

CODIERUNGSTHEORIE - PRAKTIKA 2

Wombacher Sascha

7. Dezember 2015

1. Datenstrukturwahl
2. Modulare Polynommultiplikation
3. Tabellengenerierung

DATENSTRUKTURWAHL

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 $O(N)$, $N = \text{ArrayLänge}$
 - 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

Bekannte Parameter: Poly1, Poly2, DivPoly, PolyBasis

1. Berechne: $\text{Poly1} * \text{Poly2}$, in temporäres Array (Tmp)
2. Reduziere alle Komponenten von Tmp zur PolyBasis
3. Solange $\text{Tmp.Grad} \geq \text{DivPoly.Grad}$
 - 3.1 Berechne Additions-Faktor F, mit $F = \text{PolyBasis} - \text{Tmp.Grad}$
(gilt nur wenn $\text{DivPoly}[\text{DivPoly.Grad}] \neq 1$ ist)
 - 3.2 Berechne Shift S, mit $S = \text{Tmp.Grad} - \text{DivPoly.Grad}$
 - 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 "shrunked" result
25 return toReturn;
```

MODULARE POLYNOMMULTIPLIKATION - ASSOZIATIVGESETZ ÜBERPRÜFUNG

Hierfür werden die irreduzieblen Polynome aus Aufgabe 4 verwendet

1. TODO

MODULARE POLYNOMMULTIPLIKATION - ASSOZIATIVGESETZ PROGRAMMCODE

```
1  Polynom<_ArraySize, _BaseValue>::DIVISION_POLYNOM = &div;
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 poly_a = generatePolynom(iter1);
13
14     for (int iter2 = 0; iter2 < _TestLength; ++iter2){
15         auto poly_b = generatePolynom(iter2);
16
17         for (int iter3 = 0; iter3 < _TestLength; ++iter3){
18             auto poly_c = generatePolynom(iter3);
19             auto abc = (poly_a * poly_b) * poly_c;
20             auto bca = poly_a * (poly_b * poly_c);
21             if (abc != bca)
22                 throw std::runtime_error("Error: wrong implementation");
23         }
24     }
25 }
26 std::cout << "\rmultiplication found no errors" << std::endl;
```

TABELLENGENERIERUNG

1. TODO

TABELLENGENERIERUNG - PROGRAMMCODE

Anmerkung: Teile des Programmcodes (Tabllenlienien) wurden entfernt (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 poly_a = generatePolynom(iter1);
15
16     for (int iter2 = 0; iter2 < tableSize; ++iter2){
17         auto poly_b = generatePolynom(iter2);
18
19         // foo is a function which performs addition or multiplication
20         std::cout << foo(poly_a, poly_b);
21     }
22 }
```

VIELEN DANK FÜR DIE AUFMERKSAMKEIT!
