residuo a módulo m, implica encontrar un número racional "pequeño" x/y tal que $x/y \equiv a \bmod (m)$. Si se sabe que existe un número racional tan pequeño, en lugar de calcular x/y directamente como un número racional, se puede calcular x/y módulo m, lo que da el residuo a, y luego se recupera x/y a través de reconstrucción racional. Este segundo enfoque es a menudo más eficiente, ya que se han reemplazado los cálculos por los racionales, posiblemente involucrando costosos cálculos del mcd (en Sage **gcd**), por cálculos modulares.

> **LEMMA**. Sea $a, m \in \mathbb{N}$, con $0 < a < m$. Existe al menos un par de elementos coprimos enteros $x, y \in \mathbb{Z}$, tal que $x/y \equiv a \bmod (m)$, con $0 < |x|, y \leq \sqrt{m/2}$.

Para un par x, y no siempre existe: por ejemplo, tome $a = 2$ y $m = 5$. El algoritmo de reconstrucción racional se basa en el algoritmo euclidiano extendido. El máximo común divisor extendido de m y a calcula una secuencia de números enteros $a_i = \alpha_i m + \beta_i a$, donde los a_i son decrecientes y los coeficientes $\alpha_i, \beta i$ aumentan en valor absoluto. Por lo tanto, basta con detenerse tan pronto como $|\alpha_i|, |\beta_i| \leq \sqrt{m/2}$, y la solución es entonces $x/y = a_i/\beta_i$. Este algoritmo se implementa en la función Sage **racional_reconstruction**, que devuelve x/y cuando existe una solución; generando un error si no:

```
In [25]: rational_reconstruction(411,1000)

Out[25]: -13/17

In [26]: mod(17*411,1000)
         # El resultado es 987 que se corresponde con el módulo -13.
```

```
Out[26]: 987

In [27]: # sage: rational_reconstruction(409,1000)
         # En este caso Sage advierte de un error.

ArithmeticError                           Traceback (most recent call last)
...
ArithmeticError: rational reconstruction of 409 (mod 1000) does not exist
```

Para ilustrar el uso de la reconstrucción racional, considera el computo del enésimo número armónico. En matemáticas, se define el n-ésimo número armónico como la suma de los recíprocos de los primeros n números naturales: $H_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$. Este también es igual a n veces el inverso de la media armónica.

Un cálculo nativo usando números racionales debería ser como el siguiente:

```
In [28]: def harmonic(n):
             return add([1/x for x in range(1,n+1)])
```

Conocemos que H_n puede ser escrito en la forma p_n/q_n com los enteros p_n, q_n, donde $q_n = lcm(1,2,\ldots,n)$. También conocemos que $H_n \leq \log(n) + 1$, el cual nos permite acotar p_n. Esto conduce a la siguiente función, que encuentra H_n usando aritmética modular y reconstrucción racional:

```
In [29]: def harmonic_mod(n,m):
             return add([1/x % m for x in range(1,n+1)])

In [30]: def harmonic2(n):
             q = lcm(range(1,n+1))
             pmax = RR(q*(log(n)+1))
             m = (2*pmax^2)          # originalmente "ZZ(2*pmax^2)"; pero da error
             m = ceil(m/q)*q + 1
             a = harmonic_mod(n,m)
             return rational_reconstruction(a,m)
```

En este ejemplo, la función **harmonic2** no es más eficiente que la función original **harmonic**, pero ilustra el método. No siempre es necesario conocer una cota rigurosa para x e y, ya que una estimación aproximada "a ojo" será suficiente, siempre que se pueda verificar fácilmente que x/y es la solución correcta. Se puede generalizar el método de reconstrucción racional para manejar numeradores x y denominadores y de diferentes tamaños. [ver, por ejemplo, la Sección 5.10 del libro]

Los siguientes ejemplos muestran la "eficiencia" de los algoritmos; para n = 1000.

```
In [31]: n = 1000;
         %timeit harmonic(n)

100 loops, best of 3: 9.89 ms per loop


In [32]: %timeit harmonic2(n)

10 loops, best of 3: 24.9 ms per loop
```

0.0.4 6.1.4 El Teorema Chino del Residuo

Otra aplicación de la aritmética modular se relaciona con el uso del teorema chino del residuo, o CRT. Dado dos módulos coprimo m y n, y dos clases residuales $a \bmod (m)$ y $b \bmod (n)$, si tomamos un entero x tal que $x \equiv a \bmod (m)$ y $x \equiv b \bmod (n)$. El teorema chino del residuo nos permite recuperar x del módulo del producto mn. Para ver como esto trabaja, se deduce de $x \equiv a \bmod (m)$, que x tiene la forma $x = a + \lambda m$, con $\lambda \in \mathbb{Z}$. Sustituyendo en $x \equiv b \bmod (n)$ se obtiene $\lambda \equiv \lambda_0 \bmod (n)$; donde $\lambda_0 = (b-a)/m \bmod (n)$. Aqui $x = x_0 + \mu mn$, donde $x_0 = a + \lambda_0 m$, y μ es un entero arbitrario.

Una variante simple del Teorema Chino del residuo, puede consideral el caso de módulos diversos: $m1, m2, \ldots, mk$. El comnado Sage para encontar $x0$, dados a, b, m, n es: **crt(a,b,m,n)**.

```
In [33]: a = 2; b = 3; m = 5; n = 7; lambda0 = (b-a)/m % n; a + lambda0 * m
         17

Out[33]: 17

In [34]: crt(2,3,5,7)

Out[34]: 17
```

Retornado al computo de H_n. Primero calculamos $H_n \bmod (mi)$ para $i = 1, 2, \ldots, k$, y luego optenemos $H_n \bmod (m1 \ldots mk)$ usando el remanente Chino, y finalmente tendremos el valor de H_n por roconstrucción racional:

```
In [35]: def harmonic3(n):
             q = lcm(range(1,n+1))
             pmax = RR(q*(log(n)+1))
             B = ZZ(2*pmax^2)
             a = 0; m = 1; p = 2^63
             while m < B:
                 p = next_prime(p)
                 b = harmonic_mod(n,p)
                 a = crt(a,b,m,p)
                 m = m*p
             return rational_reconstruction(a,m)

         harmonic(100) == harmonic3(100)

Out[35]: True
```

La función Sage, **crt** puede ser usada también cuando el módulo m y n no son coprimos. Si $g = gcd(mn)$; entonces la solución existirá si y solo si, $a \equiv b \bmod (g)$:

```
In [36]: crt(15,1,30,4)

Out[36]: 45

In [37]: #  crt(15,2,30,4)

ValueError                                Traceback (most recent call last)
...
ValueError: No solution to crt problem since gcd(30,4) does not divide 15-2
```

0.0.5 6.2 Primalidad

Probar si un número entero es primo es una operación fundamental para un paquete de software de computos simbólico. Incluso, con desconocimiento del usuario, el software lleva a cabo estas pruebas miles de veces por segundo. Por ejemplo, para factorizar un polinomio en $\mathbb{Z}[x]$, uno comienza factorizándolo en $\mathbb{F}_p[x]$, para algún número primo p, y por lo tanto uno debe encontrar un primo adecuado.

Comandos utilizables	
Anillo de enteros módulo n	**IntegerModRing(n)**
Campos Finitos con q elementos	**GF(q)**
Test de seudo primalidad	**is_pseudoprime(n)**
Test de primalidad	**is_prime(n)**

Existen dos principales clases de test de primalidad.

Una primera clase, más eficientes, llamadas test de *seudoprimalidad*, los cuales están generalmente basados en la forma del Pequeño Teorema de Fermat; el cual dice que si p es primo, entonces cada entero a, con $0 < a < p$, es un elemento del grupo multiplicativo $\mathbb{Z}/p\mathbb{Z}^*$, y $a^{p-1} \equiv 1 \bmod (p)$. Se usan valores pequeños para a $(2, 3, \dots)$ para incrementar el computo de $a^{p-1} \bmod (p)$. Si $a^{p-1} \not\equiv 1 \bmod (p)$, entonces p es ciertamente no-primo. Si $a^{p-1} \equiv 1 \bmod (p)$, tampoco se puede concluir que p sea primo o no; decimos que p es un pseudoprimo (Fermat) para la base a. Lo intuitivo es que un entero p -que es seudoprimo para muchas bases- tiene una mayor probabilidad de ser primo (pero ... ver más abajo). Las pruebas de pseudoprimalidad comparten la propiedad de que cuando devuelven el veredicto Falso, el número es ciertamente factorizable, mientras que cuando devuelven Verdadero, no es posible una conclusión definitiva.

La segunda clase consiste en verdaderas pruebas de primalidad. Estas pruebas siempre devuelven una respuesta correcta, pero pueden ser menos eficientes que las pruebas de seudoprimalidad, especialmente para números que son seudoprimos para muchas bases y, en particular, para números primos reales. Muchos paquetes de software solo proporcionan pruebas de seudoprimalidad, a pesar de que el nombre de la función correspondiente (isprime, por ejemplo), a veces, lleva al usuario a creer que se proporciona una verdadera prueba de primalidad. Sage proporciona dos funciones diferentes: **is_pseudoprime** para seudo-primalidad, y **is_prime** para verdadera primalidad:

```
In [38]: p = previous_prime(2^400)

In [39]: %timeit is_pseudoprime(p)

1000 loops, best of 3: 1.39 ms per loop


In [40]: %timeit is_prime(p)

1 loop, best of 3: 728 ms per loop
```

En el ejemplo, vemos que la prueba de primalidad es más costosa; mientras sea posible, por lo tanto, se prefiere usar **is_pseudoprime**. Algunos algoritmos de prueba de primalidad proporcionan un certificado que permite una verificación posterior independiente del resultado, a menudo

de manera más eficiente que la prueba en sí. Sage no proporciona dicho certificado en la versión actual, pero se puede construir uno usando el teorema de Pocklington:

> **Teorema de Pocklington**. Sea $n > 1$ un entero impar, tal que $n - 1 = FR$, con $F \geq \sqrt{n}$. Si para cada factor primo p de F, existe un a tal que $a^{n-1} \equiv 1 \bmod (n)$ y $a^{(n-1)/p} - 1$ es coprimo a n, entonces n es primo.

Considere por ejemplo, $n = 2^{31} - 1$. La factorización de $n - 1$ es $2 * 3^2 * 7 * 11 * 31 * 151 * 331$. Podemos tomar $F = 151 * 331$, y $a = 3$ sastisface la condición para ambos factores $p = 151$ y $p = 331$. Aquí es suficiente probar la primalidad de 151 y 331 para deducir que n es primo. Este test usa la exponenciación modular de una manera importante. Otro ejemplo en wiki

Números de Carmichael

Los número de Carmichael son enteros compuestos n, que son seudo-primos a todas las bases coprimas con n. El Pequeño Teorema de Fermat es insuficiente para distinguirlos de los primos, sin importar el número de bases probadas. El más pequeño número de Carmichael es 561 = 3.11.17. Un número de Carmichael debe tener al menos tres factores primos: para suponer que $n = pq$ es un número de Carmichael con p, q primos y $p < q$; por la definición de los números de Carmichael, si a es una raíz primitva módulo q entonces $a^{(n-1)} \equiv 1 \bmod (n)$ implica que la misma congruencia también tiene módulo q, y por lo tanto que $n - 1$ es un múltiplo de $q - 1$. Luego n debe tener la forma $q + \lambda q(q - 1)$, ya que es un múltiplo de q y $n - 1$ es un multiplo de $q - 1$; ahora $n = pq$ implica $p = \lambda(q - 1) + 1$, lo que contradice a $p < q$. Si $n = pqr$, entonces n es un número de Carmichael si $a^{n-1} \equiv 1 \bmod (p)$, y similarmente módulo q y r, el Teorema Chino del Residuo implica $a^{n-1} \equiv 1 \bmod (n)$. Una condición suficiente es que $n - 1$ sea divisible por cada uno de $p - 1$, $ q - 1$ y $r - 1$:

```
In [41]: [560 % (x-1) for x in [3,11,17]]
```

```
Out[41]: [0, 0, 0]
```

Ejercicio: Escriba una función Sage para contar los número de Carmichael $n = pqr \leq N$, con p, q, r primos impares distintos. £Cuántos números se pueden encontrar para $N = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch a encontrado 20138200 números de Carmichael menores que 10^{21}.)

```
In [42]: def count_primes1(n):
             return add([1 for p in range(n+1) if is_prime(p)])
         %timeit count_primes1(10^5)
```

```
1 loop, best of 3: 974 ms per loop
```

La función sería más rápida si se usan seudoprimos:

```
In [43]: def count_primes2(n):
             return add([1 for p in range(n+1) if is_pseudoprime(p)])
         %timeit count_primes2(10^5)
```

```
1 loop, best of 3: 302 ms per loop
```

En este ejeplo, vale la pena usar un bucle en lugar de construir una lista de 10^5 elementos, y nuevamente: usar **is_pseudoprime** es más rápida que **is_prime**:

```
In [44]: def count_primes3(n):
             s = 0; p = 2
             while p <= n: s += 1; p = next_prime(p)
             return s
         %timeit count_primes3(10^5)
```

```
10 loops, best of 3: 69.5 ms per loop
```

```
In [45]: def count_primes4(n):
             s = 0; p = 2
             while p <= n: s += 1; p = next_probable_prime(p)
             return s
         %timeit count_primes4(10^5)
```

```
10 loops, best of 3: 65.3 ms per loop
```

Usando el iterador **prime_range** es todavía más rápido:

```
In [46]: def count_primes5(n):
             s = 0
             for p in prime_range(n): s += 1
             return s
         %timeit count_primes5(10^5)
```

```
10 loops, best of 3: 13.3 ms per loop
```

0.0.6 6.3 Factorización y Logaritmo Discreto

Uno puede decir que un entero a es un cuadrado, o un residuo cuadrático, módulo n, si existe x tal que $a \equiv x^2 \bmod (n)$. Si no se cumple, se dice que a es un cuadrático no residual módulo n. Cuando $n = p$ es primo, existe una prueba para decidir eficientemente si a es un residuo cuadrático, usando el cálculo del símbolo de Jacobi de a y p, denotado: $(a|p)$: el cual toma el valor $\{-1, 0, 1\}$, donde $(a|p) = 0$ cuando a es multiplo de p; el valor $(a|p) = 1$, cuando a es un cuadrado módulo p; y toma el valor $(a|p) = -1$, cuando a no es un cuadrado módulo p. La complejidad del computo del symbolo de Jacobi $(a|n)$ es esencialmente la misma que la de computar el **gcd** (máximo comun divisor) de a y n, llamado $O(M(l)\log(l)$, donde l es el tamaño de n, y $M(l)$ es el costo de multiplicar dos enteros de tamaño l. Sin embargo, en la implementación del símbolo de Jacobi -tanto como con el gcd- no todos tienen esta complejidad (en Sage, **a.jacobi** computa $(a|n)$:

```
In [47]: p = (2^42737+1)//3; a = 3^42737
         %timeit a.gcd(p)
```

```
100 loops, best of 3: 10.9 ms per loop
```

```
In [48]: %timeit a.jacobi(p)

100 loops, best of 3: 13.8 ms per loop
```

Cuando n es compuesto, encontrar la solución de $x^2 \equiv a \bmod (n)$ es tan arduo como factorizar n. Sin embargo, aquellos simbolos de Jacobi, que son relativamente simple de computar, solamente dan información parcia: si $(a|n = -1)$, entonces no existe solución, ya que la existencia de un asolución implicaría $(a|p) = 1$ para todos los primos factores p de n, $(a|n) = 1$; pero si $(a|n) = +1$, no implica que a sea un cuardado módulo n cuando n es compuesto.

Sea n un entero positivo, y digamos que g es un generador de el grupo multiplicativo módulo n (asumiremos aquí que n es tal que este grupo es cíclico), y sea a coprimo a n. Por definición del hecho de que g es un generador, existe un entero x tal que $g^x = a \bmod (n)$. El *problema del logaritmo discreto* consiste en encontrar tal entero x. El método **log** proporciona una solución a este problema:

```
In [49]: p = 10^10+19; a = mod(17,p); a.log(2)

Out[49]: 6954104378

In [50]: mod(2,p)^6954104378

Out[50]: 17
```

El mejor algoritmo conocido para computar el logaritmo discreto tiene el mismo orden de complejidad, tanto para una función del tamaño de n, como para factorizar n. Sin embargo, la corriente implementación del logaritmo discreto en Sage, no es muy eficiente:

```
In [51]: p = 10^37+43; a = mod(17,p)
         %time r=a.log(2)

CPU times: user 4min 7s, sys: 1.48 s, total: 4min 8s
Wall time: 5min 30s
```

Secuencias de alícuotas

> La secuencias de alícuotas asociadas a un entero positivo n es la secuencia correcta (s_k) definida por: $S_O = n$ y $s_{k+1} = \sigma(s_k) - s_k$, donde $\sigma(s_k)$ es la suma de los divisores positivos de n, por ejemplo, s_{k+1} es la suma de los divisores *propios* de s_k, excluído el mismo s_k . La iteración se detiene cuando $s_k = 1$, tal que s_{k-1} es primo, o cuando la secuencia (s_k) entra en un ciclo. Por ejemplo, iniciando en $n = 30$, se tiene:
>
> $$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$
>
> Cuando el ciclo tiene una longitud de uno, decimos que el entero inicial es *perfecto*, como en: $6 = 1 + 2 + 3$ y $28 = 1 + 2 + 4 + 7 + 14$ son perfectos. Cuando el ciclo tiene una longitud de dos, el segundo entero en el ciclo es llamado *amigable* y forma un *par amigable* , por ejemplo: 220 y 284. Cuando el ciclo tiene longitud tres o mayor, los enteros en el ciclo es llamado *sociable*.

0.0.7 6.4 Aplicaciones

6.4.1 La Constante δ La constante δ es la generalización bi-dimensional de la constante de Euler γ. Esta es definida como sigue:

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log(n) \right)$$

Donde r_k es el radio del circulo más pequeño que los encierra en el plano afin $\mathbb{R}^2$ conteniendo al menos el punto k de $\mathbb{Z}^2$. Por ejemplo: $r_2 = 1/2$, $R_3 = r_4 = \sqrt{2/2}$, $r_5 = 1$, $r_6 = \sqrt{5/2}$, $r_7 = 5/4$, y $r_8 = r_9 = \sqrt{2}$: