cryptoguru Documentation

Release Beta

Amaury Behague, Raphaël Roblin

February 18, 2016

CHAPTER

ONE

MODULES DOCUMENTATION

1.1 itools : *useful integer fonctions*

Home : cryptoguru's documentation | genindex | modindex | search |

`itools.crt(La, Ln, N=0)`

Uses the Chinese Remainder Theorem to deduct X = La[i] mod Ln[i] for all i < len(La) = len(Ln) X is computed modulo N which is the product of the elements of Ln.

Args:

- *La (List)*: a list of integers
- *Ln (List)*: a list of integers, which size must be len(La).

Optional args:

- *N (int)*: the final modulo. Will be computed if omitted.

Returns:

- *(int)*: the result of the reconstruction modulo N.

For all i, the following statement should be true : La[i] < Ln[i]

`itools.euclide_extended(a, b, verbose=False)`

Extended version of Euclide's algorithm.

Args:

- *a (int)*: some integer
- *b (int)*: some integer

Optional args:

- *verbose (bool)*: Set to True to get a display

Returns:

- *(int,int,int)*: Three integers, the first one is the gcd, the others are Bezout coefs.

It doesn't matter if a<b, the algorithm switches a and b if necessary. However, if a < b, the first coefficient returned will be the one associated to b (gcd,coef_b,coef_a).

`itools.exp_mod(a, n, p)`

An efficient function to compute a^n mod p.

Args:

- *a (int)*: an integer

- *n (int)*: an integer
- *p (int)*: an integer > a

Returns:

- *(int)*: a^n mod p

Actually just a custom implementation of fastexp. Uses the binary decomposition of n.

`itools.`**`gcd`**`(`*a*, *b*`)`

Computes the greatest common divisor.

Args:

- *a (int)*: some integer
- *b (int)*: some integer

Returns:

- *(int)*: The GCD.

`itools.`**`get_group`**`(`*n*, *verbose=False*`)`

Generates a "safe" group for the DLP

Args:

- *n (int)*: a prime integer : security modulo / prime seed

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int,int)*: g,p with p a prime integer such that p = k*n with some small integer k. And g is a generator of Z/pZ*

`itools.`**`get_primes`**`(`*a*, *b*, *k*, *verbose=False*`)`

Computes the list of primes between two integers.

Args:

- *a (int)*: an integer
- *b (int)*: an integer such as b > a
- *k (int)*: the number of repetitions Rabin-Miller primality test.

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(List)*: the list of pseudo-prime integers in [a,b[

Why "pseudo-prime"? Because you can never be sure that they are prime with RM primality test.

`itools.`**`ilog`**`(`*x*, *b*`)`

Integer logarithm in base b.

Args:

- *x (int)*: an integer
- *b (int)*: a base

Returns:

- *(int)*: The greatest integer l such that b**l <= x.

`itools.inversion_modulaire(a, p)`

Inverts a mod p.

Args:

- *a (int)*: some integer
- *p (int)*: some integer, greater than a

Returns:

- ***(int)*: 1/a mod p if gcd(a,p) = 1** 0 if gcd(a,p) > 1

`itools.isqrt(n)`

Integer Square Root.

Args:

- *n (int)*: an integer

Returns:

- *(int)*: The greatest int s such that s*s <= n.

Uses Newton's iterative method.

`itools.rabin_miller(n, k=1, verbose=False)`

Rabin-Miller primality test.

Args:

- *n (int)*: the integer which primality you wish to test

Optional Args:

- *k (int)*: number of repetitions of the test
- *verbose (boot)*: set to True if you want a display

Returns:

- *(bool)*: True is n is a pseudo-prime, False if n isn't prime.

If this test returns False, you are 100% sure that n isn't prime. However, if this test returns True, there's a probability of 1/(2**k) that n isn't prime.

`itools.rand_prime(a, b, k, verbose=False)`

Generates a random prime number between two integers.

Args:

- *a (int)*: an integer
- *b (int)*: an integer such as b > a
- *k (int)*: the number of repetitions Rabin-Miller primality test.

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: a random pseudo-prime between a and b.

Why "pseudo-prime"? Because you can never be sure that they are prime with RM primality test.

1.2 pyfacto : *useful factoring tools*

Home : cryptoguru's documentation | genindex | modindex | search |

`pyfacto.`**`facto_fermat`**(*n*, *verbose=False*)

Fermat's factoring algorithm.

Args:

- *n (int)*: an odd integer

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: a subfactor of n.

Not very efficient.

`pyfacto.`**`lucas_mul`**(*v*, *q*, *n*)

Computes an index multiplication in a Lucas sequence.

Args:

- *v (int)*: V[m]
- *q (int)*: the multiplicator of the index
- *n (int)*: the modulo

Returns:

- *(int)*: V[mq] mod n

V is defined by [] V[i] = A*V[i-1] - V[i-2] with some A.

This function uses the following fomulas : V[2n] = V[n]*V[n] - 2 V[m+n] = V[m]*V[n] - V[m-n]

V[qm] and V[(q+1)m] are computed at the same time to solve the index addition dependency.

`pyfacto.`**`pm1_pollard`**(*n*, *B*, *nbRM=20*, *verbose=False*)

Pollard's p-1 factoring algorithm.

Args:

- *n (int)*: an integer
- *B (int)*: smooth boundary

Optional Args:

- *nbRM (int)*: number of repeats of Rabin-Miller primality test.
- *verbose (bool)*: set to True if you want a display.

Returns:

- ***(int)*: a subfactor p of n such that p-1 is B-smooth if it exists.** 0 if attack failed.

`pyfacto.`**`pm1_pollard_auto`**(*n*, *Bmax*, *verbose=False*)

Pollard's p-1 factoring algorithm with automatic Boundary adjustment.

Args:

- *n (int)*: an integer
- *Bmax (int)*: Maximum smooth boundary

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- ***(int)*: a subfactor p of n such that p-1 is B-smooth if it exists.** 0 if attack failed.

`pyfacto.pp1_williams(n, B, nbRM=20, verbose=False)`

Williams' p+1 factoring algorithm.

Args:

- *n (int)*: an integer
- *B (int)*: smooth boundary

Optional Args:

- *nbRM (int)*: number of repeats of Rabin-Miller primality test.
- *verbose (bool)*: set to True if you want a display.

Returns:

- ***(int)*: a subfactor p of n such that p+1 is B-smooth if it exists.** 0 if attack failed.

`pyfacto.pp1_williams_auto(n, Bmax, verbose=False)`

Williams' p+1 factoring algorithm with automatic Boundary adjustment.

Args:

- *n (int)*: an integer
- *Bmax (int)*: Maximum smooth boundary

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- ***(int)*: a subfactor p of n such that p+1 is B-smooth if it exists.** 0 if attack failed.

`pyfacto.rho_pollard(n, verbose=False)`

Pollard's Rho algorithm applied to factoring.

Args:

- *n (int)*: a non-prime integer

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: a small factor of n

Simple and efficient.

`pyfacto.rho_pollard_brent(n, verbose=False)`

Brent's improved version of Pollard's Rho algorithm applied to factoring. *(source : http://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf)*

Args:

- *n (int)*: a non-prime integer

Optional Args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: a small factor of n

Clearly more efficient than the standard Pollard's Rho algorithm.

`pyfacto.`**`rho_pollard_brent_p`**(*n*, *jobs=8*, *verbose=False*)

Brent's improved & parallelized version of Pollard's Rho algorithm applied to factoring. *(source : http://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf)*

Args:

- *n (int)*: a non-prime integer

Optional Args:

- *jobs (int)*: number of threads to launch. Should be your number of virtual cores.
- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: a small factor of n

Limited efficiency due to no communication between the processes : Acceleration ~ sqrt(jobs)

1.3 pwnrsa : *efficient attacks on RSA*

Home : cryptoguru's documentation | genindex | modindex | search |

`pwnrsa.`**`gen_convergents`**(*a*, *b*, *verbose=False*, *denom_only=True*)

Generates the continued fraction representation of a/b. Very similar to Euclide's extended algorithm.

Args:

- *a (int)*: some integer
- *b (int)*: another integer

Optional args:

- *verbose (bool)*: set to True to get a display.
- *denom_only (bool)*: set to True if you only need the list of denominators.

Returns:

- *(List)*: a list of tuples (integers if denom_only is set to True) representing the continued fraction.

`pwnrsa.`**`get_pq`**(*n*, *phi*, *verbose=False*)

Returns the facorisation of an RSA integer when you know its Euler totient.

Args:

- *n (int)*: a RSA integer (n = p*q with p and q two prime numbers).
- *phi (int)*: Euler's totient for n (or a guess).

Optional Args:

- *verbose (bool)*: set to True to get a display.

Returns:

- *(double,double)*: **two real numbers which are the solution of a simple second degree polynomial equation.** If those number are integers, then phi was indeed Euler's totient for n.

This function is useful to test if a given phi is a plausible one.

`pwnrsa.`**`weger1`**(*n*, *e*, *m*, *c*, *verbose=False*)

Trivial implementation of Weger's attack on RSA that uses a plain and a cipher to test potential private exponents.

Args:

- *n (int)*: the modulo
- *e (int)*: public exponent
- *m (int)*: plain
- *c (int)*: cypher (m^e % n)

Optional Args:

- *verbose (bool)*: set to True to get a display.

Returns:

- *(int)*: **the private exponent if the attack succeeded** 0 if attack failed

For this attack to work, the private exponent d must be such that : d < (n^(3/4))/abs(p-q) p and q must close to each other.

d is then the denominator of a reduced fraction of e/(n+1-2*sqrt(n)) : In this attack we assume Phi(n) ~ (n+1-2*sqrt(n)) (since we assume p ~ q ~ sqrt(n))

`pwnrsa.`**`weger2`**(*n*, *e*, *verbose=False*)

Trivial implementation of Weger's attack on RSA which computes Phi(n) to test potential private exponents.

Args:

- *n (int)*: the modulo
- *e (int)*: public exponent

Optional Args:

- *verbose (bool)*: set to True to get a display.

Returns:

- *(int)*: **the private exponent if the attack succeeded** 0 if attack failed

For this attack to work, the private exponent d must be such that : d < (n^(3/4))/abs(p-q) p and q must close to each other.

d is then the denominator of a reduced fraction of e/(n+1-2*sqrt(n)) : In this attack we assume Phi(n) ~ (n+1-2*sqrt(n)) (since we assume p ~ q ~ sqrt(n))

`pwnrsa.`**`weger_ex`**(*n*, *e*, *B*, *jobs=8*, *verbose=False*)

Parallelized version of the extended Weger attack on RSA.

Args:

- *n (int)*: the modulo
- *e (int)*: public exponent
- *B (int)*: user bound, time complexity is ~ O(B²)

Optional Args:

- *jobs (int)*: number of threads to launch. Should be your number of virtual cores (htop to visualize).
- *verbose (bool)*: set to True to get a display.

Returns:

- ***(int)*: the private exponent if the attack succeeded** 0 if attack failed

For this attack to work, n = pq must be such that : q < p < 2p p/q ~ 1 + a/b with a,b in [0,B]

d is then the denominator of a reduced fraction of e/F, where F is such that : F = n+1 - ((2+a/b)/sqrt(1+a/b))*sqrt(n) In this attack we assume Phi(n) ~ n+1 - ((2+a/b)/sqrt(1+a/b))*sqrt(n)

`pwnrsa.`**`wiener1`**(*n*, *e*, *m*, *c*, *verbose=False*)

Trivial implementation of Wiener's attack on RSA that uses a plain and a cipher to test potential private exponents.

Args:

- *n (int)*: the modulo
- *e (int)*: public exponent
- *m (int)*: plain
- *c (int)*: cypher (m^e % n)

Optional Args:

- *verbose (bool)*: set to True to get a display.

Returns:

- ***(int)*: the private exponent if the attack succeeded** 0 if attack failed

For this attack to work, the private exponent d must be such that : d < (1/3)n^(1/4)

d is then the denominator of a reduced fraction of e/n : In this attack we assume Phi(n) ~ n.

`pwnrsa.`**`wiener2`**(*n*, *e*, *verbose=False*)

Trivial implementation of Wiener's attack on RSA which computes Phi(n) to test potential private exponents.

Args:

- *n (int)*: the modulo
- *e (int)*: public exponent

Optional Args:

- *verbose (bool)*: set to True to get a display.

Returns:

- ***(int)*: the private exponent if the attack succeeded** 0 if attack failed

For this attack to work, the private exponent d must be such that : d < (1/3)n^(1/4)

d is then the denominator of a reduced fraction of e/n : In this attack we assume Phi(n) ~ n.

1.4 pwndlp : *efficient attacks on the DLP*

Home : cryptoguru's documentation | genindex | modindex | search |

`pwndlp.`**`pohlig_hellman`**(*g*, *h*, *n*, *log_file*, *verbose=False*)

Pohlig-Hellman's algorithm to solve DLP.

Args:

- *g (int)*: a generator
- *h (int)*: an integer in <g>
- *n (int)*: the modulo
- *log_file (File)*: an opened file in which results will be written

Optional args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: x such that [x mod n-1]g = h mod n

This functions factors n-1 in prime integers in order to be able to call Pollard's Rho on the subgroups and then reconstructs the result with the CRT.

`pwndlp.`**`rho_pollard_dlp`**(*g*, *h*, *p*, *n*, *verbose=False*)
Pollard's Rho applied to DLP.

Args:

- *g (int)*: a generator
- *h (int)*: an integer in <g>
- *p (int)*: the order of <g>, must be prime (else call Pohlig-Hellman first).
- *n (int)*: the modulo

Optional args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: x such that [x]g = h mod n

`pwndlp.`**`rho_pollard_dlp_adv`**(*g*, *h*, *p*, *n*, *b*, *k*, *verbose=False*)
Improved version of Pollard's Rho applied to DLP. It uses k-adding walks.

Args:

- *g (int)*: a generator
- *h (int)*: an integer in <g>
- *p (int)*: the order of <g>, must be prime (else call Pohlig-Hellman first)
- *n (int)*: the modulo
- *b (int)*: exponent bound used to generate random walks (p seems to be the best)
- *k (int)*: number of partitions (20 or more is advised)

Optional args:

- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: x such that [x]g = h mod n

`pwndlp.`**`rho_pollard_dlp_par`**(*g*, *h*, *p*, *n*, *b*, *k*, *jobs=8*, *verbose=False*)
Improved parallelized version of Pollard's Rho applied to DLP. Uses distinguished points for optimal efficiency.

Args:

- *g (int)*: a generator
- *h (int)*: an integer in <g>
- *p (int)*: the order of <g>, must be prime (else call Pohlig-Hellman first)
- *n (int)*: the modulo
- *b (int)*: exponent bound used to generate random walks (p seems to be the best)
- *k (int)*: number of partitions (20 or more is advised)

Optional args:

- *jobs (int)*: number of threads to launch. Should be your number of virtual cores.
- *verbose (bool)*: set to True if you want a display.

Returns:

- *(int)*: x such that [x]g = h mod n

Home : cryptoguru's documentation | genindex | modindex | search |

PYTHON MODULE INDEX

i

p