CODIERUNGSTHEORIE - PRAKTIKA 2

Wombacher Sascha

13. Dezember 2015

AGENDA

1. Datenstrukturwahl

2. Modulare Polynommultiplikation

3. Tabellengenerierung

DATENSTRUKTURWAHL

DATENSTRUKTUR
- STD::ARRAY IM VERGLEICH ZU STD::VECTOR

1. Als Grunddatenstruktur wurde ein std::Array gewählt
2. Nachteile:
    - 2.1 Feste Arraylänge ist für größer werdende Polynome ungeeignet
    - 2.2 Die Abfrage des Grades benötigt im worst-case O(N), N = ArrayLänge
    - 2.3 Höherer Speicherverbrauch (viele 0en)
    - 2.4 Tendenziell langsamer
3. Vorteile:
    - 3.1 Erhöhung des Grades erstellt kein zweites Array (verglichen mit std::vector, worst case)
    - 3.2 Jeder nimmt std::vector => mal was anderes :D
    - 3.3 Bischen mehr template programming :)

MODULARE POLYNOMMULTIPLIKATION

MODULARE POLYNOMMULTIPLIKATION - ALGORITHMEN BESCHREIBUNG

Bekannte Parameter: Poly1, Poly2, DivPoly, PolyBasis

1. Berechne: Poly1 * Poly2, in temporäres Array (Tmp)
2. Reduziere alle Komponenten von Tmp zur PolyBasis
3. Solange Tmp.Grad >= DivPoly.Grad
   - 3.1 Berechne Additions-Faktor F, mit F = PolyBasis - Tmp.Grad (gilt nur wenn DivPoly[DivPoly.Grad] == 1 ist)
   - 3.2 Berechne Shift S, mit S = Tmp.Grad - DivPoly.Grad
   - 3.3 Addiere Komponentenweise das ”geshiftete”DivPoly zu Tmp
   - 3.4 Reduziere alle Komponenten von Tmp zur PolyBasis
4. gib Tmp zurück

MODULARE POLYNOMMULTIPLIKATION
- PROGRAMMCODE

```
1  // create temporary polynom with 2*_MaxLength
2  // this is the maximum grad the *—Operation can create
3  Polynom<_MaxDegree * 2, _BaseValue> tmp, tmpResult;
4
5  // calculate "multiplication" into tmp poly
6  for (int i = 0, myDegree = this—>degree(), polyDegree = poly.degree(); i <= myDegree;
       ++i){
7      for (int k = 0; k <= polyDegree; ++k)
8          tmpResult.m_Data[i + k] += poly.m_Data[k] * this—>m_Data[i];
9  }
10 tmpResult._truncateToBaseValue();
11 const int divisionDegree = DIVISION_POLYNOM—>degree();
12 for (int tmpResDegree = tmpResult.degree(); tmpResDegree >= divisionDegree;
       tmpResDegree = tmpResult.degree()){
13     int factor = _BaseValue — tmpResult.m_Data[tmpResDegree];
14     int shift = tmpResDegree — divisionDegree;
15
16     for (int i = 0; i <= divisionDegree; ++i)
17         tmpResult.m_Data[shift + i] += factor * DIVISION_POLYNOM—>m_Data[i];
18     tmpResult._truncateToBaseValue();
19 }
20 MY_TYPE toReturn;
21 for (int i = 0; i < _MaxDegree; ++i)
22     toReturn.m_Data[i] = tmpResult.m_Data[i];
23
24 // return "shrinked" result
25 return toReturn;
```

Modulare Polynommultiplikation - Assoziativgesetz Überprüfung, Seite 1

Für diese Aufgabe werden irreduzieblen Polynome aus Aufgabe 4 verwendet

- Überführe eine Ganzzahl in ein Polynom
  1. Erstelle einen Iterator $i = 0$
  2. Schreibe an Arrayposition i das Modulo der Ganzzahl mit der Polynombasis
  3. Dividiere die Ganzzahl durch die Polynombasis
  4. Inkrementiere i
  5. Wiederhole Schritt 2 - 4 solange Ganzzahl > 0 gilt

Modulare Polynommultiplikation - Assoziativgesetz Überprüfung, Seite 2

Iteriere über drei (unabhängige) Iteratoren $iter1, iter2, iter3 \epsilon [0, TableSize - 1]$

1. Konvertiere Iterator $iter1$ in Polynom $Poly1$
2. Konvertiere Iterator $iter2$ in Polynom $Poly2$
3. Konvertiere Iterator $iter3$ in Polynom $Poly3$
4. Erstelle $Poly4 = (Poly1 * Poly2) * Poly3$
5. Erstelle $Poly5 = Poly1 * (Poly2 * Poly3)$
6. Wenn $Poly4 \neq Poly5$ gilt, wirf Exception

MODULARE POLYNOMMULTIPLIKATION - ASSOZIATIVGESETZ PROGRAMMCODE

```
1  Polynom<_ArraySize, _BaseValue>::DIVISION_POLYNOM = &div; // set division poly
2  auto generatePolynom = [](int value) -> Polynom<_ArraySize, _BaseValue>{
3      Polynom<_ArraySize, _BaseValue> toReturn;
4      for (int i = 0; value; ++i){
5          toReturn.m_Data.at(i) = value % _BaseValue;
6          value /= _BaseValue;
7      }
8      return toReturn;
9  };
10 // check for error
11 for (int iter1 = 0; iter1 < _TestLength; ++iter1){
12     auto poly1 = generatePolynom(iter1);
13
14     for (int iter2 = 0; iter2 < _TestLength; ++iter2){
15         auto poly2 = generatePolynom(iter2);
16
17         for (int iter3 = 0; iter3 < _TestLength; ++iter3){
18             auto poly3 = generatePolynom(iter3);
19
20             auto poly4 = (poly1 * poly2) * poly3;
21             auto poly5 = poly1 * (poly2  * poly3);
22             if (poly4 != poly5)
23                 throw std::runtime_error("Error: wrong implementation");
24         }
25     }
26 }
27 std::cout << "\rmultiplication found no errors" << std::endl;
```

TABELLENGENERIERUNG

Tabellengenerierung - Beschreibung

Iteriere über zwei (unabhängige) Iteratoren $iter1, iter2 \epsilon [0, TableSize - 1]$

1. Konvertiere Iterator $iter1$ in Polynom *Poly1*
2. Konvertiere Iterator $iter2$ in Polynom *Poly2*
3. Gib $poly1 * poly2$ aus

TABELLENGENERIERUNG - PROGRAMMCODE

Anmerkung: Teile des Programmcodes (Tabllenlienien) wurden entfert (ansonsten genügt eine Folie nicht)

```
1  // generate polynoms from int-iterator
2  auto generatePolynom = [](int value) -> Polynom<_PolySize, _BaseValue>{
3      Polynom<_PolySize, _BaseValue> toReturn;
4      for (int i = 0; value; ++i){
5          toReturn.m_Data[i] = value % _BaseValue;
6          value /= _BaseValue;
7      }
8      return toReturn;
9  };
10 Polynom<_PolySize, _BaseValue>::DIVISION_POLYNOM = &divisionPoly;
11
12 // table itself
13 for (int iter1 = 0; iter1 < tableSize; ++iter1){
14     auto poly1 = generatePolynom(iter1);
15
16     for (int iter2 = 0; iter2 < tableSize; ++iter2){
17         auto poly2 = generatePolynom(iter2);
18
19         // foo is a function which performams addition or multiplication
20         std::cout << foo(poly1, poly2);
21     }
22 }
```

VIELEN DANK FÜR DIE AUFMERKSAMKEIT!