1	2	3	$\sum$

Wolfgang Mulzer, Katharina Klost

Algorithmen, Datenstrukturen und Datenabstraktion, Semester

TutorIn: Tobias Gleißner, Tutorium 02

Übung 02

Nicolas Höcker, Michael Wernitz

2. November 2017

1 Dünnbesetzte Polynome

Die neu erstellte Abspeicherung von sogenannten dünn besetzten Polynomen erfolgt durch Tupelbildung mit folgendem Schema.

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

wird abgespeichert als eine nach Grad des Polynoms absteigend geordnete Liste aus Tupeln mit folgender Bedingung:

$$p(x) = [(a_n, n), \ldots, (a_l, l)] \text{ ,wobei } a_i \neq 0,\ 0 \leq i, l \leq n$$

und für k (Anzahl der Nullpolynome) gilt: $k = 0$

- **Addition**
  Für die Addition zweier $p(x), q(x)$ Polynome gilt:

  $$\forall (a_i, i) \in p(x), \forall (b_l, l) \in q(x) : (a_i, i) + (b_l, l) = (a_i + b_l, i \vee l) \Leftrightarrow i = l$$

  Beispiel:

- **Multiplikation**
  Für die Multiplikation von zwei Polynomen $p(x), q(x)$ gilt:

  $$\forall (a_i, i) \in p(x), \forall (b_l, l) \in q(x) : (a_i, i) + (b_l, l) = (a_i * b_l, i * l) \quad 0 \leq i, l \leq deg(p(x), deg(q(x)))$$

  $$\forall (a_i, i) \in p(x), \forall (b_l, l) \in q(x) : (a_i, i) + (b_l, l) = (a_i * b_l, i * l + 1) \Leftrightarrow i = 1 \vee l = 1$$

  $$\forall (a_i, i) \in p(x), \forall (b_l, l) \in q(x) : (a_i, i) + (b_l, l) = (a_i * b_l, i) \Leftrightarrow l = 0 \quad 0 \leq i \leq deg(p(x)$$

  $$\forall (a_i, i) \in p(x), \forall (b_l, l) \in q(x) : (a_i, i) + (b_l, l) = (a_i * b_l, l) \Leftrightarrow i = 0 \quad 0 \leq l \leq deg(q(x)))$$

- Polynomauflösung $p(x)$ an Stelle x_0

  $$\forall (a_i, i) \in p(x) : \sum_{0 \leq i \leq n} (a_i * x_0^i)$$

  wobei zur Potenzierung der im Tutorium besprochene Algorithmus mit $O(n * log_2(n))$ angewendet wird

- **Effizienz der Algorithmen**
  1. Addition Mit dem Algorithmus zur Addition sind wir im Vergleich zu der in der Vorlesung vorgestelltem Darstellung mit k-0-Koeffizienten effizienter, da wir auf die Abspeicherung dieser verzichten, wodurch wir nur noch $(n-k)$ (n - Grad des Polynoms), anstatt n Additionen (wie in der Darstellung aus der Vorlesung) durchführen müssen. Damit unterlassen wir unnötige Additionsoperationen von k-0-Koeffizienten.
  2. Multiplikation Unsere Darstellung bewährt sich auch bei der Multiplikation von zwei Polynomen $p(x), q(x)$, da hier keine $(k * k^{'})$-0-Produkte vorkommen (wobei k bzw. $k^{'}$ - Anzahl der 0-Koeffizienten in $p(x)$ bzw. $q(x)$), da nur mit Koeffizienten $a_i \neq 0$ mit $0 \leq i \leq n$ gerechnet wird.
     Somit ist auch hier unser Algorithmus zur gewählten Darstellung effizienter als der in der Vorlesung vorgestellte.
  3. Auswertung an x_0 Die Auswertung eines Polynoms $p(x)$ an einer Stelle x_0 ist ebenso effizienter als der in der Vorlesung vorgestellte Algorithmus. Dabei können k-0-Produkte die durch Multiplikation eines 0-Koeffizienten und der i-ten Potenz von x_0 entstehen würden, sowie die k-Additionen (Anzahl der 0-Koeffizienten) weggelassen werden, was unseren Algorithmus effizienter gestaltet.
- **Implementierung der oben vorgestellten Darstellung von Polynomen**
  1. Addition

```python

def polynom_add(p1,p2):
  ergebnis_polynom = []
  p_zwischen = []

  if len(p1) == len(p2) and access_tupel(p1,0,1)>access_tupel(p2,0,1):
    p_zwischen= p2
    p2 = p1
    p1 = p_zwischen

  while len(p1)>0 or len(p2)>0:
      if len(p2) == 0 or access_tupel(p1,0,1)>access_tupel(p2,0,1):     # Fall p2 abgearbeitet oder p1 größerer Grad
          ergebnis_polynom.append(p1[0])
          p1.pop(0)
      elif len(p1) == 0 or access_tupel(p1,0,1)<access_tupel(p2,0,1):   # Fall p1 abgearbeitet oder p2 größerer Grad
          ergebnis_polynom.append(p2[0])
          p2.pop(0)
      else:                                                             # Fall p1 und p2 haben denselben Grad inne
          ergebnis_polynom.append((access_tupel(p1,0,0)+ access_tupel(p2,0,0), (access_tupel(p1,0,1))))
          p1.pop(0)
          p2.pop(0)
  return ergebnis_polynom


def access_tupel(polynom, stelle_von_tupel, position_in_tupel):    # Hilfsfunktion für den Zugriff auf die Tupelstruktur
    zwischenspeicher = polynom[stelle_von_tupel]
    zwischenspeicher2 = zwischenspeicher[position_in_tupel]
    return zwischenspeicher2
```

2. Multiplikation

```python
def polynom_mult(p1,p2):
    ergebnis_polynom = []              # Initialisiere Ergebnispolynom
    iterationen = len(p1)+len(p2)    # Verwende Multiplikationsformel aus der Vorlesung
    for i in range(0, iterationen+1):
        ergebnis = berechne_koeffizienten(p1,p2,i)
        if ergebnis[0]>0:
            ergebnis_polynom.insert(0,ergebnis)
    return ergebnis_polynom


def berechne_koeffizienten(p1,p2,i_index):         # Liefert Koeffizienten im Tupelformat bereits mit Grad zurück
    ergebnistupel = 0
    for j in range(0,i_index+1):                       # Analog zum Index 0 bis i bei der Summenformel für c_i
        zwischenspeicher_tupel1 =0                     # Initialisiere bzw. setze zurück auf 1
        zwischenspeicher_tupel2 =0
        for tupel in p1:                               # Finde das richtige Tupel mit dem Grad i in p1, falls vorhanden
            if tupel[1]==j:
                zwischenspeicher_tupel1 = tupel[0]   # Speichere Koeffizienten, falls vorhanden
        for tupel in p2:                               # Finde das richtige Tupel mit dem Grad i in p2, falls vorhanden
            if tupel[1]==(i_index-j):
                zwischenspeicher_tupel2 = tupel[0]    # speichere den Koeffizienten, falls vorhanden

        ergebnistupel = ergebnistupel + (zwischenspeicher_tupel1 * zwischenspeicher_tupel2) # Füge neu berechneten Koeffizienten zu Endergebnis hinzu

    return(ergebnistupel,i_index)
```

3. Auswertung an x_0

```python
def polynom_auswertung(x,polynom):
    ergebnis = 0

    for i in range(0,len(polynom)):
        ergebnis= ergebnis + (access_tupel(polynom,i,0)*(x**access_tupel(polynom,i,1)))

    return ergebnis
```

2 Experimentelles Sortieren

1. Bubblesort - worst-case Laufzeit: $O(n^2)$

```java
public class BubbleSort extends Sorting_Algorithms{

 private boolean swaped;
 private int arraySize;
 private double runtime;


 public BubbleSort(){
  this(new int[100]);
  super.fillArrayRandom();
 }

 public BubbleSort(int [] list){
  super(list);
 }

 public void bubblesort (){
  setSwaped(true);    // safe status if swaped or not
  int n = super.getSize();
  while(isSwaped() == true){
   setSwaped(false);
   for(int i=0; i < n-1; i++){  // check every element
     if(super.data[i] > super.data[i+1]){ // swap element if true
      int temp = super.data[i];
      super.data[i] = super.data[i+1];  // swaping
      super.data[i+1] = temp;
      setSwaped(true);  // set swaped true
     }
   }
   n = n-1;  // increase n for while loop
  }
 }
```

2. QuickSort - worst-case Laufzeit: $O(n^2)$

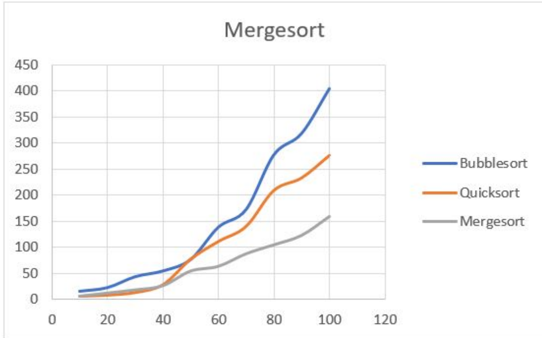
```java
public QuickSort(){
 this(new int[100]);
 super.fillArrayRandom();
}

public QuickSort(int [] array){
 super(array);
 right = super.data.length -1;
}

public void quickSort(){
 helper(left, right);
}

private void helper(int l, int r) {
 if(l < r){
  setDivider(divide(l,r)); // sorts the left and right side of the pivot element
  helper(l,getDivider()-1); // recursive call right list
  helper(getDivider()+1,r); // recursive call left list
 }
}

private int divide(int l, int r) {
  int i = l;   // index left side
  int j = r;  // index right side
  int pivot = super.data[r];

  while(i<j){
   // searches in left part of list for equal/bigger elements than pivot -> stop -> index with bigger element
   while(super.data[i] < pivot & i < r){
    i = i+1;
   }
   // searches in rigth part of list for lower elements than pivot -> index with lower element
   while(super.data[j] > pivot & j > l){
    j=j-1;
   }
   // change of elements in left/right lists
   if(i<j){
    int temp = super.data[i]; // tempory safe of element for change
    super.data[i] = super.data[j];
    super.data[j] = temp;
   }
  }
  // i is border of both parts of the list, left lower, rigth bigger than pivot
  // setting new pivot element
  if(super.data[i] > pivot){
   int help = super.data[i];
   super.data[i] = pivot;
   pivot = help;
  }
  // return index of pivot element
 return i;
}
```

3. MergeSort - worst-case Laufzeit: $O(n * log(n))$

```java
private int[] helper(int [] data) {
 if(data.length <= 1){
  return data;
 }else{
  int middle = (int) (data.length / 2);

  createLeft(data,0,middle);    // splits the lists into half everytime its called -> end: list with one element
  createRight(data,middle,data.length);

  // calls the function to build the result list
  // with recursive call of the splitted lists, so that at first the lists are splitted into list with one element and the merge function
  // combines every single list into the big result list
  return merge(helper(createLeft(data,0,middle)), helper(createRight(data,middle,data.length)));
 }

}
private static int[] merge(int[] left, int[] right) {
 int [] accuList = new int[left.length+right.length];  // creates the result list
 int indexleft = 0;  // saving the current indexes for navigating in the lists
 int indexright =0;
 int indexaccu = 0;

 // case that two not empty lists have to be combined
 while(indexleft < left.length && indexright < right.length){
  if(left[indexleft] <= right[indexright]){  // comparing first elements of the two lists
   accuList[indexaccu] = left[indexleft];  // saving into result list
   indexleft = indexleft +1;     // decrease index for comparing next left element with first right
  }else{
   accuList[indexaccu] = right[indexright];
   indexright = indexright +1;     // decrease index for comparing next right with first left
  }
  indexaccu = indexaccu +1;      // go to next index in resultlist and compare again or leave loop
 }

 // cases that one list is empty
 while( indexleft < left.length){
  accuList[indexaccu] = left[indexleft];
  indexleft = indexleft + 1;
  indexaccu = indexaccu +1;
 }
 while(indexright < right.length){
  accuList[indexaccu] = right[indexright];
  indexright = indexright +1;
  indexaccu = indexaccu + 1;
 }

 return accuList;
}
```

- Grafische Darstellung

Feldgröße n	Zeit in Millisekunden		
	Bubblesort	**Quicksort**	**Mergesort**
10	**16**	**6**	**7**
20	**23**	**8**	**13**
30	**44**	**13**	**19**
40	**55**	**28**	**27**
50	**77**	**77**	**55**
60	**139**	**111**	**64**
70	**173**	**140**	**88**
80	**278**	**209**	**105**
90	**319**	**233**	**123**
100	**405**	**276**	**159**



- Auswertung Es ist deutlich zu erkennen, dass Bubblesort mit steigenden Eingabegrößen eine deutlich längere Ausführungszeit besitzt als Quicksort. Gleiches ist beim Vergleich von Quicksort und Mergesort festzustellen. Dieses Ergebnis spiegelt die den Sortieralgorithmen entsprechenden Komplexitätsklassen von Quick-/Bubblesort mit $O(n^2)$ und $O(n * log(n))$ von Mergesort.

- Vorteile der Algorithmen

  1. Bubblesort Bubblesort ist ein stabiler in-place Algorithmus der einfach zu implementieren und verstehen ist. Daher wird er meist für lehrreiche Zwecke verwendet, um z.B.: Komplexitäts- oder Korrektheitsbeweise durchzuführen.

  2. Quicksort Quicksort hat eine sehr hohe Ausführungsgeschwindigkeit, die durch eine kurze innere Schleife erreicht wird. Es ist ein in-place Verfahren, welches mit Ausnahme des benötigten Speicherplatzes für die Rekursionsaufrufe auf dem Aufruf-Stack keinen externen Speicherplatz benötigt.

  3. Mergesort Mergesort ist ein sehr guter rekursiver Algorithmus, an dem das Divide and Conquer Prinzip sehr deutlich wird. Dieser Algorithmus hat eine Laufzeit von $O(n * log(n))$ sowohl im best-/average case als auch im worst case. Es ist zudem ein stabiles Verfahren.

- Nachteile der Sortieralgorithmen

  1. Bubblesort Bubblesort besitzen bei sehr großen n eine sehr schlechte Laufzeit, da er mehrmals durch die volle Liste iteriert und Element darin vergleicht.

  2. Quicksort Quicksort ist kein stabiles Verfahren, wodurch manche Datensätze falsch sortiert werden und es zu mehr oder weniger großen Komplikationen kommen kann.

  3. Mergesort Mergesort benötigt zur Ausführung externen Speicher, da es kein in-place Algorithmus ist.

- Verwendung der Algorithmen Wenn nur mit sehr kleinen Eingaben gearbeitet wird und die Laufzeit nicht ins Gewicht fällt, ist Bubblesort eine einfache, schnell zu erstellende und stabile Lösung. Benötigt man eine hohe Ausführungsgeschwindigkeit und die Eingaben sind teils sortierte Listen ist Quicksort zu empfehlen. Dabei muss beachtet werden, dass Quicksort nicht satbil ist und die Eingabelisten nicht zu groß sein dürfen, da die Laufzeit gegen $O(n^2)$ geht. Mergesort ist unter den drei gewählten Verfahren das mit der besten Laufzeit in allen drei Fällen. Es ist aber nicht in-place, d.h. es wird zusätzlicher Speicher gebraucht, um Mergesort anzuwenden. Bei sehr großen Eingaben ist Mergesort unter den dreien sehr zu empfehlen, da es auch dort am schnellsten ist.

- Aussagen der O-Notation Schaut man sich die Laufzeitfunktion zu den gemessenen Eingaben an erkennt man, dass diese bei Bubblesort und Quicksort den Grad 2 besitzen, sie also quadratisch und damit in der Komplexitätsklasse von $O(n^2)$ liegen, also die Funktion $f(n) = n^2$ die obere Grenze ist. Bei Mergesort lässt sich ähnliches erkennen. Dort stellt die Funktion $f(n) = n * log(n)$ die obere Grenze des Graphen der gemessenen Werte dar.

3 Türme von Hollywood

a)Sei n die Anzahl der Scheiben und die drei vertikalen Stangen als A,B,C bezeichnet, wobei A die erste und C die letzte Stange ist. Um alle Scheiben 1 bis n von A nach C zu bewegen, müssen zuerst die größte Scheibe, also die Scheibe n, nach C bewegt werden.
Um dieses Problem zu lösen, müssen wir aber erst n-1 Scheiben nach B bewegt werden, um die Ordnungseigenschaft einzuhalten. Anschließend müssen die n-1 Scheiben nach C verschoben werden.
Dafür müssen wir aber, um n-1 Scheiben nach B zu bewegen, n-2 Scheiben nach C bewegen. Danach können wir die n-1-te Scheibe nach B bewegen, um anschließend die n-2 Scheiben von C nach b zu bewegen.
Wir zerlegen analog rekursiv weiter bis wir beim Problem angelangt sind, wohin wir Scheibe 1 bewegen.

- - Sei also n die eindeutige Scheibennummer für die Funktion $bewegeScheibe(n)$, so lautet die Anzahl von Verschiebungen:

$$bewegeScheibe(n) = bewegeScheibe(n-1) + 1 + bewegeScheibe(n-1)$$
$$bewegeScheibe(1) = 1$$

  - wir können also vereinfachen zu:

$$bewegeScheibe(n) = 1 + 2 * bewegeScheibe(n-1)$$

  - Um pragmatisch die Anzahl der Scheiben für n=100 ermitteln zu können, müssen wir allerdings die geschlossene Formel für diese rekursive Funktion finden.
  -

$$\begin{aligned}
b(n) &= 1 + 2 * (b(n-1)) \\
&= 1 + 2 * (1 + 2 * (b(n-2))) \\
&= 1 + 2 * (1 + 2 * (1 + 2 * (b(n-3)))) \\
&= 1 + 2 + 4 + 8 * b(n-3) \\
&= 1 + 2^1 + 2^2 + 2^3 * b(n-1) \\
&= 1 + 2^1 + \cdots + 2^{n-1} * b(n) \\
&= 1 + 2^1 + \cdots + 2^{n-1} \\
&= \ldots
\end{aligned}$$

  - Beweis durch Induktion nach n

Beweis. ∗ I.A.: $n = 1$

$$2^1 - 1 = 2 - 1 = 1 = b(1)$$
$$n = 2$$
$$\begin{aligned} b(2) &= b(1) + 1 + b(1) \\ &= 1 + 2 + 1 \\ &= 3 \end{aligned}$$

$$2^2 - 1 = 3 = b(2)$$

∗ I.V.: $b(n) = 2^n - 1$

∗ I.B.: $b(n + 1) = 2^{n+1} - 1$

∗ I.S.: $n \Rightarrow (n + 1)$

$$\begin{aligned} b(n + 1) &= 1 + 2 * b(n) \\ &\overset{IV}{=} 1 + 2(2^n - 1) \\ &= (2^n - 1) + 1 + (2^n - 1) \\ &= 2^{n+1} - 1 + 1 - 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

□

– Also gilt für $n = 100$:

$$\begin{aligned} 2^{100} - 1 &= 1267650600228229401496703205375\text{Tage} \\ &\approx 3.473.015.343.091.039.456.155.351.248\text{Jahre} \\ &\approx 3.473.015\text{Trilliarden Jahre} \end{aligned}$$

- Seien A,B,C Stangen und A Startstange und C Endstange mit n Scheiben.
  So kann das Problem erneut rekursiv gelöst werden. Wir ermitteln die Formel durch Beobachtung. Zu Beginn wollen wir Scheibe n von A nach C bringen. Dafür können wir die anderen Scheiben weitestgehend so verschieben, dass sie von A nach B verlegt werden.
  z.B.: bei n=6, S = Scheibe, Ziel: Scheibe n nach C verschieben

$$\begin{aligned} S1 :&\quad A \to B \\ S2 :&\quad A \to C \Rightarrow S1 : B \to C \\ S3 :&\quad A \to B \Rightarrow S1 : C \to B, S2 : C \to B \\ S4 :&\quad A \to C \Rightarrow S2 : B \to C, S1 : B \to C, S3 : B \to C \\ &\quad \dots \\ S6 :&\quad \dots \end{aligned}$$

  Wir können beobachten, $\lceil \frac{n-1}{2} \rceil$ Scheiben aufsteigend geordnet sind und die folgenden $\lfloor \frac{n-1}{2} \rfloor$ Elemente absteigend geordnet sind (Betrachtung des Stapels von oben).
  Anschließend kann man das oberste bis vorletzte Element con B nach A schieben, sodass die geforderte Ordnung erhalten bleibt und setzt Scheibe 5 nach C.
  Dies wiederholt man so lange bis C geordnet und vollständig ist.

- Formel: Wir benötigen für den ersten Schritt, um die Scheibe n nach C zu bekommen folgende Anzahl von Schritten.

$$\begin{aligned} bewege(n) &= (n-1) + bewege(n-1) \\ bewege(1) &= 1 - (n-1) \end{aligned}$$

- mit (n-1) bezeichnen wir den Abzug der letzen Verschiebung auf C, da nicht alle Elemente am Ende auf C liegen müssen, sondern nur Scheibe n

- Anschließend müssen wir noch für jede umzuschiebene Scheibe, also (n-1) auf B oder A die Scheiben $1, \ldots, (n-2)$ auf den jeweils anderen Stapel schichten und Scheibe n-1 nach C verschieben. Das wiederholen wir bis C fertig ist.
  Also:

$$\begin{aligned} bewege_zuletzt(n) &= n + bewege_zuletzt(n-1) \\ bewege_zuletzt(1) &= 1 \end{aligned}$$

  Fassen wir beide Formeln zusammen, erhalten wir:

$$\begin{aligned} bewege(n) &= (n-1) + bewege(n-1) \\ bewege(1) &= 1 - (n-1) \end{aligned}$$

  Also:

$$\sum_{i=1}^{n-1} i \quad + \quad \sum_{i=1}^{n} i \quad = \quad 2 * \left(\sum_{i=1}^{n} i \right) - n$$

- setzen wir nun n=100 ein erhalten wir:

$$\begin{aligned} 2 * \left(\sum_{i=1}^{100} i \right) - 100 &= 2 * * \frac{100^2 + 100}{2} - 100 \\ &= 10000 \end{aligned}$$

- folglich bräuchte man ca. 10000 Tage zum verschieben, was ca. 27 Jahre währen