Polynómy v NTL

Obsah

1 Moduly na prácu s polynómami

Na prácu s polynómami je možné v NTL použiť viacero modulov. Každý z nich má k dispozícii iné funkcie a je vhodný na použitie na iný účel. Základné moduly ktoré pracujú s polynómami sú *ZZ_X, ZZ_pX, ZZ_pEX, GF2X a GF2EX.*

- Trieda *ZZ_X* implementuje polynómy v ZZ[X], teda polynómy jednej premennej s celočíselnými koeficientami. Násobenie polynómov je implementované použitím 4 rôznych algoritmov:
  1. klasickým
  2. Karatsuba
  3. Schoenhage & Strassen - vykonáva FFT(Fast Fourier transform) s modulom, ktoré je "Fermatove číslo"vhodnej veľkosti. Toto násobenie je dobré pre polynómy s veľmi veľkými koeficientami a stredného stupňa.
  4. CRT/FFT - vykonáva FFT s modulom niekoľkých malých prvočísel. Je vhodný pre polynómy so strednými koeficientami a veľmi veľkého stupňa.

  Výber algoritmu je heuristický a nemusí byť vždy ideálny.
- Trieda *ZZ_pX* implementuje aritmetiku polynómov modulo p. Táto je implementovaná použitím FFT, skombinovanej s CRT(Chinese Remainder Theorem). Polynómy malého stupňa sú násobené buď pomocou klasického algoritmu, alebo Karatsubovým algoritmom.
- Trieda *ZZ_pEX* reprezentuje polynómy nad *ZZ_pE*, a tak môže byť použitá napríklad na aritmetiku v GF(p^n)[X]. Tam kde to nie je matematicky potrebné (napr. výpočet GCD), ZZ_pE nemusí byť nutne pole.
- Trieda *GF2X* implementuje aritmetiku polynómov modulo 2. Táto aritemetika je implementovaná klasickými postupmi a Karatsubom.
- Trieda *GF2EX* reprezentuje polynómy nad *GF2E* a tak môže byť použitá napr. na aritmetiku v GF(2^n)[X]. Tam kde to nie je matematicky potrebné (napr. výpočet), *GF2E* nemusí byť nutne pole.

2 Porovnávanie polynómov

Knižnica NTL poskytuje na porovnananie dvoch polynómov dva klasické operátory == (rovnosť), ! = (nerovnosť). Ďalej existujú funkcie *IsZero*, a *IsOne*. Obe fungujú pre všetky dátové typy, ktoré v NTL slúžia na reprezentáciu polynómov

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

Funkcia IsZero vracia 1, ak sú všetky koeficienty polyómu nulové, inak vracia 0. Funkcia IsOne vracia 1, ak je absolútny člen polynómu rovný 1 a všetky ostatné členy sú nulové, inak vracia 0.

3 Aritmetika polynómov

3.1 Sčítanie polynómov

Na sčítanie a odčítanie polynómov slúžia klasické operátory $+, -, + =, - =, ++, --$. Môžu sa použiť aj funkcie *add*, *sub*, *negate*. Tieto funkcie sa dajú použiť, pre všetky dátové typy, ktoré v NTL reprezentujú polynómy.

3.1.1 add

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

Do premennej *ZZ_pX x*, priradí súčet polynómov *ZZ_pX a* + *ZZ_pX b*.

3.1.2 sub

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

Do *ZZ_pX x*, priradí rozdiel polynómov *ZZ_pX a* - *ZZ_pX b*.

3.1.3 negate

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

_pX x, priradí negáciu polynómu *ZZ_pX a*.

3.2 Násobenie polynómov

Na násobenie polynómov sa používa operátor $*$. Na násobenie a umocňovanie sa používajú aj funkcie *mul*, *sqr*, *power*. Funkcie *mul*, *sqr* sú dostupné v každom module, ktorý pracuje s polyómami. Funkcia *power* je dostupná iba v moduloch *ZZ_pX, ZZ_pEX, GF2EX*.

3.2.1 mul

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

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

3.2.2 sqr

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

sqr do *ZZ_pX x* priradí druhú mocninu *ZZ_pX* a^2.

3.2.3 power

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

Funkcia do premennej *ZZ_pX x* priradí e-tu mocninu *ZZ_pX* a^e. Exponent e musí byť kladný, inak nastane chyba *negative exponent*.

3.3 Delenie polynómov

Vo všetkých moduloch je možné polynómy aj deliť. Umožnujú to buď klasické operátory / a %, alebo funkcie *div, rem, DivRem*. V module ZZX sú navyše k dispozícii aj funkcie *content, PrimitivePart, PseudoDivRem, PseudoDiv a PseudoRem*

3.3.1 DivRem

```
void DivRem(GF2X& q, GF2X& r, const GF2X& a, const GF2X& b); // q = a/b, r = a%b
```

Funkcia *DivRem* do premennej q priradí podiel a/b a do premennej r zvyšok po delení $r\%b$.

3.3.2 div

```
void div(GF2X& q, const GF2X& a, const GF2X& b); // q = a/b
```

Do q vracia podiel a/b.

3.3.3 rem

```
void rem(GF2X& r, const GF2X& a, const GF2X& b); // r = a%b
```

Funkcia *rem* do premennej q priradí zvyšok po delení a/b teda *a mod b*.

3.3.4 divide

```
long divide(GF2X& q, const GF2X& a, const GF2X& b);
```

Funkcia *divide* do premennej q priradí 1, ak b delí a, inak priradí 0.

3.3.5 content

```
void content(ZZ& d, const ZZX& f);
ZZ content(const ZZX& f);
```

Do d vracia najvačšie číslo k, ktoré delí polynóm f, teda k/f.

3.3.6 PrimitivePart

```
void PrimitivePart(ZZX& pp, const ZZX& f);
ZZX PrimitivePart(const ZZX& f);
```

Do premennej *pp* vracia *f(x)/k*, kde k je content *f*.

3.3.7 PseudoDivRem

```
void PseudoDivRem(ZZX& q, ZZX& r, const ZZX& a, const ZZX& b);
```

Vykoná pseudo-delenie: vypočíta *q,r* s *deg(q)* $<$ *deg(r)*, a $LeadCoeff(b)^{(deg(a)-deg(b)+1)}$, *a=b*q + r*.

3.3.8 PseudoDiv

```
void PseudoDiv(ZZX& q, const ZZX& a, const ZZX& b);
ZZX PseudoDiv(const ZZX& a, const ZZX& b);
```

Rovnaké ako *PseudoDivRem* ale počíta iba q.

3.3.9 PseudoRem

```
void PseudoRem(ZZX& r, const ZZX& a, const ZZX& b);
ZZX PseudoRem(const ZZX& a, const ZZX& b);
```

Rovnaké ako *PseudoDivRem* ale počíta iba r.

4 Bitové posuny

Vo všetkých moduloch je možné používať bitové posuny. Môžeme na ne použiť klasické operátory >>(RightShift) a <<(LeftShift). Takisto sú aj definované funkcie LeftShift a RightShift. LeftShift o n miest znamená násobenie x^n, RightShift o n miest znamená delenie x^n. V moduloch $GF2X$ a $GF2EX$ je možné použiť aj funkciu $MulByX$.

4.1 LeftShift

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

Do premennej *ZZ_pX x*, priradí ľavý posun premennej *ZZ_pX x* o n miest. Teda $x = a * X^n$. Napríklad $Leftshift(x^5 + x + 1, 2) = (x^7 + x^3 + x^2)$.

4.2 RightShift

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

Do premennej *ZZ_pX x*, priradí pravý posun premennej *ZZ_pX x* o n miest. Teda $x = a/X^n$. Napríklad $Rightshift(x^5 + x + 1, 2) = (x^3)$.

4.3 MulByX

```
void MulByX(GF2X& x, const GF2X& a);
```

Do premennej *GF2X x* priradí hodnotu *GF2X a*, vynásobenú X.

5 GCD

5.1 GCD

Pre polynómy všetkých typov je v NTL možné hľadať najväčší spoločný deliteľ. Je možné na to použiť 2 funkcie a to GCD a $XGCD$. Ak sa nejaký koeficient zadá záporný, vezme sa k nemu inverzný prvok.

```
void GCD(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& b);
ZZ_pX GCD(const ZZ_pX& a, const ZZ_pX& b);
// x = GCD(a, b),  x je vzdy monicky (alebo rovny nule, ak a==b==0).
```

Funkcia *GCD* priradí do premennej x $GCD(a, b)$ najväčší spoločný deliteľ polynómov a a b.

5.2 XGCD

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

Funkcia *GCD* priradí do premennej d $GCD(a, b)$ najväčší spoločný deliteľ polynómov a a b. Do premenných s a t sú priradené také polynómy aby platila rovnosť *a*s + b*t = d* . Ak sa nejaký koeficient zadá záporný, vezme sa k nemu inverzný prvok.

6 Input / Output

Všetky triedy umožňujú vstup a výstup polynómov. I/O formát */a_ 0 a_ 1 ... a_ n/*, reprezentuje polynóm $a_0 + a_1 * X + ... + a_n * X^n$. Na vstupe môže byť polynóm s lubovoľnými celočíselnými koeficientami, ktoré sú redukované modulo p (v pripade *GF2* redukované modulo 2).

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

7 Utility routines

V tejto časti sa nachádza niekoľko užitočných funkcií, ktoré môžu byť potrebné pri práci s polynómami. Sú tu obsiahnuté rôzne funkcie, ako napr. zistenie stupňa polynómu, zistenie koeficientov, derivácii, alebo vytváranie monických polynómov. Nasledujúce funkcie sú implementované vo všetkých moduloch.

7.1 Zistenie koeficientov polynómu

7.1.1 deg

```
long deg(const ZZ_pX& a);  // return deg(a); deg(0) == -1.
```

Funkcia *deg* vracia stupeň polynómu.

7.1.2 coeff

```
const ZZ_p& coeff(const ZZ_pX& a, long i);
```

Funkcia *coeff* sa používa na zistenie koeficientu polynómu na i-tom mieste. Ak $i < 0$ tak vracia 0.

7.1.3 LeadCoeff

```
const ZZ_p& LeadCoeff(const ZZ_pX& a);
```

LeadCoeff sa používa na zistenie koeficientu vedúceho prvku.

7.1.4 ConstTerm

```
const ZZ_p& ConstTerm(const ZZ_pX& a);
```

Na zistenie absolútneho člena polynómu slúži funkcia *ConstTerm*.

7.1.5 IsX

```
long IsX(const ZZ_pX& a); // testuje či x = X
```

Funkcia *IsX* slúži na otestovanie, či je polynóm rovný x.

7.2 Nastavenie koeficientov polynómu

7.2.1 SetCoeff

```
void SetCoeff(ZZ_pX& x, long i, const ZZ_p& a);
void SetCoeff(ZZ_pX& x, long i, long a);
```

Na nastavenie hodnoty polynómu x na i-tom mieste na hodnotu a sa používa funkcia *SetCoeff*. Ak $i < 0$ tak sa vyvolá chyba *negative index*.

7.2.2 SetX

```
void SetX(ZZ_pX& x); // x sa nastaví na monomial X
```

Pomocou funkcie *SetX* nastavíme akýkoľvek polynóm na "monomial", teda polynóm ktorý ma iba jeden koeficient a to pri x rovný 1 a všetky ostatné sa rovnajú 0.

7.2.3 reverse

```
void reverse(ZZ_pX& x, const ZZ_pX& a);
```

reverse slúži na priradenie koeficientov polynómu v opačnom poradí. $x = X^{deg(a)} * a(1/X)$.

7.2.4 MakeMonic

```
void MakeMonic(ZZ_pX& x);
```

Funkcia *MakeMonic* vytvorí monický polynóm (koeficient pri najvyššom stupni nastaví na 1). Táto funkcia definovaná iba pre ZZ_pX a ZZ_pEX.

7.3 ostatné

7.3.1 diff

```
void diff(ZZ_pX& x, const ZZ_pX& a);
```

Funkcia *diff* vracia prvú deriváciu polynómu.

7.3.2 VectorCopy

```
void VectorCopy(vec_ZZ_p& x, const ZZ_pX& a, long n);
```

Funkcia *VectorCopy* zapíše do vektoru vybraný počet koeficientov polynómu. Ak sa tento počet nerovná veľkosti polynómu, tak je zvyšok buď doplnený nulami, alebo orezaný.

7.3.3 MaxBits

```
long MaxBits(const ZZX& f);
```

Funkcia *MaxBits* udáva koľko bitov je potrebných na reprezentáciu najväčšieho koeficientu polynómu. Funkcia je k dispozícii iba v triede ZZX.

7.4 Funkcie v GF2X

V module $GF2X$ sú definované funkcie *weight, GF2XFromBytes, BytesFromGF2X, NumBits, NumBytes.*

7.4.1 weight

```
long weight(const GF2X& a);
```

Funkcia *weight* vracia váhu polynómu, teda počet jeho nenulových koeficientov.

7.4.2 NumBits, NumBytes

```
long NumBits(const GF2X& a);
long NumBytes(const GF2X& a);
```

Funkcia *NumBits* vracia počet bitov polynómu a *NumBytes* počet bytov polynómu. Platí $NumBytes = \lceil Numbits/8 \rceil$.

7.4.3 GF2XFromBytes

```
void GF2XFromBytes(GF2X& x, const unsigned char *p, long n);
GF2X GF2XFromBytes(const unsigned char *p, long n);
```

Konverzia bytového vektora na polynóm. *x=sum(p[i]* $* X^{8*i}$*, i = 0..n-1)*, kde bity *p[i]* sú interpretované ako polynóm prirodzeným spôsobom (p[i] = 1 je interpretované ako 1, p[i] = 2 je interpretované ako X, p[i] = 3 je interpretované ako X+1, atď.). Ak nastane situácia, že znak je reprezentovaný viac ako 8 bitmi, berie sa iba 8 bitov najnižšieho rádu.

7.4.4 BytesFromGF2X

```
void BytesFromGF2X(unsigned char *p, const GF2X& a, long n);
```

Konverzia polynóm na bytový vektor. *p[0..n-1] sú vypočítané ako a = sum(p[i]* $* X^{8*i}$*, i = 0..n-1) mod* X^{8*i}, kde hodnoty *p[i]* sú interpretované ako polynóm podobne ako v predchádzajúcom príklade *GF2XFromBytes.*

8 Náhodné polynómy

Všetky triedy okrem ZZX poskytujú funkcie na generovnanie náhodných polynómov.

8.1 random

```
void random(ZZ_pX& x, long n); // generate a random polynomial of degree < n
```

Funkcia *random* do polynómu *ZZ_pX x* priradí náhodne vygenerovaný polynóm stupňa $< n$.

9 Zisťovanie funkčnej hodnoty polynómov a príbuzné problémy

Triedy *ZZ_pX, ZZ_pEX a GF2EX* poskytujú funkcie na zisťovanie funkčnej hodnoty polynómov, ich interpoláciu podobné problémy.

9.1 BuildFromRoots

```
void BuildFromRoots(ZZ_pX& x, const vec_ZZ_p& a);
```

BuildFromRoots z koreňov zadaných vo vektore *vec_ZZ_p a* vygeneruje polynóm *ZZ_pX x* v tvare *(X-a[0]) ... (X-a[n-1])*, dĺžky a.length().

9.2 eval

```
void eval(ZZ_p& b, const ZZ_pX& f, const ZZ_p& a); // b = f(a)
```

Pomocou *eval* do premennej *b* priradíme funkčnú hodnotu *f(a)* funkcie *f* v bode *a*.

9.3 interpolate

```
void interpolate(ZZ_pX& f, const vec_ZZ_p& a, const vec_ZZ_p& b);
// interpoluje polynóm aby platilo f(a[i]) = b[i].
// p musí byť prvočíslo, inak nastane chyba
// division by non-invertible element
```

Funkcia *interpolate* do premennej *ZZ_pX f* priradí polynóm interpolovaný z vektorov *vec_ZZ_p a* a *vec_ZZ_p b*, tak aby platilo *f(a[i]) = b[i]*.

10 Aritmetika x^n

Aritmetika x^n a jej funkcie sú implementované vo všetých triedach, ktoré pracujú s polynómami. Všetky funkcie pracujú s modulom x^n a požadujú aby $n >= 0$, inak nastane chyba *bad args*.

10.1 trunc

```
void trunc(ZZ_pX& x, const ZZ_pX& a, long n); // x = a % X^n
```

trunc do premennej *ZZ_pX x* priradí polynóm, ktorý je výsledkom *ZZ_pX a* modulo X^n.

10.2 MulTrunc

```
void MulTrunc(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& b, long n);
// x = (a*b)%X^n
```

Pomocou *MulTrunc* do premennej *ZZ_pX x* priradíme polynóm, ktorý je výsledkom násobenia *ZZ_pX a* a *ZZ_pX b* modulo X^n.

10.3 SqrTrunc

```
void SqrTrunc(ZZ_pX& x, const ZZ_pX& a, long n); // x = a^2 % X^n
```

Funkcia *SqrTrunc* do premennej *ZZ_pX x* priradí polynóm, ktorý je výsledkom druhej mocniny *ZZ_pX a* modulo X^n.

10.4 InvTrunc

```
void InvTrunc(ZZ_pX& x, const ZZ_pX& a, long n);
// počíta x = a^{-1} % X^m.  Musí mať invertibilný ConstTerm(a).
// inak nastane chyba division by non-invertible element
```

Funkcia *InvTrunc* do premennej *ZZ_pX x* priradí polynóm, ktorý je inverzný k vstupnému polynómu *ZZ_pX a* modulo X^n, teda počíta $x = a^{-1} \ mod \ X^m$.

11 Modulárna aritmetika bez počiatočných podmienok

Táto časť obsahuje aritmetiku modulo f. Všetky vstupy a výstupy sú polynómy stupňa nižšieho ako *deg(f)* a zároveň *deg(f) > 0*. Funkcie sú definované pre všetky triedy, okrem funkcii *InvMod* a *InvModStatus*, ktoré nie sú definované pre triedu *ZZX*. Funkcie sú podobné ako v časti 10, v tejto časti, ale môžeme použiť ľubovoľný polynóm.

Ak chceme robiť veľa výpočtov s fixným *f*, tak pre lepšiu výkonnosť použijeme dátovú štruktúru *ZZ_pXModulus* a príbuzné funkcie uvedené nižšie.

11.1 MulMod

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

Funkcia *MulMod* do premennej *ZZ_pX x* priradí súčin polynómov *ZZ_pX a* a *ZZ_pX b* modulo polynóm *ZZ_pX f*.

11.2 SqrMod

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

MulMod do premennej *ZZ_pX x* priradí druhú mocninu polynómu *ZZ_pX a* modulo polynóm *ZZ_pX f.*

11.3 MulByXMod

```
void MulByXMod(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& f); // x = (a * X) mod f
```

Funkcia *MulByXMod* násobí premennú *ZZ_pX a* X a do *ZZ_pX x* priradí výsledok $x = (a * X)$ modulo f.

11.4 InvMod

```
void InvMod(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& f);
// x = a^{-1} % f, ak a nieje invertibilne nastane chyba non-invertible element.
```

InvMod vracia inverzný polynóm k *a modulo f.*

11.5 InvModStatus

```
long InvModStatus(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& f);
```

Ak *GCD(a,f) = 1* funkcia *InvModStatus* vracia 0 a nastaví $x = a^{-1} mod\ f$. Inak vracia 1 a nastaví $x = GCD(a,f)$.

12 Modulárna aritmetika s predvypočítanou informáciou

Ak potrebujeme robiť veľa výpočtov aritmetického modulo s fixným *f*, môžeme si vytvoriť *ZZ_pXModulus F* pre *f*. Do *ZZ_pXModulus F* sa uloží dopredu vypočítaná informácia o *f*, čo urýchli nasledovné výpočty. Väčšina funkcií je implementovaná pre všetky moduly okrem *ZZX*. Konverzie sú automatické, takže *ZZ_pXModulus* môže byť použitý všade, kde *ZZ_pX* a naopak.

12.0.1 build

```
void build(ZZ_pXModulus& F, const ZZ_pX& f);
```

Predvypočíta informácie o f a uloží ich v F na použitie pre ďalšie výpočty. Deklarácia *ZZ_pXModulus F(f)* je ekvivalentná s *ZZ_pXModulus F; build(F, f).*

12.1 Násobenie a umocňovanie

V nasledujúcich príkladoch je *f* polynóm dodaný do *build* a *n = deg(f)*

12.1.1 MulMod

```
void MulMod(ZZ_pX& x, const ZZ_pX& a, const ZZ_pX& b, const ZZ_pXModulus& F);
ZZ_pX MulMod(const ZZ_pX& a, const ZZ_pX& b, const ZZ_pXModulus& F);
```

Do x vracia polynóm $x = (a * b)$ *mod f* a zároveň *deg(a), deg(b)* $< n$.

12.1.2 SqrMod

```
void SqrMod(ZZ_pX& x, const ZZ_pX& a, const ZZ_pXModulus& F);
ZZ_pX SqrMod(const ZZ_pX& a, const ZZ_pXModulus& F);
```

Do x sa vracia $x = a^2$ *mod f, deg(a)* $< n$.

12.1.3 PowerMod

```
void PowerMod(ZZ_pX& x, const ZZ_pX& a, const ZZ& e, const ZZ_pXModulus& F);
ZZ_pX PowerMod(const ZZ_pX& a, const ZZ& e, const ZZ_pXModulus& F);

void PowerMod(ZZ_pX& x, const ZZ_pX& a, long e, const ZZ_pXModulus& F);
ZZ_pX PowerMod(const ZZ_pX& a, long e, const ZZ_pXModulus& F);
```

Funkcia do x vracia $x = a^e$ *mod f, deg(a)* $< n$, *deg(GCD(a,f)) = 1*, (e môže byť záporné), inak vracia chybu *bad args*.

12.1.4 PowerXMod

```
void PowerXMod(ZZ_pX& x, const ZZ& e, const ZZ_pXModulus& F);
ZZ_pX PowerXMod(const ZZ& e, const ZZ_pXModulus& F);

void PowerXMod(ZZ_pX& x, long e, const ZZ_pXModulus& F);
ZZ_pX PowerXMod(long e, const ZZ_pXModulus& F);
```

Vracia $x = X^e$ *mod f* (e môže byť záporné).

```
void PowerXPlusAMod(ZZ_pX& x, const ZZ_p& a, const ZZ& e,
                    const ZZ_pXModulus& F);

ZZ_pX PowerXPlusAMod(const ZZ_p& a, const ZZ& e,
                           const ZZ_pXModulus& F);

void PowerXPlusAMod(ZZ_pX& x, const ZZ_p& a, long e,
                    const ZZ_pXModulus& F);

ZZ_pX PowerXPlusAMod(const ZZ_p& a, long e,
                           const ZZ_pXModulus& F);
```

Funkcia *PowerXPlusAMod* je definovaná iba pre moduly *ZZ_pX a ZZ_pEX*.
Do x priradí $x = (X + a)^e$ *mod f* (e môže byť záporné).

12.2 Delenie a zvyšky

12.2.1 rem

```
void rem(ZZ_pX& x, const ZZ_pX& a, const ZZ_pXModulus& F);
```

Do x vracia $x = a \ mod \ f$.

12.2.2 DivRem

```
void DivRem(ZZ_pX& q, ZZ_pX& r, const ZZ_pX& a, const ZZ_pXModulus& F);
// q = a/f, r = a%f
```

Funkcia *DivRem* vracia $q=a/f$, $r=a \ mod \ f$.

12.2.3 div

```
void div(ZZ_pX& q, const ZZ_pX& a, const ZZ_pXModulus& F);
```

Počíta $x = a \ div \ f$.

12.3 Ostatné funkcie

Táto časť obsahuje 2 funkcie, ktoré sú dostupné v triede *ZZ_pX*.

12.3.1 build

```
void build(ZZ_pXMultiplier& B, const ZZ_pX& b, const ZZ_pXModulus& F);
```

Ak potrebujeme veľa krát počítať $a * b$ mod f je efektívnejšie používať pre b, *ZZ_pXMultiplier B* , ktorý predvypočíta informácie o b a uloží ich v B a zároveň $deg(b) < deg(F)$. Pracuje podobne ako *ZZ_pXModulus*.

12.3.2 MulMod

```
void MulMod(ZZ_pX& x, const ZZ_pX& a, const ZZ_pXMultiplier& B,
                                       const ZZ_pXModulus& F);
```

Do x priradí $x = (a * b) \ mod \ F;$ a zároveň $deg(a) < deg(F)$.

13 Modular composition

Modular composition je problém výpočtu $g(h) \ mod \ f$ pre polynómy f,g a h. V knižnici NTL tento problém riešime pomocou Brent & Kung algoritmu, ktorý použije $O(n^{1/2})$ násobení polynómov a $O(n^2)$ skalárnych operácií (n = deg(f)). Nasledovné funkcie sú dostupné vo všetkých moduloch okrem *ZZX*.

13.1 CompMod

```
void CompMod(ZZ_pX& x, const ZZ_pX& g, const ZZ_pX& h, const ZZ_pXModulus& F);
ZZ_pX CompMod(const ZZ_pX& g, const ZZ_pX& h,
                    const ZZ_pXModulus& F);
```

Do x priradí $x = g(h) \ mod \ f; \ deg(h) < n$.

13.2 Comp2Mod

```
void Comp2Mod(ZZ_pX& x1, ZZ_pX& x2, const ZZ_pX& g1, const ZZ_pX& g2,
              const ZZ_pX& h, const ZZ_pXModulus& F);
```

Počíta $x_i = g_i(h) \ mod \ f$ *(i=1,2); deg(h)* $< n$

13.3 Comp3Mod

```
void Comp3Mod(ZZ_pX& x1, ZZ_pX& x2, ZZ_pX& x3,
              const ZZ_pX& g1, const ZZ_pX& g2, const ZZ_pX& g3,
              const ZZ_pX& h, const ZZ_pXModulus& F);
```

Funkcia vracia $x_i = g_i(h) \ mod \ f$ *(i=1..3); deg(h)* $< n$.

14 Composition s predvypočítanou informáciou

Ak budeme používať jediný polynóm h na výpočet s viacerými polynómami g, potom by sme si mali vytvoriť *ZZ_pXArgument* pre h a následne použiť postup na kompozíciu uvedený nižšie. Tento postup vytvorí a uloži $h, h^2, ...h^m \ mod \ f$. Po tomto pred-výpočte, zloženie polynómu stupňa približne n s h vyžaduje n/m násobení *mod f* a n^2 skalárnych násobení. Takže zvyšovanie hodnoty m zvyšuje požiadavky na miesto a čas pred-výpočtu, ale znižuje čas na kompozíciu.

14.1 build

```
void build(ZZ_pXArgument& H, const ZZ_pX& h, const ZZ_pXModulus& F, long m);
```

Predvypočíta informáciu o h. $m>0$, *deg(h)*$<n$.

14.2 CompMod

```
void CompMod(ZZ_pX& x, const ZZ_pX& g, const ZZ_pXArgument& H,
             const ZZ_pXModulus& F);

ZZ_pX CompMod(const ZZ_pX& g, const ZZ_pXArgument& H,
                    const ZZ_pXModulus& F);



extern long ZZ_pXArgBound;
```

Na začiatku je rovná 0. Ak túto premennú nastavíme na hodnotu vačšiu ako 0, potom sa pri kompozícii alokuje tabuľka veľkosti približne *ZZ_pXArgBound*. Nastavenie tejto hodnoty ovplyvní všetky postupy kompozície, umocňovania a minimálnych polynómov, ktoré sú uvedené nižšie a nepriamo ovplyvní veľa ďalších postupov v *ZZ_pXFactoring*.

15 Power projection routines

15.1 project

```
void project(ZZ_p& x, const ZZ_pVector& a, const ZZ_pX& b);
ZZ_p project(const ZZ_pVector& a, const ZZ_pX& b);
```

x = skalárny súčin vektorov a a b.

15.2 ProjectPowers

```
void ProjectPowers(vec_ZZ_p& x, const vec_ZZ_p& a, long k,
                   const ZZ_pX& h, const ZZ_pXModulus& F);

vec_ZZ_p ProjectPowers(const vec_ZZ_p& a, long k,
                   const ZZ_pX& h, const ZZ_pXModulus& F);
```

Vypočíta vektor *(project(a,1), project(a,h), ..., project(a, h^{k-1} mod f)).*

15.3 ProjectPowers

```
void ProjectPowers(vec_ZZ_p& x, const vec_ZZ_p& a, long k,
                   const ZZ_pXArgument& H, const ZZ_pXModulus& F);

vec_ZZ_p ProjectPowers(const vec_ZZ_p& a, long k,
                   const ZZ_pXArgument& H, const ZZ_pXModulus& F);
```

Rovnaká funkcia ako predošlá, ale používa pred-vypočítaný argument *ZZ_pXArgument*, čiže výpočet je rýchlejší.

15.4 UpdateMap

```
void UpdateMap(vec_ZZ_p& x, const vec_ZZ_p& a,
               const ZZ_pXMultiplier& B, const ZZ_pXModulus& F);

vec_ZZ_p UpdateMap(const vec_ZZ_p& a,
               const ZZ_pXMultiplier& B, const ZZ_pXModulus& F);
```

Funkcia vypočíta vektor *(project(a,b), project(a,$(b * X)$ mod f), ..., project(a, $(b * X^{n-1}$ mod f)).* Pri jej použití platí obmedzenie *a.length() <= deg(F).* Vstup môže mať orezané nuly pri vysokom ráde, výstup ich bude mať vždy orezané.

16 Minimálne polynómy

Všetky tieto funkcie musia byť použité s inicializovaným prvočíslom p. Používa sa algritmus z [Shoup, J. Symbolic Comp. 17:371-391, 1994] and [Shoup, J. Symbolic Comp. 20:363-397, 1995], založený Berlekamp/Massey algoritme.

Funkcie sú dostupné vo všetkých moduloch okrem ZZX.

16.1 MinPolySeq

```
void MinPolySeq(ZZ_pX& h, const vec_ZZ_p& a, long m);
ZZ_pX MinPolySeq(const vec_ZZ_p& a, long m);
```

Funkcia počíta minimálny polynóm postupnosti. Premenná m stanovuje hranicu veľkosti stupňa polynómu. Zároveň musí platiť a.length() >= 2*m.

16.2 ProbMinPolyMod

```
void ProbMinPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F, long m);
ZZ_pX ProbMinPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F, long m);

void ProbMinPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F);
ZZ_pX ProbMinPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F);
```

Funkcia vypočíta monický minimálny polynóm ak *(g mod f)>0*. Hranica na stupeň minimálneho polynómu je *(m=a)*. Algoritmus je pravdepodobnostný, vždy vracia deliteľ minimálneho polynómu, alebo minimálny polynóm s pravdepodobnosťou *(=< m/p)*.

16.3 MinPolyMod

```
void MinPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F, long m);
ZZ_pX MinPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F, long m);

void MinPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F);
ZZ_pX MinPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F);
```

Pracuje rovnako a predošlá funkcia, ale vracia vždy minimálny polynóm.

16.4 IrredPolyMod

```
void IrredPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F, long m);
ZZ_pX IrredPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F, long m);

void IrredPolyMod(ZZ_pX& h, const ZZ_pX& g, const ZZ_pXModulus& F);
ZZ_pX IrredPolyMod(const ZZ_pX& g, const ZZ_pXModulus& F);
```

Rovnaká funkcia ako *MinPolyMod*, ale predpokladá, že f je ireducibilný, alebo prinajmenšom, že minimálny polynóm polynómu g je sám ireducibilný. Algoritmus je deterministický (a vždy vracia správny výsledok).

17 Composition a minimálne polynómy

Nasledujúce funkcie sú implementované v triedach *ZZ_pEX, GF2EX*. Požadujú aby bolo definované prvočíslo p, ale ZZ_pE nemusí byť pole.

17.1 CompTower

```
void CompTower(ZZ_pEX& x, const ZZ_pX& g, const ZZ_pEXArgument& h,
             const ZZ_pEXModulus& F);

ZZ_pEX CompTower(const ZZ_pX& g, const ZZ_pEXArgument& h,
             const ZZ_pEXModulus& F);

void CompTower(ZZ_pEX& x, const ZZ_pX& g, const ZZ_pEX& h,
             const ZZ_pEXModulus& F);

ZZ_pEX CompTower(const ZZ_pX& g, const ZZ_pEX& h,
             const ZZ_pEXModulus& F);
```

x=g(h) mod f.

17.2 ProbMinPolyTower

```
void ProbMinPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F,
                      long m);

ZZ_pX ProbMinPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F, long m);

void ProbMinPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F);

ZZ_pX ProbMinPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F);
```

Používa pravdepodobnostný algoritmus na výpočet minimálneho polynómu g *mod* f nad *ZZ_p*. Parameter m je hranica stupňa minimálneho polynómu (default = deg(f) * ZZ_pE::degree()). Vo všeobecnosti bude výsledok deliteľ skutočného minimálneho polynómu. Pre správne výsledky je potrebné použiť funkcie *MinPoly* uvedené nižšie.

17.3 MinPolyTower

```
void MinPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F, long m);

ZZ_pX MinPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F, long m);

void MinPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F);

ZZ_pX MinPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F);
```

Rovnaká ako predchádzajúca, ale výsledok je vždy správny.

17.4 IrredPolyTower

```
void IrredPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F, long m);

ZZ_pX IrredPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F, long m);
```

```
void IrredPolyTower(ZZ_pX& h, const ZZ_pEX& g, const ZZ_pEXModulus& F);

ZZ_pX IrredPolyTower(const ZZ_pEX& g, const ZZ_pEXModulus& F);
```

Rovnaká funkcia ako predchádzajúce, ale predpokladá, že minimálny polynóm je ireducibilný a používa mierne rýchlejší deterministický algoritmus.

18 Stopy , normy a resultant

Funkcie *TraceMod* a *TraceVec* je možné použiť vo všetkých triedach pracujúcich s polynómami. *NormMod* a *resultant* nie sú definované v *GF2X*. *CharPolyMod* je definované iba v *ZZX a ZZ_pX*. *discriminant a MinPolyMod* pracuje iba v triede *ZZX*.

18.1 TraceMod

```
void TraceMod(ZZ_pE& x, const ZZ_pEX& a, const ZZ_pEXModulus& F);
ZZ_pE TraceMod(const ZZ_pEX& a, const ZZ_pEXModulus& F);

void TraceMod(ZZ_pE& x, const ZZ_pEX& a, const ZZ_pEX& f);
ZZ_pE TraceMod(const ZZ_pEX& a, const ZZ_pEXModulus& f);
```

Do x priradí *x = Trace(a mod f); deg(a) < deg(f).* $Trace(\alpha) = \sum_{i=1}^{n-1} \alpha^{p^n}$

18.2 TraceVec

```
void TraceVec(vec_ZZ_pE& S, const ZZ_pEX& f);
vec_ZZ_pE TraceVec(const ZZ_pEX& f);
```

Do *S[]* priradí *S[i] = Trace(X^i mod f), i = 0..deg(f)-1; 0 < deg(f).* Horeuvedené funkcie implemtujú asymptoticky rýchly algoritmus hľadania stopy z [von zur Gathen and Shoup, Computational Complexity, 1992]

18.3 NormMod

```
void NormMod(ZZ_pE& x, const ZZ_pEX& a, const ZZ_pEX& f);
ZZ_pE NormMod(const ZZ_pEX& a, const ZZ_pEX& f);
```

Do x priradí *x = Norm(a mod f); 0 < deg(f), deg(a) < deg(f).*$Norm(\alpha) = \prod_{i=1}^{n-1} \alpha^{p^n}$

18.4 resultant

```
void resultant(ZZ_pE& x, const ZZ_pEX& a, const ZZ_pEX& b);
ZZ_pE resultant(const ZZ_pEX& a, const ZZ_pEX& b);
```

Do x priradí *x = resultant(a, b).* *NormMod* a *resultant* požadujú aby *ZZ_pE* bolo pole.

18.5 CharPolyMod

```
void CharPolyMod(ZZ_pX& g, const ZZ_pX& a, const ZZ_pX& f);
ZZ_pX CharPolyMod(const ZZ_pX& a, const ZZ_pX& f);
```

Do *g* priradí charakteristický polynóm *(a mod f); 0<def(f), deg(g)<deg(f)*. Táto funkcia pracuje pre lubovoľné *f*. Ak *f* je ireducibilný je výhodnejšie použiť *IrredPolyMod* a potom umocňovať, ak je to potrebné (pretože v tomto prípade je *CharPoly* mocninou *IrredPoly*.

19 Incremental Chinese remaindering

19.1 CRT

```
long CRT(ZZX& a, ZZ& prod, const zz_pX& A);
long CRT(ZZX& a, ZZ& prod, const ZZ_pX& A);
```

Ak *p* je aktuálny *zz_p/ZZ_p* modulus pre ktorý platí *(p, prod) = 1* vypočíta *a'* také že

a' = a mod prod a zároveň

a' = A mod p

s koeficientami z intervalu *(-p*prod/2, p*prod/2)*. Nastaví *a := a', prod := p*prod* a vracia 1, ak sa hodnota *a* zmenila.

20 Ostatné

Ak *f* je premenná reprezentujúca polynóm, tak pri všetkých dátových typoch, môžeme k jednotlivým členom pristupovať použitím *f.rep*. Absolútny člen získame použitím *f.rep[0]* a vedúci člen je *f.rep[f.rep.length()-1]*, samozrejme okrem prípadu, keď *f.rep.length() == 0*. Vedúci člen polynómu je vždy nenulový (kým je polynóm nenulový). Môžeme voľne pristupovať a modifikovať *f.rep*, ale vždy by sme sa mali ubezpečiť, že vedúci člen polynómu je nenulový, čo môžeme dosiahnuť volaním *f.normalize()*.

Nasledovné operácie sú definované pre všetky triedy pracujúce s polynómami.

20.1 Nastavenie a normalizácia polynómu

20.1.1 clear

```
void clear(ZZ_pX& x)
```

Funkcia clear priradí *x=0*.

20.1.2 set

```
void clear(ZZ_pX& x)
```

Funkcia clear priradí *x=1*.

20.1.3 normalize

```
void ZZ_pX::normalize();
```

Volaním *f.normalize()* odtránime vedúce nuly zo začiatku polynómu.

20.1.4 SetMaxLength

```
void ZZ_pX::SetMaxLength(long n);
```

Volaním *f.SetMaxLength(n)* alokujeme v pre polynóm miesto pre n koeficientov. Polynóm ktorý f reprezentuje ostáva nezmenený.

20.2 Konštrukcia a deštrukcia

20.2.1 kill

```
void ZZ_pX::kill();
```

Volanie *f.kill()* naství f na 0 a uvoľní pamäť držanú *f*.

20.2.2 ZZ_pX

```
ZZ_pX::ZZ_pX(INIT_SIZE_TYPE, long n);
```

ZZ_pX(INIT_SIZE, n) inicializuje polynóm rovný 0, ale alokuje miesto pre n koeficientov.

20.3 zero

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

Read - only referencia na 0.

20.4 swap

```
void swap(ZZ_pX& x, ZZ_pX& y);
```

Vymení hodnoty v premenných x *a* y (zmenou pointerov).