Sortierverfahren 2

- Sie kennen das Prinzip: «Teile und Herrsche»
- Sie kennen zwei schnelle Sortierverfahren: Quick-Sort, Distribution-Sort
- Sie kenne das externe Sortierverfahren: Merge-Sort
- Sie können Algorithmen mittels Parallelisierung optimieren

Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Caesar gewinnt gegen Vercingetorix.

Teile und Herrsche

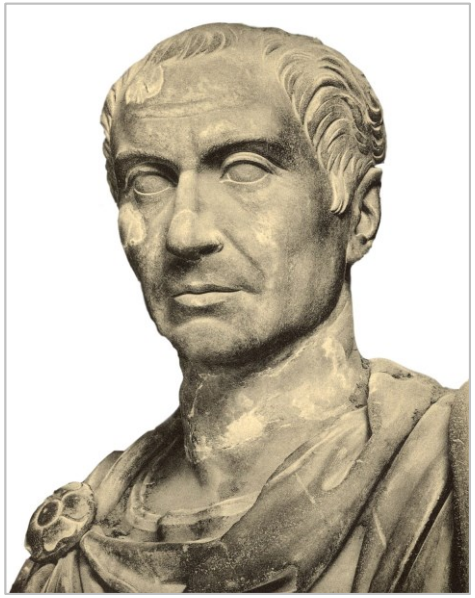
Das Prinzip «Teile und Herrsche»

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Teile und Herrsche - abgekürzt: **TUH**

Andere Bezeichnungen dieses Prinzips:

- Divide et impera
- Divide and conquer
- Divide and rule



- Divide et impera wird - fälschlicherweise - Cäsar zugeschrieben.
- Zerlege das Problem in kleinere, einfacher zu lösende Teile. Spezialfall: Teil = Ursprungsproblem mit kleinerem Bereich.
- 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 {
   Teilen: Zerlege die Menge in Teilmengen;
   Ausführen: Sortiere jede der Teilmengen;
   Vereinigen: Füge die Teilmengen geordnet zusammen;
}
```

TUH-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 Teile möglichst gleich gross sein.

Idee des Quick-Sort

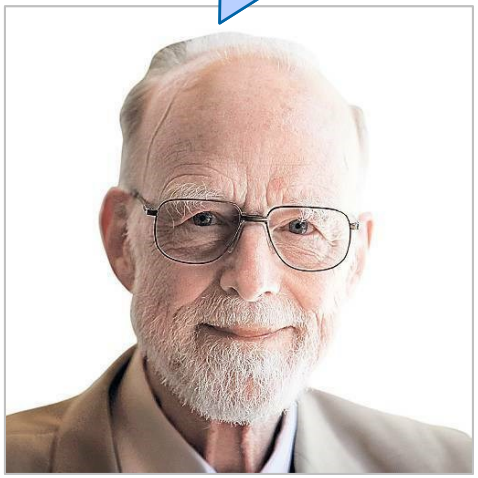
Quick-Sort

- 1960: von dem britischen Informatiker C.A.R. Hoare erfunden.

- Entstehung:
  - 1960 waren noch keine schnellen Sortieralgorithmen bekannt.
  - man versuchte damals Sortierverfahren durch raffinierte Assemblerprogrammierung zu beschleunigen.
  - Hoare zeigte mit seiner Aussage auch, dass es sinnvoller sein kann, nach besseren Algorithmen zu suchen, als vorhandene Algorithmen durch ausgefeilte Programmierung zu beschleunigen.
  - Ja, das Hoare-Kalkül ist auch von ihm.

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

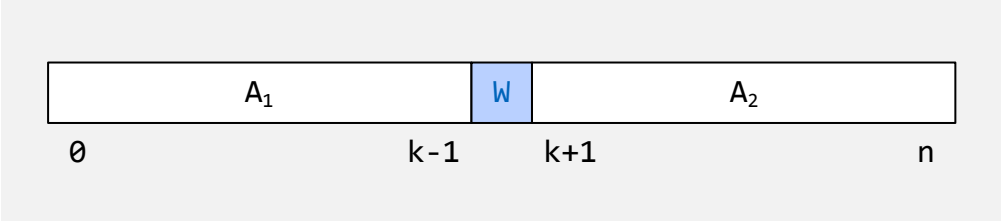
Ich stelle fest, dass es zwei Wege gibt, ein Software-Design zu erstellen, entweder so einfach, dass es offensichtlich keine Schwächen hat, oder so kompliziert, dass es keine offensichtlichen Schwächen hat. Die erste Methode ist weitaus schwieriger.



Quick-Sort

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 A_1 und A_2 mit folgenden Eigenschaften:



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

- Wenn jetzt A_1 und A_2 sortiert werden, ist das Problem gelöst.

Quick-Sort

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

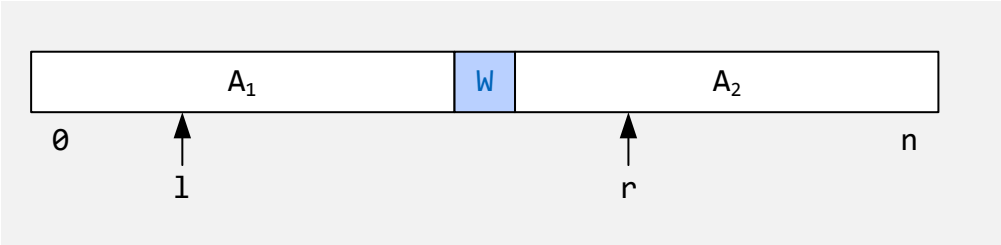
Noch zu lösende Probleme der Partitionierung:

1. Finden eines Wertes W, so dass A_1 und A_2 möglichst gleich gross sind.
2. Aufteilen von A auf A_1 und A_2.

Quick-Sort

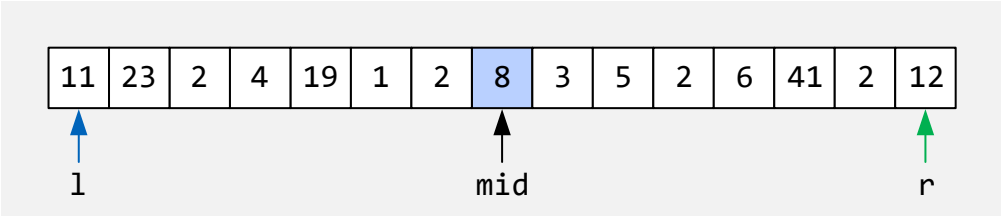
Die Partitionierung erfolgt durch:

- Wähle einen Wert `W` aus `A` (das Element `W` wird allenfalls auch verschoben).
- Suche von links kommend ein Element, das auf der falschen Seite ist, d.h. `A[l]` ≥ `W`.
- Suche von rechts kommend ein Element, das auf der falschen Seite ist, `A[r]` ≤ `W`.
- Vertausche die gefundenen Werte von `A[l]` und `A[r]`.
- Wiederhole obige Schritte bis `l` und `r` sich kreuzen, d.h. `l` ≥ `r`.



Quick-Sort: Aufgabe

Führen Sie die erste Partionierung mit dem Algorithmus auf der vorhergehenden Seite an folgendem Beispiel durch (das mittlere Element 8 sei W):



Quick-Sort: Wahl des Pivots

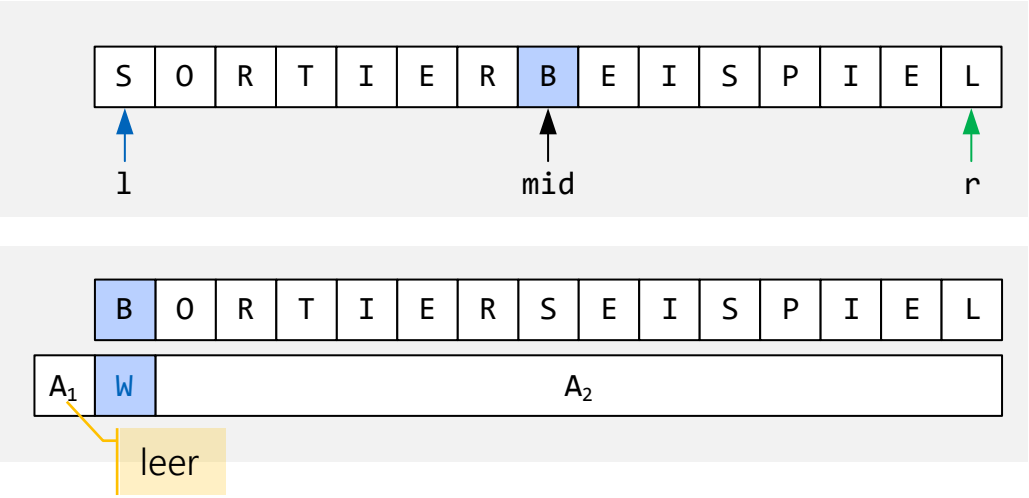
- Bestimmen des genauen Median Wertes aufwendig → Laufzeitvorteil von Quick-Sort ginge wieder verloren.
- `W` wird lediglich geschätzt.
- Folgende Pivotwahlen sind möglich (Franz. Pivot = Drehpunkt):
  - `A[l]` das (der Position nach) linke Element von A;
  - `A[r]` das (der Position nach) rechte Element von A;
  - `A[mid]` das (der Position nach) mittlere Element von A mit `mid = (l+r)/2`

- Strategie 1: Nehme eines der drei Elemente.
- Strategie 2: Nehme das (wertmässig) mittlere der drei Elemente.
- Strategie 3: Nehme das arithmetisch Mittel der drei Elemente.

Quick-Sort: Schlechte Pivotwahl

- Gutes Pivot-Element ist nicht unbedingt das Element, welches der Position nach aktuell in der Mitte liegt.
- Möglichkeit einer ungünstigen Verteilung der Daten: Durch die Partitionierung können in eine Hälfte der Partition sehr viele und in die andere Hälfte sehr wenige Daten gelangen.

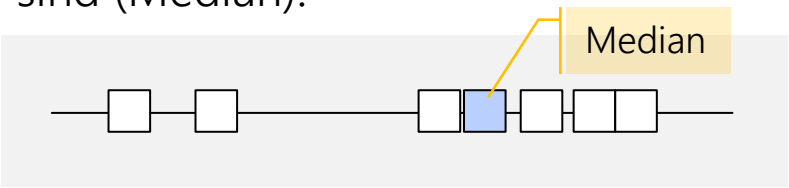
- Bei unserem Beispiel ergibt sich folgende Situation: Bei der Wahl von A[mid] als Pivot-Element ergibt sich ungünstige Partitionierung: A_1 ist leer und A_2 ist nur ein Element kleiner als A.



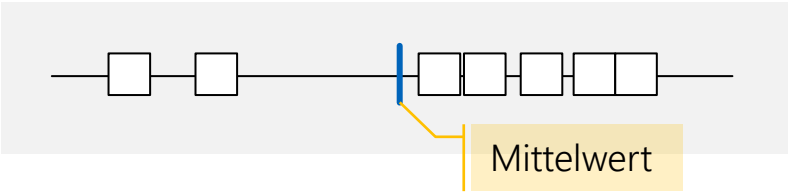
Quick-Sort: Pivot und Median

Wahl des Pivot:

- Von entscheidender Bedeutung für die Effizienz von Quick-Sort.
- Optimal wäre ein Element, das A in zwei gleich grosse Teile partitioniert.
- W sollte so bestimmt werden, dass gleich viele Werte grösser wie kleiner als W sind (Median):

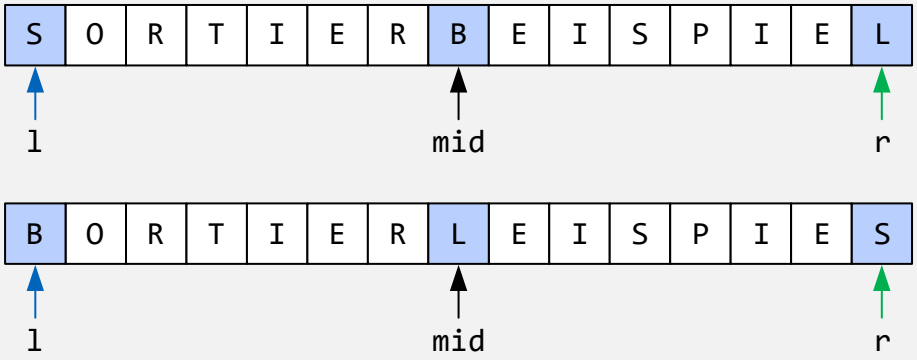


- Vorsicht: Median ist im Allgemeinen nicht gleich dem Mittelwert ($\sum x / n$):



Quick-Sort: Partition mit Median-Methode

Die drei Elemente **A[l]**, **A[mid]** und **A[r]** werden vorsortiert und man nimmt das dem Werte nach mittlere Element dieser drei Werte.



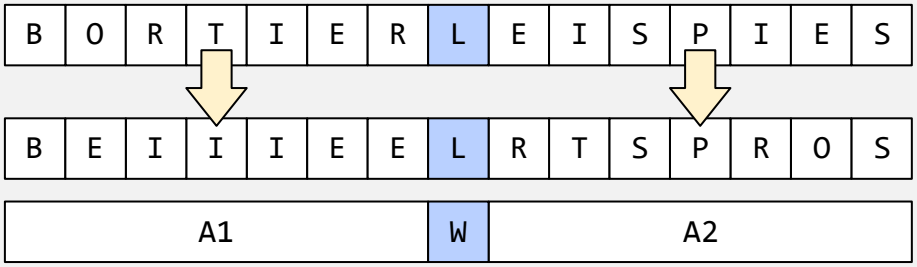

```
int mid = (l+r)/2;
if (a[l] > a[mid]) swap(a, l, mid);
if (a[mid] > a[r]) swap(a, mid, r);
if (a[l] > a[mid]) swap(a, l, mid);
int p = a[mid];
if (r-l > 2) {...
```

Wert des Pivot-Elements bestimmen.

Teile und Herrsche...

Quick-Sort: Partition mit Median-Methode

- Dadurch ergibt sich (im Beispiel) eine bessere Pivot Wahl und ausgeglichenere Hälften:



- Optimale Partitionierung erreicht:
  - 15 Elemente aufgespalten in zwei Partitionen mit je 7 Elementen.
  - Muss aber nicht immer so sein.

Quick-Sort: Partition-Methode

- Teile den Bereich in zwei Bereiche «alle kleiner» und «alle grösser» als Pivot.
- Gebe Index der Grenze zurück.

```
static int partition (int[] arr, int left, int right) {
   int pivot = arr[(left + right) / 2];
   while (left <= right) {
      while (arr[left] < pivot) { left++; }
      while (arr[right] > pivot) { right--; }
      if (left <= right) {
         swap(arr, left, right);
         left++;
         right--;
      }
   }
   return left;
}
```

Hier wird direkt das Element in der Mitte als Pivot verwendet.

Finde linkes Element ≥ Pivot.

Finde rechtes Element ≤ Pivot.

l + r schon gekreuzt? Dann sind wir fertig.

Noch nicht fertig → Element austauschen und vorfahren.

Startposition der rechten Partition.

Quick-Sort-Algorithmus

Die eigentliche Sort-Methode ist rekursiv:

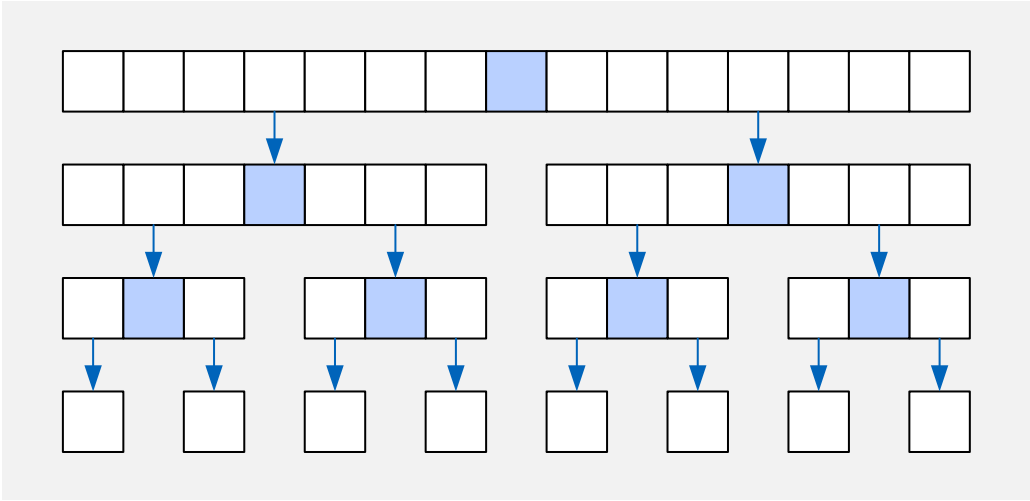
```
static void quickSort(int[] a){
   quickSort(a, 0, a.length-1);
}
```

```
static void quickSort(int[] arr, int left, int right) {
   if (left < right) {
      int mid = partition (arr, left, right);
      quickSort(arr, left, mid - 1);
      quickSort(arr, mid, right);
   }
}
```

Rekursiver Aufruf.

Quick-Sort: Aufwand

- Die Rekursionstiefe ist $\log_2(n)$, wenn bei jeder Partitionierung eine gleichmässige Aufteilung der Daten erfolgt:
  - 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 $n \cdot \log_2(n)$.
- Die Ordnung von Quick-Sort ist in diesem Fall daher $O(n \cdot \log(n))$.



Quick-Sort: Ungünstigster Fall

Ungünstigster Fall:

- Jeder Partitionierungs-Schritt enthält eine leere Partition → Baum wird zu Kette (Liste) mit n Elementen: man sagt auch der Baum entartet oder degeneriert.
- In diesem Fall ist der Aufwand proportional zu $O(n^2)$.
- Dies tritt jedoch nur in extra konstruierten Fällen auf, oder wenn man Pech hat. Im Normalfall ist die Partitionierung bei Quick-Sort nahezu optimal.

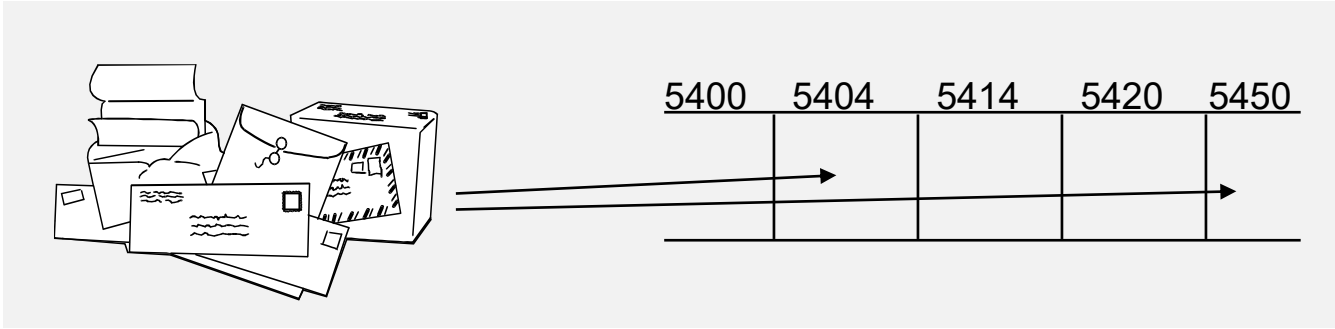
Quick-Sort ist immer die erste Wahl, wenn grössere Mengen von ungeordneten Daten sortiert werden müssen.

Aber besser iterativ als rekursiv!

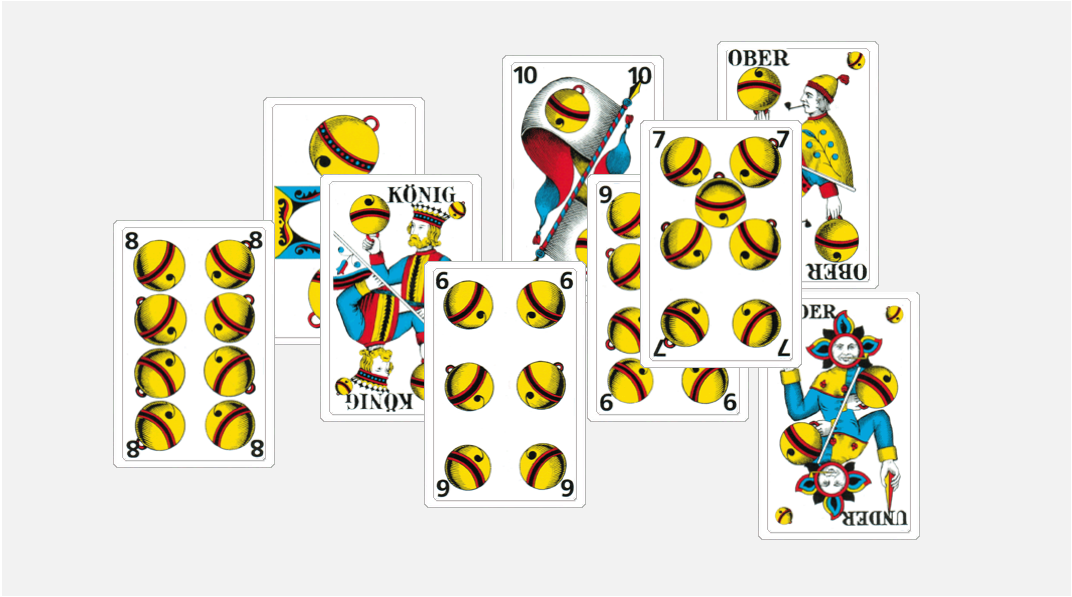
Distribution-Sort

Distribution-Sort

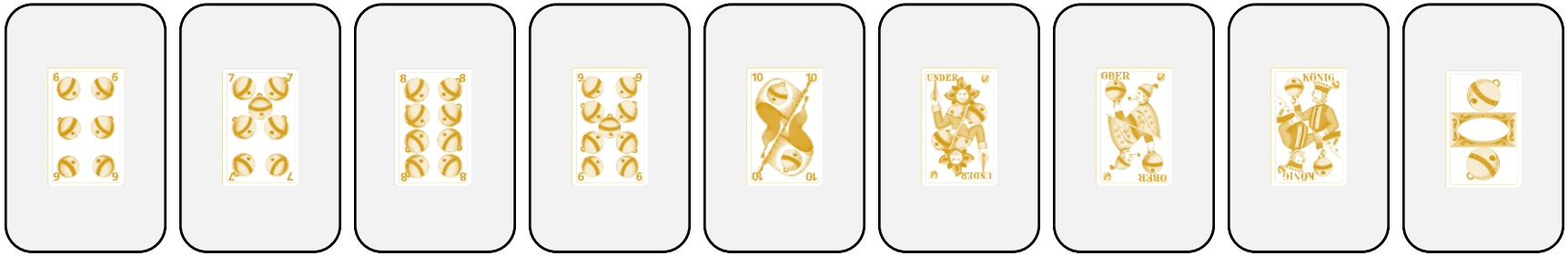
- Die bisher diskutierten Sortieralgorithmen basieren auf den Operationen: Vergleichen zweier Elemente und ev. Vertauschen zweier Elemente (swap).
- Im Gegensatz dazu kommt Distribution-Sort ohne Vergleiche zwischen den Elemente aus.
- Die zu sortierenden Elemente werden (Teile und Herrsche):
  - entsprechend dem Sortierschlüssel in Fächer verteilt: O(n)
  - zusammengetragen: O(n)
- Bsp.: Briefe werden in die entsprechenden Fächer nach Postleitzahl sortiert.



Distribution-Sort: Beispiel

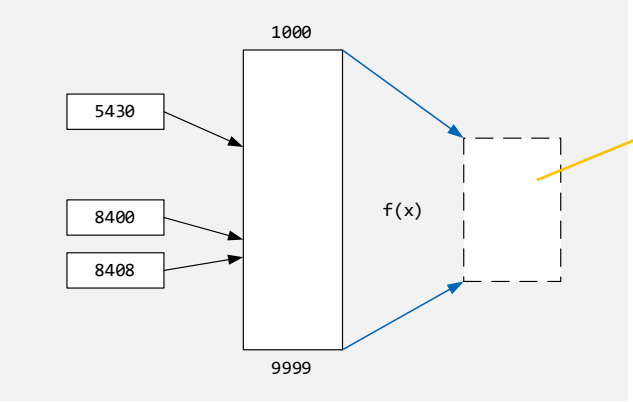


direktes Einfügen



Distribution-Sort

- Grundprinzip wie beim direkten Adressieren (vergl. Hashtable):



Allenfalls mit kleinerem Bildbereich (Abbildung mittels f(x)).

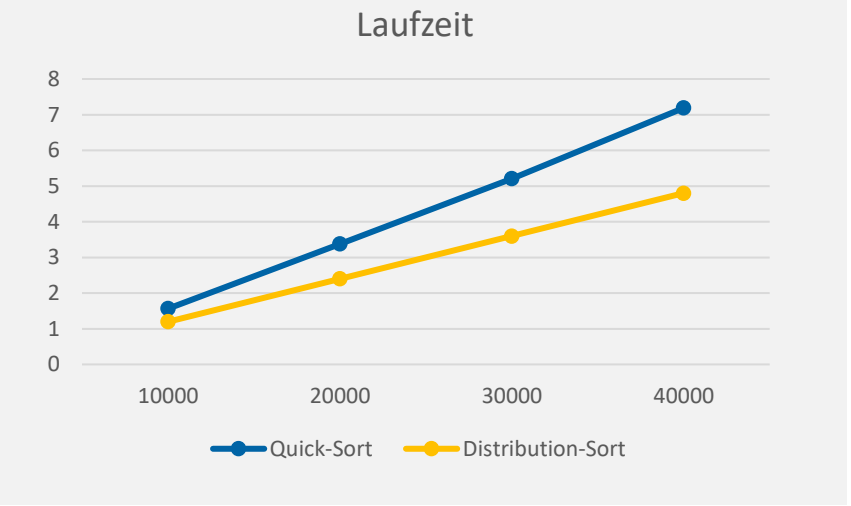
- Vorteile:
  - Schneller geht's nicht.
  - Linearer Algorithmus: die Komplexität ist also O(n).
- Nachteile:
  - Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
  - Geht nur bei Schlüsseln, die einen kleinen Wertebereich haben, oder auf einen solchen abgebildet werden können, ohne dass die Ordnung verloren geht.
  - Allgemeines Hashing funktioniert nicht: Wieso?
- Distribution-Sort ist mit Abstand der schnellste Algorithmus zum Sortieren.
  - Es handelt sich aber nicht um ein allgemein anwendbares Sortierverfahren.

Laufzeitvergleich schneller Sortieralgorithmen

Sortierung: Laufzeitvergleich

unsortiert	n = 10'000	n = 20'000	n = 30'000	n = 40'000
Quick-Sort	1.57	3.38	5.21	7.19
Distribution-Sort	1.2	2.4	3.6	4.8

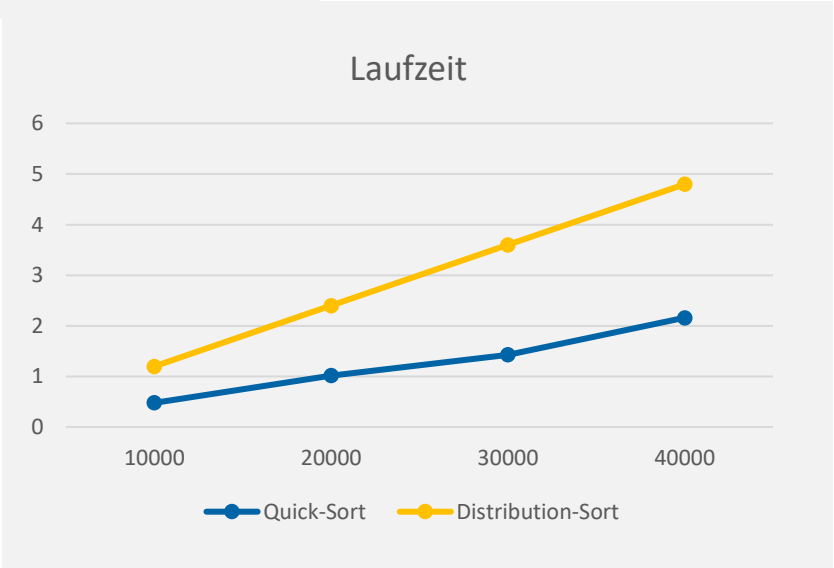
Ergebnisse der Laufzeitmessungen in Millisekunden.



Sortierung: Laufzeitvergleich

sortiert	n = 10'000	n = 20'000	n = 30'000	n = 40'000
Quick-Sort	0.48	1.02	1.43	2.16
Distribution-Sort	1.2	2.4	3.6	4.8

Ergebnisse der Laufzeitmessungen in Millisekunden.



- Sortierung in der Praxis recht 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.

- Es kann sich sogar lohnen, den Datenbestand auf die «Sortiertheit» vorab zu untersuchen → dann eventuell Insertion-Sort verwenden.

Sortierung: Laufzeitvergleich Übung

Ein $O(n \cdot \log_2(n))$ Sortieralgorithmus brauche 1 Sekunde für 10’000 Elemente; wie lange braucht er für 100'000 Elemente?

Sortieralgorithmen: Untere Schranke Aufwand

Ein Sortieralgorithmus, der darauf beruht, dass Elemente untereinander verglichen werden, kann im Worst-Case bestenfalls eine Komplexität von O(n·log(n)) 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.

Merge-Sort

Merge-Sort: Extern

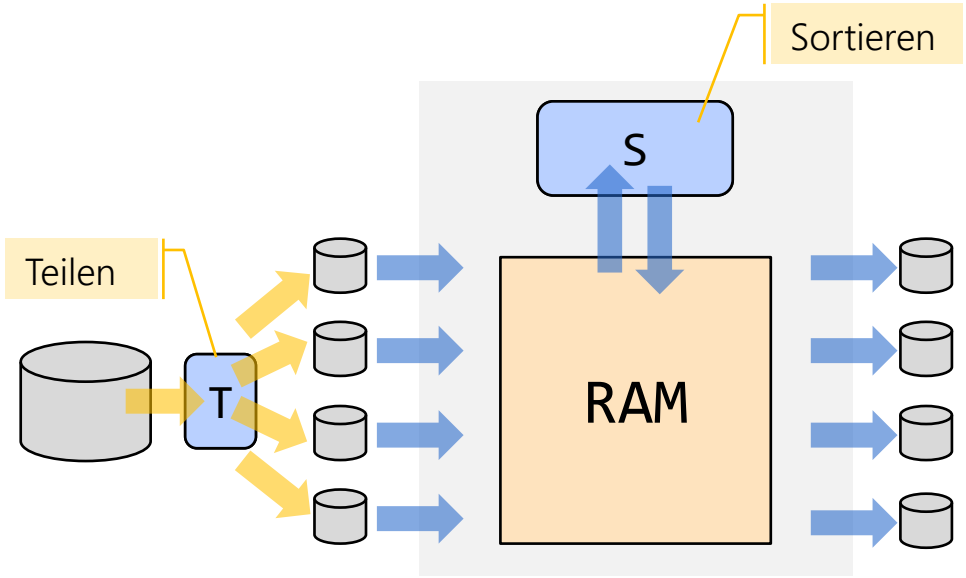
- Erstmals vorgestellt durch John von Neumann 1945.
- Beim externen Sortieren liegen die Daten in einer Datei auf der Festplatte. Zwei Arten des Zugriffs sind möglich:
  - Sequentieller Zugriff.
  - Der beliebige Zugriff auf die Elemente wäre zwar möglich, ergibt aber einen grossen Effizienzverlust.

- Annahmen:
  - Datenstrom wird sequentiell gelesen.
  - Jeweils nur ein Teil der Daten passt in den Hauptspeicher.

Merge-Sort: 1. Phase: Sortieren-Verteilen (Sort)

1. Lade jeweils einen Teil der Datei in den Speicher (Teile und Herrsche).
2. Sortiere diesen Teil mit schnellem internem Verfahren.
3. Schreibe diesen Teil sortiert in eine die Ausgabedateien.

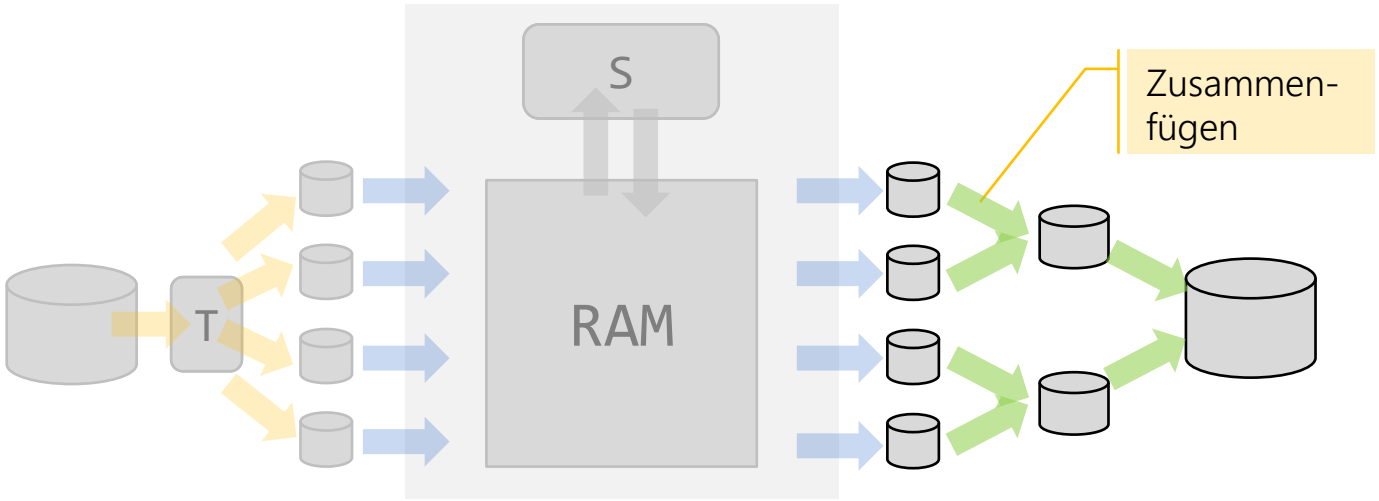
Es entstehen Folgen von sortierten Abschnitten in den Ausgabedateien.



Merge-Sort: 2. Phase: Mischen (Merge)

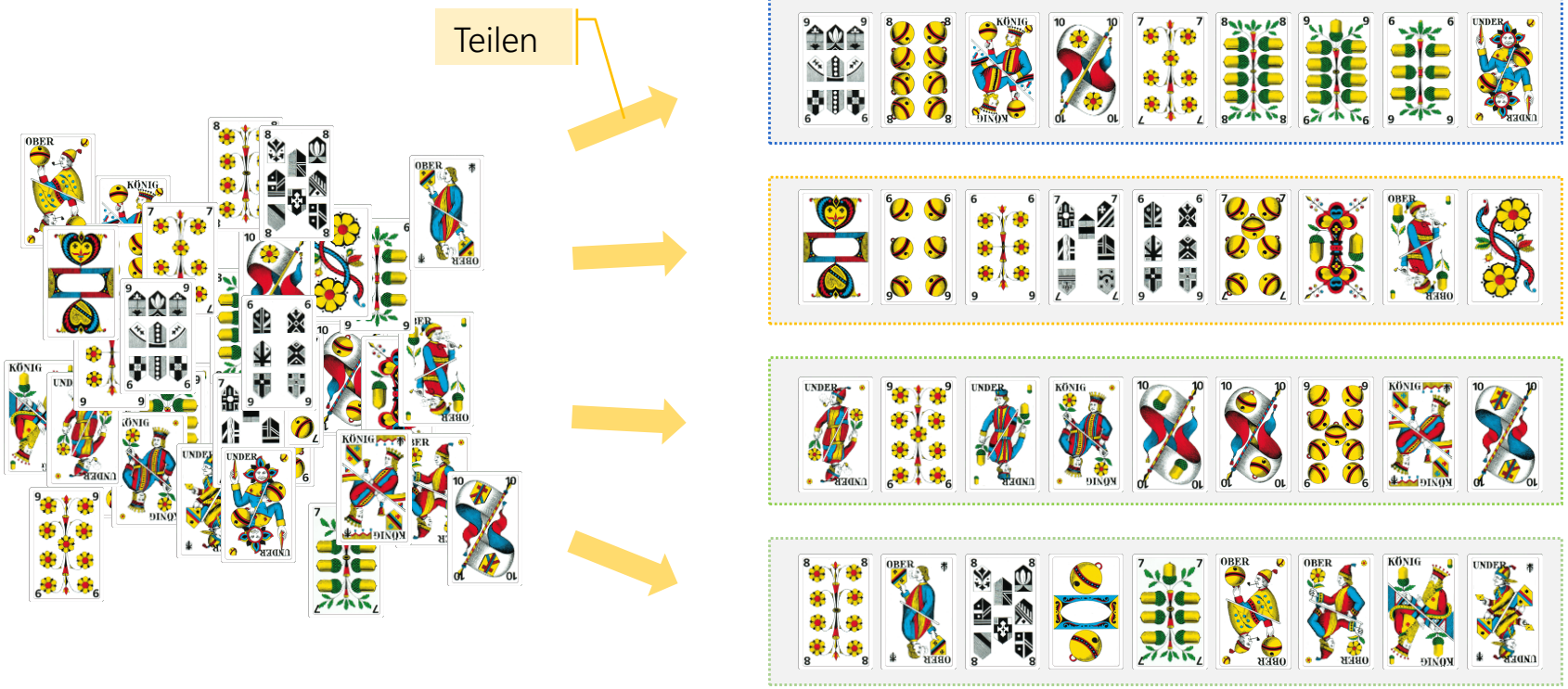
1. Lese von beiden Dateien das erste Element
2. Schreibe das Kleinere und lese das Nächste von der gleichen Datei

→ Länge der geordneten Abschnitte hat sich verdoppelt. So lange wiederholen, bis vollständig zusammengefügt.

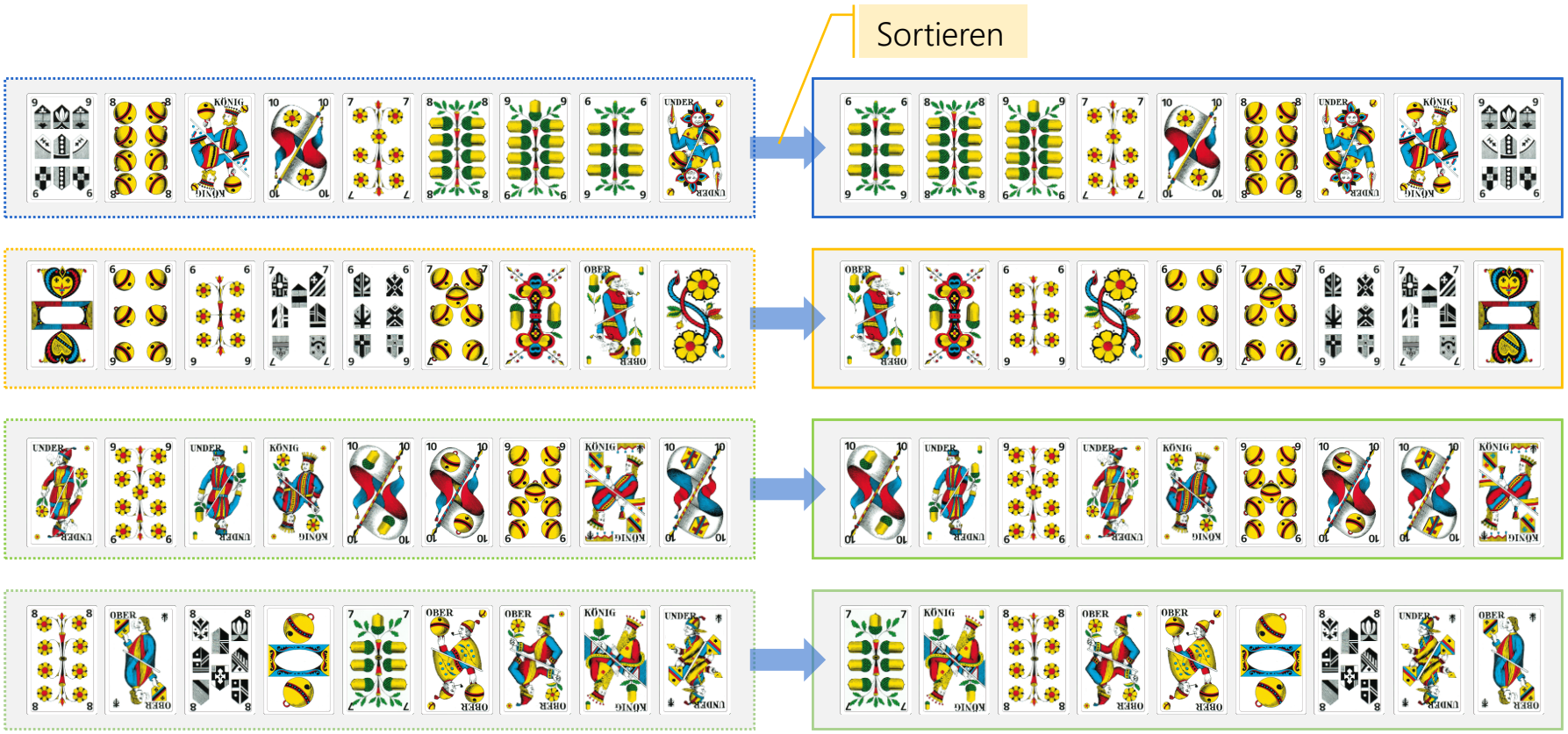


Merge-Sort: Beispiel

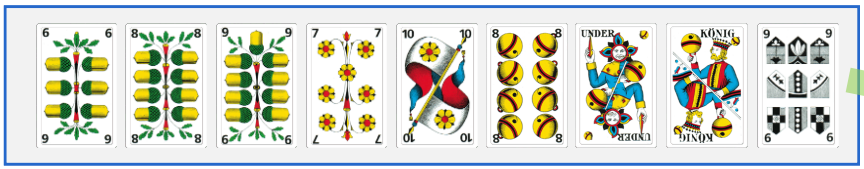
Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering



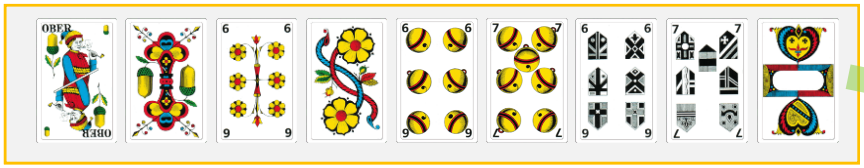
Merge-Sort: Beispiel

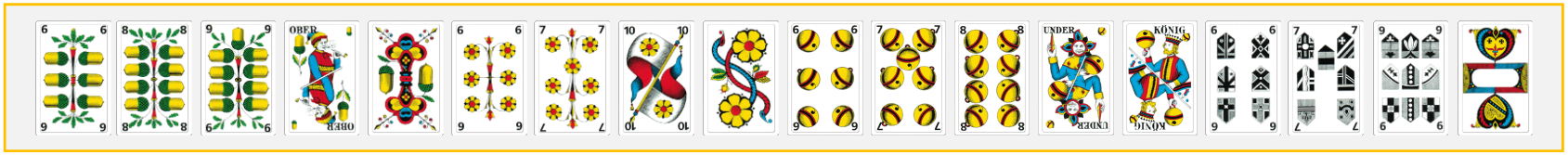


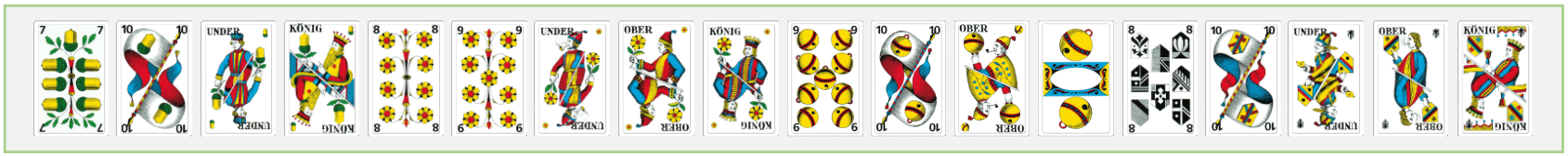
Merge-Sort: Beispiel



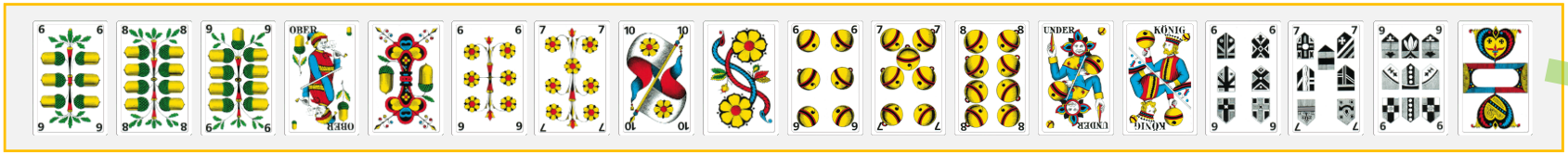
Zusammen-
fügen

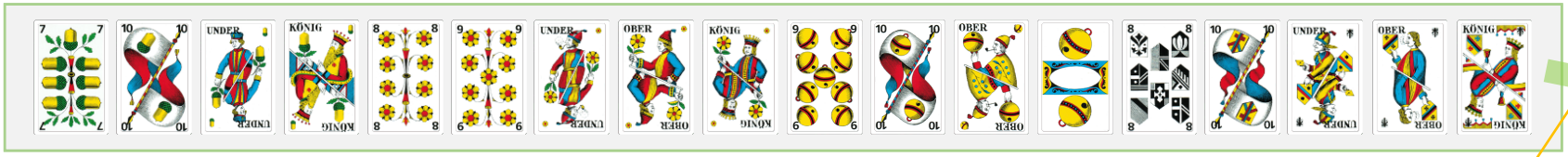






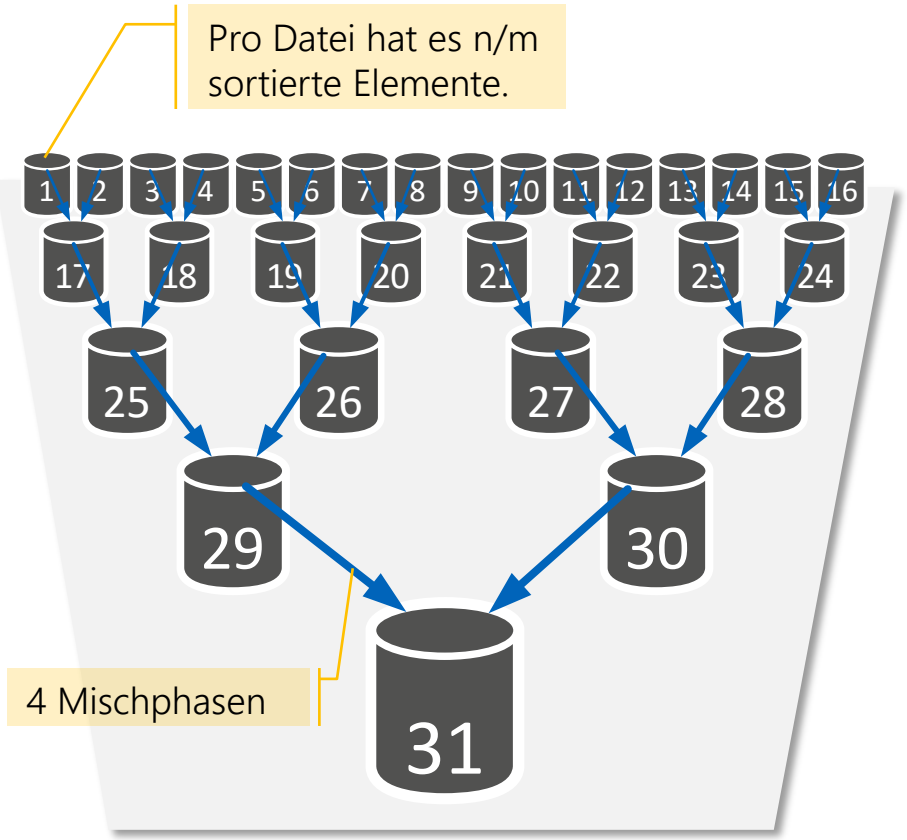
Merge-Sort: Beispiel





Zusammen-
fügen

Merge-Sort: Laufzeit



Annahmen:

- Zeit für internes Sortieren kann vernachlässigt werden.
- Beispiel: 16 Sequenzen (sortierte Abschnitte aus Sort-Phase), je 2 Eingabe-Dateien während Mischphase → $\log_2(16)$ Mischphasen

Bei insgesamt n Elementen und m Eingabedateien:

- Aufwand Sortierphase (Erstellen der Dateiteile): n
- Aufwand je Mischphase: n
- Anzahl Mischphasen: $\log_2(m)$
- Gesamtaufwand Sortieren und Mischen: $n + n \cdot \log_2(m)$ → $O(n \cdot \log(m))$

Mischphase

Sortierphase

Merge-Sort: Intern

Der Merge-Sort kann auch als internes Sortierverfahren (z.B. auf Arrays) angewendet werden.

- Der Aufwand beträgt in diesem Fall O(n·log(n)), da n=m.
- Das Verfahren benötigt jedoch zusätzlichen Speicherplatz der Grössenordnung O(n), ist also kein Inplace-Verfahren.
- Das Verfahren ist im Gegensatz zum Quick-Sort stabil.

```
Methode Mergesort (A) {
   if (A.size()) <= 1 return A;
   else {
      halbiere A in A1 und A2;
      A1 = Mergesort(A1);
      A2 = Mergesort(A2);
      return Merge(A1, A2)
   }
}
```

Liste

Zusammenführen der sortierten Listen (nächste Folie).

Merge-Sort: Intern

```
Methode Merge(linkeListe, rechteListe); {
   neueListe;
   while (!(linkeListe.isEmpty() OR rechteListe.isEmpty)) {
      if (linkeListe(0) <= rechteList(0) {
         neueListe.add(linkeListe(0)); linkeListe.remove(0);
      }
      else {
         neueListe.add(rechteListe(0)); rechteListe.remove(0);
      }
   }
   while (!linkeListe.isEmpty()) {
      neueListe.add(linkeListe(0)); linkeListe.remove(0);
   }
   while (!rechteListe.isEmpty()) {
      neueListe.add(rechteListe(0)); rechteListe.remove(0);
   }
   return neueListe;
}
```

Eine der beiden Listen ist jetzt leer.

Wahl des Sortierverfahrens

Wahl des Sortierverfahrens

Auswahlkriterien des geeigneten Sortierverfahrens:

- Internes oder externes Verfahren
  - Sortieren im Hauptspeicher
  - Sortieren im Hauptspeicher und Sekundärspeicher (Platte)
- Methode des Algorithmus:
  - Vertauschen / Auswählen / Einfügen
  - Rekursion
  - Mehrphasen: Sortieren-Mischen
- Nach Effizienz:
  - Laufzeit: $O(n^2)$ oder $O(n \cdot \log(n))$ oder sogar $O(n)$
  - Speicherbedarf/Inplace: Wieviel Speicher wird zusätzlich zu dem für die zu sortierenden Daten benötigt?

Wahl des Sortierverfahrens

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Allgemein:

- Stabilität: Wird die Reihenfolge von Datensätzen mit gleichem Sortierkriterium durch den Algorithmus geändert?
- Der Algorithmus setzt eine bestimmte Struktur der Schlüssel voraus (z.B. Distribution-Sort).
- Wenige Datensätze (weniger als 1'000), für Laufzeit unerheblich: Möglichst einfachen Sortieralgorithmus wählen (also Insertion-Sort, Selection-Sort oder Bubble-Sort).
- 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 angepasst.
- Sehr viele Daten: externes Sortierverfahren in Kombination mit schnellem internem Sortierverfahren.

Klasse mit statischen Hilfsmethoden zum Array.

Klasse Arrays und sort()-Methode

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

```
static void sort()
```

Z.B. `static void sort(int[] a)`

- sort() in Java verwendet in den meisten Fällen einen Dual-Pivot-Quick-Sort: Es werden drei statt zwei Teile gebildet (im Durchschnitt etwas schneller).
- Nicht stabil, daher von Java nur für primitive Typen verwendet.
- Aufwand: O(n·log(n))

Stabilität ist hier irrelevant.

```
static void sort(Object[] a)
static void sort(T[] a, Comparator<? super T> c)
```

- Java verwendet den stabilen, iterativen Merge-Sort (intern, nicht extern).
- Aufwand Laufzeit: O(n·log(n)), kein Inplace-Verfahren.

Schauen wir nächste Woche an.

```
static void parallelSort()
```

- Analog sort(), verwendet das Fork/Join-Framework zum parallelen Sortieren von Unterarrays.

```
static <T extends Comparable<? super T>> void parallelSort(T[] a)
static <T> void parallelSort(T[] a, Comparator<? super T> cmp)
```

- Java unterteilt das Array in Unter-Arrays, bis sie eine bestimmte Grösse unterschreiten, sortiert diese mit der passenden sort()-Methode und fügt sie wieder zum Resultat zusammen.

Array von Objekten.

Klasse mit statischen Hilfsmethoden zur Collection.

Klasse Collections und sort()-Methode

Liste

```
static <T extends Comparable<? super T>> void sort(List<T> list)
static <T> void sort(List<T> list,Comparator<? super T> c)
```

- Java verwendet den stabilen, iterativen Merge-Sort (intern, nicht extern).
- Basiert auf dem Interface List mit der Methode sort().
- Aufwand: O(n·log(n))

Optimierung durch Kombination von Algorithmen

Optimierungen von Quick-Sort

- Quick-Sort ist schnell, aber wegen der Rekursion hat er einen relativ grossen initialen Aufwand («Footprint»).
- Für wenige (~10..1000) zu sortierende Daten ist ein einfaches Sortierverfahren schneller.
- Idee: Unterhalb einer bestimmten Länge des Intervalls nicht mehr Quick-Sort verwenden, sondern z.B. Insertion-Sort; bringt ca.10%.

```
static void quickerSort(int[] a, int left, int right) {
   if (right - left < THRESHOLD)
      insertionSort(a, left, right);
   else {
      int l = partition (a, left, right);
      quickerSort(arr, left, l - 1);
      quickerSort(arr, l , right);
   }
}
```

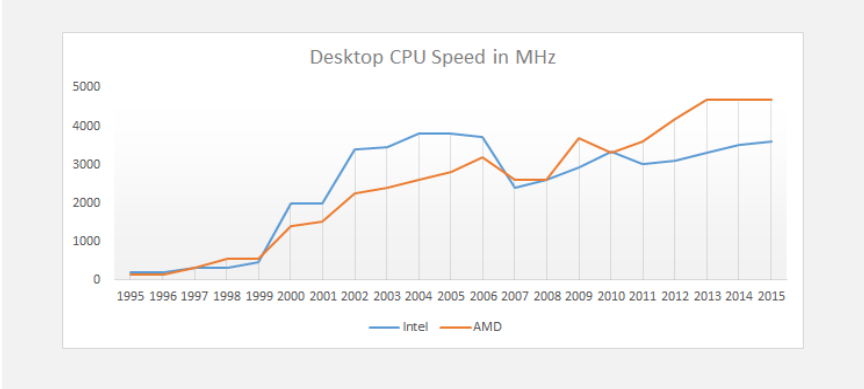
Unterhalb dieser Grösse wird nicht mehr Quick-Sort eingesetzt...

Optimierung durch Parallelisierung

Parallelisierung: Speed vs. Power Consumption

- Das grösste Problem heute ist die Wärmeabgabe der Chips.
- Jeder CMOS Schaltvorgang braucht Energie, Verkleinerungen erhöht die Leckströme.
- Die Energiedichte innerhalb einer CPU ist grösser als diejenige im Kern eines Kernreaktors.

- Computer Industrie ist für 1.4% (2020) der weltweiten CO2 Emissionen verantwortlich[1)] und verbraucht 3.6% des Stroms
- Rechenzentren: Stromkosten p.a. ~ HW Kosten.



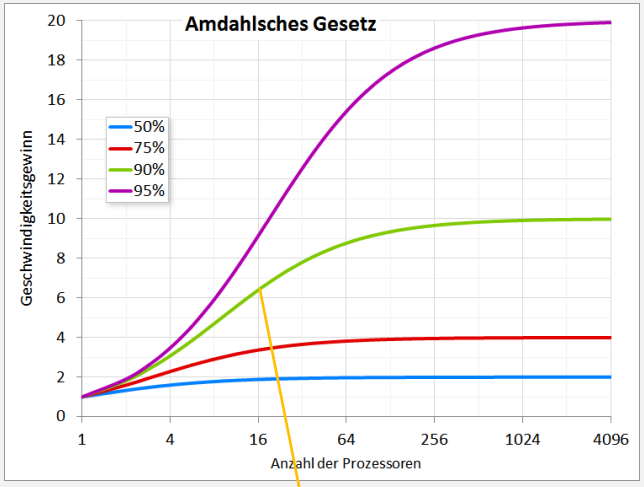
Parallelisierung: Beschleunigung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering

Moderne Multicore-CPUs sprechen eigentlich dafür aber - Amdahl's Law:

- Die Geschwindigkeitssteigerung ist durch den sequentiellen Anteil im Programm limitiert.
- p: ist der Anteil des Programms, der parallelisiert werden kann.
- s: Zahl der Prozessoren.
- Es kommt aber noch ein Overhead durch die Steuerung der Threads dazu.

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{s}}$$



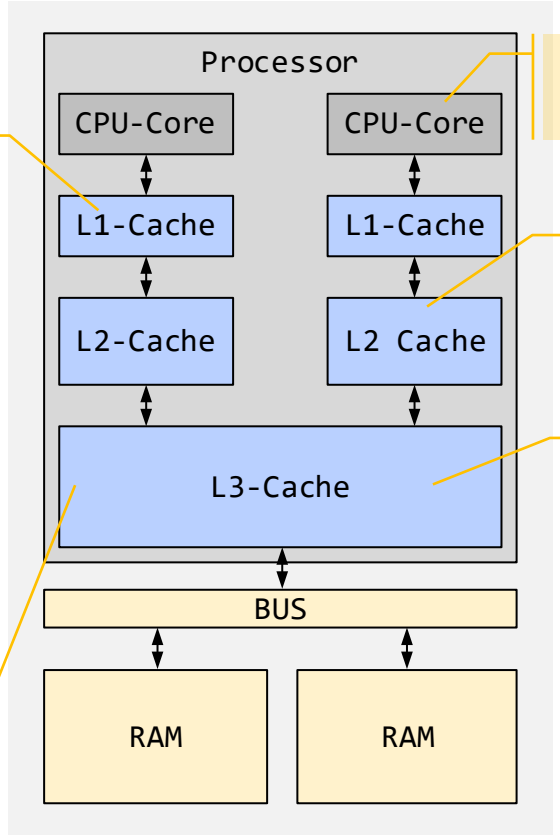
bei 16 Kernen und p=90% ist der Geschwindigkeitsgewinn etwas grösser als 6-fach.

Parallelisierung: Hardware-Memory-Architektur

- Zugriffzeiten:
  - Register: < 1 Zyklus
  - L1: ~ 4 Zyklen
  - L2: ~ 11 Zyklen
  - L3: ~ 40 Zyklen
  - RAM: ~ 150 – 300 Zyklen
  - SSD: ~ 200'000 Zyklen (~ 0.1 ms)
  - HD: ~ 20'000'000 Zyklen (~10 ms)

- Hauptspeicherzugriffe sind «relativ» langsam.

i9-9900:
64KB je Core



i9-9900: 8 Cores,
16 Threads

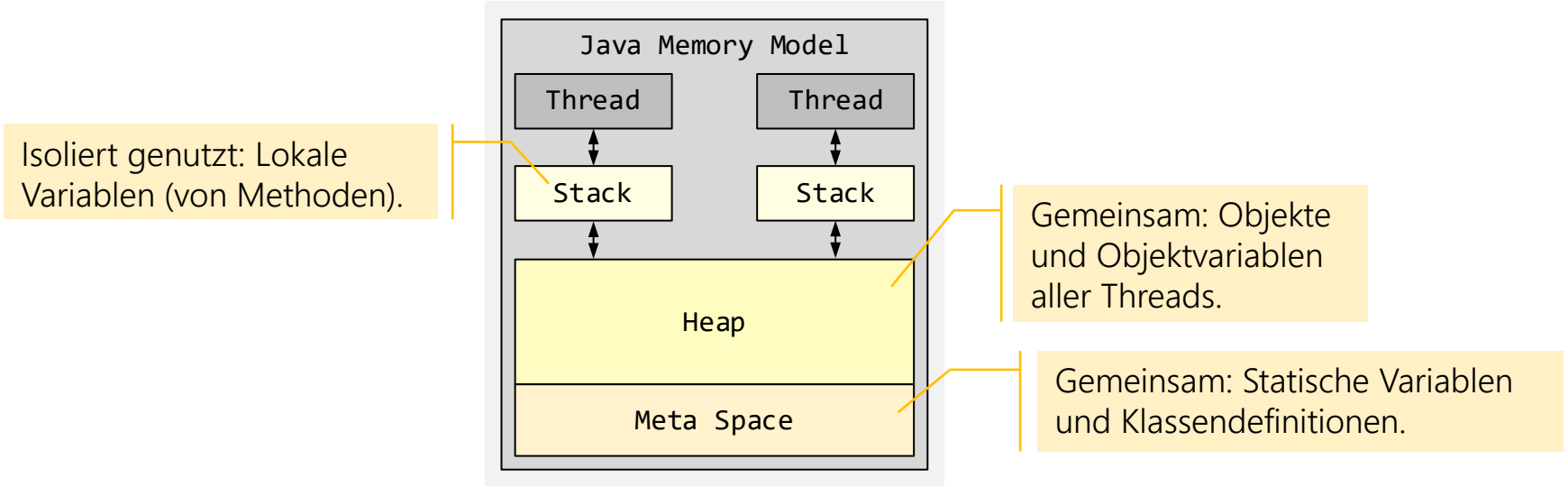
i9-9900:
256KB je Core.

i9-9900:
16MB shared

L3-Cache ist vor allem für den Datenaustausch zwischen den Cores (Cache-Kohärenz-Protokoll).

Parallelisierung: Java-Memory-Modell (JMM)

- JMM abstrahiert von physikalischem Memory-Modell. Das JMM gibt an:
  - Wie und wann verschiedene Threads Werte sehen können, die von anderen Threads in gemeinsame Variablen geschrieben wurden.
  - Wie der Zugriff auf gemeinsame Variablen bei Bedarf synchronisiert werden kann.
- Im JMM hat jeder Thread:
  - Eigenen Thread-Stack, der Zugriff auf andere Thread-Stacks ist nicht möglich.
  - Eigene Informationen über ausgeführte Methoden (Call Stack).
  - Eigene lokale Variablen für primitive Datentypen jeder Methode (Stack).
  - Objekte liegen im gemeinsamen Heap – nicht im Stack.



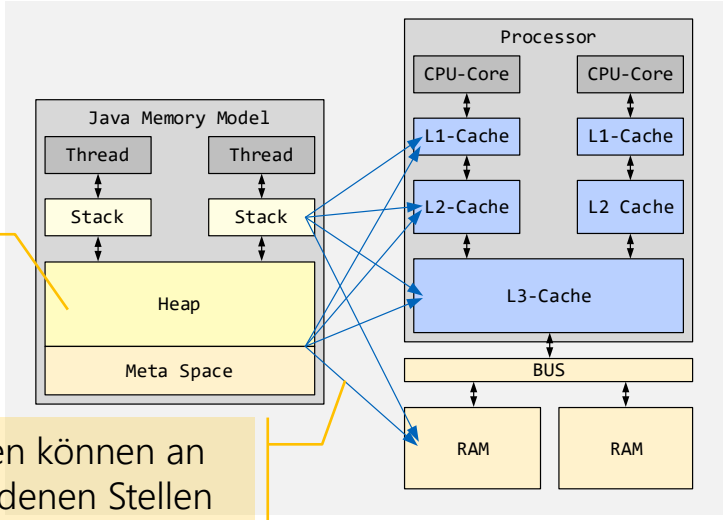
Parallelisierung: Memory-Synchronisation in Java

- Die Hardware-Memory-Architektur unterscheidet nicht zwischen Stack und Heap. Das führt zu Problemen:
  1. Änderungen von Variablen sind eventuell nicht sichtbar (da erst im Cache).
  2. Es können Race-Conditions auftreten (wenn Reihenfolge Lesen/Schreiben wichtig ist).

Nichtdeterministische Programmfehler ☹.

- Veränderungen aus Cache (des Threads) werden synchronisiert bei:
  1. Thread-Start und –Beendigung, sowie bei einem Thread-Wechsel auf dem CPU-Core.
  2. Beim ersten Lesen von final-Variablen.
  3. Das Schlüsselwort volatiles stellt sicher, dass eine Objekt- oder Klassen-Variable direkt aus dem RAM gelesen und bei einer Aktualisierung in das RAM zurückgeschrieben wird, verhindert Race-Condition nicht.
  4. Explizite Synchronisation mittels synchronized keyword → verhindert Race-Condition.

Für Daten im Heap und im Meta Space muss mittels volatile und synchronized angegeben werden, wie und wann diese mit dem RAM abgeglichen werden.



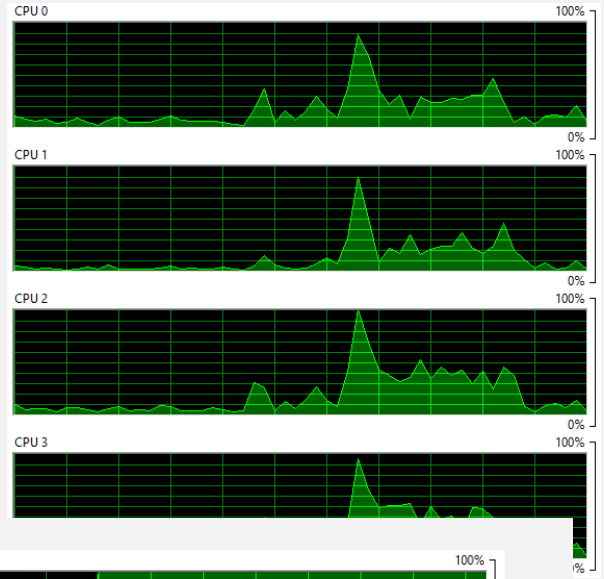
Die Daten können an verschiedenen Stellen im Speicher liegen.

Parallelisierung: Prozessor Auslastung

- Single Thread:

Der Algorithmus wird sequentiell auf 4 CPUs (Timeslices!) ausgeführt.

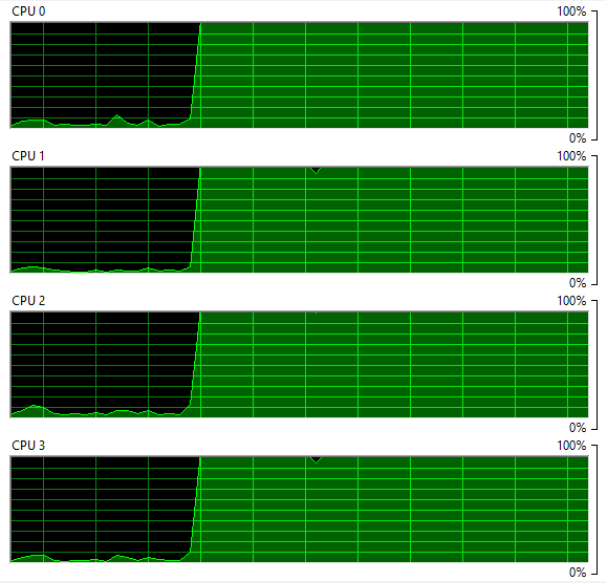
```
"main"@1 in group "main": RUNNING
"Attach Listener"@804: RUNNING
"Common-Cleaner"@795 in group "InnocuousThreadGroup": WAIT
"Finalizer"@802: WAIT
"Notification Thread"@791: RUNNING
"Reference Handler"@801: RUNNING
"Signal Dispatcher"@803: RUNNING
```



- Multi Thread:

Der Algorithmus wird parallel auf 4 CPUs ausgeführt.

```
"Finalizer"@866: WAIT
"main"@1 in group "main": WAIT
"Notification Thread"@782: RUNNING
"Reference Handler"@865: RUNNING
"Signal Dispatcher"@867: RUNNING
"Thread-1"@796 in group "main": WAIT
"Thread-10"@837 in group "main": WAIT
"Thread-11"@836 in group "main": WAIT
"Thread-12"@847 in group "main": RUNNING
"Thread-13"@842 in group "main": WAIT
"Thread-14"@845 in group "main": RUNNING
"Thread-15"@846 in group "main": WAIT
"Thread-16"@870 in group "main": RUNNING
"Thread-17"@855 in group "main": RUNNING
"Thread-18"@854 in group "main": RUNNING
"Thread-19"@851 in group "main": RUNNING
"Thread-2"@808 in group "main": WAIT
```



Parallelisierung: Naiver Quick-Sort

- Je ein neuer Thread für jede Partitionsaufgabe.
- Java-Threads werden auf Betriebssystem-Threads abgebildet: «Kernel Level Threads» (es gibt mehr Threads als Kerne)

- Vorteile:
  - Die Rechenzeitzuteilung kann besser (von BS) verwaltet werden.
  - Sind mehrere CPU-Kerne im Rechner vorhanden, können diese ausgenutzt werden.
- Aber:
  - Erzeugung und Zerstörung von Threads kostet (viel) Zeit.
  - Instanziierte Threads belegen Speicher.
  - Ineffektiv, mehr Threads zu erzeugen, als die Prozessoren gleichzeitig handhaben können, da immer einige inaktiv (idle) sein werden.
  - Quick-Sort eignet sich relativ schlecht für parallele Ausführung, da der parallelisierbare Anteil klein ist (Amdahl's Gesetze). Allenfalls ist Performance sogar schlechter…

Parallelisierung: Naiver Quick-Sort

```
class NaiveParallelQuicksort extends Thread {
   int[] arr; int left, right;

   public NaiveParallelQuicksort(int[] arr, int left, int right) {
      this.arr = arr; this.left = left, this.right = right; }

   public static void sort(int[] arr) {
      Thread root = new NaiveParallelQuicksort(arr, 0, arr.length - 1);
      root.start();
      root.join();
   }

   public void run() {
      int mid = 0;
      Thread t1 = null;
      Thread t2 = null;

      if (left < right) {
         mid = partition(arr, left, right);
         t1 = new NaiveParallelQuicksort(arr, left, mid - 1);
         t1.start();
         t2 = new NaiveParallelQuicksort(arr, mid , right);
         t2.start();
         if (t1 != null) t1.join();
         if (t2 != null) t2.join();
      }
   }
```

Hier im Bsp. explizit nicht als Implementation von Runnable.

Setzen der Parameter.

Aufruf der run()-Methode.

Konstruktor von NaiveParallelQuicksort.

Sortieren der Partitionen mittels paralleler Threads.

Warten bis beide (Sub-)Threads beendet sind.

Parallelisierung: Verbesserter naiver Quick-Sort

```
class NaiveParallelQuicksort2 extends Thread {
   ...
   private final int SPLIT_THRESHOLD = 100000;

   public void run() {
      int mid = 0;
      Thread t1 = null;
      Thread t2 = null;

      if (left < right) {
         mid = partition(arr, left, right);
         if (mid - left > SPLIT_THRESHOLD) {
            t1 = new NaiveParallelQuicksort2(arr, left, mid - 1);
            t1.start();
         } else {
            Quicksort.quickSort(arr, left, mid - 1);
         }
         if (right - mid > SPLIT_THRESHOLD) {
            t2 = new NaiveParallelQuicksort2(arr, mid, right);
            t2.start();
         } else {
            Quicksort.quickSort(arr, mid, right);
         }
      if (t1 != null) t1.join();
      if (t2 != null) t2.join();
   }
}
```

Nur falls Task genügend gross ist parallel ausführen.

Sequentielle Ausführung des Tasks.

Zusammenfassung

- Teiles und Herrsche Prinzip
- Quick-Sort
  - Partitionierung
  - Wahl des Pivots
  - Sequentielle Optimierungen
- Distribution-Sort
- Merge-Sort
- Optimierung durch Parallelisierung
  - Java Memory-Modell
  - Threads

Nerd-Zone

Quick-Sort Funktional

Quick-Sort Funktional


```
public class Main {
   private static Function<Integer, Predicate<Integer>> smallerThan = x -> y -> y < x;

   public static List<Integer> qsort(List<Integer> l){
      if(l.isEmpty()) return new ArrayList<>();
      return Stream.concat(Stream.concat(
         qsort(l.stream().skip(1).filter(smallerThan.apply(l.get(0)))
            .collect(Collectors.toList())).stream(),
         Stream.of(l.get(0))),
         qsort(l.stream().skip(1).filter(smallerThan.apply(l.get(0)).negate())
            .collect(Collectors.toList())).stream()).collect(Collectors.toList());
   }

   public static void main(String[] args) {
      List<Integer> l = Arrays.asList(5, 6, 7, 23, 4, 5645, 6, 1223, 44453, 60182,
         2836, 23993, 1);
      System.out.println(qsort(l));
   }
}
```

```
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```