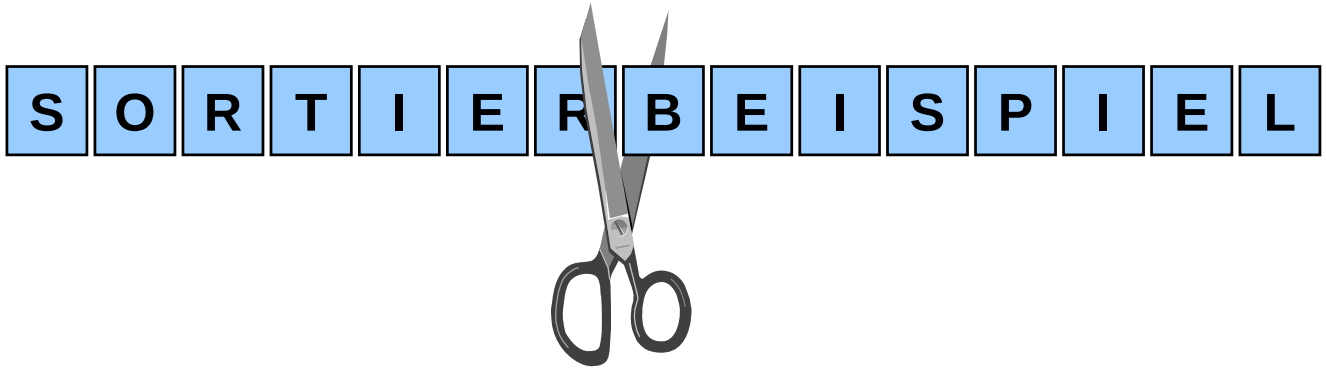
Sortierverfahren 2



- Sie kennen das Prinzip: "Teile und Herrsche"
- Sie kennen drei schnelle Sortierverfahren
  - Quick Sort
  - Distribution Sort
  - Merge Sort
- Sie wissen, wie die Laufzeit abgeschätzt wird
- Sie wissen, wie sich die Algorithmen bei vorsortierten Daten verhalten
- Sie wissen wie man Datensätze sortiert, die die Grösse des Hauptspeichers überschreiten
  - Sortier-Misch Verfahren
- Sie wissen, wann Sie welchen Sortieralgorithmus wählen müssen.

Das Prinzip "Teile und Herrsche"

Teile und Herrsche
Andere Bezeichnungen dieses Prinzips:
Divide et impera.
Divide and conquer.

- Divide et impera (Julius Cäsar).
  Divide and rule.

- Zerlege das Problem in kleinere Teile
- Löse die so erhaltenen Teilprobleme
- Füge die Teillösungen wieder zu einem Ganzen zusammen

Teile und Herrsche bei Sortieralgorithmen

- Sortieralgorithmen nach dem Prinzip Teile und Herrsche.

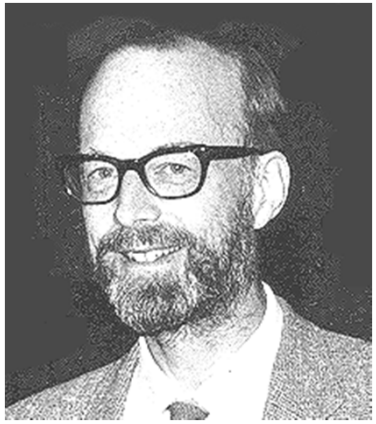
```
if (Menge der Datenobjekte klein genug)
       Ordne sie direkt;
else {
       Teile: Zerlege die Menge in Teilmengen;
       Herrsche: Sortiere jede der Teilmengen;
       Vereinige: Füge die Teilmengen geordnet zusammen;
      }
```

- Solche Algorithmen sind typischerweise rekursiv:

```
Sort (Menge a)
       if (Menge der Datenobjekte klein genug)
              Ordne sie direkt;
       else {
              Zerlege in zwei Teilmengen
              Sort(Teilmenge1); Sort(Teilmenge2)
              Füge Teilmengen geordnet zusammen
       }
}
```

Bei der Zerlegung sollten die Teilmengen möglichst gleich gross sein.

Quick-Sort (1/5)



- Wurde **1960** von dem britischen Informatiker **C.A.R. Hoare** entdeckt.

- Entstehung
  - 1960 waren noch keine schnellen Sortieralgorithmen bekannt.
  - man versuchte damals Sortierverfahren durch raffinierte Assemblerprogrammierung zu beschleunigen.

- Quick-Sort mit naheliegenden Verbesserungen ist
  - einer der schnellsten bekannten allgemeinen Sortieralgorithmen
  - theoretisch gut verstanden.

Hoare zeigte dadurch, dass es sinnvoller sein kann, nach **besseren Algorithmen** zu suchen, als vorhandene Algorithmen durch ausgefeilte Programmierung zu beschleunigen.

Quick-Sort (2/5)

- Die Grundidee besteht darin, das vorgegebene Problem nach dem bereits genannten Motto Teile und Herrsche in einfachere Teilaufgaben zu zerlegen.
  - nehme irgendeinen Wert W der Teil von A ist – zum Beispiel den mittleren
  - konstruiere eine Partitionierung des Sortierfeldes **A** in Teilmengen $\mathbf{A_1}$ und $\mathbf{A_2}$ mit folgenden Eigenschaften:

- $\mathbf{A} = \mathbf{A_1} \cup \mathbf{A_2} \cup \{W\}$
- Alle Elemente von $\mathbf{A_1}$ sind $\leq W$ (aber evtl. noch unsortiert).
- Alle Elemente von $\mathbf{A_2}$ sind $\geq W$ (aber evtl. noch unsortiert).

Wenn jetzt $\mathbf{A_1}$ und $\mathbf{A_2}$ sortiert werden, ist das Problem gelöst.

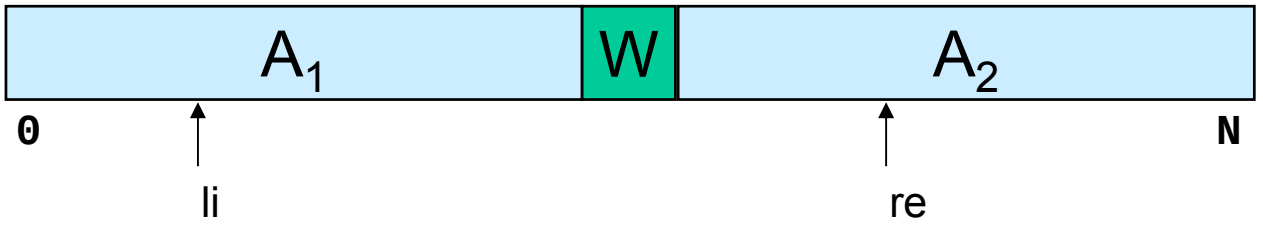
Quick-Sort (3/5)

```
Methode Sortiere (A){
      Konstruiere die Partition A = A1 W A2;
      Sortiere A1;
      Sortiere A2;
 }
```

noch zu lösendes Problem:

- Partitionen erzeugen
- finden eines Wertes W, so dass $\mathbf{A_1}$ und $\mathbf{A_2}$ möglichst gleich gross sind.

Quick-Sort (4/5)



- Die Konstruktion der Partition erfolgt durch:
  - wähle ein Element W (Pivot)
  - suche von links ein Element, das auf falscher Seite ist, d.h. A[li] ≥ W
  - suche von rechts ein Element, das auf falscher Seite ist, A[re] ≤ W
  - vertauschen von A[li] und A[re]
  - wiederhole obige Schritte bis li und re sich kreuzen, d.h. li > re.
- Achtung: Der Ort des Pivot-Elements bleibt im Allgemeinen nicht fix! Am Schluss hat man einfach die Garantie, dass alle Elemente A[i], i≤re, kleiner und alle Elemente A[j], j≥li, grösser oder gleich W sind.
- Die folgende Funktion wird aufgerufen mit:

```
static void quickSort(char[] A){
        int hi = A.length-1;
        rekQuickSort(A, 0, hi);
}
```

Quick-Sort (5/5)

```
void rekQuickSort(char[] A, int lo, int hi) {
  int li  = lo;
  int re  = hi;
  int mid = (li+re)/2;
  char w = A[mid];
  do{
      while (A[li] < w) li++;
      while (w < A[re]) re--;
      if (li <= re) {swap(A, li, re); li++; re--;}
  } while (li <= re);

  if (lo < re) rekQuickSort(A, lo, re);
  if (li < hi) rekQuickSort(A, li, hi);
 }
}
```

Demo

partitionieren

rekursiv aufrufen

Demo

Quick-Sort: Wahl des Pivots (1/2)

- W wird auch Pivot genannt
  - Für die Effizienz von QuickSort wäre ein Element ideal, das A in **zwei gleich grosse Teile** partitioniert, d.h., dass gleich viele Werte grösser wie kleiner als W sind (sog. *Median*)

- Genaue Bestimmung des Medianes aufwendig
  → Laufzeitvorteil von QuickSort ginge wieder verloren.

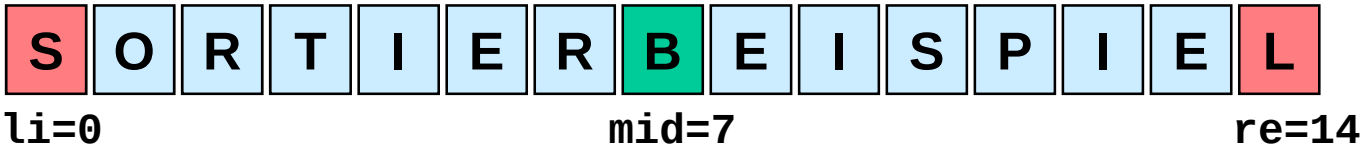
W wird lediglich geschätzt

- folgende Schätzungen sind möglich
  - A[li] das (der Position nach) linke Element von A;
  - A[re] das (der Position nach) rechte Element von A;
  - A[mid] das (der Position nach) mittlere Element von A mit mid = (li+re)/2

- Strategie 1: Nimm eines der drei Elemente
- Strategie 2: Nimm das (wertmässig) mittlere der drei Elemente

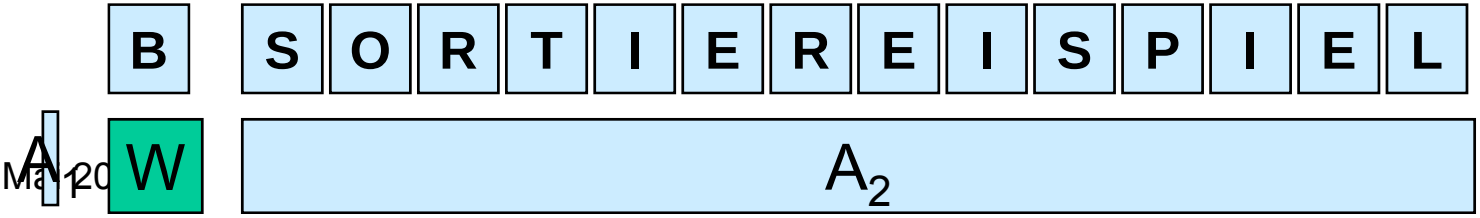
Quick-Sort: Wahl des Pivots(2/2)

- Die Wahl des linken oder des rechten Elements ist bei teilweise vorsortierten Daten schlecht.
- Normalerweise wird das mittlere Element genommen. Dieses ist aber nicht unbedingt das Element, welches der Position nach in der Mitte liegt.
- Die Möglichkeit einer ungünstigen Verteilung der Daten, wenn:
  - durch die Partitionierung in eine Hälfte der Partition sehr viele und in die andere Hälfte sehr wenige Daten gelangen. Bei unserem Beispiel ergibt sich folgende Situation:



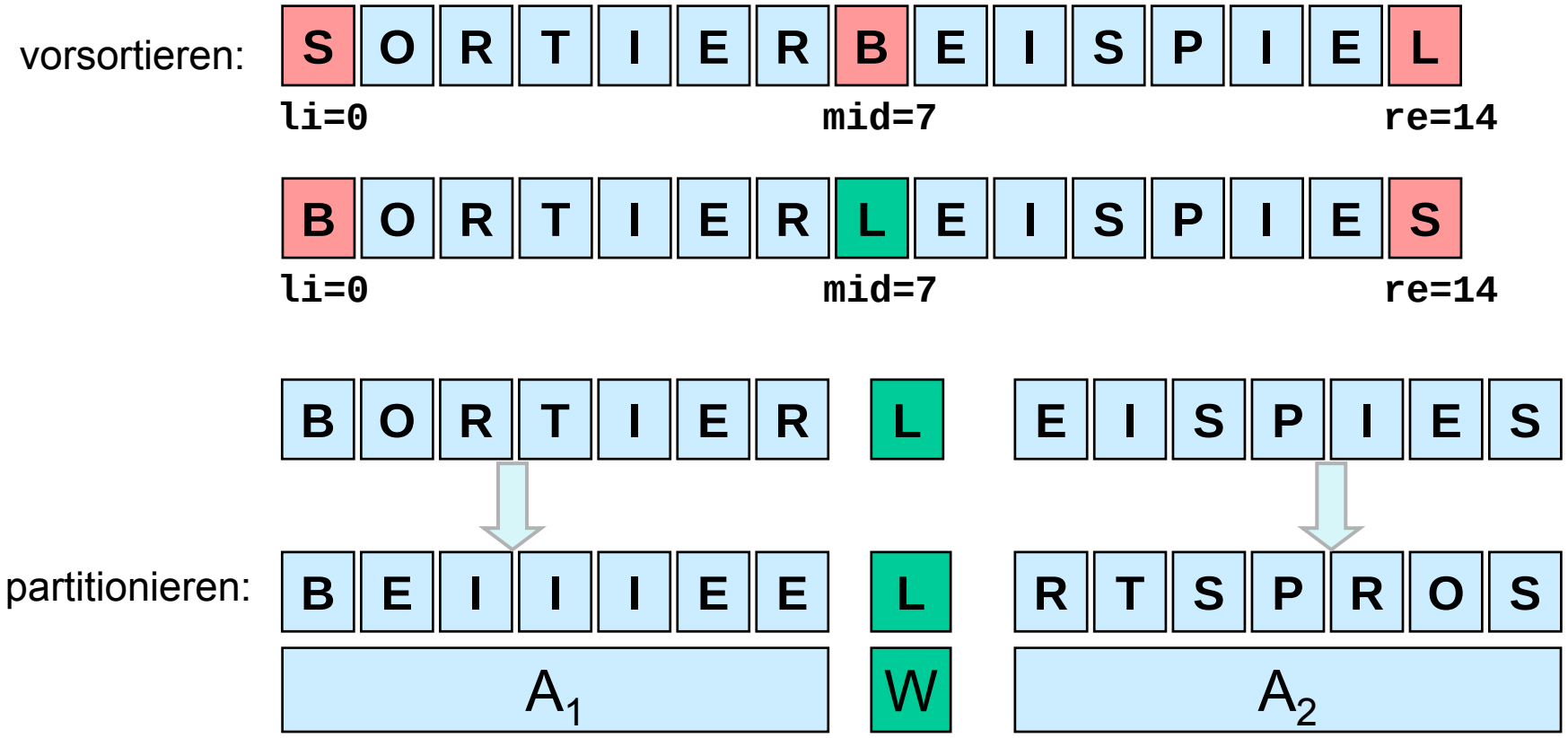
Bei der Wahl von A[mid] als Pivot-Element ergibt sich ungünstige Partitionierung:

A_1 ist leer und A_2 ist gerade mal ein Element kleiner als A.



Quick-Sort: Die Median Methode (1/3)

- Die drei Elemente A[li], A[mid] und A[re] werden vorsortiert und man nimmt das dem Werte nach **mittlere dieser drei Werte**.



Quick-Sort: Die Median Methode (2/3)

- Im Beispiel wird eine optimale Partionierung erreicht
  - **15** Elemente aufgespalten in zwei Partitionen mit je **7** Elementen

- **Vorsortierung** vermindert die Rekursionstiefe von QuickSort
  - rekursive Weitersortierung nur wenn wir mehr als drei Elemente haben

Quick-Sort: Die Median Methode (3/3)

vorsortieren

partitionieren

rekursiv aufrufen

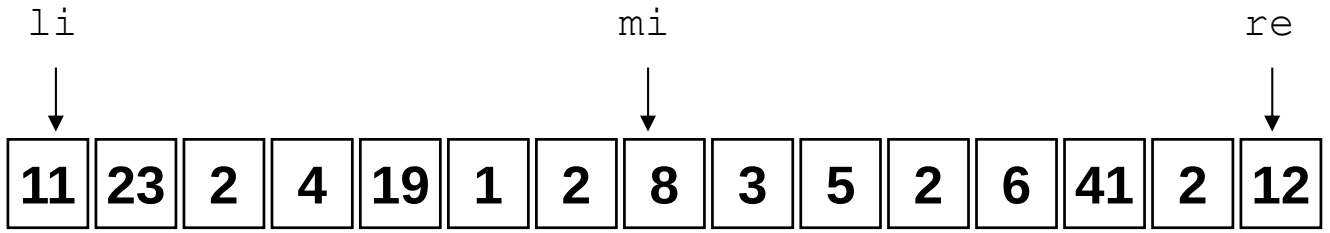
```
static void rekQuickSort(char[] A, int lo, int hi) {
  int li  = lo;
  int re  = hi;
  int mid = (li+re)/2;
  if (A[li]  > A[mid]) swap(A, li, mid);
  if (A[mid] > A[re])  swap(A, mid, re);
  if (A[li]  > A[mid]) swap(A, li, mid);

  if ((re - li) > 2){
    char w = A[mid];
    do{
      while (A[li] < w) li++;
      while (w < A[re]) re--;
      if (li <= re) {swap(A, li, re); li++; re--;}
    } while (li <= re);

  if (lo < re) rekQuickSort(A, lo, re);
  if (li < hi) rekQuickSort(A, li, hi);
 }
}
```

Aufgabe

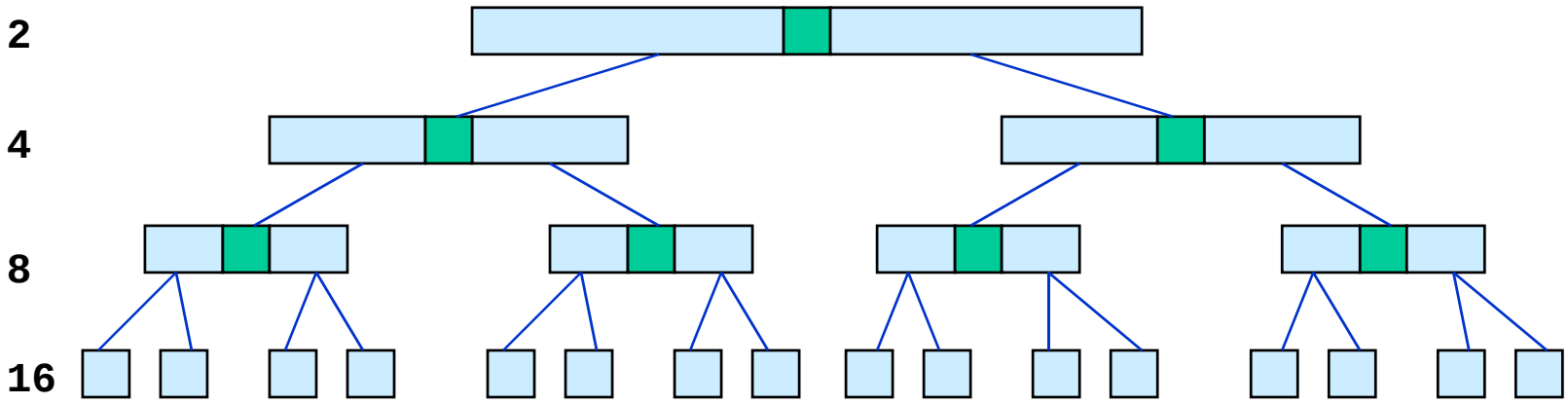
- Führen Sie die erste Partionierung mit dem Algorithmus auf der vorhergehenden Seite an dem folgenden Beispiel durch.



Quick-Sort: Aufwand (1/2)

- ein Bereich muss $\log_2(N)$ mal geteilt werden:
  - Es entsteht dabei ein binärer Partitionenbaum mit Tiefe **$\log_2(N)$**.
  - Der Aufwand, auf jeder Schicht diese komplett zu partitionieren, ist proportional zu **N**.
- der Gesamtaufwand ist somit proportional zu **$N*\log_2(N)$**.

- Die Ordnung von Quicksort ist somit **O(N*log(N))**,
  - wenn bei jeder Partitionierung eine gleichmässige Aufteilung der Daten erfolgt



Quick-Sort: Aufwand (2/2)

- ungünstigster Fall:
  - jeder Partitionierungs-Schritt ergibt eine leere Partition und den (nur um das Pivot-Element reduzierten) Rest
  - → Partitionierungs-Baum degeneriert zu einer Liste mit N Elementen.
- In diesem Fall ist der Aufwand proportional zu $O(N^2)$.

- Dies tritt jedoch nur in extra konstruierten Fällen auf
- Im Normalfall ist die Partitionierung bei QuickSort nahezu optimal

QuickSort stellt die **erste Wahl** dar, wenn grosse Mengen ungeordnete Daten sortiert werden müssen.

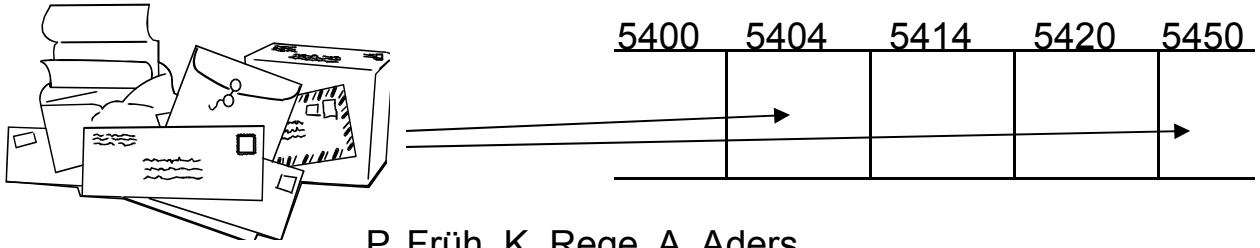
Distribution- oder BucketSort (1/2)

Die bisher diskutierten Sortieralgorithmen basieren auf den Operationen: Vergleichen zweier Elemente und ev. Vertauschen zweier Elemente (Swap).

- allgemein anwendbare Sortieralgorithmen haben bestenfalls O(N*log(N)) Aufwand, wie zum Beispiel **Quick-Sort**.
- es wurde **bewiesen**, dass Sortieralgorithmen, die auf Vergleichen aufbauen, mindestens den Aufwand O(N*log(N)) haben.

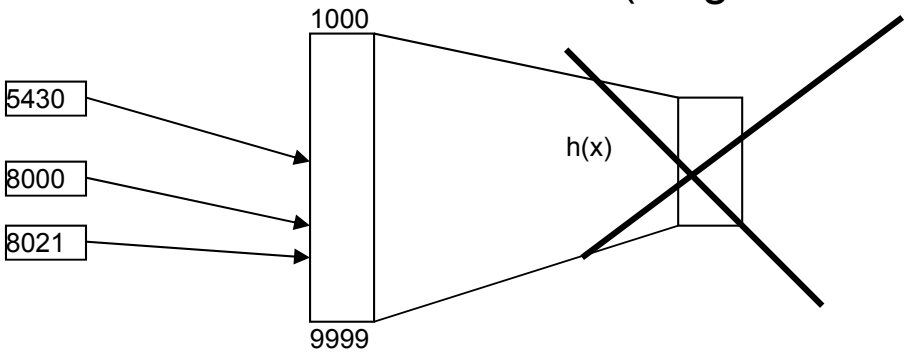
Im Gegensatz dazu kommt Distribution-Sort **ohne Vergleiche** aus.

- Die zu sortierenden Elemente werden
  - entsprechend dem Sortierschlüssel in Fächer verteilt: O(N)
  - zusammengetragen: O(N)
- Bsp: Briefe werden in die entsprechenden Fächer nach Plz. sortiert.



Distribution- oder BucketSort (2/2)

•Grundprinzip wie beim direkter Adressieren (vergl. Hashtable):



•Vorteile
- schneller geht's nicht
- linearer Algorithmus: die Komplexität ist also O(N)

•Nachteile
- Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
- Geht nur bei Schlüsseln, die einen **kleinen Wertebereich** haben oder auf einen **solchen abgebildet** werden können, ohne dass die **Ordnung** verloren geht.
- Hashing funktioniert nicht: wieso?

Distribution-Sort **mit Abstand der schnellste Algorithmus zum Sortieren**.Es handelt aber nicht um ein allgemein anwendbares Sortierverfahren.

Laufzeitvergleich von Sortieralgorithmen (1/3)

- Testbedingungen
  - Array mit N=10000, N=20000, N=30000 und N=40000 ganzer positiver Zahlen.
  - erst dann fallen nennenswerte Laufzeiten an.
  - mit Hilfe eines Zufallszahlengenerators wird ein Array mit der gewünschten Zahl von Elementen erzeugt (der Startwert immer gleich).

- Die Daten werden jeweils zweimal sortiert:
  - in einer zufälligen, unsortierten Reihenfolge
  - sortiert
  - (umgekehrt sortiert)

Einschub Zufallszahlen: Zufallszahlengeneratoren

- Es werden **Folgen von Zahlen** erzeugt, die möglichst viele Eigenschaften von Zufallszahlen besitzen (gleichverteilt, unkorrelliert → diskretes weisses Rauschen).
- Es wird eine Funktion bestimmt, bei der die Auswahl der nächsten Zahl scheinbar zufällig erfolgt: es kommt erst nach langer Zeit zu einer Wiederholung der Folge → **Pseudozufallszahlen**
- Lineare Kongruenz-Methode: $r_{i+1} = (a*r_i + c) \bmod p$

```
int z;
double random() {
    int a = 897; c = 2111; p = 123456;
    z = (a*z + c) % p;
    return z/p;
}
```

http://www.cs.pitt.edu/~kirk/cs1501/animations/Random.html

- gute Wahl von a,c und p ist schwierig; kann nur durch statistische Analysen überprüft werden
- Vorteil: es kann mehrmals die gleiche Folge erzeugt werden.
- Durch entsprechende Transformationen können aus gleichverteilten Zufallszahlen solche mit andern Verteilungen (z.B. normalverteilt) erzeugt werden.

Einschub Zufallszahlen: Zufallszahlen mit Java

- Die statische Methode `Math.random()` liefert gleichverteilte Zufallszahlen im Bereich [0..1[
- für beliebigen anderen Bereich: r = (int)(k*Math.random())
- Die Klasse `Random` erlaubt grössere Flexibilität.
- `Random(long seed)`
  Erzeugt einen Zufallszahlengenerator. *seed* ist der Startwert für die erzeugten Zufallszahlen. Damit kann mehrmals die gleiche Zufallssequenz erzeugt werden.
- `Random()`
  Erzeugt einen Zufallszahlengenerator. Der Startwert wird aus der aktuellen aktuellen Tageszeit in Millisekunden bestimmt.
- `nextInt(int n)` liefert eine pseudozufällige, gleichverteilte Integer-Zahl im Bereich [o..n[
- `nextDouble()` liefert eine pseudozufällige, gleichverteilte Double-Zahl im Bereich [o..1[
- `nextGaussian()` liefert eine pseudozufällige, Gauss-verteilte Double-Zahl mit Mittelwert 0.0 und Standard-Abweichung 1.0.

Einschub Zufallszahlen, Mersenne-Twister

- Der Java Zufallszahlengenerator arbeitet mit der linearen Kongruenz-Methode und einem 48 Bit Seed
- Der wohl beste Zufallszahlengenerator ist der 1997 entdeckte *Mersenne-Twister* (http://en.wikipedia.org/wiki/Mersenne_twister). Er hat eine Periode von $2^{19937} - 1$ und ist gleichverteilt bis zur 623. Dimension.
- Er wird unter anderem eingesetzt in Matlab.
- Die ursprüngliche Form des Mersenne-Twisters ist aber ungeeignet für kryptographische Applikationen.

Laufzeitvergleich von Sortieralgorithmen (2/3)

Die Ergebnisse in Sekunden der Laufzeitmessung bei zufällig geordneten Daten:

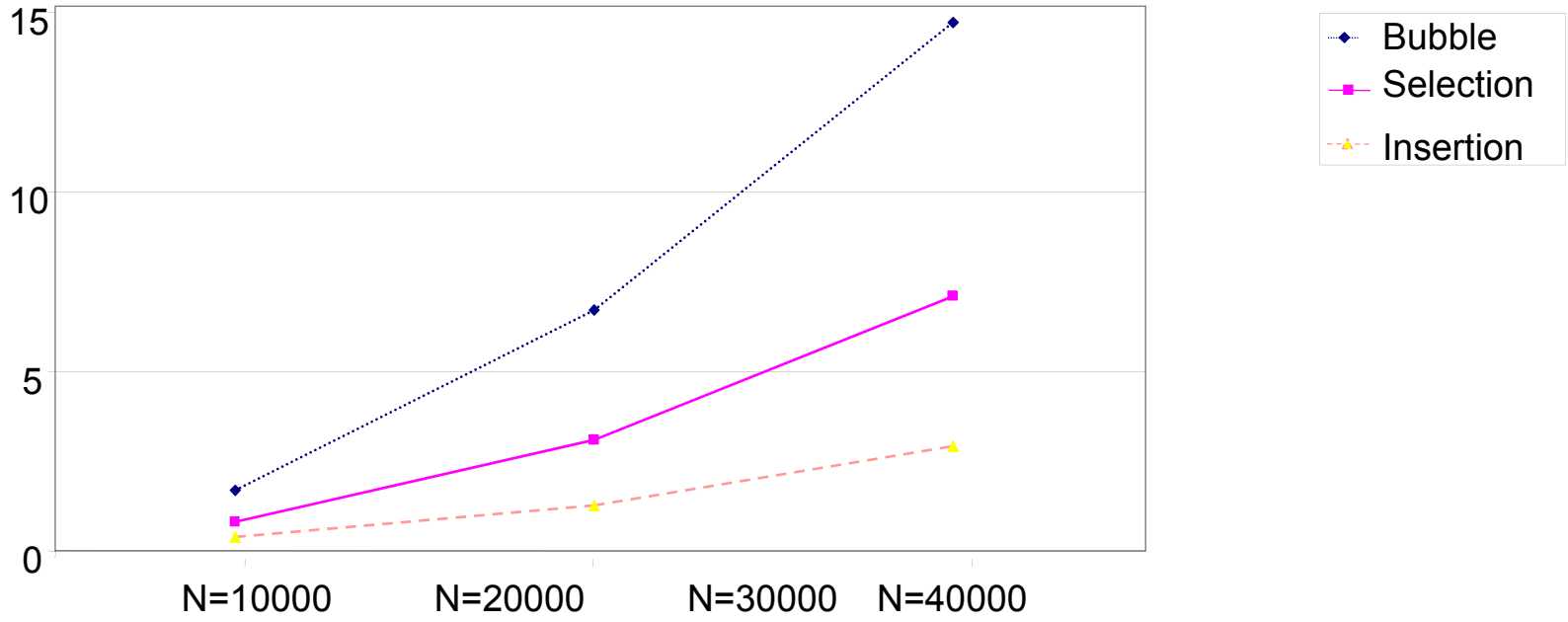
	N=10000	N=20000	N=30000	N=40000
Bubble	1.7	6.7	14.7	26.1
Selection	0.8	3.1	7.1	12.6
Insertion	0.4	1.25	2.9	6.7

... und bei vorsortierten Daten:

	N=10000	N=20000	N=30000	N=40000
Bubble	0	0	0	0
Selection	0.75	3.1	7.1	12.6
Insertion	0	0	0	0

Laufzeitvergleich von einfachen Sortieralgorithmen (3/3)

Laufzeitverhalten bei zufällig geordneten Daten:



Laufzeit bei vorsortierten Daten

- **BubbleSort** und **Insertion Sort** gut bei vorsortierten Daten
- folgende Situation in der Praxis sehr häufig
  - unsortierter Datenbestand wird einmalig eingebracht.
  - danach ändern sich in dem Datenbestand jeweils nur wenige Datensätze:
    - einige wenige Datensätze werden neu eingebracht.
    - einige wenige Datensätze werden geändert oder gelöscht.
- Eine Menge von vorsortierten Daten
- In dieser Situation sind **BubbleSort** und **InsertionSort** geeignete Algorithmen.

Laufzeitvergleich schneller Sortieralgorithmen (1/4)

- Testbedingungen
  - Array mit N=1000000, N=2000000, N=3000000 und N=4000000 ganzer positiver Zahlen (100 mal grösser als bei einfachen Sortieralgorithmen).
  - erst dann fallen nennenswerte Laufzeiten an.
  - mit Hilfe eines Zufallszahlengenerators wird ein Array mit der gewünschten Zahl von Elementen erzeugt.

- Die Daten werden jeweils dreimal sortiert:
  - in einer zufälligen, unsortierten Reihenfolge
  - sortiert
  - (umgekehrt sortiert)

Laufzeitvergleich schneller Sortieralgorithmen (2/4)

Laufzeitmessung bei zufällig geordneten Daten (in Sekunden):

	N=1000000	N=2000000	N=3000000	N=4000000
Quick	0.8	1.7	2.6	3.5
Distribution	0.6	1.2	1.8	2.5

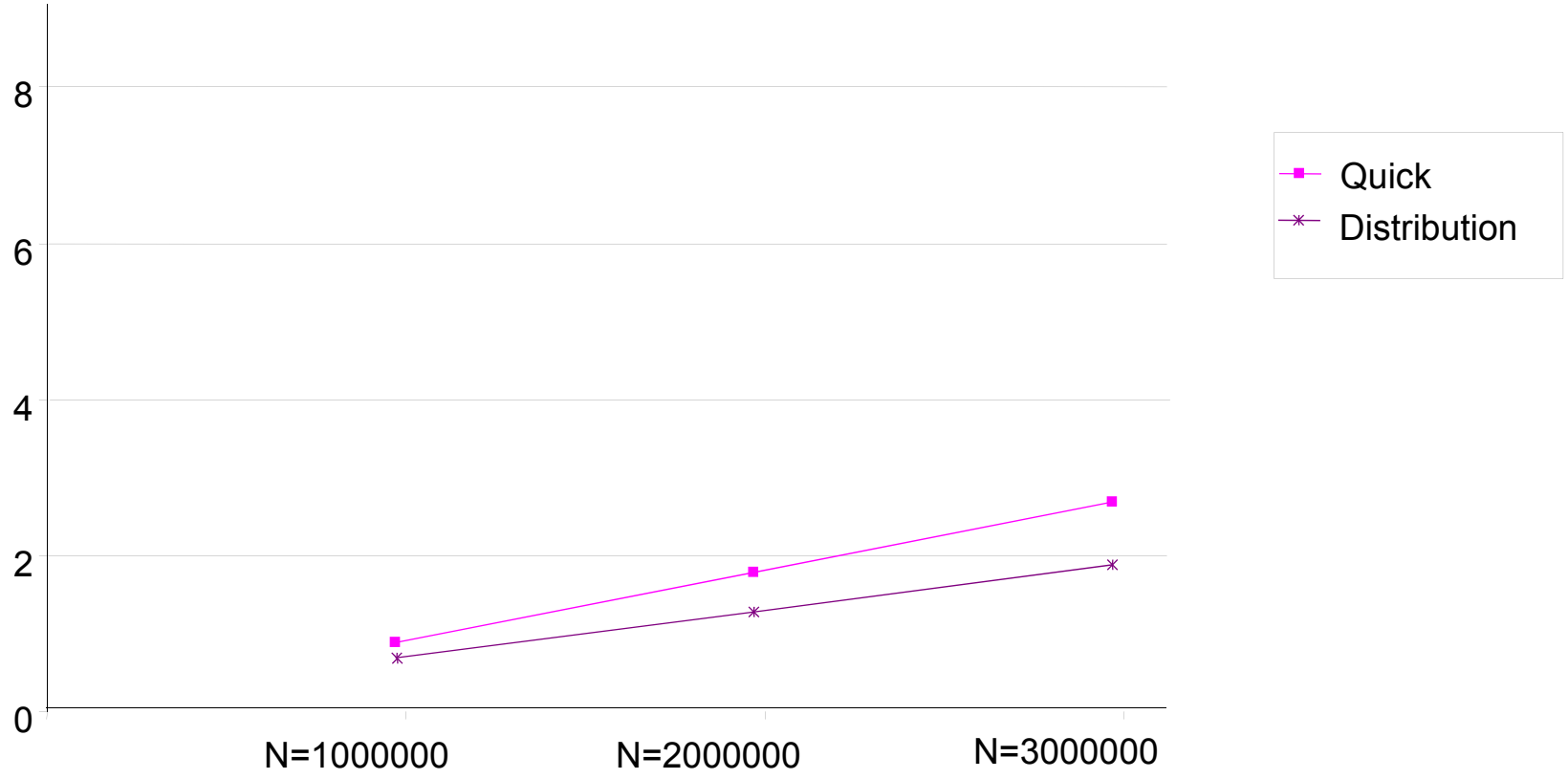
Laufzeitvergleich schneller Sortieralgorithmen (3/4)

... und bei vorsortierten Daten:

	N=1000000	N=2000000	N=3000000	N=4000000
Quick	0.4	0.8	1.3	1.7
Distribution	0.6	1.2	1.9	2.5

Laufzeitvergleich schneller Sortieralgorithmen (4/4)

zufällig geordnete Daten



Übung

- Ein O(N*log N) Sortieralgorithmus brauche 1 Sekunde für 10’000 Elemente. Wie lange braucht er für 100'000 Elemente?

Komplexitätsschranke für Sortieralgorithmen (Optimalitätssatz)

Satz: Ein Sortieralgorithmus, der darauf beruht, dass Elemente **untereinander verglichen** werden, kann bestenfalls eine Komplexität von O(N*log(N)) im Worst Case haben.

- Distribution-Sort fällt nicht darunter, da er
  - **nicht** auf dem Vergleich von Elementen untereinander beruht.
  - auf **Verteilen** und **Zusammentragen** von Datensätzen mit Hilfe von Fächern basiert.

Externe Sortierverfahren: Zwei-Phasen Sortieren

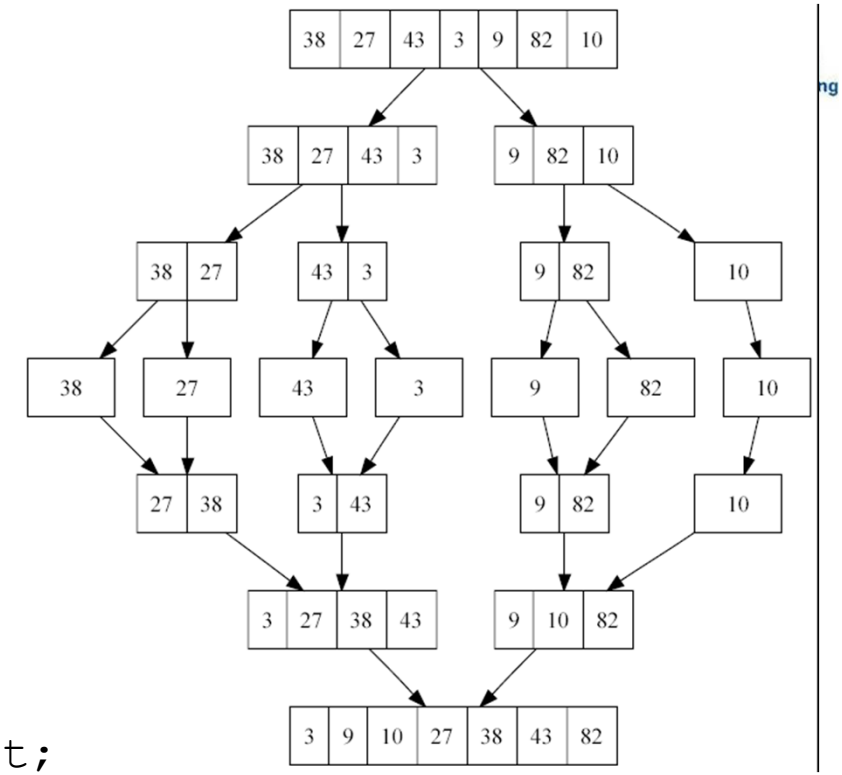
- Beim externem Sortieren liegen die Daten in einer Datei auf der Festplatte. Zwei Arten des Zugriffs sind möglich:
  - sequentieller Zugriff
  - der beliebige Zugriff auf die Elemente wäre zwar möglich, ergäbe aber einen grossen Effizienzverlust (siehe Vorlesung BS)

- Modell:
  - Magnetband, das nur sequentiell gelesen werden kann
  - nur jeweils ein Teil der Daten passen in den Hauptspeicher

Merge-Sort

2 Schritte:

- Aufteilen
- Mischen

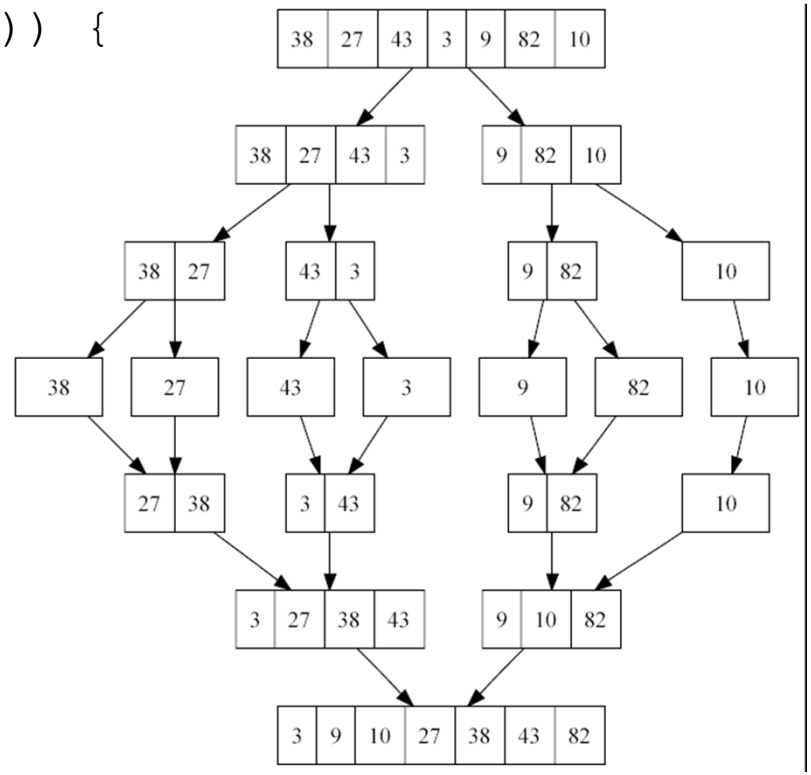



```
List mergesort (List list) {
  List left, right, result;
  if (list.length() ≤ 1) return list;
  else {                    // Aufteilen
    int m = list.length() / 2;
    for (int i=0; i<m; i++) left.add(list.item(i);
    for (int i=m; i<list.length(); i++) right.add(list.item(i));
    left = mergesort(left);
    right = mergesort(right);
    result = merge (left, right); // Mischen
    return result;
  }
}
```

Merge-Sort

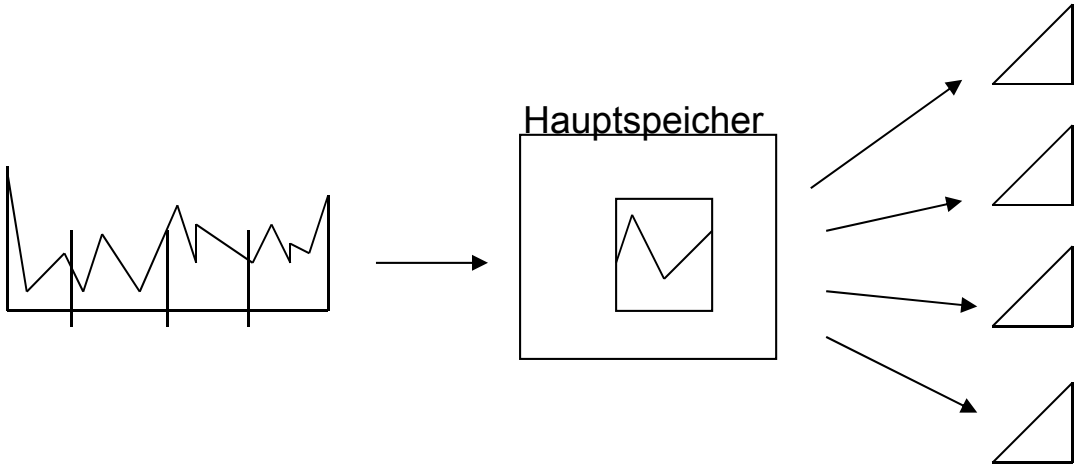
Demo

```
List merge (List left, List right) {
  List result;
  while (left.length() > 0 && right.length() > 0){
    if (left.get(0) ≤ right.get(0)) {
      result.add(left.get(0));
      left.remove(0);
    } else {
      result.add(right.get(0));
      right.remove(0);
    }
  }
  if (left.length() > 0)
    result.addAll(left);
  else result.addAll(right);
  return result;
}
```



1. Phase: Aufteilen

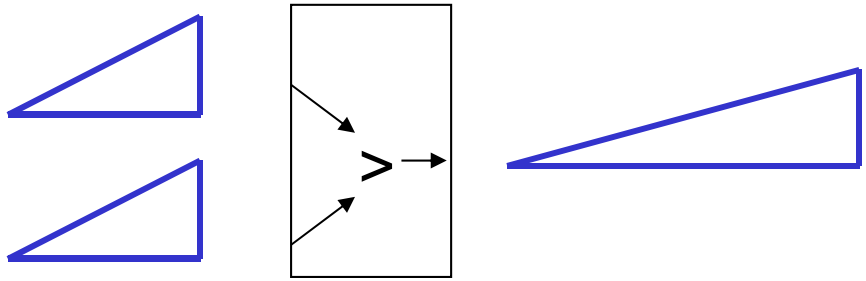
- In der Praxis wird das Verteilen nicht bis zur Listenlänge 1 durchgeführt, sondern:
  - Man lädt Teile der Datei in den Arbeitsspreicher,
  - sortiert diese Teile mit einem schnellen internen Verfahren
  - und gibt sie aus.
- Es entstehen mehrere sortierte Ausgabedateien



2. Phase: Mischen

- 2. Phase: Mischen
  - lese von jeweils zwei Dateien das erste Element
  - schreibe das kleinere und lese das nächste von der gleichen Datei

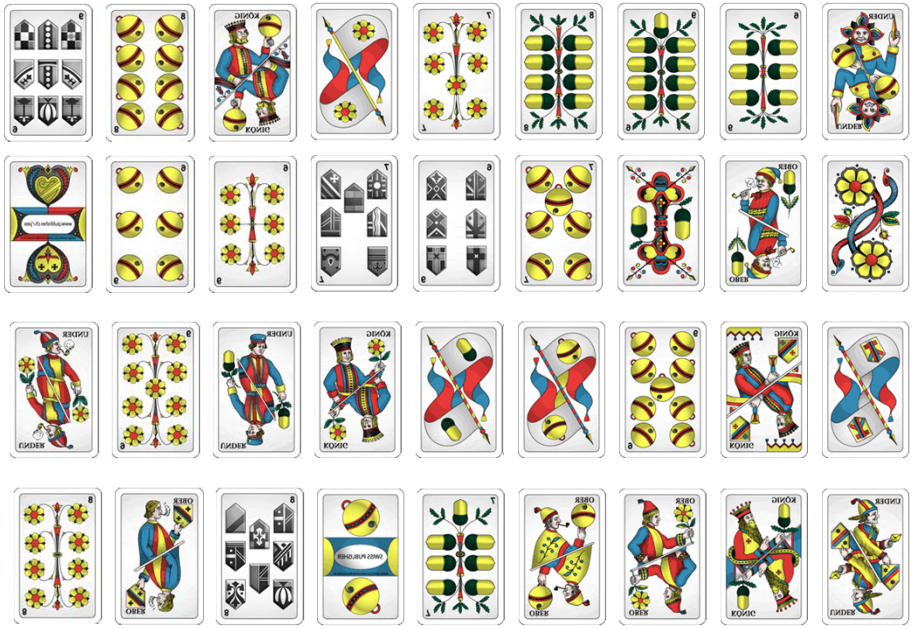
- → Anzahl Dateien hat sich halbiert
  - output → input
  - solange wiederholen, bis vollständig sortiert

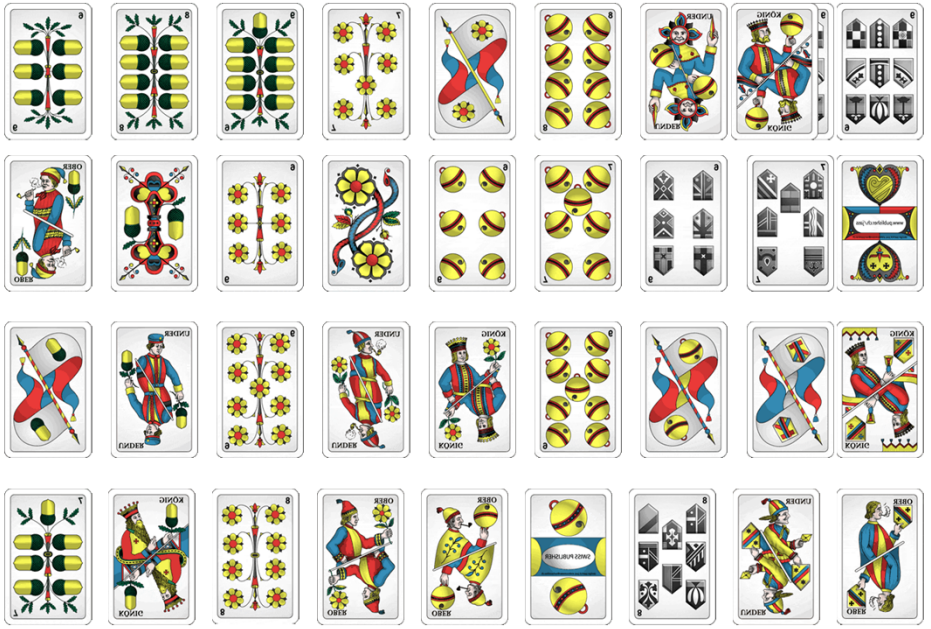


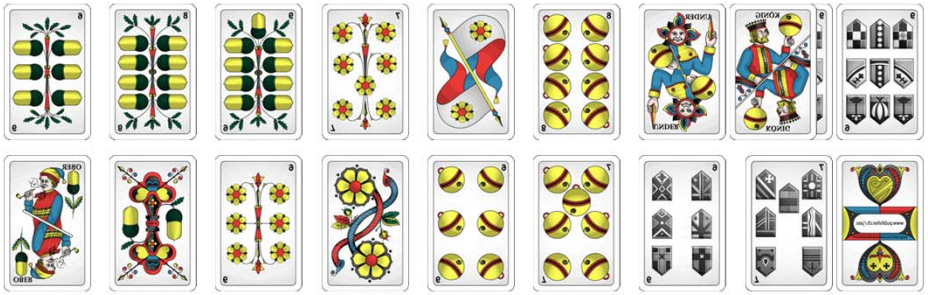
Beispiel Jasskarten

⇒

intern sortieren







mischen

mischen

mischen

Laufzeit

- Annahme:
  - Zeit für internes Sortieren kann vernachlässigt werden
  - Beispiel $n=2^{16}$ Elemente in m=16 Dateien
  - Jeweils 2 Eingabe Dateien für das Mischen (es können natürlich auch mehr als 2 Eingabedateien gleichzeitig gemischt werden)
    - 1. Phase: 2*8 Dateien mit je 2^{12} Elementen. Aufwand: 2^{15}
    - 2. Phase: 2*4 Dateien mit je 2^{13} Elementen. Aufwand: 2^{15}
    - 3. Phase: 2*2 Dateien mit je 2^{14} Elementen. Aufwand: 2^{15}
    - 4. Phase: 2*1 Datei mit je 2^{15} Elementen. Aufwand: 2^{15}
  - → $\log_2(16).n/2$

  - Falls nur 2 sortierte Dateien gemischt werden O(n)
  - Für n >> RAM → O(n.log(n))

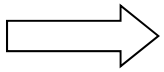
Stabile Sortieralgorithmen

- Ein wichtiger Punkt bei Sortieralgorithmen ist die Art wie Elemente mit gleichem Schlüssel behandelt werden.
- Sei S = ((k_0, e_0), ... , (k_{n-1}, e_{n-1})) eine Sequenz von Elementen: Ein Sortieralgorithmus heisst *stabil (stable)*, wenn für zwei beliebige Elemente (k_i, e_i) und (k_j, e_j) mit gleichem Schlüssel $k_i = k_j$ und i < j (d.h. Element i kommt vor Element j), i < j auch noch nach dem Sortieren gilt (Element i kommt immer noch vor Element j).

Stabile Sortieralgorithmen

Vreni	SI06b
Max	SI06b
Moni	SI06a
Sepp	SI06a
Köbi	SI06b
Fritz	SI06b
Jenny	SI06a

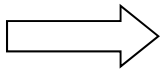
Sortiere nach Namen



Fritz	SI06b
Jenny	SI06a
Köbi	SI06b
Max	SI06b
Moni	SI06a
Sepp	SI06a
Vreni	SI06b

Fritz	SI06b
Jenny	SI06a
Köbi	SI06b
Max	SI06b
Moni	SI06a
Sepp	SI06a
Vreni	SI06b

Sortiere nach Klassen



Jenny	SI06a
Moni	SI06a
Sepp	SI06a
Fritz	SI06b
Köbi	SI06b
Max	SI06b
Vreni	SI06b

Innerhalb Klasse nach Namen sortiert

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Stabile Sortieralgorithmen

Algorithmus	Effizienz	Stabilität
Bubble Sort	$O(n^2)$	stabil
Insertion Sort	$O(n^2)$	stabil
Selection Sort	$O(n^2)$	instabil
Quick Sort	$O(n \log n)$	instabil
Merge Sort	$O(n \log n)$	stabil*
Heap Sort	$O(n \log n)$	instabil
Shell Sort	$O(n^{1.5})$	instabil

* Wenn das interne Sortieren mit einem stabilen Sortieralgorithmus erfolgt

Auswahl des Sortieralgorithmus

- **wenige Datensätze** (weniger als 1000),
  - Laufzeit **unerheblich**,
  - möglichst einfachen Sortieralgorithmus wählen (also Insertion-Sort, Selection-Sort oder Bubble-Sort). Stabilität beachten!
  - Für Selection-Sort im Vergleich zu Insertion-Sort spricht eigentlich nichts
- **vorsortierte** Datenbestände
  - dann Insertion- oder Bubble-Sort.
- **viele ungeordnete Daten**
  - dann Quick-Sort bevorzugen.
- **viele Daten, ungeordnet, sehr oft zu sortieren**
  - Distribution-Sort an das spezielle Problem anzupassen
- **sehr viele Daten**
  - externes Sortierverfahren in Kombination mit schnellem internem

Zusammenfassung

- Teile und herrsche
- Quicksort
- DistributionSort
- Laufzeitvergleiche
- Zufallszahlen
- Externe Sortierverfahren: MergeSort
- Stabilität
- Auswahlkriterien