GNU Multiple Precision - GNU MP - GMP Cheat Sheet

by andystp via cheatography.com/54020/cs/14512/

Header

```
#include<gmp.h>
```

Data Types

mpz_t	multiple precision integer
mpq_t	multiple precision fraction
mpf_t	float is an arbitrary precision mantissa with a limited precision exponent
mp_exp_t	floating point functions accept and return exponents in this type
mp_limb_t	a limb means the part of a multi-precision number that ts in a single machine word (normally 32bit or 64bit)
mp_size_t	counts of limbs of a multi-precision number represented
mp_bitcnt_t	counts of bits of a multi-precision number
gmp_randstate_t	random state means an algorithm selection and current state data

Function Class prefixes

mpz_	Functions for signed integer arithmetic
mpq_	Functions for rational number arithmetic
mpf_	Functions for floating-point arithmetic
mpn_	Fast low-level functions that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs.

Initialization Functions

void mpz_init (mpz_t x)	Initialize x, and set its value to 0.
void mpq_init (mpq_t x)	Initialize x and set it to 0/1.
void mpf_init (mpf_t x)	Initialize x to 0.
void mp[z,q,f]_inits (mp[z,q,f] x, ...)	Initialize a NULL-terminated list of mp[z,q,f]_t variables, and set their values to 0.
void mp[z,q,f]_clear (mp[z,q,f]_t x)	Free the space occupied by x. Call this function for all mp[z,q,f]_t variables when you are done with them.
void mp[z,q,f]_clears (mp[z,q,f]_t x, ...)	Free the space occupied by a NULL-terminated list of mp[z,q,f]_t variables.

Assignment Functions

void mpz_set (mpz_t rop, const mpz_t op)	Set the value of rop from op.
void mpz_set_ui (mpz_t rop, unsigned long int op)	"
void mpz_set_si (mpz_t rop, signed long int op)	"
void mpz_set_d (mpz_t rop, double op)	truncate
void mpz_set_q (mpz_t rop, const mpq_t op)	"
void mpz_set_f (mpz_t rop, const mpf_t op)	"
void mpq_set (mpq_t rop, const mpq_t op)	Assign rop from op.

Assignment Functions (cont)

void mpq_set_z (mpq_t rop, const mpz_t op)	"
void mpq_set_ui (mpq_t rop, unsigned long int op1, unsigned long int op2)	Set the value of rop to op1/op2. Note that if op1 and op2 have common factors, rop has to be passed to mpq_canonicalize before any operations are performed on rop.
void mpq_set_si (mpq_t rop, signed long int op1, unsigned long int op2)	"
void mpf_set (mpf_t rop, const mpf_t op)	Set the value of rop from op.
void mpf_set_ui (mpf_t rop, unsigned long int op)	"
void mpf_set_si (mpf_t rop, signed long int op)	"
void mpf_set_d (mpf_t rop, double op)	"
void mpf_set_z (mpf_t rop, const mpz_t op)	"
void mpf_set_q (mpf_t rop, const mpq_t op)	"
int mpz_set_str (mpz_t rop, const char *str, int base)	Set the value of rop from str, a null-terminated C string in base base. White space is allowed in the string, and is simply ignored. Should return 0, if error returns -1.
int mpq_set_str (mpq_t rop, const char *str, int base)	Set rop from a null-terminated string str in the given base. Should return 0, if error returns -1.
int mpf_set_str (mpf_t rop, const char *str, int base)	et the value of rop from the string in str. Should return 0, if error returns -1.
void mpz_swap (mpz_t rop1, mpz_t rop2)	Swap the values rop1 and rop2 efficiently.
void mpq_swap (mpq_t rop1, mpq_t rop2)	Swap the values rop1 and rop2 efficiently.
void mpf_swap (mpf_t rop1, mpf_t rop2)	Swap rop1 and rop2 efficiently. Both the values and the precisions of the two variables are swapped.

Conversion Functions

unsigned long int mpz_get_ui (const mpz_t op)	Return the value of op as an unsigned long.
signed long int mpz_get_si (const mpz_t op)	If op fits into a signed long int return the value of op. Otherwise return the least significant part of op, with the same sign as op.
double mpz_get_d (const mpz_t op)	Convert op to a double, truncating if necessary (i.e. rounding towards zero).
double mpz_get_d_2exp (signed long int *exp, const mpz_t op)	Convert op to a double, truncating if necessary (i.e. rounding towards zero), and returning the exponent separately.
char *mpz_get_str (char* str, int base, const mpz_t op)	Convert op to a string of digits in base base. The base argument may vary from 2 to 62 or from -2 to -36.

Conversion Functions (cont)	
double mpq_get_d (const mpq_t op)	Convert op to a double, truncating if necessary (i.e. rounding towards zero).
void mpq_set_d (mpq_t rop, double op)	Set rop to the value of op. There is no rounding, this conversion is exact.
void mpq_set_f (mpq_t rop, const mpf_t op)	Set rop to the value of op. There is no rounding, this conversion is exact.
char *mpq_get_str (char* str, int base, const mpq_t op)	Convert op to a string of digits in base base. The base may vary from 2 to 36. The string will be of the form 'num/den', or if the denominator is 1 then just 'num'.
double mpf_get_d (const mpf_t op)	Convert op to a double, truncating if necessary (i.e. rounding towards zero).
double mpf_get_d_2exp (signed long int *exp, const mpf_t op)	Convert op to a double, truncating if necessary (i.e. rounding towards zero), and with an exponent returned separately.
long mpf_get_si (const mpf_t op)	Convert op to a long or unsigned long, truncating any fraction part. If op is too big for the return type, the result is undefined.
unsigned long mpf_get_ui (const mpf_t op)	Convert op to a long or unsigned long, truncating any fraction part. If op is too big for the return type, the result is undefined.
char *mpf_get_str (char* str, mp_exp_t *expptr, int base, size_t n_digits, const mpf_t op)	Convert op to a string of digits in base base. The base argument may vary from 2 to 62 or from -2 to -36. Up to n_digits digits will be generated. Trailing zeros are not returned. No more digits than can be accurately represented by op are ever generated. If n_digits is 0 then that accurate maximum number of digits are generated.

Arithmetic Functions	
void mpz_add (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to op1 + op2.
void mpz_add_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Set rop to op1 + op2.
void mpz_sub (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to op1 - op2.
void mpz_sub_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Set rop to op1 - op2.
void mpz_ui_sub (mpz_t rop, unsigned long int op1, const mpz_t op2)	Set rop to op1 - op2.
void mpz_mul (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to op1 times op2.

Arithmetic Functions (cont)	
void mpz_mul_si (mpz_t rop, const mpz_t op1, long int op2)	Set rop to op1 times op2.
void mpz_mul_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Set rop to op1 times op2.
void mpz_addmul (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to rop + op1 times op2.
void mpz_addmul_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Set rop to rop + op1 times op2.
void mpz_submul (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to rop - op1 times op2.
void mpz_submul_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Set rop to rop - op1 times op2.
void mpz_mul_2exp (mpz_t rop, const mpz_t op1, mp_bitcnt_t op2)	Set rop to op1 times 2 raised to op2. This operation can also be defined as a left shift by op2 bits.
void mpz_neg (mpz_t rop, const mpz_t op)	Set rop to -op.
void mpz_abs (mpz_t rop, const mpz_t op)	Set rop to the absolute value of op.
void mpq_add (mpq_t sum, const mpq_t addend1, const mpq_t addend2)	Set sum to addend1 + addend2.
void mpq_sub (mpq_t difference, const mpq_t minuend, const mpq_t subtrahend)	Set difference to minuend - subtrahend.
void mpq_mul (mpq_t product, const mpq_t multiplier, const mpq_t multiplicand)	Set product to multiplier times multiplicand.
void mpq_mul_2exp (mpq_t rop, const mpq_t op1, mp_bitcnt_t op2)	Set rop to op1 times 2 raised to op2. This operation can also be defined as a left shift by op2 bits.
void mpq_div (mpq_t quotient, const mpq_t dividend, const mpq_t divisor)	Set quotient to dividend/divisor.
void mpq_div_2exp (mpq_t rop, const mpq_t op1, mp_bitcnt_t op2)	Set rop to op1 divided by 2 raised to op2.
void mpq_neg (mpq_t negated_operand, const mpq_t operand)	Set negated_operand to -operand.
void mpq_abs (mpq_t rop, const mpq_t op)	Set rop to the absolute value of op.
void mpq_inv (mpq_t inverted_number, const mpq_t number)	Set inverted_number to 1/number. If the new denominator is zero, this routine will divide by zero.
void mpf_add (mpf_t rop, const mpf_t op1, const mpf_t op2)	Set rop to op1 + op2.
void mpf_add_ui (mpf_t rop, const mpf_t op1, unsigned long int op2)	Set rop to op1 + op2.

Arithmetic Functions (cont)	
void mpf_sub (mpf_t rop, const mpf_t op1, const mpf_t op2)	Set rop to op1 - op2.
void mpf_ui_sub (mpf_t rop, unsigned long int op1, const mpf_t op2)	Set rop to op1 - op2.
void mpf_sub_ui (mpf_t rop, const mpf_t op1, unsigned long int op2)	Set rop to op1 - op2.
void mpf_mul (mpf_t rop, const mpf_t op1, const mpf_t op2)	Set rop to op1 times op2.
void mpf_mul_ui (mpf_t rop, const mpf_t op1, unsigned long int op2)	Set rop to op1 times op2.
void mpf_div (mpf_t rop, const mpf_t op1, const mpf_t op2)	Set rop to op1/op2.
void mpf_ui_div (mpf_t rop, unsigned long int op1, const mpf_t op2)	Set rop to op1/op2.
void mpf_div_ui (mpf_t rop, const mpf_t op1, unsigned long int op2)	Set rop to op1/op2.
void mpf_sqrt (mpf_t rop, const mpf_t op)	Set rop to the square root of op.
void mpf_sqrt_ui (mpf_t rop, unsigned long int op)	Set rop to the square root of op.
void mpf_pow_ui (mpf_t rop, const mpf_t op1, unsigned long int op2)	Set rop to op1 raised to the power op2.
void mpf_neg (mpf_t rop, const mpf_t op)	Set rop to -op.
void mpf_abs (mpf_t rop, const mpf_t op)	Set rop to the absolute value of op.
void mpf_mul_2exp (mpf_t rop, const mpf_t op1, mp_bitcnt_t op2)	Set rop to op1 times 2 raised to op2. This operation can also be defined as a left shift by op2 bits.
void mpf_div_2exp (mpf_t rop, const mpf_t op1, mp_bitcnt_t op2)	Set rop to op1 divided by 2 raised to op2.

Integer Division Functions
void mpz_cdiv_q (mpz_t q, const mpz_t n, const mpz_t d)
void mpz_cdiv_r (mpz_t r, const mpz_t n, const mpz_t d)
void mpz_cdiv_qr (mpz_t q, mpz_t r, const mpz_t n, const mpz_t d)
unsigned long int mpz_cdiv_q_ui (mpz_t q, const mpz_t n, unsigned long int d)
unsigned long int mpz_cdiv_r_ui (mpz_t r, const mpz_t n, unsigned long int d)
unsigned long int mpz_cdiv_qr_ui (mpz_t q, mpz_t r, const mpz_t n, unsigned long int d)
unsigned long int mpz_cdiv_ui (const mpz_t n, unsigned long int d)
void mpz_cdiv_q_2exp (mpz_t q, const mpz_t n, mp_bitcnt_t b)
void mpz_cdiv_r_2exp (mpz_t r, const mpz_t n, mp_bitcnt_t b)
void mpz_fdiv_q (mpz_t q, const mpz_t n, const mpz_t d)
void mpz_fdiv_r (mpz_t r, const mpz_t n, const mpz_t d)
void mpz_fdiv_qr (mpz_t q, mpz_t r, const mpz_t n, const mpz_t d)

Integer Division Functions (cont)

unsigned long int mpz_fdiv_q_ui (mpz_t q, const mpz_t n, unsigned long int d)	
unsigned long int mpz_fdiv_r_ui (mpz_t r, const mpz_t n, unsigned long int d)	
unsigned long int mpz_fdiv_qr_ui (mpz_t q, mpz_t r, const mpz_t n, unsigned long int d)	
unsigned long int mpz_fdiv_ui (const mpz_t n, unsigned long int d)	
void mpz_fdiv_q_2exp (mpz_t q, const mpz_t n, mp_bitcnt_t b)	
void mpz_fdiv_r_2exp (mpz_t r, const mpz_t n, mp_bitcnt_t b)	
void mpz_tdiv_q (mpz_t q, const mpz_t n, const mpz_t d)	
void mpz_tdiv_r (mpz_t r, const mpz_t n, const mpz_t d)	
void mpz_tdiv_qr (mpz_t q, mpz_t r, const mpz_t n, const mpz_t d)	
unsigned long int mpz_tdiv_q_ui (mpz_t q, const mpz_t n, unsigned long int d)	
unsigned long int mpz_tdiv_r_ui (mpz_t r, const mpz_t n, unsigned long int d)	
unsigned long int mpz_tdiv_qr_ui (mpz_t q, mpz_t r, const mpz_t n, unsigned long int d)	
unsigned long int mpz_tdiv_ui (const mpz_t n, unsigned long int d)	
void mpz_tdiv_q_2exp (mpz_t q, const mpz_t n, mp_bitcnt_t b)	
void mpz_tdiv_r_2exp (mpz_t r, const mpz_t n, mp_bitcnt_t b)	
Divide n by d, forming a quotient q and/or remainder r. For the 2exp functions, d=2^b. The rounding is in three styles, each suiting different applications.	cdiv rounds q up towards +infinity, and r will have the opposite sign to d. The c stands for “ceil”.
	fdiv rounds q down towards -infinity, and r will have the same sign as d. The f stands for “floor”.
	tdiv rounds q towards zero, and r will have the same sign as n. The t stands for “truncate”.
In all cases q and r will satisfy n=q*d+r, and r will satisfy 0<=abs(r)<abs(d).	
The q functions calculate only the quotient, the r functions only the remainder, and the qr functions calculate both. Note that for qr the same variable cannot be passed for both q and r, or results will be unpredictable.	
For the ui variants the return value is the remainder, and in fact returning the remainder is all the div_ui functions do. For tdiv and cdiv the remainder can be negative, so for those the return value is the absolute value of the remainder.	
For the 2exp variants the divisor is 2^b. These functions are implemented as right shifts and bit masks, but of course they round the same as the other functions.	
For positive n both mpz_fdiv_q_2exp and mpz_tdiv_q_2exp are simple bitwise right shifts. For negative n, mpz_fdiv_q_2exp is effectively an arithmetic right shift treating n as twos complement the same as the bitwise logical functions do, whereas mpz_tdiv_q_2exp effectively treats n as sign and magnitude.	
void mpz_mod (mpz_t r, const mpz_t n, const mpz_t d)	Set r to n mod d. The sign of the divisor is ignored; the result is always non-negative.
unsigned long int mpz_mod_ui (mpz_t r, const mpz_t n, unsigned long int d)	Set r to n mod d. The sign of the divisor is ignored; the result is always non-negative.

Integer Division Functions (cont)	
	mpz_mod_ui is identical to mpz_fdiv_r_ui above, returning the remainder as well as setting r. See mpz_fdiv_ui above if only the return value is wanted.
void mpz_divexact (mpz_t q, const mpz_t n, const mpz_t d)	Set q to n/d. These functions produce correct results only when it is known in advance that d divides n.
void mpz_divexact_ui (mpz_t q, const mpz_t n, unsigned long d)	Set q to n/d. These functions produce correct results only when it is known in advance that d divides n.
	These routines are much faster than the other division functions, and are the best choice when exact division is known to occur, for example reducing a rational to lowest terms.
int mpz_divisible_p (const mpz_t n, const mpz_t d)	Return non-zero if n is exactly divisible by d, or in the case of mpz_divisible_2exp_p by 2^b.
int mpz_divisible_ui_p (const mpz_t n, unsigned long int d)	Return non-zero if n is exactly divisible by d, or in the case of mpz_divisible_2exp_p by 2^b.
int mpz_divisible_2exp_p (const mpz_t n, mp_bitcnt_t b)	Return non-zero if n is exactly divisible by d, or in the case of mpz_divisible_2exp_p by 2^b.
	n is divisible by d if there exists an integer q satisfying n = q*d. Unlike the other division functions, d=0 is accepted and following the rule it can be seen that only 0 is considered divisible by 0.
int mpz_congruent_p (const mpz_t n, const mpz_t c, const mpz_t d)	Return non-zero if n is congruent to c modulo d, or in the case of mpz_congruent_2exp_p modulo 2^b.
int mpz_congruent_ui_p (const mpz_t n, unsigned long int c, unsigned long int d)	Return non-zero if n is congruent to c modulo d, or in the case of mpz_congruent_2exp_p modulo 2^b.
int mpz_congruent_2exp_p (const mpz_t n, const mpz_t c, mp_bitcnt_t b)	Return non-zero if n is congruent to c modulo d, or in the case of mpz_congruent_2exp_p modulo 2^b.
	n is congruent to c mod d if there exists an integer q satisfying n = c + q*d. Unlike the other division functions, d=0 is accepted and following the rule it can be seen that n and c are considered congruent mod 0 only when exactly equal.

Integer Exponentiation Functions

void mpz_powm (mpz_t rop, const mpz_t base, const mpz_t exp, const mpz_t mod)	Set rop to (base raised to exp) modulo mod.
void mpz_powm_ui (mpz_t rop, const mpz_t base, unsigned long int exp, const mpz_t mod)	Set rop to (base raised to exp) modulo mod.
	Negative exp is supported if an inverse base^-1 mod mod exists (see mpz_invert in Number Theoretic Functions). If an inverse doesn't exist then a divide by zero is raised.
void mpz_powm_sec (mpz_t rop, const mpz_t base, const mpz_t exp, const mpz_t mod)	Set rop to (base raised to exp) modulo mod.
	It is required that exp > 0 and that mod is odd.
	This function is designed to take the same time and have the same cache access patterns for any two same-size arguments, assuming that function arguments are placed at the same position and that the machine state is identical upon function entry. This function is intended for cryptographic purposes, where resilience to side-c-hannel attacks is desired.
void mpz_pow_ui (mpz_t rop, const mpz_t base, unsigned long int exp)	Set rop to base raised to exp. The case 0^0 yields 1.
void mpz_ui_pow_ui (mpz_t rop, unsigned long int base, unsigned long int exp)	Set rop to base raised to exp. The case 0^0 yields 1.

Integer Root Extraction Functions

int mpz_root (mpz_t rop, const mpz_t op, unsigned long int n)	Set rop to the truncated integer part of the nth root of op. Return non-zero if the computation was exact, i.e., if op is rop to the nth power.
void mpz_rootrem (mpz_t root, mpz_t rem, const mpz_t u, unsigned long int n)	Set root to the truncated integer part of the nth root of u. Set rem to the remainder, u-root**n.
void mpz_sqrt (mpz_t rop, const mpz_t op)	Set rop to the truncated integer part of the square root of op.
void mpz_sqrtrem (mpz_t rop1, mpz_t rop2, const mpz_t op)	Set rop1 to the truncated integer part of the square root of op, like mpz_sqrt. Set rop2 to the remainder op-rop1*rop1, which will be zero if op is a perfect square.
	If rop1 and rop2 are the same variable, the results are undefined.
int mpz_perfect_power_p (const mpz_t op)	Return non-zero if op is a perfect power, i.e., if there exist integers a and b, with b>1, such that op equals a raised to the power b.

Integer Root Extraction Functions (cont)	
	Under this definition both 0 and 1 are considered to be perfect powers. Negative values of op are accepted, but of course can only be odd perfect powers.
int mpz_perfect_square_p (const mpz_t op)	Return non-zero if op is a perfect square, i.e., if the square root of op is an integer. Under this definition both 0 and 1 are considered to be perfect squares.

Integer Number Theoretic Functions	
int mpz_probab_prime_p (const mpz_t n, int reps)	Determine whether n is prime. Return 2 if n is definitely prime, return 1 if n is probably prime (without being certain), or return 0 if n is definitely non-prime.
	This function performs some trial divisions, then reps Miller-Rabin probabilistic primality tests. A higher reps value will reduce the chances of a non-prime being identified as “probably prime”. A composite number will be identified as a prime with a probability of less than 4^(-reps). Reasonable values of reps are between 15 and 50.
void mpz_nextprime (mpz_t rop, const mpz_t op)	Set rop to the next prime greater than op.
	This function uses a probabilistic algorithm to identify primes. For practical purposes it’s adequate, the chance of a composite passing will be extremely small.
void mpz_gcd (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to the greatest common divisor of op1 and op2. The result is always positive even if one or both input operands are negative. Except if both inputs are zero; then this function defines gcd(0,0) = 0.
unsigned long int mpz_gcd_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)	Compute the greatest common divisor of op1 and op2. If rop is not NULL, store the result there.
	If the result is small enough to fit in an unsigned long int, it is returned. If the result does not fit, 0 is returned, and the result is equal to the argument op1. Note that the result will always fit if op2 is non-zero.

Integer Number Theoretic Functions (cont)

void mpz_gcdext (mpz_t g, mpz_t s, mpz_t t, const mpz_t a, const mpz_t b)	Set g to the greatest common divisor of a and b, and in addition set s and t to coefficients satisfying *as + bt* = g. The value in g is always positive, even if one or both of a and b are negative (or zero if both inputs are zero). The values in s and t are chosen such that normally, abs(s) < abs(b) / (2 g) and abs(t) < abs(a) / (2 g), and these relations define s and t uniquely. There are a few exceptional cases:
	If abs(a) = abs(b), then s = 0, t = sgn(b).
	Otherwise, s = sgn(a) if b = 0 or abs(b) = 2 g, and t = sgn(b) if a = 0 or abs(a) = 2 g.
	In all cases, s = 0 if and only if g = abs(b), i.e., if b divides a or a = b = 0.
	If t is NULL then that value is not computed.
void mpz_lcm (mpz_t rop, const mpz_t op1, const mpz_t op2)	Set rop to the least common multiple of op1 and op2. rop is always positive, irrespective of the signs of op1 and op2. rop will be zero if either op1 or op2 is zero.
void mpz_lcm_ui (mpz_t rop, const mpz_t op1, unsigned long op2)	Set rop to the least common multiple of op1 and op2. rop is always positive, irrespective of the signs of op1 and op2. rop will be zero if either op1 or op2 is zero.
int mpz_invert (mpz_t rop, const mpz_t op1, const mpz_t op2)	Compute the inverse of op1 modulo op2 and put the result in rop. If the inverse exists, the return value is non-zero and rop will satisfy 0 <= rop < abs(op2) (with rop = 0 possible only when abs(op2) = 1, i.e., in the somewhat degenerate zero ring). If an inverse doesn't exist the return value is zero and rop is undefined. The behaviour of this function is undefined when op2 is zero.
int mpz_jacobi (const mpz_t a, const mpz_t b)	Calculate the Jacobi symbol (a/b). This is defined only for b odd.
int mpz_legendre (const mpz_t a, const mpz_t p)	Calculate the Legendre symbol (a/p). This is defined only for p an odd positive prime, and for such p it's identical to the Jacobi symbol.
int mpz_kronecker (const mpz_t a, const mpz_t b)	Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.
int mpz_krone-cker_si (const mpz_t a, long b)	Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.
int mpz_krone-cker_ui (const mpz_t a, unsigned long b)	Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.

Integer Number Theoretic Functions (cont)	
int mpz_si_kronecker (long a, const mpz_t b)	Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.
int mpz_ui_kronecker (unsigned long a, const mpz_t b)	Calculate the Jacobi symbol (a/b) with the Kronecker extension (a/2)=(2/a) when a odd, or (a/2)=0 when a even.
mp_bitcnt_t mpz_remove (mpz_t rop, const mpz_t op, const mpz_t f)	Remove all occurrences of the factor f from op and store the result in rop. The return value is how many such occurrences were removed.
void mpz_fac_ui (mpz_t rop, unsigned long int n)	Set rop to the factorial of n: mpz_fac_ui computes the plain factorial n!, mpz_2fac_ui computes the double-factorial n!!, and mpz_mfac_uiui the m-multi-factorial n!^(m).
void mpz_2fac_ui (mpz_t rop, unsigned long int n)	Set rop to the factorial of n: mpz_fac_ui computes the plain factorial n!, mpz_2fac_ui computes the double-factorial n!!, and mpz_mfac_uiui the m-multi-factorial n!^(m).
void mpz_mfac_uiui (mpz_t rop, unsigned long int n, unsigned long int m)	Set rop to the factorial of n: mpz_fac_ui computes the plain factorial n!, mpz_2fac_ui computes the double-factorial n!!, and mpz_mfac_uiui the m-multi-factorial n!^(m).
void mpz_primorial_ui (mpz_t rop, unsigned long int n)	Set rop to the primorial of n, i.e. the product of all positive prime numbers <=n.
void mpz_bin_ui (mpz_t rop, const mpz_t n, unsigned long int k)	Compute the binomial coefficient n over k and store the result in rop. Negative values of n are supported by mpz_bin_ui, using the identity bin(-n,k) = (-1)^k * bin(n+k-1,k), see Knuth volume 1 section 1.2.6 part G.
void mpz_bin_uiui (mpz_t rop, unsigned long int n, unsigned long int k)	Compute the binomial coefficient n over k and store the result in rop. Negative values of n are supported by mpz_bin_ui, using the identity bin(-n,k) = (-1)^k * bin(n+k-1,k), see Knuth volume 1 section 1.2.6 part G.
void mpz_fib_ui (mpz_t fn, unsigned long int n)	mpz_fib_ui sets fn to to F[n], the n'th Fibonacci number. mpz_fib2_ui sets fn to F[n], and fnsub1 to F[n-1].
void mpz_fib2_ui (mpz_t fn, mpz_t fnsub1, unsigned long int n)	mpz_fib_ui sets fn to to F[n], the n'th Fibonacci number. mpz_fib2_ui sets fn to F[n], and fnsub1 to F[n-1].

Integer Number Theoretic Functions (cont)

	These functions are designed for calculating isolated Fibonacci numbers. When a sequence of values is wanted it's best to start with mpz_fib2_ui and iterate the defining F[n+1]=F[n]+F[n-1] or similar.
void mpz_lucnum_ui (mpz_t ln, unsigned long int n)	mpz_lucnum_ui sets ln to to L[n], the n'th Lucas number. mpz_lucnum2_ui sets ln to L[n], and lnsub1 to L[n-1].
void mpz_lucnum2_ui (mpz_t ln, mpz_t lnsub1, unsigned long int n)	mpz_lucnum_ui sets ln to to L[n], the n'th Lucas number. mpz_lucnum2_ui sets ln to L[n], and lnsub1 to L[n-1].
	These functions are designed for calculating isolated Lucas numbers. When a sequence of values is wanted it's best to start with mpz_lucnum2_ui and iterate the defining L[n+1]=L[n]+L[n-1] or similar.
	The Fibonacci numbers and Lucas numbers are related sequences, so it's never necessary to call both mpz_fib2_ui and mpz_lucnum2_ui. The formulas for going from Fibonacci to Lucas can be found in Lucas Numbers Algorithm, the reverse is straightforward too.