1 Príklady

Modulárna Aritmetika

NTL podporuje aj modulárnu aritmetiku celých čísel. Trieda `ZZ_p` reprezentuje celé čísla `modulo p`. Napriek zápisu, `p` nemusí byť prvčíslo, len v prípadoch keď je to matematicky požadované.

Triedy `vec_ZZ_p`, `mat_ZZ_p a ZZ_pX` reprezentujú vektory, matice a polynómy `modulo p`, a pracujú rovnako ako zodpovedajúce triedy pre `ZZ`.
Nasledujúci program načítava prvočíslo `p` a polynóm `f modulo p` a rozloží ho.

```
#include <NTL/ZZ_pXFactoring.h>

NTL_CLIENT

int main()
{
   ZZ p;
   cin >> p;
   ZZ_p::init(p);

   ZZ_pX f;
   cin >> f;

   vec_pair_ZZ_pX_long factors;

   CanZass(factors, f);  // calls "Cantor/Zassenhaus" algorithm

   cout << factors << "\n";

}
```

Ako program beží, NTL sleduje 'aktuálne modulo' pre triedu `ZZ_p`, ktoré môže byť inicializované alebo zmenené použitím príkazu `ZZ_p::init`. To-

to musí byť spravené pred deklaráciou premenných a výopočtami, ktoré závisia na module.
Z výkonnostných dôvodov treba poznamenať, že NTL nezabezpečuje kontrolu použitia premenných pri zmene modula. V takomto prípade, správanie programu je nepredvídateľné.

Ďalšie dva príklady sú na ukážku príbuzných tried `ZZ_p`. Prvý príklad slúži na sčítavanie vektorov:

```
#include <NTL/vec_ZZ_p.h>

NTL_CLIENT

void add(vec_ZZ_p& x, const vec_ZZ_p& a, const vec_ZZ_p& b)
{
   long n = a.length();
   if (b.length() != n) Error("vector add: dimension mismatch");

   x.SetLength(n);
   long i;
   for (i = 0; i < n; i++)
      add(x[i], a[i], b[i]);
}
```

A tento druhý na skalárny súčin vektorov:

```
#include <NTL/vec_ZZ_p.h>

NTL_CLIENT

void InnerProduct(ZZ_p& x, const vec_ZZ_p& a, const vec_ZZ_p& b)
{
   long n = min(a.length(), b.length());
   long i;
   ZZ accum, t;

   accum = 0;
   for (i = 0; i < n; i++) {
      mul(t, rep(a[i]), rep(b[i]));
      add(accum, accum, t);
   }

   conv(x, accum);
}
```

Druhý príklad objasňuje dve veci. Prvou je použitie funkcie `rep`, ktorá vracia odkaz na `ZZ_p`, ako ZZ s rozsahom `0` a `p-1`. Druhou je algoritmus, v ktorom sú realizované výpočty nad `ZZ` . Táto technika výrazne redukuje potrebný počet delení a spôsobuje rýchlejší chod programu.

Trieda `ZZ_p` podporuje všetky aritmetické operácie. Používa sa hlavne na prácu s vektormi, s maticami, a s polynómami `modulo p`. Na jednoduchšie úlohy z modulárnej aritmetiky je jednochšie pracovať len s triedou ZZ.

Obzvlášť, keď chceme pracovať s viacerými rôznymi modulami: prepínanie modula je podporované, ale použitie je trošku obtiažne.

Trieda `ZZ_pX` tiež podporuje všetky aritmetické operácie.
Pozri `ZZ p.txt` kvôli podrobnostiam o `ZZ_p`; `ZZ pX.txt` kvôli podrobnostiam o `ZZ_pX`; `ZZ pXFactoring.txt` kvôli podrobnostiam o faktorizácii polynómov nad `ZZ_p`; `vec ZZ p.txt` kvôli podrobnostiam o `vec_ZZ_p`;`mat ZZ p.txt` kvôli podrobnostiam o `mat_ZZ_p`.

V nasledujúcom príklade je aparát, ktorý objasňuje uloženie a obnovenie modula. Tento program vyžaduje za vstup polynóm a prvočíslo reprezentované číslami a testuje, či je daný polynóm rozložiteľný modulo prvočíslo.

```
#include <NTL/ZZX.h>
#include <NTL/ZZ_pXFactoring.h>

NTL_CLIENT

long IrredTestMod(const ZZX& f, const ZZ& p)
{
   ZZ_pBak bak;
   bak.save();  // save current modulus in bak

   ZZ_p::init(p);  // set the current modulus to p

   return DetIrredTest(to_ZZ_pX(f));

   // old modulus is restored automatically when bak is destroyed
   // upon return
}
```

Mechanizmus prepínania modula, je v skutočnosti trošku univerzálnejší a premenlivejší, ako daný príklad ukazuje.

Funkcia konverzie na `to_ZZ_pX`, je jedna z mnohých konverzných funkcií knižnice NTL. Mohla sa tiež použiť aj procedurálna forma:

```
ZZ_pX f1;
conv(f1, f);
return DetIrredTest(f1);
```

Predpokladajme, že v hore uvedenom príklade je `p` vopred známe a malé prvočíslo s jednoduchou presnosťou. V takomto prípade NTL poskytuje triedu `zz_p`, ktorá sa správa rovnako ako `ZZ_p`, spolu s odpovedajúcimi triedami `vec_zz_p`, `mat_zz_p` a `zz_pX`. Rozhrania ku všetkým programom sú obyčajne rovnaké, ako pre `ZZ_p` avšak sú výkonnejšie časovo aj priestorovo.

Predchádzajúci program pre malé prvočísla, môže byť programovaný nasledovne:

```
#include <NTL/ZZX.h>
#include <NTL/lzz_pXFactoring.h>

NTL_CLIENT

long IrredTestMod(const ZZX& f, long p)
{
   zz_pBak bak;
   bak.save();

   zz_p::init(p);

   return DetIrredTest(to_zz_pX(f));
}
```

Nasledovný program (v podstate rovnaký, aký je implementovaný v NTL) na výpočet GCD (NSD) polynómov s celočíselnými zložkami. Používa modulárny prístup: GCD-čka sú počítané modulo malé prvočísla a výsledky sú zložené použitím CRT (Čínska zvyšková veta). Malé prvočísla sú špeciálne vybraté "FFT prvočísla", ktoré umožnujú rýchle polynomické počty.

```
#include <NTL/ZZX.h>

NTL_CLIENT

void GCD(ZZX& d, const ZZX& a, const ZZX& b)
{
```

```
if (a == 0) {
   d = b;
   if (LeadCoeff(d) < 0) negate(d, d);
   return;
}

if (b == 0) {
   d = a;
   if (LeadCoeff(d) < 0) negate(d, d);
   return;
}

ZZ c1, c2, c;
ZZX f1, f2;

content(c1, a);
divide(f1, a, c1);

content(c2, b);
divide(f2, b, c2);

GCD(c, c1, c2);

ZZ ld;
GCD(ld, LeadCoeff(f1), LeadCoeff(f2));

ZZX g, res;

ZZ prod;

zz_pBak bak;
bak.save();

long FirstTime = 1;

long i;
for (i = 0; ;i++) {
   zz_p::FFTInit(i);
   long p = zz_p::modulus();

   if (divide(LeadCoeff(f1), p) ||
   || divide(LeadCoeff(f2), p)) continue;

   zz_pX G, F1, F2;
```

```
      zz_p  LD;

      conv(F1, f1);
      conv(F2, f2);
      conv(LD, ld);

      GCD(G, F1, F2);
      mul(G, G, LD);


      if (deg(G) == 0) {
         res = 1;
         break;
      }

      if (FirstTime || deg(G) < deg(g)) {
         prod = 1;
         g = 0;
         FirstTime = 0;
      }
      else if (deg(G) > deg(g)) {
         continue;
      }

      if (!CRT(g, prod, G)) {
         PrimitivePart(res, g);
         if (divide(f1, res) && divide(f2, res))
            break;
      }

   }

   mul(d, res, c);
   if (LeadCoeff(d) < 0) negate(d, d);
}
```

Pozri `lzz p.txt` kvôli podrobnostiam o `zz_p`; `lzz pX.txt` kvôli podrobnostiam o `zz_pX`; `lzz pXFactoring.txt` kvôli podrobnostiam o faktorizácii polynómov nad `zz_p`; `vec lzz p.txt` kvôli podrobnostiam o `vec_zz_p`;`mat lzz p.txt` kvôli podrobnostiam o `mat_zz_p`.

Aritmetika modulo 2 je dôležitým špeciálnym prípadom, pre ktorú NTL poskytuje triedu `GF2`, ktorá sa správa rovnako ako `ZZ_p` pre `p==2`, spolu

so zodpovedajúcimi triedami `vec_GF2`, `mat_GF2`, a `GF2X`. Rozhrania ku všetkým programom sú obyčajne rovnaké, ako pre `ZZ_p`, ale sú výkonnejšie časovo aj priestorovo.

Tento príklad znázorňuje triedy `GF2X` a `mat_GF2` pomocou jednoduchého programu na testovanie rozložiteľnosti polynómu nad `GF2` použitím lineárnej algebry.

```
#include <NTL/GF2X.h>
#include <NTL/mat_GF2.h>

NTL_CLIENT

long MatIrredTest(const GF2X& f)
{
   long n = deg(f);

   if (n <= 0) return 0;
   if (n == 1) return 1;

   if (GCD(f, diff(f)) != 1) return 0;

   mat_GF2 M;

   M.SetDims(n, n);

   GF2X x_squared = GF2X(2, 1);

   GF2X g;
   g = 1;

   for (long i = 0; i < n; i++) {
      VectorCopy(M[i], g, n);
      M[i][i] += 1;
      g = (g * x_squared) % f;
   }

   long rank = gauss(M);

   if (rank == n-1)
      return 1;
   else
      return 0;
}
```

Všimnime si príkaz,

```
g = (g * x_squared) % f;
```

ktorý by mohol byť nahradený s výkonnejším kódom,

```
MulByXMod(g, g, f);
```

ale toto by nedosiahlo významnú dobu výpočtu, pretože je to Gausova eliminačná metóda, ktorá je hlavnou zložkou doby výpočtu.

Pozri `GF2.txt` kvôli podrobnostiam o `GF2`; `GF2X.txt` kvôli podrobnostiam o `GF2X`; `GF2XFactoring.txt` kvôli podrobnostiam o faktorizácii polynómov nad `GF2`; `vec GF2.txt` kvôli podrobnostiam o `vec_GF2`;`mat GF2.txt` kvôli podrobnostiam o `mat_GF2`.

MODUL: `ZZ_p` zo `ZZ_p.txt`

SÚHRN:

Trieda `ZZ_p` je používaná na reprezentáciu čísel `modulo p`. Modulo `p` môže byť ľubovoľné kladné číslo, nemusí byť nevyhnutne prvočíslo.
Objekty triedy `ZZ_p` sú reprezentované, ako `ZZ` s rozsahom `0` až `p-1`.
Bežiaci program trvá na nastavení 'aktuálneho modula', na ktorú slúži príkaz `ZZ_p::init(p)`. Aktuálne modulo by malo byť nastavené pred vytvaráním objektov `ZZ_p`.
Modulo je možné meniť a je mechanizmus na jeho uloženie a obnovenie (pozri dole triedy `ZZ_Bak` a `ZZ_pContext`).

```
#include <NTL/ZZ.h>
#include <NTL/ZZVec.h>

class ZZ_p {
public:

   ZZ_p(); // nastavené na 0

   ZZ_p& operator=(const ZZ_p& a); // priradenie
   ZZ_p& operator=(long a); // priradenie

   ZZ_p(const ZZ_p& a); // kopírovací konštruktor
```

```
    ~ZZ_p(); // deštruktor

    static void init(const ZZ& p);
    // ZZ_p::init(p) nastavenie modula na p (p > 1)

    static const ZZ& modulus();
    // ZZ_p::modulus() udelí odkaz na dočasné
    // modulo
};
```