Big Integer

Veľké celé čísla

Obsah

1 Prvý príklad

Prvý príklad využíva triedu ZZ, ktorá reprezentuje "veľké celé čísla": čísla ľubovolnej dĺžky. Tento program načíta dve veľké celé čísla a a b, následne vypočíta $(a+1) \times (b+1)$.

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

NTL_CLIENT

int main()
{
   ZZ a, b, c;

   cin >> a;
   cin >> b;
   c = (a+1)*(b+1);
   cout << c << "\n";
}
```

Program deklaruje tri premenné a,b a c zo ZZ. Hodnoty a a b sú načítané zo štandartného vstupu. Hodnota c je vypočítaná ako $(a+1) \times (b+1)$. Nakoniec hodnota c je zobrazená v štandartnom výstupe.

Poznámky:

- Všimnite si, že sa môže počítať s veľkými ZZ tak, ako pri normálnych číslach, pretože väčšina štandardných výpočtov a operácií môže byť použitá v priamom a prirodzenom spôsobe. Kompilujúci program **C++** a programy knižnice **NTL** sa automaticky postarajú o všetko, čo súvisi s riadením pamäte a dočasnými objektami.

- Makro NTL_CLIENT. Keď kompilujeme **NTL** je nastevený štandartne **ISO** režim, rozšíriteľný o

  ```
  using namespace std;
  using namespace NTL;
  ```

  Keď kompilovanie **NTL** prebieha v tradičnom móde, rozšíri sa do prázdneho reťazca.

2 Druhý príklad

Tu je program, ktorý načíta zoznam celých čísel zo štandartného vstupu a vytlačí súčet ich štvorcov.

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

NTL_CLIENT

int main()
{
   ZZ acc, val;

   acc = 0;
   while (SkipWhiteSpace(cin)) {
      cin >> val;
      acc += val*val;
   }

   cout << acc << "\n";
}
```

Funkcia *SkipWhiteSpace* je definovaná v **NTL**. Preskočí biele znaky a vráti 1 ak za nimi niečo nasleduje. Táto funkcia je užitočná, pretože **NTL**-ové vstupné operátory zvýšia chybu ak vstup chýba alebo je zle formovaný. Týmto sa odlišuje od štandartnej vstupno/výstupnej knižnice, ktorá nezvýši chybu.

Nezvyšovnie chyby, alebo aspoň vyvolanie výnimky, je zlý nápad, pretože vyvolávač vstupu/výstupu musí stále kontrolovať štatút vstupného prúdu.

3 Tretí príklad

Program pre jednoduché modulárne umocňovanie pre počítanie a^e mod n. Predsa len **NTL** už poskytuje dômyselnejšie riešenie.

```
ZZ PowerMod(const ZZ& a, const ZZ& e, const ZZ& n)
{
   if (e == 0) return to_ZZ(1);

   long k = NumBits(e);

   ZZ res;
   res = 1;

   for (long i = k-1; i >= 0; i--) {
      res = (res*res) % n;
      if (bit(e, i) == 1) res = (res*a) % n;
   }

   if (e < 0)
      return InvMod(res, n);
   else
      return res;
}
```

Poznámky:

- Všimnite si, že ako alternatívu, by sme mohli implementovať vnútornú slučku nasledovným spôsobom:

  ```
  res = SqrMod(res, n);
  if (bit(e, i) == 1) res = MulMod(res, a, n);
  ```

- Môžeme ju taktiež napísať iným spôsobom napr.:

  ```
  SqrMod(res, res, n);
  if (bit(e, i) == 1) MulMod(res, res, a, n);
  ```

Toto objasňuje dôležitý bod o programovacom rozhraní **NTL**. Pre každú funkciu v **NTL** je procedurálna verzia, ktorá uloží výsledky do prvého argumentu. Efektívny dôvod pre používanie procedurálnych variant je: pri každej iterácií cez vyššie uvedenú slučku, spôsobí funkčná forma $SqrMod$ dočasný ZZ objekt pre vytvorenie a zničenie, zatiaľ čo procedurálna verzia nevytvorí žiadne dočasné objekty. Pokiaľ je výkon kritický, preferovaná bude procedurálna verzia. Hoci je to zvyčajne nezmyselné, pracovný výkon sa zvýši. Môžme si to argumentovať tým, že modulárne umocňovanie je dosť dôležitá rutina, ktorá by mala byť vykonávaná tak rýchlo ako je možné.

Poznámka:

- Ak je verzia funkcie funkčná, môže byť prirodzene volaná s operátorom. Napríklad, **NTL** poskytuje tretí argument *mul* pre ZZ násobenie a funkčná verzia, ktorej meno je operátor $*$, a nie *mul*.

Zatiaľ uvažujeme o dočasných objektoch, posudzujeme prvú verziu vnútornej slučky. Vykonanie príkazu:

```
res = (res*res) % n;
```

Výsledok vytvorenia dvoch dočasných objektov, jeden pre produkt a druhý pre výsledok operácie *mod*, ktorého hodnota je skopírovaná do výsledku. Samozrejme, kompilujúci program automaticky vytvára vpravý čas kód pre čistenie dočasných a iných lokálnych objektov. Čiže, programátor sa tým nemusí zaoberať resp. znepokojovať.

4 Štvrtý príklad

Tento príklad je zaujímavejší. Program vyzve používateľa pre zadanie vstupu a aplikuje ako prvé jednoduchý pravdepodobnostný test. Všimnite si, že **NTL** už poskytuje o niečo sofistikovanejší prvotný test.

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

NTL_CLIENT

long witness(const ZZ& n, const ZZ& x)
{
   ZZ m, y, z;
   long j, k;

   if (x == 0) return 0;

   // compute m, k such that n-1 = 2^k * m, m odd:

   k = 1;
   m = n/2;
   while (m % 2 == 0) {
      k++;
      m /= 2;
   }

   z = PowerMod(x, m, n); // z = x^m % n
   if (z == 1) return 0;

   j = 0;
   do {
      y = z;
      z = (y*y) % n;
      j++;
   } while (j < k && z != 1);

   return z != 1 || y != n-1;
}


long PrimeTest(const ZZ& n, long t)
{
   if (n <= 1) return 0;

   // first, perform trial division by primes up to 2000

   PrimeSeq s;  // a class for quickly generating primes in sequence
   long p;

   p = s.next();  // first prime is always 2
```

```
   while (p && p < 2000) {
      if ((n % p) == 0) return (n == p);
      p = s.next();
   }

   // second, perform t Miller-Rabin tests

   ZZ x;
   long i;

   for (i = 0; i < t; i++) {
      x = RandomBnd(n); // random number between 0 and n-1

      if (witness(n, x))
         return 0;
   }

   return 1;
}

int main()
{
   ZZ n;

   cout << "n: ";
   cin >> n;

   if (PrimeTest(n, 10))
      cout << n << " is probably prime\n";
   else
      cout << n << " is composite\n";
}
```

Poznámky:

- V **NTL** existuje väčšie množstvo spôsobov ako vypočítať rovnakú vec. Napríklad uvažujme výpočet m a k vo funkci *witness*. Môžme ju napísať nasledovne:

  ```
  k = 1;
     m = n >> 1;
     while (!IsOdd(m)) {
        k++;
        m >>= 1;
     }
  ```

  V podstate tento zápis nie je v skutočnosti oveľa výkonnejší ako pôvodný zápis, pretože implementácia optimalizuje násobenie a delenie do 2.

Nasledujúci program je výkonnejší:

```
k = 1;
while (bit(n, k) == 0) k++;
m = n >> k;
```

Ale iba v tom prípade, ak obsahuje vstavanú **NTL** funkciu, ktorá robí práve to, čo chceme:

```
m = n-1;
k = MakeOdd(m);
```

5 Konštruktory, úlohy a konverzie

Mali ste možnosť vidieť množstvo príkladov zahŕňajúcich ZZ, poďme sa trochu podrobnejšie pozrieť na ZZ rozhranie. Ak deklarujete nejaký objekt typu ZZ, štandartný konštruktor sa inicializuje na hodnotu 0. Ako sme už videli, obsahuje operátor riadenia, ktorý umožňuje kopírovať jednu hodnotu zo ZZ do ďalšej. Všimnite si, že tieto kópie sú "skryté", skutočný údaj je skopírovaný, nie je to len pointer (ukazovateľ). Samozrejme, ak množstvo priestoru prideleného cieľovou adresou je nedostatočné, priestor bude automaticky pridelený.

Poznámky

- Môže taktiež priradiť hodnotu typu *long* k ZZ:

```
ZZ x;
x = 1;
```

- Nemôže sa zapísať nasledovne(inicializujem ZZ):

```
ZZ x = 1;  // error
```

- V prrincípe návrhu sa **NTL** vyhýba iplicitným konverziam, ale žiaľ, jediný spôsob ako umožniť inicializáciu v **C++** je definovať nejaký implicitný konverzný operátor.

```
ZZ x = to_ZZ(1);
```

- Nasledovný príklad je jeden z **NTL** prevodových podprogramov pre veľmi veľké konštanty:

```
ZZ x = to_ZZ("99999999999999999999");
```

- Tento príklad objasní prevodové podprogramy v ich funkčných formách. Príslušné procedurálne formy sú volané príkazom *conv*:

```
ZZ x;
conv(x, 1);
conv(x, "99999999999999999999");
```

6 Funkčnosť

Všetky základné aritmetické operácie sú podporované vrátane porovnávania, aritmetických výpočtov, posunov a bitových logických operácií. Normálnym spôsobom sa môžu miešať ZZ a *long* čísla vo výrazoch. Ako už bolo spomenuté, **NTL** nepodporuje implicitnú konverziu typov. Pre základné operácie, jednoducho preťaží operátory alebo funkcie skôr než docieli oblasť "podpory logiky". Ak jeden vstup je ZZ a iný *long*(alebo niečo, čo implicitne konvertuje *long* na *int*), vstup *long* je efektívne prekonvertovaný na ZZ. Potom, kdekoľvek bude možné, sa efektívne urobí implementácia. Obvykle sa vyhýba vytvoreniu dočasných ZZ.

Existujú taktiež aj procedurálne verzie pre všetky základné operácie:

```
add, sub, negate, mul, sqr, div, rem, DivRem,
LeftShift, RightShift,
bit\_and, bit\_or, bit\_xor
```

Existuje mnoho ďalších funkcií:

- GCD – computes greatest common divisor of two integers
- $XGCD$ – extended Euclidean algorithm
- $AddMod, SubMod, NegateMod, MulMod, SqrMod, InvMod, PowerMod$ – routines for modular arithmetic, including inversion and exponentiation
- $NumBits$ – length of binary representation
- *bit* – extract a bit
- $ZZFromBytes, BytesFromZZ$ – convert between octet strings and ZZs
- $RandomBnd, RandomBits, RandomLen$ – routines for generating pseudo-random numbers
- $GenPrime, ProbPrime$ – routines for generating primes and testing primality
- *power* – (non-modular) exponentiation
- $SqrRoot$ – integer part of square root
- $Jacobi, SqrRootMod$ – Jacobi symbol and modular square root

Väčšina týchto funkcií je tiež typu *long*. Ako obvykle sú zastúpené obe varianty; funkčná a procedurálna.