Aritmetika celých čísel a rozšírení polí/okruhov

Obsah

1 Moduly na prácu s celými číslami a rozšírením polí/okruhov

Na prácu s celými číslami je možné použiť nasledovné moduly:

- `ZZ`: veľké celé čísla
- `ZZ_p`: veľké celé čísla modulo p
- `zz_p`: celé čísla mod p s jednoduchou presnosťou (http://en.wikipedia.org/wiki/Single_precision)
- `GF2`: celé čísla mod 2.

Na prácu s rozšírením polí/okruhov je možné použiť moduly:

- `ZZ_pE`: rozšírenie poľa/okruhu ZZ_p
- `zz_pE`: rozšírenie poľa/okruhu zz_p
- `GF2E`: rozšírenie poľa/okruhu GF2.

- **Trieda *ZZ*** je vhodná na reprezentáciu celých čísel ľubovolnej dĺžky so znamienkom. Obsahuje funkcie pre všetky základné aritmetické operácie, ako aj pre niektoré rozšírené aritmetické operácie napr. testovanie prvočíselnosti, generovanie malých prvočísel, rýchla modulárna aritmetika na číslach s jednoduchou presnosťou. Miesto v pamäti je automaticky spravované konštruktormi a deštruktormi.
- **Trieda *ZZ_p*** je vhodná na reprezentáciu celých čísel modulo p. Modulus p môže byť kladné celé číslo, **nemusí byť prvočíslo**. Objekty triedy *ZZ_p* sú reprezentované ako *ZZ* v rozsahu 0 ... p-1. Pred vytvorením *ZZ_p* objektov je potrebné inicializovať modulus p pomocou príkazu *ZZ_p::init(p)*. Vždy môže byť **inicializované len jedno *ZZ_p***. Zmena modulu je vysvetlená v texte nižšie.
- **Trieda *zz_p*** je vhodná na reprezentáciu celých čísel modulo p, kde $1 \leq p \leq NTL\_SP\_BOUND$. NTL_SP_BOUND je väčšinou 2^{30} na 32 bitových počítačoch a 2^{50} na 64 bitových počítačoch. Modulus p môže byť kladné celé číslo, **nemusí byť prvočíslo**. Objekty triedy *zz_p* sú reprezentované ako *long* v rozsahu 0 ... p-1. Pred vytvorením *ZZ_p* objektov je potrebné inicializovať modulus p pomocou príkazu *zz_p::init(p)*. Vždy môže byť **inicializované len jedno *ZZ_p***. Zmena modulu je vysvetlená v texte nižšie.
- **Trieda *GF2*** reprezentuje pole GF(2). Trieda slúži hlavne pre vybudovanie jednotných rozhraní pre triedy konečných polí.
- **Trieda *ZZ_pE/zz_pE*** je vhodná na reprezentáciu polynómov v *Z_p[X]* modulo polynóm P. Modulus P môže byť reprezentovaný polynómom so stupňom deg(P) > 0, **nemusí byť ireducibilný**. Modulus p, ktorý

definuje Z_p, nemusí byť prvočíslo. Objekty triedy *ZZ_pE/zz_pE* sú reprezentované ako *ZZ_pX/zz_pX* stupňa $<$ deg(P). Pred vytvorením *ZZ_pE/zz_pE* objektov je potrebné inicializovať modulus P pomocou príkazu *ZZ_pE::init(P)*. Zmena modulu je vysvetlená v texte nižšie.

- **Trieda *GF2E*** je vhodná na reprezentáciu polynómov v *F_2[X]* modulo polynóm P. Modulus P môže byť reprezentovaný polynómom so stupňom deg(P) > 0, **nemusí byť ireducibilný**. Objekty triedy *GF2E* sú reprezentované ako *GF2X* stupňa $<$ deg(P). Pred vytvorením *GF2E* objektov je potrebné inicializovať modulus P pomocou príkazu *GF2E::init(P)*. Zmena modulu je vysvetlená v texte nižšie.
  Poznámka: ak je P reprezentovaný ako trojčlen $X^n + X^k + 1$ alebo päťčlen $X^n + X^{k3} + X^{k2} + X^{k1} + 1$, alebo formou $X^n + g$, kde g je nízkeho stupňa, výkon bude zlepšený. Takéto polynómy môžme zostrojiť pomocou funkcií BuildSparseIrred a BuildIrred v triede *GF2XFactoring*.

1.1 Upozornenie

V texte sa na viacerých miestach uvádza, že dané funkcie platia pre všetky triedy. Znamená to, že sa dajú použiť pre triedy *ZZ*, *ZZ_p*, *zz_p*, *GF2*, *ZZ_pE*, *zz_pE* a *GF2E*, ak nie je uvedené inak.

1.2 #include

Pre triedy používané v tomto dokumente je potrebné do programu zahrnúť nasledovné hlavičkové súbory:

```
#include <NTL/tools.h>
#include <NTL/ZZ.h>
#include <NTL/ZZ_p.h>
#include <NTL/zz_p.h>
#include <NTL/GF2.h>
#include <NTL/ZZ_pE.h>
#include <NTL/zz_pE.h>
#include <NTL/GF2E.h>

NTL_CLIENT
```

2 Aritmetické operácie

2.1 Porovnanie

Knižnica NTL poskytuje na porovnananie dvoch objektov popisovaných tried dva klasické operátory == (**rovnosť**), ! = (**nerovnosť**). Ďalej existujú funkcie *IsZero*, a *IsOne*. Obe fungujú pre všetky dátové typy.

```
long IsZero(const ZZ& a);
long IsOne(const ZZ& a);
```

Funkcia IsZero vracia 1, ak má objekt hodnotu 0, inak vráti 0.
Funkcia IsOne vracia 1, ak má objekt hodnotu 1, inak vráti 0.

Trieda *ZZ* ďalej poskytuje operátory porovnania <, >, <= a >= a funkcie

```
long sign(const ZZ& a);
long compare(const ZZ& a, const ZZ& b);
```

Funkcia **sign** vracia hodnotu -1, ak má objekt záporné znamienko, hodnotu 0, ak má objekt hodnotu 0 a hodnotu 1, ak má objekt kladné znamienko. Funkcia **compare** vracia rovnaké hodnoty ako funkcia sign, ale testuje znamienko rozdielu (a-b).

Promotions(povýšenia) Pravidlá povýšenia určujú ako je možné konvertovať dátové typy bez straty údajov. Nižšie uvedené operátory pretypujú uvedené dátové typy automaticky. Ak potrebujeme pretypovať dátový typ explicitne, na konverziu použijeme funkciu *to_ T*, kde T predstavuje návratový typ.

Operátory ==, ! = povyšujú údajové typy v tabuľke:
Operátory porovnania a funkcia compare podporuje povýšenie z *long* na *ZZ*.

cieľ:	zdroj
ZZ_p, zz_p, GF2	long
ZZ_pE	long, ZZ_p
zz_pE	long, zz_p
GF2E	long, GF2

Uvedené platí len pre objekty *a* a *b*.

2.2 Sčítanie a odčítanie

Na sčítanie a odčítanie objektov slúžia klasické operátory $+, -, + =, - =$ $, ++, --$ ako aj unárny operátor $-$, ktorý zmení znamienko objektu. Operátory $++, --$ fungujú aj ako prefix a postfix. V procedurálnej forme existujú funkcie *add, sub, negate*. Tieto funkcie sa dajú použiť pre všetky dátové typy.

```
void add(ZZ& x, const ZZ& a, const ZZ& b); // x = a + b
```

Do premennej *ZZ x* priradí súčet *ZZ a* + *ZZ b*.

```
void sub(ZZ& x, const ZZ& a, const ZZ& b); // x = a - b
```

Do premennej *ZZ x* priradí rozdiel *ZZ a* - *ZZ b*.

```
void negate(ZZ& x, const ZZ& a); // x = -a
```

Do premennej *ZZ x* priradí hodnotu *ZZ a*, ale s opačným znamienkom.

Trieda *ZZ* obsahuje naviac funkcie *SubPos* a *abs*.

```
void SubPos(ZZ& x, const ZZ& a, const ZZ& b); // x = a-b
void abs(ZZ& x, const ZZ& a); // x = |a|
ZZ abs(const ZZ& a);
```

Funkcia *SubPos* priradí premennej *ZZ x* rozdiel *ZZ a* - *ZZ b*, ale predpokladá $a \geq b \geq 0$. Ak neplatí $a \geq b$, program vyhodí **chybu** (Internal error: can't free this _ntl_verylong) a ukončí sa. Ak neplatí $a, b \geq 0$, program vyhodí **chybu** (negative size allocation in _ntl_zsetlength) a ukončí sa.
Funkcia *abs* priradí do *ZZ x* absolútnu hodnotu *ZZ a*.

Promotions(povýšenia) Pravidlá povýšenia určujú ako je možné konvertovať dátové typy bez straty údajov. Nižšie uvedené operátory pretypujú uvedené dátové typy automaticky. Ak potrebujeme pretypovať dátový typ explicitne, na konverziu použijeme funkciu *to_ T*, kde T predstavuje návratový typ.

Binárne operátory $+, -$ a funkcie *add, sub* povýšia *long* na *ZZ, ZZ_p, zz_p, GF2*.
Operátory $+, -$ a funkcie *add, sub* povýšia údajové typy v tabuľke:
Uvedené platí len pre objekty *a* a *b*.

cieľ:	zdroj
ZZ_pE	long, ZZ_p
zz_pE	long, zz_p
GF2E	long, GF2

2.3 Násobenie

Na násobenie sa používajú operátory $*, *=$. Na **násobenie** a **umocňovanie** sú dostupné funkcie *mul* a *sqr*. Funkcie aj operátory sú dostupné vo všetkých triedach.

```
void mul(ZZ& x, const ZZ& a, const ZZ& b); // x = a * b
```

Do premennej *ZZ x*, priradí výsledok násobenia premenných *ZZ a * ZZ b*.

```
void sqr(ZZ& x, const ZZ& a); // x = a^2
ZZ sqr(const ZZ& a);
```

Do premennej *ZZ x* priradí druhú mocninu *ZZ* a^2.

Promotions(povýšenia) Pravidlá povýšenia určujú ako je možné konvertovať dátové typy bez straty údajov. Nižšie uvedené operátory pretypujú uvedené dátové typy automaticky. Ak potrebujeme pretypovať dátový typ explicitne, na konverziu použijeme funkciu *to_ T*, kde T predstavuje návratový typ.

Operátor $*$ a funkcia *mul* podporujú povýšenie údajových typov v tabuľke:

Uvedené platí len pre objekty *a* a *b*.

cieľ:	zdroj
ZZ	long
ZZ_p	long
zz_p	long
GF2	long
ZZ_pE	long, ZZ_p
zz_pE	long, zz_p
GF2E	long, GF2

2.4 Delenie

Na delenie je možné použiť operátory $/, /=$ a funkcie *div* a *inv*, pričom fungujú vo všetkých triedach s výnimkov ZZ, ktorá má iné základné funkcie.

```
void div(ZZ_p& x, const ZZ_p& a, const ZZ_p& b);
```

Funkcia *div* priradí do premennej x podiel a/b.

```
void inv(ZZ_p& x, const ZZ_p& a); // x = 1/a
ZZ_p inv(const ZZ_p& a);
```

Funkcia *inv* priradí do premennej x hodnotu $1/a$.

Ak delitele nemajú inverzný prvok, program vyhodí **chybu** (ZZ_p: division by non-invertible element). Toto sa dá vyriešiť **definovaním chybového handlera**

```
void H(const ZZ_p& b)
```

a nastavením

```
ZZ_p::DivHandler = H
```

Potom, ak $b! = 0$ a b nemá inverzný prvok, funkcia H je vyvolaná s argumentom b. Ak sa to stane, p nebude prvočíslo a GCD(p,rep(b)) je netriviálny. Funkcia H môže mať napr. tvar:

```
void H (const ZZ_p& b)
{
  ZZ p;
  ntl_found = 2;
  p = GCD(rep(b), ZZ_p::modulus());
  gmp_of_ntl (factor_found, p);
}
```

Promotions(povýšenia) Pravidlá povýšenia určujú ako je možné konvertovať dátové typy bez straty údajov. Nižšie uvedené operátory pretypujú uvedené dátové typy automaticky. Ak potrebujeme pretypovať dátový typ explicitne, na konverziu použijeme funkciu *to_ T*, kde T predstavuje návratový typ.

cieľ:	zdroj
ZZ_p, zz_p, GF2	long
ZZ_pE	long, ZZ_p
zz_pE	long, zz_p
GF2E	long, GF2

Operátor / a procedúra *div* povýši údajové typy v tabuľke:

Uvedené platí len pre objekty a a b.

2.4.1 ZZ

Trieda umožňuje použiť operátory $/, \%, / =, \% =$ a funkcie *DivRem, div, rem* a *divide* pre testovanie deliteľnosti.

```
void DivRem(ZZ& q, ZZ& r, const ZZ& a, const ZZ& b);
```

Funkcia vypočíta podiel a/b, ten zaokrúhli smerom dolu a výsledok uloží do q. Zvyšok po delení uloží do r.

```
void div(ZZ& q, const ZZ& a, const ZZ& b);
```

Funkcia vypočíta podiel a/b, ten zaokrúhli smerom dolu a výsledok uloží do q.

```
void rem(ZZ& r, const ZZ& a, const ZZ& b);
```

Funkcia vypočíta podiel a/b, zvyšok po delení uloží do r.

Funkcie počítajú aj so zápornými číslami.

2.4.2 Varianty pre čísla s jednoduchou presnosťou

```
long DivRem(ZZ& q, const ZZ& a, long b);
```

Funkcia vypočíta podiel a/b, ten zaokrúhli smerom dolu a výsledok uloží do q. Vráti hodnotu zvyšku po delení.

```
long rem(const ZZ& a, long b);
```

Funkcia vráti hodnotu zvyšku po delení.
Funkcie počítajú aj so zápornými číslami.

2.4.3 Testovanie deliteľnosti

```
long divide(ZZ& q, const ZZ& a, const ZZ& b);
long divide(ZZ& q, const ZZ& a, long b);
```

Ak $b|a$, uloží výsledok delenia a/b do q a vráti 1, inak vráti 0.

```
long divide(const ZZ& a, const ZZ& b);
long divide(const ZZ& a, long b);
```

Ak $b|a$ vráti 1, inak vráti 0. Funkcie počítajú aj so zápornými číslami.

2.5 Umocňovanie

Na umocňovanie slúži funkcia *power*. Funkcia vypočíta $\mathbf{a^e}$ a výsledok uloží do x. Funkcionálna forma výsledok vráti. **Platia bežné povýšenia.** Platí pre všetky triedy okrem *ZZ*. Ak je e záporné, vyhodí **chyb**u (undefined inverse in _ntl_zinvmode).

```
void power(ZZ_p& x, const ZZ_p& a, const ZZ& e); // x = a^e
ZZ_p power(const ZZ_p& a, const ZZ& e);
```

ZZ Trieda ZZ obsahuje nasledovné funkcie:

```
void power(ZZ& x, const ZZ& a, long e); // x = a^e (e >= 0)
ZZ power(const ZZ& a, long e);

void power(ZZ& x, long a, long e);

funkcionálna forma:
ZZ power_ZZ(long a, long e);
long power_long(long a, long e);

void power2(ZZ& x, long e); // x = 2^e (e >= 0)
ZZ power2_ZZ(long e);
```

3 Generovanie náhodných prvkov

Funkcia *random* slúži na generovanie náhodných prvkov danej triedy. Platí pre všetky triedy okrem *ZZ*, ktorá má viac funkcií popísaných v kapitole 8.6 a 8.9.

3.0.1 random

```
void random(ZZ_p& x); náhodné číslo uloží do x
ZZ_p random_ZZ_p(); náhodné číslo vráti
```

4 Input/Output

```
istream& operator>>(istream& s, ZZ& x);
ostream& operator<<(ostream& s, const ZZ& a);
```

Načítané čísla sú vždy **redukované** modulo p (v prípade GF2 redukované modulo 2), prípadne modulo príslušný polynóm P.

5 Normy a stopy

Triedy *ZZ_pE, zz_pE* podporujú výpočet normy, triedy *ZZ_pE, zz_pE, GF2E* umožňujú výpočet stopy.

Definícia 1 *Nech* $\alpha \in GF(q^m) = F \supset GF(q) = K$. *Potom* $N_{(F/K)}(\alpha) = \alpha^{1+q+q^2+...+q^{m-1}}$ *sa nazýva norma prvku* α.

Definícia 2 *Nech* $\alpha \in GF(q^m) = F \supset GF(q) = K$. *Potom* $Tr_{(F/K)}(\alpha) = \alpha + \alpha^q + ... + \alpha^{q^{m-1}}$ *sa nazýva stopa prvku* α.

5.0.2 norm

```
void norm(ZZ_p& x, const ZZ_pE& a);   // x = norma prvku a
ZZ_p norm(const ZZ_pE& a);
```

5.0.3 trace

```
void trace(ZZ_p& x, const ZZ_pE& a);  // x = stopa prvku a
ZZ_p trace(const ZZ_pE& a);
```

6 Zmena modula

Všetky triedy okrem *GF2* (tu je modulo vždy rovné 2) umožňujú **zmenu modula a uloženie súčasného modula**. Na to slúži trieda *"názov triedy"Bak*, napr. *ZZ_pBak*.

```
ZZ_pBak bak;
bak.save();     // uloženie práve používaného modula
ZZ_p::init(p);  // nastavenie modula na novú želanú hodnotu p
// ...
bak.restore();  // obnovenie pôvodného modula
```

Pri **zmene** modula treba dávať pozor na objekty vytvorené pri predchádzajúcom module, ktorý je uložený. Pri jeho **obnovení** môžeme tieto objekty znovu používať. Používanie týchto objektov pri zmenenom module môže viesť k neočakávanému správaniu, pravdepodobne k **chybe**. Nemôžme tiež očakávať, že správanie objektov bude správne, keď znovu nastavíme hodnotu modula na pôvodnú pomocou init, ale neobnovíme ju z ...Bak. To znamená, že po zmene modula je vhodné vytvoriť nové objekty, ktoré budeme používať pre nový modulus. Po vrátení sa k pôvodonému modulu môžme znovu používať pôvodné objekty, takto sa vyhneme akýmkoľvek chybám.

Na všeobecnejšiu zmenu kontextu používame triedu *"názov triedy"Context*, napr. *ZZ_pContext*. Objekty triedy *"názov triedy"Context*, narozdiel od triedy *"názov triedy"Bak* môžu byť kopírované a inicializované.

7 Ostatné funkcie

Všetky triedy majú spoločné funkcie *clear, set, swap, zero.*

7.0.4 clear, set, swap, zero

```
void clear(ZZ& x); // nastaví x = 0
void set(ZZ& x);   // nastaví x = 1
void swap(ZZ& x, ZZ& y); // vymení x a y (výmena pointerov, ak je to možné)
static const ZZ& ZZ::zero(); // vráti read-only odkaz na nulu
```

Ďalšie funkcie

7.0.5 ModulusSize

```
static long ZZ_p::ModulusSize();
```

ZZ_p::ModulusSize() vráti veľkosť modula p

7.0.6 ModulusInverse

```
static double zz_p::ModulusInverse();
```

Vráti 1.0/(double(zz_p::modulus())).

7.0.7 modulus

```
static long GF2::modulus();
```

GF2::modulus() vráti hodnotu modulu triedy GF2, teda hodnotu 2.

7.0.8 WordLength

```
static long GF2X::WordLength();
```

GF2E::size() vráti počet slov potrebných na obsiahnutie polynómu stupňa $<$ GF2E::degree(). Napr.

```
GF2X P;
long a;
SetCoeff(P,0);
SetCoeff(P,4);
SetCoeff(P,33);

GF2E::init(P);
a=GF2E::WordLength();
```

vráti na 32bitovej architektúre 2 slová.

7.0.9 cardinality

```
static ZZ& ZZ_pE::cardinality();
```

vráti kardinalitu , t.j. $p^{ZZ\_pE::degree()}$ Platí aj pre triedy *zz_pE, GF2E.*

7.1 zz_p

7.1.1 init

```
static void zz_p::init(long p, long maxroot);
```

Výkonnejšie ako zz_p::init(p). Pri aritmetike modulo polynóm n-tého stupňa, nastavte maxroot na NextPowerOfTwo(n)+1 (ďalšia mocnina dvojky rovná alebo väčšia n, plus 1, t.j. pri polynóme 7-meho stupňa nastavíme 9). Je to účinné pri faktorizacii polynómu stupňa n modulo p, ak poznáte n vopred. Ak sa pokúšate nastaviť maxroot príliž malé, program vyhodí príslušnú chybu.

7.1.2 FFTInit

```
static void zz_p::FFTInit(long i);
```

Funkcia nastaví modulo na i-te FFT prvočíslo (počítajúc od 0). FFT prvočísla sú NTL_SP_NBITS-bitové prvočísla, kde p-1 je deliteľné veľkou mocninou dvojky. Aritmetika polynómov modulo p môže byť efektívne implementovaná použitím FFT. Zvyšovaním *i* sa znižuje mocnina dvojky, ktorá delí *p-1* a sprísňujú sa obmedzenia na stupne polynómov, ktorými sa ide násobiť.

7.1.3 zz_pContext

```
zz_pContext::zz_pContext(long p, long maxroot);
```

Konštruktor pre zz_pContext s rovnakou sémantikou ako zz_p::init(p, maxroot).

```
zz_pContext::zz_pContext(INIT_FFT_TYPE, long i);
```

Konštruktor pre zz_pContext s rovnakou sémantikou ako zz_p::FFTInit(i), volá sa zz_pContext(INIT_FFT, i).

8 Ďalšie funkcie triedy ZZ

8.1 GCD

8.1.1 GCD

```
void GCD(ZZ& d, const ZZ& a, const ZZ& b);
ZZ GCD(const ZZ& a, const ZZ& b);
```

Táto funkcia používa binárny GCD algoritmus na výpočet najväčšieho spoločného deliteľa. Výsledok uloží do d, alebo vráti ako návratovú hodnotu funkcie. Výsledok nikdy nie je záporný. Ak je jeden člen nula, do d sa uloží druhý člen. Ak sú oba členy nula, výsledok je nula.

8.1.2 XGCD

```
void XGCD(ZZ& d, ZZ& s, ZZ& t, const ZZ& a, const ZZ& b);
//  d = gcd(a, b) = a*s + b*t.
```

Koeficienty s a t sú vypočítané štandardným Euklidovým algoritmom aplikovaným na $|a|$ a $|b|$, znamienka sú neskôr prispôsobené podľa znamienok a a b. Ak $a = 78$ a $b = 34$, tak sa znamienka s a t prispôsobia tak, aby rovnica platila $2 = gcd(78, 34) = 78 * 7 + 34 * (-16)$.

Špeciálne verzie s jednoduchou presnosťou

```
long GCD(long a, long b);
```

Návratová hodnota je gcd(a,b), je vždy nezáporná.

```
void XGCD(long& d, long& s, long& t, long a, long b);
//  d = gcd(a, b) = a*s + b*t.
```

Ako varianta vyššie.

8.2 Modulárna aritmetika

Nasledujúce funkcie slúžia na aritmetiku modulo n, kde n $>$ 1. Predpokladá sa, že všetky argumenty okrem exponentov budú v rozsahu 0 .. n-1. Niektoré funkcie kontrolujú rozsah a vyhodia chybu, ak nesedí. Iné rozsah nekontrolujú a ich správanie môže byť neočakávané.

8.2.1 AddMod

```
void AddMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n); // x = (a+b)%n
ZZ AddMod(const ZZ& a, const ZZ& b, const ZZ& n);
```

Funkcia neredukuje záporné hodnoty (a+b) modulom n.

8.2.2 SubMod

```
void SubMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n); // x = (a-b)%n
ZZ SubMod(const ZZ& a, const ZZ& b, const ZZ& n);
```

Pri zápornej hodnote a alebo b vyhodí **chybu** - negative size allocation in _ntl_zsetlength a ukončí program.

8.2.3 NegateMod

```
void NegateMod(ZZ& x, const ZZ& a, const ZZ& n); // x = -a % n
ZZ NegateMod(const ZZ& a, const ZZ& n);
```

Ak je $a > n$ vyhodí **chybu** a ukončí program. Ak je $a < 0$, do x uloží nulu.

8.2.4 MulMod

```
void MulMod(ZZ& x, const ZZ& a, const ZZ& b, const ZZ& n); // x = (a*b)%n
ZZ MulMod(const ZZ& a, const ZZ& b, const ZZ& n);
```

Funguje aj za podmienky $a, b \leq 0$.

8.2.5 SqrMod

```
void SqrMod(ZZ& x, const ZZ& a, const ZZ& n); // x = a^2 % n
ZZ SqrMod(const ZZ& a, const ZZ& n);
```

Funguje aj za podmienky $a \leq 0$.

8.2.6 InvMod

```
void InvMod(ZZ& x, const ZZ& a, const ZZ& n);
ZZ InvMod(const ZZ& a, const ZZ& n);
// x = a^{-1} mod n (0 <= x < n); vyhodí chybové hlásenie,
keď inverzný prvok nie je definovaný
```

Chybové hlásenia vyhodí ak $a \geq n$ (first input too big), $a < 0$ (first input negative) a ak inverzný prvok nie je definovaný (undefined inverse in _ntl_zinvmod).

8.2.7 InvModStatus

```
long InvModStatus(ZZ& x, const ZZ& a, const ZZ& n);
// ak gcd(a,n) = 1, vráti hodnotu 0, x = a^{-1} mod n;
inak vráti hodnotu 1, x = gcd(a, n)
```

Platia rovnaké chybové hlásenia ako pre InvMod.

8.2.8 PowerMod

```
void PowerMod(ZZ& x, const ZZ& a, const ZZ& e, const ZZ& n);
ZZ PowerMod(const ZZ& a, const ZZ& e, const ZZ& n);

void PowerMod(ZZ& x, const ZZ& a, long e, const ZZ& n);
ZZ PowerMod(const ZZ& a, long e, const ZZ& n);

// x = a^e % n (e môže byť záporné)
```

Pre a musí platiť $0 < a < n$, inak vyhodí **chybu** bad args.

Funkcie *AddMod, SubMod, MulMod* (procedurálne aj funkcionálne formy) podporujú povýšenie z *long* na *ZZ*, platí pre *a, b*.

8.3 Modulárna aritmetika s jednoduchou presnosťou

Tieto funkcie implementujú modulárnu aritmetiku s jednoduchou presnosťou. Všetky vstupy musia byť z rozsahu 0 .. n-1, n je modulo. Modulo n musí byť z rozsahu 2..NTL_SP_BOUND-1. **Rozsah vstupov nie je kontrolovaný** (Pozri predchádzajúcu podsekciu).

8.3.1 AddMod

```
long AddMod(long a, long b, long n); // vráti (a+b)%n
```

8.3.2 SubMod

```
long SubMod(long a, long b, long n); // vráti (a-b)%n
```

8.3.3 NegateMod

```
long NegateMod(long a, long n); // vráti (-a)%n
```

8.3.4 MulMod

```
long MulMod(long a, long b, long n); // vráti (a*b)%n

long MulMod(long a, long b, long n, double ninv);
// vráti (a*b)%n.  ninv = 1/((double) n).  Ak je n fixované
pre veľa násobení, funguje rýchlejšie.
```

8.3.5 MulMod2

```
long MulMod2(long a, long b, long n, double bninv);
// vráti (a*b)%n.  bninv = ((double) b)/((double) n).  Ak je
n aj b fixované pre veľa násobení, funguje rýchlejšie.
Táto funkcia je zastaralá, použite MulModPrecon (pozri nižšie)
pre lepší výkon.
```

8.3.6 MulDivRem

```
long MulDivRem(long& q, long a, long b, long n, double bninv);
// vráti (a*b)%n, nastaví q = (a*b)/n.  bninv = ((double) b)/((double) n)
```

8.3.7 InvMod

```
long InvMod(long a, long n);
// počíta a^{-1} mod n.  Ak nie je definované, vyhodí chybu.
```

8.3.8 PowerMod

```
long PowerMod(long a, long e, long n);
// počíta a^e mod n (e môže byť záporné)
```

Nasledujú varianty *MulMod2*, ktoré sú podstatne rýchlejšie na niektorých počítačoch. Implementácia je rozdielna, podľa nastavení NTL_SPMM_ULL and NTL_SPMM_UL. Prednastavené hodnoty predstavujú implementáciu ako *MulMod2*. Najlepším riešením je nechať nastaviť optimálne hodnoty wizardom.

```
typedef mulmod_precon_t  /*  závisí od implementácie */ ;

mulmod_precon_t PrepMulModPrecon(long b, long n, double ninv);
// pripraví predbežné podmienky. ninv = 1/((double) n)

long MulModPrecon(long a, long b, long n, mulmod_precon_t bninv);
// vráti (a*b)%n.  bninv = MulModPrecon(b, n, ninv).

// Príklad použitia:
```

```
//    long a, b, n, c;
//      ...
//    double ninv = 1/((double) n);
//    mulmod_precon_t bninv = PrepMulModPrecon(b, n, ninv);
//     ...
//    c = MulModPrecon(a, b, n, bninv);  // c = (a*b) % n
```

8.4 Operátory posunu

Posun doľava o n znamená násobenie 2^n.
Posun doprava o n znamená delenie 2^n. (znamienko je konštantné)
Je možné použiť operátory $<<, >>, <<=, >>=$ alebo funkcie

8.4.1 LeftShift

```
void LeftShift(ZZ& x, const ZZ& a, long n);
ZZ LeftShift(const ZZ& a, long n);
```

8.4.2 RightShift

```
void RightShift(ZZ& x, const ZZ& a, long n);
ZZ RightShift(const ZZ& a, long n);
```

8.5 Bits and Bytes

8.5.1 MakeOdd

```
long MakeOdd(ZZ& x);
//funkcia delí číslo x dvojkou dovtedy, kým je x párne,
ak je prvý krát nepárne, uloží ho do x a vráti počet delení,
ak je čislo x nepárne, nezmení ho a vráti hodnotu 0
```

8.5.2 NumTwos

```
long NumTwos(const ZZ& x);
//vráti maximálne e také, že 2^e delí x, ak x=0 vráti 0
```

8.5.3 IsOdd

```
long IsOdd(const ZZ& a);
//vráti hodnotu 1, ak je a nepárne, vráti 0 ak je párne
```

8.5.4 NumBits

```
long NumBits(const ZZ& a);
long NumBits(long a);
//vráti počet bitov v binárnej reprezentácii |a|, NumBits(0)=0
```

8.5.5 bit

```
long bit(const ZZ& a, long k);
long bit(long a, long k);
//vráti k-ty bit čisla |a|, pozícia 0 predstavuje rádovo najnižší bit
//ak je k < 0 alebo k >= NumBits(a), vráti 0.
```

8.5.6 trunc

```
void trunc(ZZ& x, const ZZ& a, long k);
// x = k rádovo najnižších bitov z |a|.
// ak k <= 0, x = 0.

// dve funkcionálne formy:
ZZ trunc_ZZ(const ZZ& a, long k);
long trunc_long(const ZZ& a, long k);
```

8.5.7 SetBit

```
long SetBit(ZZ& x, long p);
// vráti pôvodnú hodnotu p-teho bitu z |x|, a nahradí ho hodnotou 1
// ak mal hodnotu 0; rádovo najnižší bit je bit 0; vyhodí chybu ak p < 0;
// znamienko x sa nemení
```

8.5.8 SwitchBit

```
long SwitchBit(ZZ& x, long p);
// vráti pôvodnú hodnotu p-teho bitu z |a|, a zmení hodnotu
// p-teho bitu z a; rádovo najnižší bit je bit 0; vyhodí chybu ak p < 0;
// znamienko x sa nemení
```

8.5.9 weight

```
long weight(const ZZ& a); // vráti Hammingovu váhu z |a|
long weight(long a);
```

8.5.10 Bitové boolovské operácie, procedurálne formy

```
void bit_and(ZZ& x, const ZZ& a, const ZZ& b); // x = |a| AND |b|
void bit_or(ZZ& x, const ZZ& a, const ZZ& b); // x = |a| OR |b|
void bit_xor(ZZ& x, const ZZ& a, const ZZ& b); // x = |a| XOR |b|
```

Namiesto procedurálnej formy môžeme použiť aj operátory pre bitové boolovské operácie, a to &, |, ^, pri ktorých dochádza k povýšeniu z *long* na *ZZ*. Ďalšie operátory sú &=, |=, ^=.

8.5.11 Konverzie medzi bytovými sekvenciami a ZZ

```
void ZZFromBytes(ZZ& x, const unsigned char *p, long n);
ZZ ZZFromBytes(const unsigned char *p, long n);
// x = sum(p[i]*256^i, i=0..n-1).
```

```
// Poznámka: ak je char viac ako 8 bitov,
// iba rádovo najnižších 8 bitov z p[i] je použitých

void BytesFromZZ(unsigned char *p, const ZZ& a, long n);
// Počíta p[0..n-1] ako abs(a) == sum(p[i]*256^i, i=0..n-1) mod 256^n.

long NumBytes(const ZZ& a);
long NumBytes(long a);
// vráti počet základných 256 číslic potrebných na reprezentáciu abs(a).
// NumBytes(0) == 0.
```

8.6 Pseudonáhodné čísla

Funkcie generujú vysoko kvalitné, kryptograficky silné pseudonáhodné čísla. Sú implementované tak, že ich správanie je nezávislé od hardvéru a implementácie dátového typu long integer. Iné funkcie v NTL používajú pseudonáhodné čísla, a preto veľkosť slova na počítači môže mať dopad na postupnosť čísel užívateľského programu.

8.6.1 SetSeed

```
void SetSeed(const ZZ& s);
```

Inicializuje generátor s počiatkom *s*. Z *s* sa vypočíta odtlačková funkcia a vygeneruje sa **počiatočný stav**, takže nie je potrebné aby *s* bolo náhodné, iba aby malo vysokú entropiu. Pokiaľ nie je funkcia SetSeed volaná pred funkciami uvedenými nižšie, použije sa prednastavený počiatočný stav. Zavolaním SetSeed kde s == 0, t.j. **SetSeed(ZZ::zero())**, sa znovu nastaví počiatočný stav do **prednastavenej hodnoty**. Potom je vždy generované rovnaké číslo. Funkcia ZZFromBytes (vyššie) môže byť potrebná.

8.6.2 RandomBnd

```
void RandomBnd(ZZ& x, const ZZ& n);
ZZ RandomBnd(const ZZ& n);
long RandomBnd(long n);
// x = pseudonáhodné číslo v rozsahu 0..n-1, alebo 0 if n <= 0
```

8.6.3 RandomBits

```
void RandomBits(ZZ& x, long l);
ZZ RandomBits_ZZ(long l);
long RandomBits_long(long l);
// x = pseudonáhodné číslo v rozsahu 0..2^l-1.
```

8.6.4 RandomLen

```
void RandomLen(ZZ& x, long l);
ZZ RandomLen_ZZ(long l);
long RandomLen_long(long l);
// x = pseudonáhodné číslo s presne l bitmi alebo
// 0 ak l <= 0.
```

```
unsigned long RandomBits_ulong(long l);
// vráti pseudonáhodné číslo v rozsahu 0..2^l-1
```

8.6.5 RandomWord

```
unsigned long RandomWord();
// vráti slovo typu long naplnené pseudonáhodnými bitmi.
// ekvivalentné s RandomBits_ulong(NTL_BITS_PER_LONG).
```

8.7 Čínska zvyšková veta

8.7.1 CRT

```
long CRT(ZZ& a, ZZ& p, const ZZ& A, const ZZ& P);
long CRT(ZZ& a, ZZ& p, long A, long P);
```

Funkcia požaduje vstupy *a, p, A, P* z rovníc

$$x \equiv a \bmod p$$

$$x \equiv A \bmod P,$$

ktoré musia spĺňať podmienky $0 \leq A < P$, $(p, P) = 1$. Funkcia vypočíta x, pre ktoré bude platiť $-\frac{p*P}{2} < x \leq \frac{p*P}{2}$, a uloží ho do a. Do p uloží modulo $p * P$. Vráti hodnotu 1, ak prepísal premennú a, inak vráti 0.

Ak **neplatí** $(p, P) = 1$, program vyhodí chybové hlásenie (undefined inverse in _ntl_zinvmod).

8.8 Rekonštrukcia racionálneho čísla

8.8.1 ReconstructRational

```
long ReconstructRational(ZZ& a, ZZ& b, const ZZ& x, const ZZ& m,
                         const ZZ& a_bound, const ZZ& b_bound);
```

Funkcia počíta parametre a, b, ktoré spĺňajú podmienky

1. $a \equiv b * x \bmod m$
2. $|a| \leq a\_bound$, $0 < b <= b\_bound$, a
3. $\gcd(m, b) = \gcd(a, b)$.

Funkcia vráti 1, ak našla vyhovujúce a, b a nastaví ich hodnoty do premenných a, b, ak také a, b nenašla, vráti 0. Podmienky pre vstupné parametre:

$0 \leq x < m$
$m > 2 * a\_bound * b\_bound$
$a\_bound \geq 0, b\_bound > 0$.

Ak existuje a, b spĺňajúce podmienky 1, 2 a podmienku

$$\gcd(m, b) = 1,$$

potom sú hodnoty a, b vrátené funkciou, ak spĺňa aj podmienku $\gcd(a, b) = 1$. Toto je treba po volaní funkcie dodatočne otestovať.
Funkcia je implementovaná použitím Lehmerovho rozšíreného Euklidovho algoritmu.

8.9 Testovanie a generovanie prvočísel

8.9.1 GenPrime

```
void GenPrime(ZZ& n, long l, long err = 80);
ZZ GenPrime_ZZ(long l, long err = 80);
long GenPrime_long(long l, long err = 80);
```

Funkcia **GenPrime** generuje náhodné prvočíslo n dĺžky v bitoch l také, že pravdepodobnosť, že výsledné n je zložené číslo je ohraničená 2^{-err}. Funkcia volá *RandomPrime* uvedenú nižšie a používa výsledky Damgarda, Landrocka a Pomeranca na optimalizáciu čísel, ktoré vyhoveli Miller-Rabinovmu testu.

8.9.2 GenGermainPrime

```
void GenGermainPrime(ZZ& n, long l, long err = 80);
ZZ GenGermainPrime_ZZ(long l, long err = 80);
long GenGermainPrime_long(long l, long err = 80);
```

Sophie-Germain prvočíslo je také prvočíslo p, pre ktoré platí, že $p_1 = 2 * p + 1$ je tiež prvočíslo. Funkcia **GenGermainPrime** generuje náhodné Sophie-Germain prvočíslo n dĺžky v bitoch l také, že pravdepodobnosť, že n alebo *2*n+1* sú neni prvočísla je ohraničená 2^{-err}.

8.9.3 ProbPrime

```
long ProbPrime(const ZZ& n, long NumTrials = 10);
long ProbPrime(long n, long NumTrials = 10);
```

Funkcia **ProbPrime** testuje, či je n prvočíslo. Opakuje do *NumTrials* Miller-witness testov. Ak n je prvočíslo, vráti hodnotu 1, ak nie je, vráti 0.

8.9.4 RandomPrime

```
void RandomPrime(ZZ& n, long l, long NumTrials=10);
ZZ RandomPrime_ZZ(long l, long NumTrials=10);
long RandomPrime_long(long l, long NumTrials=10);
```

Funkcia **RandomPrime** generuje náhodné prvočíslo n dĺžky l-bitov. Používa funkciu *ProbPrime*.

8.9.5 NextPrime

```
void NextPrime(ZZ& n, const ZZ& m, long NumTrials=10);
ZZ NextPrime(const ZZ& m, long NumTrials=10);
```

Funkcia **NextPrime** generuje ďalšie najmenšie prvočíslo n, ktoré je $\geq m$. Používa funkciu *ProbPrime*.

```
long NextPrime(long m, long NumTrials=10);
```

Verzia funkcie **NextPrime** na čísla s jednoduchou presnosťou. Výsledok bude vždy ohraničený hranicou NTL_ZZ_SP_BOUND a funkcia vyhodí chybové hlásenie ak táto podmienka nemôže byť splnená.

8.9.6 MillerWitness

```
long MillerWitness(const ZZ& n, const ZZ& w);
```

Funkcia **MillerWitness** vracia hodnotu 0, ak test indikuje, že n môže byť prvočíslo a hodnotu 1, aj test indikuje, že n je zložené. Funkcia testuje hodnotu w ako Millerovho svedka, či n je zložené. Predpokladom je, že n je nepárne a kladné číslo, a platí $0 \leq w < n$.

8.10 Druhá odmocnina a Jacobiho symbol

8.10.1 SqrRoot

```
void SqrRoot(ZZ& x, const ZZ& a); // x = floor(a^{1/2}) (a >= 0)

ZZ SqrRoot(const ZZ& a);
long SqrRoot(long a);
```

Funkcia vráti druhú odmocninu z čísla a zaokrúhlenú smerom nadol.

8.10.2 Jacobi

```
long Jacobi(const ZZ& a, const ZZ& n);
```

Funkcia počíta Jacobiho symbol z a a n, pričom predpokladá, že $0 \leq a < n$ a n je nepárne.

8.10.3 SqrRootMod

```
void SqrRootMod(ZZ& x, const ZZ& a, const ZZ& n);
ZZ SqrRootMod(const ZZ& a, const ZZ& n);
```

Funkcia počíta druhú odmocninu z a modulo n. Predpokladá, že n je nepárne prvočíslo, a je kvadratický zvyšok modulo n, pričom $0 \leq a < n$.

8.11 Ostatné funkcie

8.11.1 log

```
double log(const ZZ& a);
```

Funkcia vráti aproximáciu $log(a)$ s dvojitou presnosťou.

8.11.2 NextPowerOfTwo

```
long NextPowerOfTwo(long m);
```

Funkcia vráti najmenšie nezáporné k, také, že $2^k \geq m$.

8.11.3 size

```
long ZZ::size() const;
```

a.size() vráti počet zzigit v $|a|$. Veľkosť 0 je 0. Trieda ZZ je reprezentovaná postupnosťou "zzigit", kde každý zzigit je medzi 0 a $2^{NTL\_ZZ\_NBITS-1}$.

8.11.4 SetSize

```
void ZZ::SetSize(long k)
```

a.SetSize(k) nezmení hodnotu a, iba alokuje pamäť pre k zzigit.

8.11.5 SinglePrecision

```
long ZZ::SinglePrecision() const;
```

a.SinglePrecision() testuje či abs(a) $<$ NTL_SP_BOUND.

8.11.6 WideSinglePrecision

```
long ZZ::WideSinglePrecision() const;
```

a.WideSinglePrecision() testuje či abs(a) $<$ NTL_WSP_BOUND.

8.11.7 digit

```
long digit(const ZZ& a, long k);
```

Funkcia vráti k-ty zzigit z $|a|$, pozícia 0 je rádovo najnižší zzigit.

8.11.8 kill

```
void ZZ::kill();
```

a.kill() nastaví a na 0 a uvoľní pamäť premennej a.

8.11.9 ZZ

```
ZZ::ZZ(INIT_SIZE_TYPE, long k);
```

ZZ(INIT_SIZE, k) inicializuje na 0, ale pamäť je alokovaná tak, že čísla x, ak x.size() $\leq$ k, môže byť uložené bez realokácie.

8.11.10 zero

```
static const ZZ& ZZ::zero();
```

ZZ::zero() vráti odkaz na nulu.