technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2020/21

Kapitel 13: Datenstrukturen und Algorithmen

M.Sc. Roman Kalkreuth
Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik

Inhalt

Hashing

- Motivation
- Grobentwurf
- ADT Liste (ergänzen)
- ADT HashTable
- Anwendung
-
- **Mergesort**
- Konzept
- Laufzeitanalyse
- Realisierung (mit Schablonen)

Motivation

Gesucht: Datenstruktur zum Einfügen, Löschen und Auffinden von Elementen

⇒ Binäre Suchbäume!

Problem: Binäre Suchbäume erfordern eine **totale Ordnung** auf den Elementen

Totale Ordnung

Jedes Element kann mit jedem anderen verglichen werden:

Entweder a < b oder a > b oder a = b. Beispiele: $\mathbb{N}$, $\mathbb{R}$, { A, B, …, Z }, …

Partielle Ordnung

Es existieren unvergleichbare Elemente: a || b

Beispiele: $\mathbb{N}^2$, $\mathbb{R}^3$ …

$$\begin{pmatrix} 2 \\ 5 \end{pmatrix} < \begin{pmatrix} 8 \\ 6 \end{pmatrix}; \quad \begin{pmatrix} 2 \\ 5 \end{pmatrix} \| \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

Idee: durch lexikographische Ordnung total machen! **Aber:** Degenerierte Bäume!

Motivation

Gesucht: Datenstruktur zum Einfügen, Löschen und Auffinden von Elementen
Problem: Totale Ordnung nicht auf natürliche Art vorhanden
Beispiel: Vergleich von Bilddaten, Musikdaten, komplexen Datensätzen

⇒ Lineare Liste!

Funktioniert, jedoch mit **ungünstiger Laufzeit:**
1. Feststellen, dass Element nicht vorhanden: N Vergleiche auf Gleichheit
2. Vorhandenes Element auffinden: im Mittel (N+1) / 2 Vergleiche

(bei diskreter Gleichverteilung)

⇒ Alternative Suchverfahren notwendig ⇒ ***Hashing***

Idee

Jedes Element e bekommt einen **numerischen** „Stempel“ h(e),

1. der sich aus dem **Dateninhalt** von e berechnet
2. Aufteilen der Menge von N Elementen in **M disjunkte Teilmengen**, wobei M die Anzahl der möglichen Stempel ist
   → Elemente mit **gleichem Stempel** kommen in **dieselbe Teilmenge**
3. Suchen nach Element e nur noch in Teilmenge für Stempel h(e)

Laufzeit (**Annahme:** alle M Teilmengen ungefähr gleich groß)

a) Feststellen, dass Element nicht vorhanden: N / M Vergleiche auf Gleichheit

b) Vorhandenes Element auffinden: im Mittel (N / M +1) / 2 Vergleiche

(bei diskreter Gleichverteilung)

⇒ deutliche Beschleunigung!

Grobentwurf

Jedes Element e ∈ E bekommt einen **numerischen** „Stempel“ h(e),
der sich aus dem **Dateninhalt** von e berechnet

> Funktion h: E → { 0, 1, …, M – 1 } heißt **Hash-Funktion** (*to hash*: zerhacken)
> Anforderung: sie soll zwischen 0 und M – 1 gleichmäßig verteilen

1. Elemente mit **gleichem Stempel** kommen in **dieselbe Teilmenge**

2. > M Teilmengen werden durch M lineare Listen realisiert (ADT Liste),
   > Tabelle der Größe M enthält für jeden Hash-Wert eine Liste

3. Suchen nach Element e nur noch in Teilmenge für Stempel h(e)

> Suche nach e → Berechne h(e); h(e) ist Index für Tabelle[h(e)] (vom Typ Liste)
> Suche in dieser Liste nach Element e

Grobentwurf

Weitere Operationen auf der Basis von „Suchen“

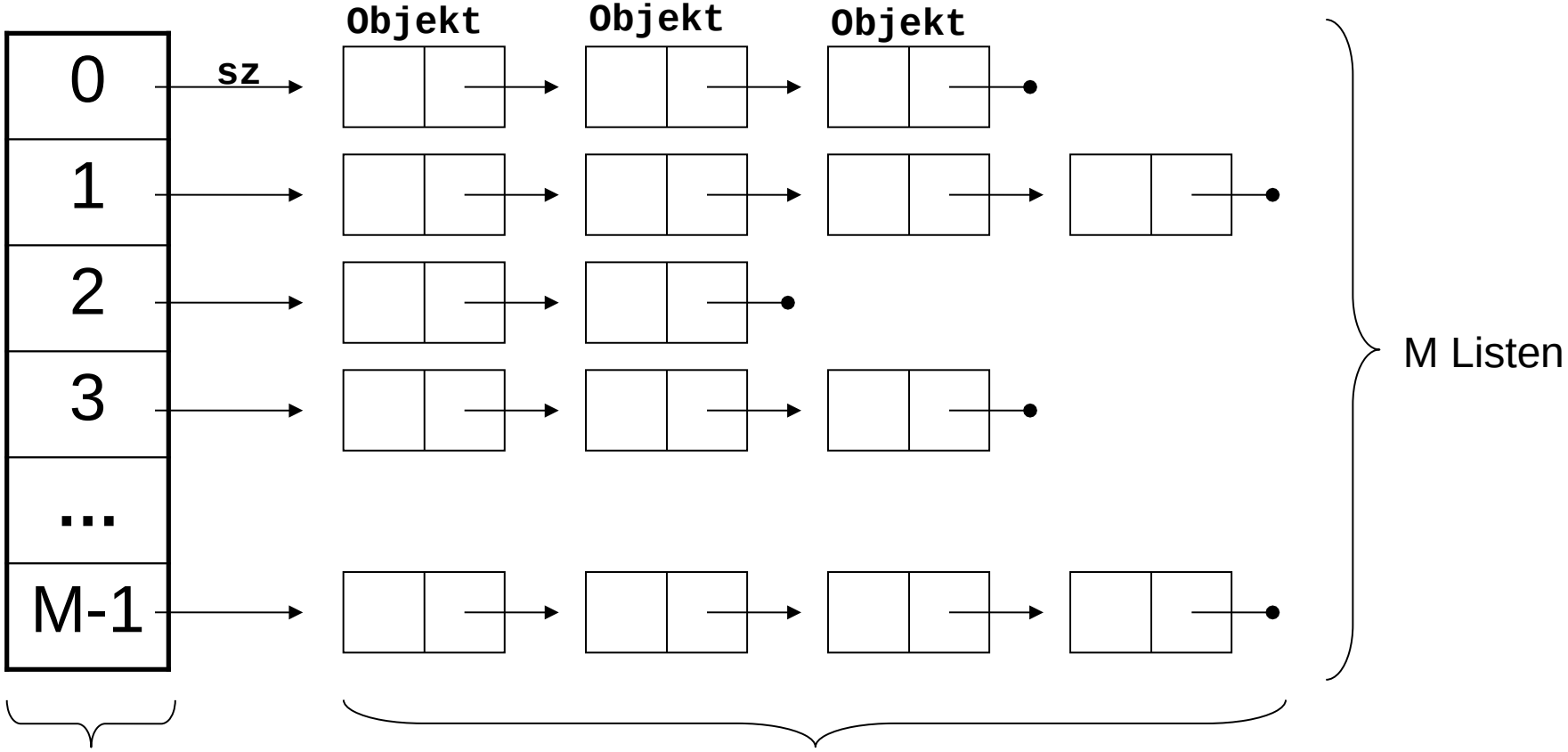
Einfügen von Element e

→ Suche nach e in Liste für Hash-Werte h(e)
Nur wenn e **nicht** in dieser Liste, dann am Ende der Liste einfügen

- **Löschen** von Element e
- → Suche nach e in Liste für Hash-Werte h(e)
  Wenn e in der Liste **gefunden** wird, dann aus der Liste entfernen

Auch denkbar: **Ausnahme werfen**, falls
einzufügendes Element schon existiert oder zu löschendes Element nicht vorhanden

Grobentwurf



Was ist zu tun?

Wähle Datentyp für die Nutzinformation eines Elements

⇒ **hier:** realisiert als Schablone

1. Realisiere den ADT **`Liste`** zur Verarbeitung der Teilmengen

   ⇒ Listen kennen und haben wir schon; jetzt nur ein paar Erweiterungen

2. Realisiere den ADT **`HashTable`**

   ⇒ Verwende dazu den ADT **`Liste`** und eine Hash-Funktion

3. Konstruiere eine Hash-Funktion h: E → { 0,1, …, M – 1}

   ⇒ **Kritisch** wg. Annahme, dass h() gleichmäßig über Teilmengen verteilt!

```
template<typename T> class Liste {
public:
  Liste();
  Liste(const Liste& liste);
  void append(const T& x);
  void prepend(const T& x);
  bool empty();
  bool is_elem(const T& x);
  void clear();
  void remove(const T& x);
  void print();
  ~Liste();
protected:
  struct Objekt {
    T      data;
    Objekt *next;
  } *sz, *ez;
  void clear(Objekt *obj);
  Objekt *remove(Objekt *obj, const T& x);
  void print(Objekt *obj);
};
```

ADT Liste

öffentliche Methoden, z.T. überladen

privater lokaler Datentyp

private rekursive Funktionen

ADT Liste

```
template<typename T> Liste<T>::Liste()
: sz(nullptr), ez(nullptr) {
}
```

Konstruktor

```
template<typename T> Liste<T>::~Liste() {
  clear();
}
```

Destruktor

```
template<class T> void Liste<T>::clear() {
  clear(sz);
  sz = ez = nullptr;
}
```

public clear : gibt Speicher frei, initialisiert zu leerer Liste

```
template<typename T>
void Liste<T>::clear(Objekt *obj) {
  if (obj == nullptr) return;
  clear(obj->next);
  delete obj;
}
```

private Hilfsfunktion von ***public clear*** löscht Liste rekursiv!

ADT Liste

öffentliche Methode:

```
template<typename T> void Liste<T>::remove(const T& x){
  sz = remove(sz, x); if (sz == nullptr) ez = nullptr;
}
```

private überladene Methode:

```
template<typename T>
typename Liste<T>::Objekt* Liste<T>::remove(
                                    Objekt *obj, const T& x) {
  if (obj == nullptr) return nullptr; // oder: Ausnahme!
  if (obj->data == x) {
    Objekt *tmp = obj->next;          // Zeiger retten
    delete obj;                       // Objekt löschen
    return tmp;                       // Zeiger retour
  }
  obj->next = remove(obj->next, x);   // Rekursion
  if (obj->next == nullptr) ez = obj;
  return obj;   }
```

ADT Liste

öffentliche Methode:

```
template<typename T> void Liste<T>::print()
{
  print(sz);
}
```

private überladene Methode:

```
template<typename T>
void Liste<T>::print(Objekt *obj) {
  static int cnt = 1;   // counter
  if (obj != nullptr) {
    cout << obj->data;
    cout << (cnt++ % 6 ? "\t" : "\n");
    print(obj->next);
  }
  else {
    cnt = 1;
    cout << "(end of list)" << endl;
  }
}
```

← Speicherklasse **`static`** : Speicher wird nur einmal angelegt

ADT HashTable

```
template<typename T> class HashTable {
private:
  Liste<T> *table;
protected:
  unsigned int maxBucket;
public:
  HashTable(int aMaxBucket);
  virtual int Hash(T& aElem) = 0;        // rein virtuell!
  bool Contains(T& aElem) {
    return table[Hash(aElem)].is_elem(aElem); }
  void Delete(T& aElem) {
    table[Hash(aElem)].remove(aElem); }
  void Insert(T& aElem) {
    table[Hash(aElem)].append(aElem); }
  void Print();
  ~HashTable();
};
```

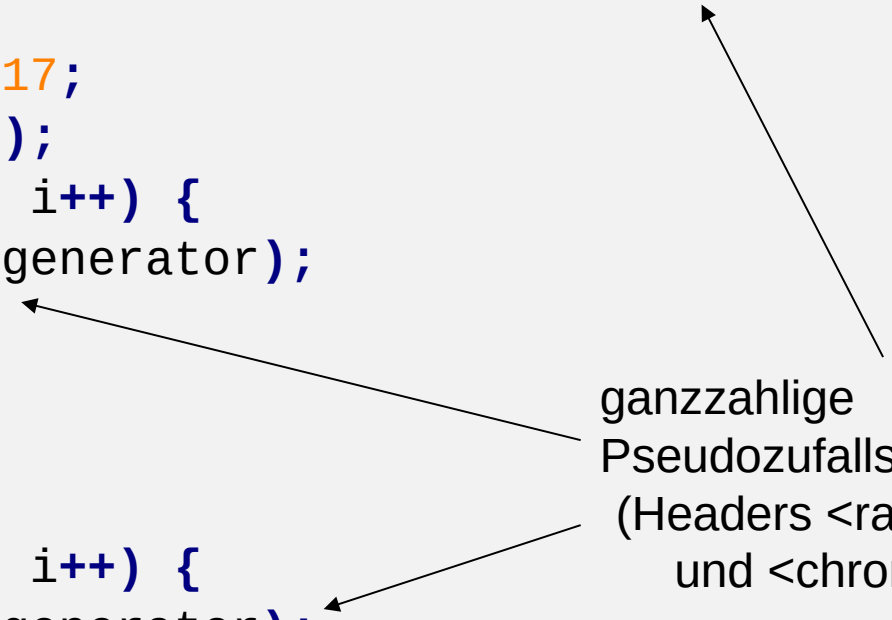
ADT HashTable

```
template<typename T>
HashTable<T>::HashTable(int aMaxBucket):maxBucket(aMaxBucket) {
  if (maxBucket < 2) throw "invalid bucket size";
  table = new Liste<T>[maxBucket];
}

template<typename T>
HashTable<T>::~HashTable() {
  delete[] table;
}

template<typename T>
void HashTable<T>::Print() {
  for (unsigned int i = 0; i < maxBucket; i++) {
    cout << "\nBucket " << i << " :\n";
    table[i].print();
  }
}
```

ADT HashTableInt

```
class HashTableInt : public HashTable<int> {
public:
  HashTableInt(int aMaxBucket) : HashTable(aMaxBucket) {}
  int Hash(int& aElem) { return aElem % maxBucket; }
};
```

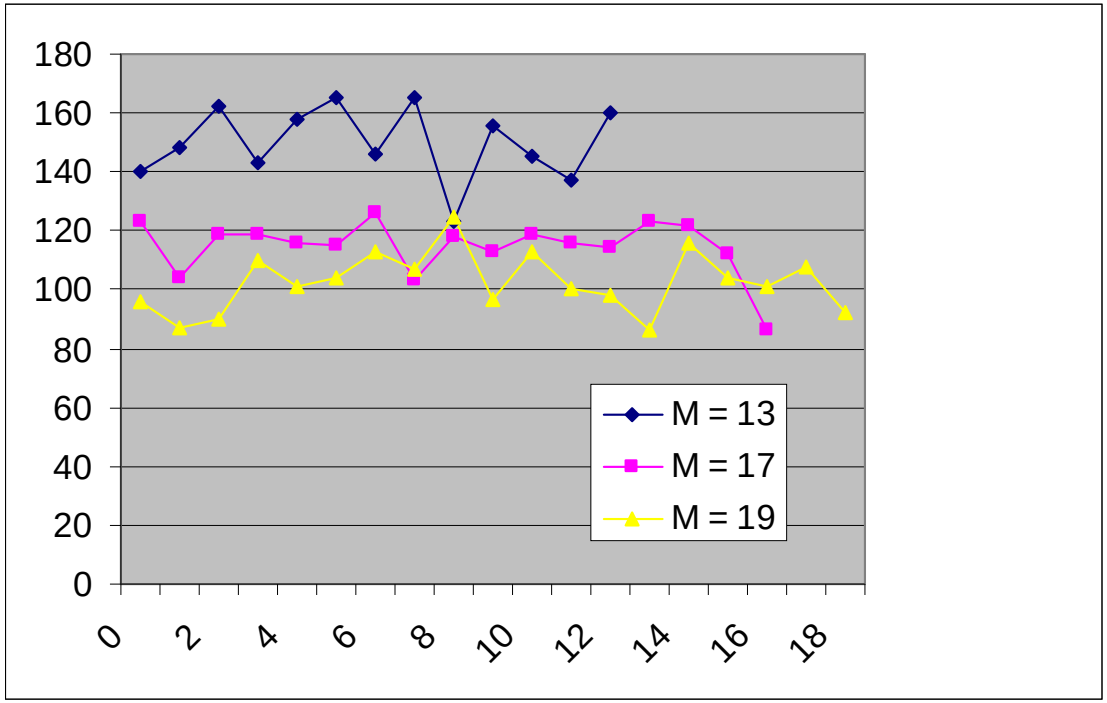
```
int main() {
  default_random_engine generator(
    chrono::system_clock::now().time_since_epoch().count());
  uniform_int_distribution<int> distribution(1,10000);

  unsigned int maxBucket = 17;
  HashTableInt ht(maxBucket);
  for (int i = 0; i < 2000; i++) {
    int num = distribution(generator);
    ht.Insert(num);
  }

  int hits = 0;
  for (int i = 0; i < 2000; i++) {
    int num = distribution(generator);
    if (ht.Contains(num)) hits++;
  }
  cout << "Treffer: " << hits << endl;
}
```



ganzzahlige Pseudozufallszahlen (Headers <random> und <chrono>)

Ausgabe (z.B.): **`Treffer: 367`**

ADT HashTable: Verteilung von 2000 Zahlen auf M Buckets

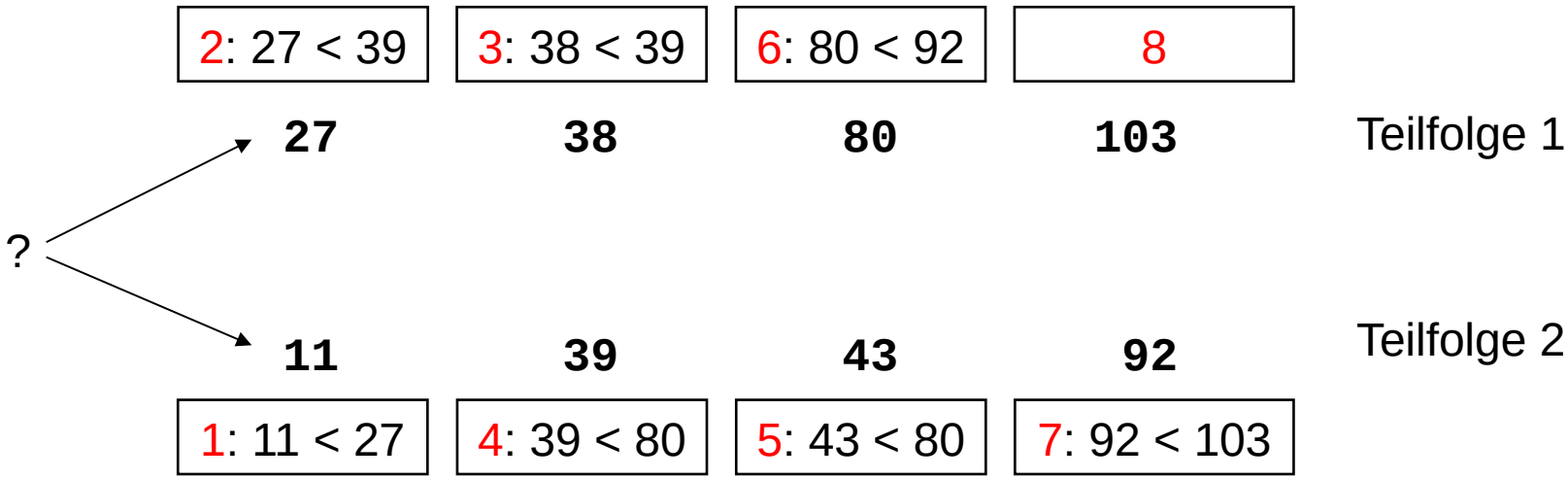


M	Mittel-wert	Std.-Abw.
13	149	13,8
17	114	8,1
19	102	6,7

⇒ Hash-Funktion ist wohl OK

Mergesort

Beobachtung:
Sortieren ist einfach, wenn man zwei sortierte Teilfolgen hat.



Mergesort

- **Eingabe:** unsortiertes Feld von Zahlen
- **Ausgabe:** sortiertes Feld
- Algorithmisches Konzept: **„Teile und herrsche“** (*divide and conquer*)
  - Zerlege Problem solange in Teilprobleme bis Teilprobleme lösbar
  - Löse Teilprobleme
  - Füge Teilprobleme zur Gesamtlösung zusammen

Hier:

1. Zerteile Feld in Teilfelder bis Teilproblem lösbar (→ bis Feldgröße = 2)
2. Sortiere Felder der Größe 2 (→ einfacher Vergleich zweier Zahlen)
3. Füge sortierte Teilfelder durch Mischen zu sortierten Feldern zusammen

Mergesort

Programmentwurf

Annahme:
Feldgröße ist
Potenz von 2

- Teilen eines Feldes → einfach!

1. Sortieren
   a) eines Feldes der Größe 2 → einfach!
   b) eines Feldes der Größe > 2 → rekursiv durch Teilen & Mischen
2. Mischen → nicht schwer!

Mergesort: Version 1

```
void Msort(int const size, int a[]) {
  if (size == 2) {  // sortieren
    if (a[0] > a[1]) Swap(a[0], a[1]);
    return;
  }
  // teilen
  int k = size / 2;
  Msort(k, &a[0]);
  Msort(k, &a[k]);
  // mischen
  Merge(k, &a[0], &a[k]);
}
```

sortieren (einfach)

sortieren durch Teilen & Mischen

```
void Swap(int& a, int& b) {
  int c = b; b = a; a = c;
}
```

Werte vertauschen per Referenz

Mergesort: Version 1

```
void Merge(int const size, int a[], int b[]) {
  int* c = new int[2*size];
  // mischen
  int i = 0, j = 0;
  for (int k = 0; k < 2 * size; k++)
    if ((j == size) || (i < size && a[i] < b[j]))
      c[k] = a[i++];
    else
      c[k] = b[j++];
  // umkopieren
  for (int k = 0; k < size; k++) {
    a[k] = c[k];
    b[k] = c[k+size];
  }
  delete[] c;
}
```

dynamischen Speicher anfordern

dynamischen Speicher freigeben

Mergesort: Version 1

```
void Print(int const size, int a[]) {
  for (int i = 0; i < size; i++) {
    cout << a[i] << "\t";
    if ((i+1) % 8 == 0) cout << endl;
  }
  cout << endl;
}
int main() {
  default_random_engine gen(…);
  uniform_int_distribution<int> distri(1,10000);
  int const size = 32;
  int a[size];
  for (int k = 0; k < size; k++) a[k] = distri(gen);
  Print(size, a);
  Msort(size, a);
  Print(size, a);
}
```

Hilfsfunktion

Programm zum Testen

Mergesort: Version 1

Ausgabe:

```
6887    6812    3408    2927    3554    750     6440    4764
2980    6534    1331    5168    6336    9404    7169    7622
2853    7654    8780    1074    4465    3883    1885    323
150     9664    1873    5029    6423    5373    6258    6374

150     323     750     1074    1331    1873    1885    2853
2927    2980    3408    3554    3883    4465    4764    5029
5168    5373    6258    6336    6374    6423    6440    6534
6812    6887    7169    7622    7654    8780    9404    9664
```

OK, funktioniert für **`int`** … was ist mit **`char, float, double`** … ?

⇒ **Idee:** Schablonen!

Mergesort: Version 2

```
template <class T> void Msort(int const size, T a[])
{
  if (size == 2) {  // sortieren
    if (a[0] > a[1]) Swap<T>(a[0], a[1]);
    return;
  }
  // teilen
  int k = size / 2;
  Msort<T>(k, &a[0]);
  Msort<T>(k, &a[k]);
  // mischen
  Merge<T>(k, &a[0], &a[k]);
}
```

```
template <class T> void Swap(T& a, T& b) {
    T c = b; b = a; a = c;
}
```

Mergesort: Version 2

```
template <class T> void Merge(int const size, T a[], T b[]) {
  T* c = new T[2*size];
  // mischen
  int i = 0, j = 0;
  for (int k = 0; k < 2 * size; k++) {
    if ((j == size) || (i < size && a[i] < b[j]))
      c[k] = a[i++];
    else
      c[k] = b[j++];
  }
  // umkopieren
  for (int k = 0; k < size; k++) {
    a[k] = c[k];
    b[k] = c[k+size];
  }
  delete[] c;
}
```

Mergesort: Version 2

```
template <class T> void Print(int const size, T a[]) { … }
```

```
int main() {
    default_random_engine gen(

chrono::system_clock::now().time_since_epoch().count());
    uniform_real_distribution<float> distri(1.0,1000.0);

    int const size = 32;
    float a[size];

    for (int k = 0; k < size; k++) a[k] = distri(gen);

    Print<float>(size, a);
    Msort<float>(size, a);
    Print<float>(size, a);
}
```

Mergesort: Version 2

Ausgabe:

```
977.659 142.785 365.544 23.6122 423.959 784.038 696.633 206.966
133.042 452.624 48.4892 949.978 445.117 751.544 16.9055 591.667
278.982 726.154 863.679 557.759 817.62  673.558 993.751 864.057
915.418 12.9085 347.95  23.4945 443.875 855.105 306.114 182.264

12.9085 16.9055 23.4945 23.6122 48.4892 133.042 142.785 182.264
206.966 278.982 306.114 347.95  365.544 423.959 443.875 445.117
452.624 557.759 591.667 673.558 696.633 726.154 751.544 784.038
817.62  855.105 863.679 864.057 915.418 949.978 977.659 993.751
```

Mergesort: Version 2

Schablone instanziiert mit Typ **`string`** funktioniert auch.

Schablone instanziiert mit Typ **`Complex`** **funktioniert nicht**! Warum?

Vergleichsoperatoren sind nicht überladen für Typ **`Complex`**!

in **`Msort`**:
```
if (a[0] > a[1]) Swap<T>(a[0], a[1]);
```

in **`Merge`**:
```
if ((j == size) || (i < size && a[i] < b[j]))
```

Entweder Operatoren überladen oder überladene Hilfsfunktion (z.B. **`Less`**):

```
bool Less(Complex &x, Complex &y) {
  if (x.Re() < y.Re()) return true;
  return (x.Re() == y.Re() && x.Im() < y.Im());
}
```

hier: lexikographische Ordnung