

# Sortiervverfahren 2



- Sie kennen das Prinzip: "Teile und Herrsche"
- Sie kennen zwei schnelle Sortiervverfahren: Quicksort, Distribution Sort
- Sie können Algorithmen mittels Parallelisierung optimieren

# Teile und Herrsche

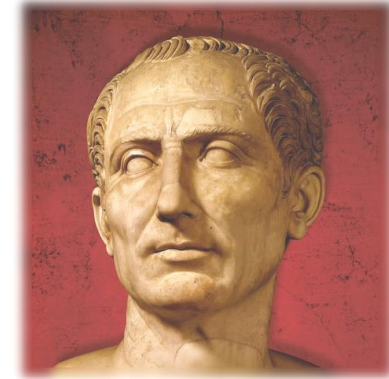
# Das Prinzip "Teile und Herrsche"

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 Sortialgorithmen

- Sortialgorithmen 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;  
}
```

- 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 Teile möglichst gleich gross sein.

## Idee des Quicksort

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

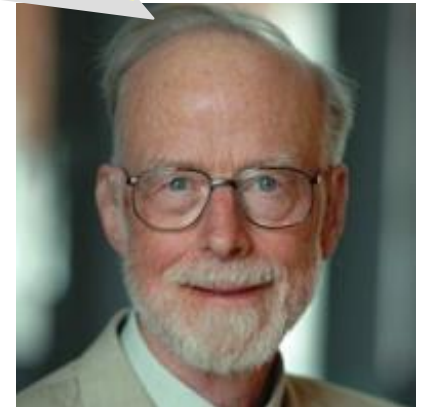
Ich stelle fest, dass es zwei Wege gibt, ein Software-Design zu erstellen, entweder so einfach, dass es offensichtlich keine Schwächen hat, oder so kompliziert, dass es keine offensichtlichen Schwächen hat. Die erste Methode ist weitaus schwieriger.

- Entstehung

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

- Quicksort mit naheliegenden Verbesserungen ist

- einer der **schnellsten bekannten** allgemeinen Sortieralgorithmen
- theoretisch gut verstanden.



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

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 noch evtl. unsortiert).
- Alle Elemente von  $A_2$  sind  $\geq W$  (aber noch evtl. unsortiert).

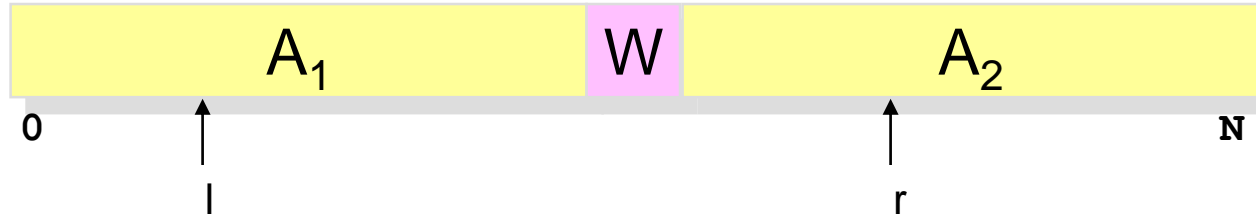
Wenn jetzt  $A_1$  und  $A_2$  sortiert werden, ist das Problem gelöst.

```
Methode Sortiere (A) {  
    Konstruiere die Partition  $A = A_1 \cup A_2$ ;  
    Sortiere  $A_1$ ;  
    Sortiere  $A_2$ ;  
}
```

noch zu lösendes Problem:

- finden eines Wertes  $W$ , so dass  $A_1$  und  $A_2$  möglichst gleich gross sind.

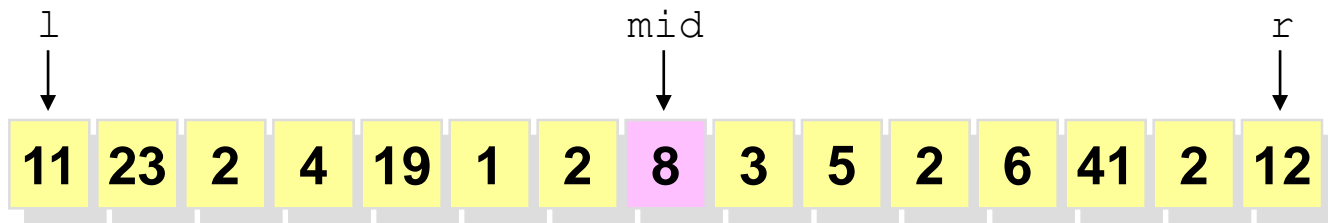




Die Konstruktion der Partition erfolgt durch:

- wähle ein Element  $W$
- suche von links ein Element, das auf falscher Seite ist, d.h.  $A[l] \geq W$
- suche von rechts ein Element, das auf falscher Seite ist,  $A[r] \leq W$
- vertausche von  $A[l]$  und  $A[r]$
- wiederhole obige Schritte bis  $l$  und  $r$  sich kreuzen, d.h.  $l \geq r$ .

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

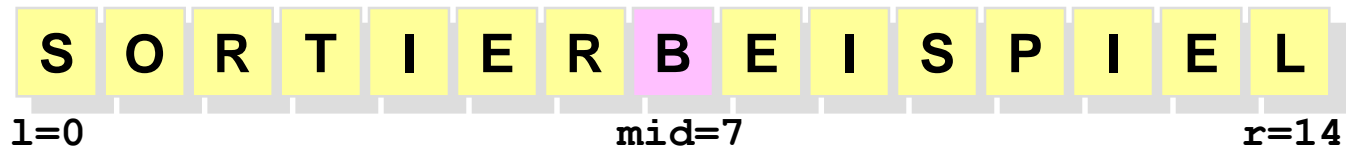


Bestimmen des genau Median Wertes aufwendig -> Laufzeitvorteil von Quicksort ginge wieder verloren

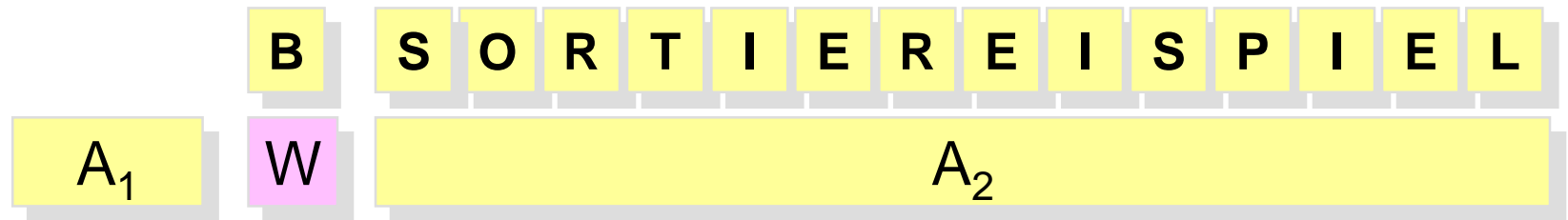
## W wird lediglich geschätzt

- 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

- Gutes Pivot-Element ist nicht unbedingt das Element, welches der Position nach in der Mitte liegt.
- die Möglichkeit einer ungünstigen Verteilung der Daten:
  - 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  $A$ .

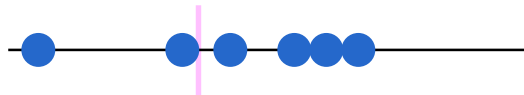


# Quicksort: Wahl des Pivots

- von entscheidender Bedeutung für die Effizienz von Quicksort
- optimal wäre ein Element, das A in **zwei gleich grosse Teile** partitioniert
- W muss so bestimmt werden, dass gleich viele Werte grösser wie kleiner als W sind:

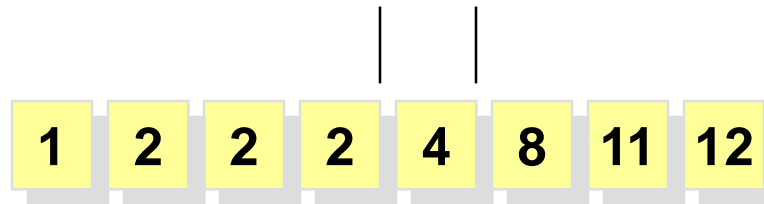


- Vorsicht: ist i.A. nicht gleich dem Mittelwert:  $\sum x / n$ :



# Aufgabe: Anwendung Median, Mittelwert

- Ein Lehrer möchte, dass der Schnitt einer Prüfung genau 4 beträgt. Welche Punktzahl muss er für eine 4 fordern; Punktedurchschnitt 5.25\*.



- Ein anderer Lehrer möchte, dass lediglich die Hälfte der Schüler eine genügende Note bekommt. Welche Punktzahl muss er jetzt für eine 4 fordern (ohne Runden)\*\*.

\* natürlich wird ein Lehrer das nie so machen. Noten werden immer gerecht vergeben und anhand eines vorher festgelegten Bewertungsmassstabes bestimmt.

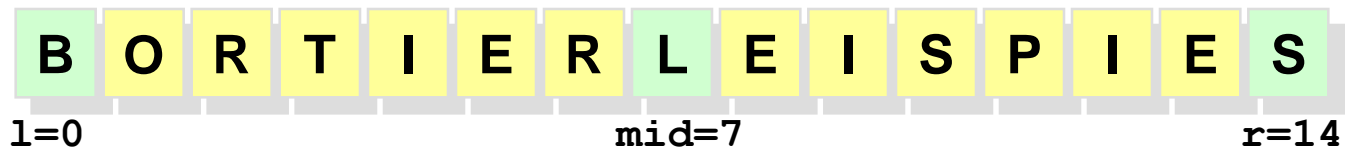
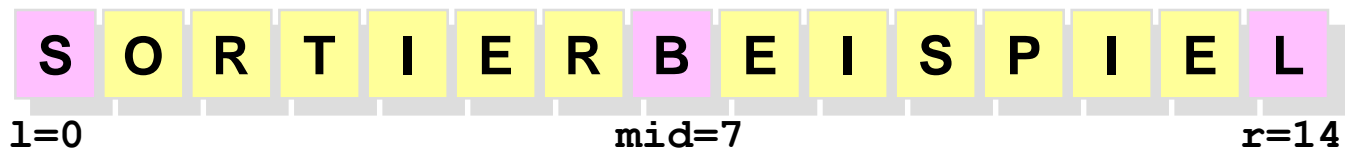
\*\* spätestens jetzt dürfte klar sein, dass das Beispiel völlig aus der Luft gegriffen ist und absolut keinen Bezug zur Realität hat.

# Pivotwahl nach sog. Median Methode

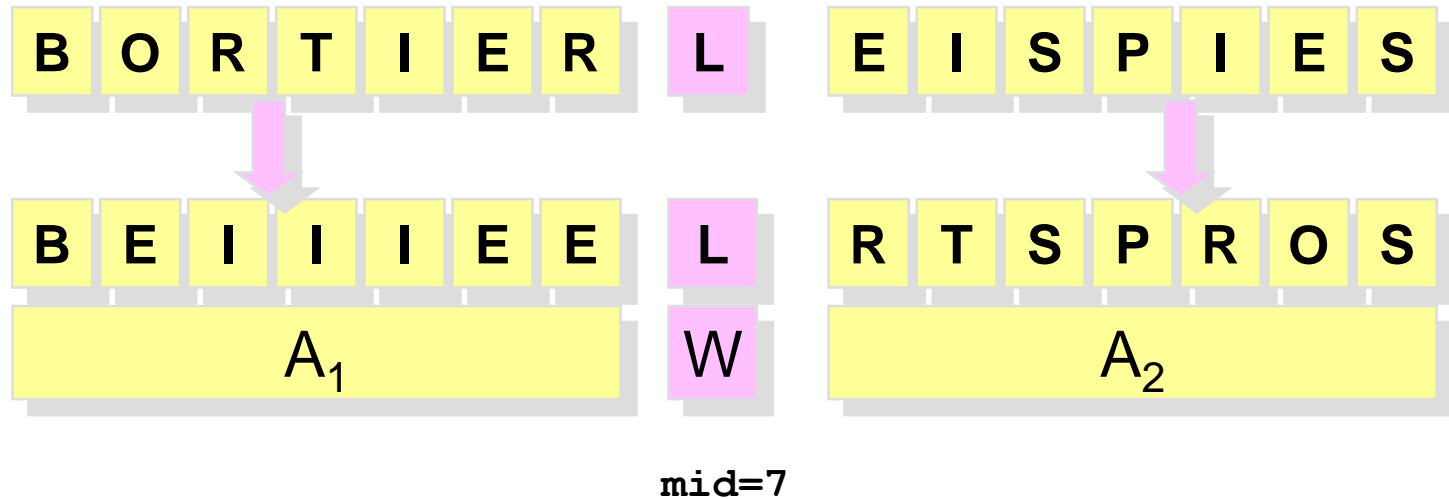
Die drei Elemente  $A[l]$ ,  $A[mid]$  und  $A[r]$  werden **vorsortiert** und man nimmt das dem Werte nach **mittlere dieser drei Werte**.

vorsortieren:

```
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) {...
```



- Dadurch ergibt sich eine bessere Pivot Wahl und ausgeglichene Hälften



- Optimale Partionierung erreicht
  - 15 Elemente aufgespalten in zwei Partitionen mit je 7 Elementen
  - muss aber nicht immer so sein



- teile den Bereich in zwei "alle kleiner" und "alle grösser" als Pivot
- gebe Index der Grenze zurück

```
public static int partition (int[] a, int left, int right) {  
    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(a, left, right);  
                left++;  
                right--;  
            }  
        }  
        return left;  
    }  
}
```

finde linkes Element  
grösser Pivot

finde rechtes Element  
kleiner Pivot

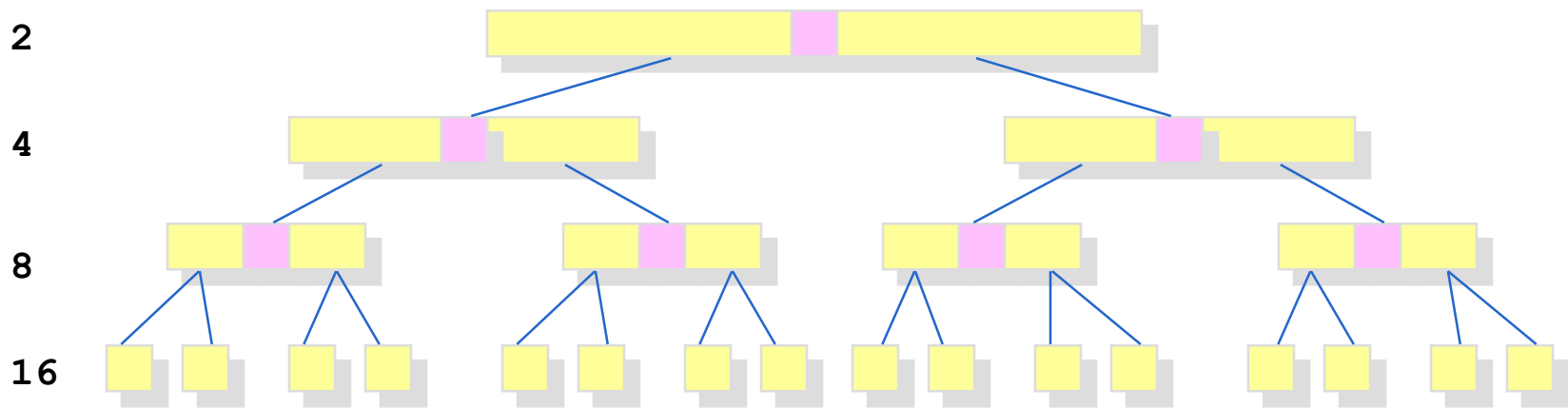
schon gekreuzt?

index der neuen Grenze

die eigentliche Sort-Methode ist rekursiv:

```
void quickSort(int[] a, int left, int right) {  
    if (left < right) {  
        int mid = partition (a, left, right);  
        quickSort(arr, left, mid-1);  
        quickSort(arr, mid , right);  
    }  
}  
  
public void quickSort(int[] a){  
    quickSort(a, 0, a.length-1);  
}
```

- 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  $N \times \log_2(N)$ .
- Die Ordnung von Quicksort ist somit  $O(N \times \log(N))$ ,
  - wenn bei jeder Partitionierung eine gleichmässige Aufteilung der Daten erfolgt

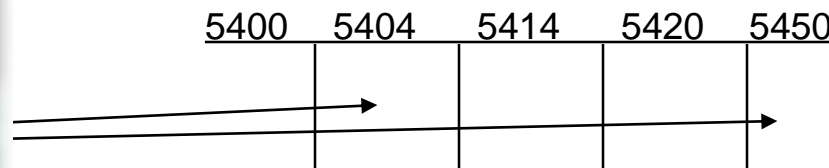


- ungünstigster Fall:
  - jeder Partitionierungs-Schritt nur jeweils das Vergleichselement
  - -> Baum wird zu Kette (Liste) mit  $N$  Elementen:  
man sagt auch der Baum **entartet** oder **degeneriert**
- In diesem Fall ist der Aufwand proportional zu  $O(N^2)$ .
- Dies tritt jedoch nur in extra konstruierten Fällen auf
- Im Normalfall ist die Partitionierung bei Quicksort nahezu optimal

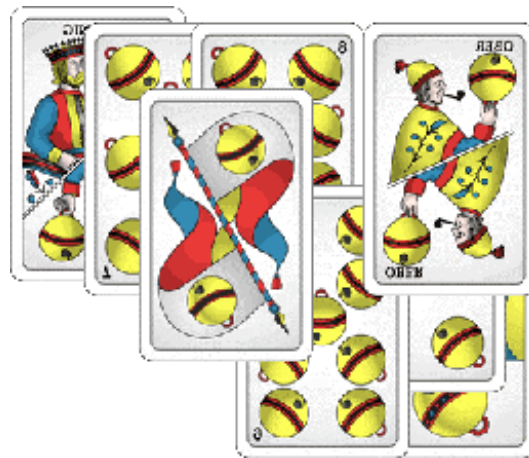
Quicksort immer die **erste Wahl**, wenn grössere Mengen von ungeordneten Daten sortiert werden müssen.

# Distribution Sort

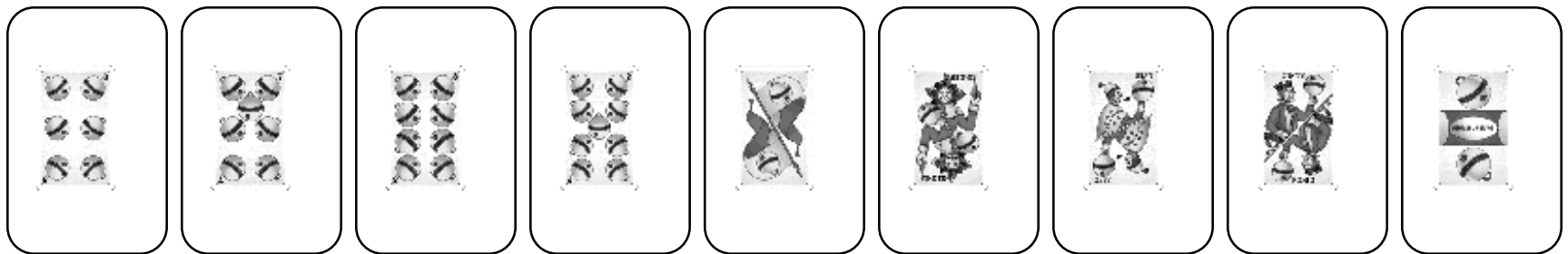
- Die bisher diskutierten Sortialgorithmen basieren auf den Operationen: **Vergleichen** zweier Elemente und ev. **Vertauschen** zweier Elemente (swap).
- Im Gegensatz dazu kommt **Distribution-Sort ohne Vergleiche** aus.
- Die zu sortierenden Elemente werden
  - entsprechend dem Sortierschlüssel in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



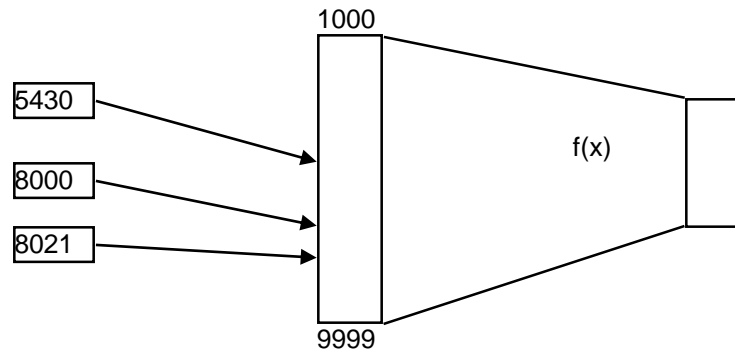
# Distribution-Sort Beispiel



**direktes Einfügen**



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



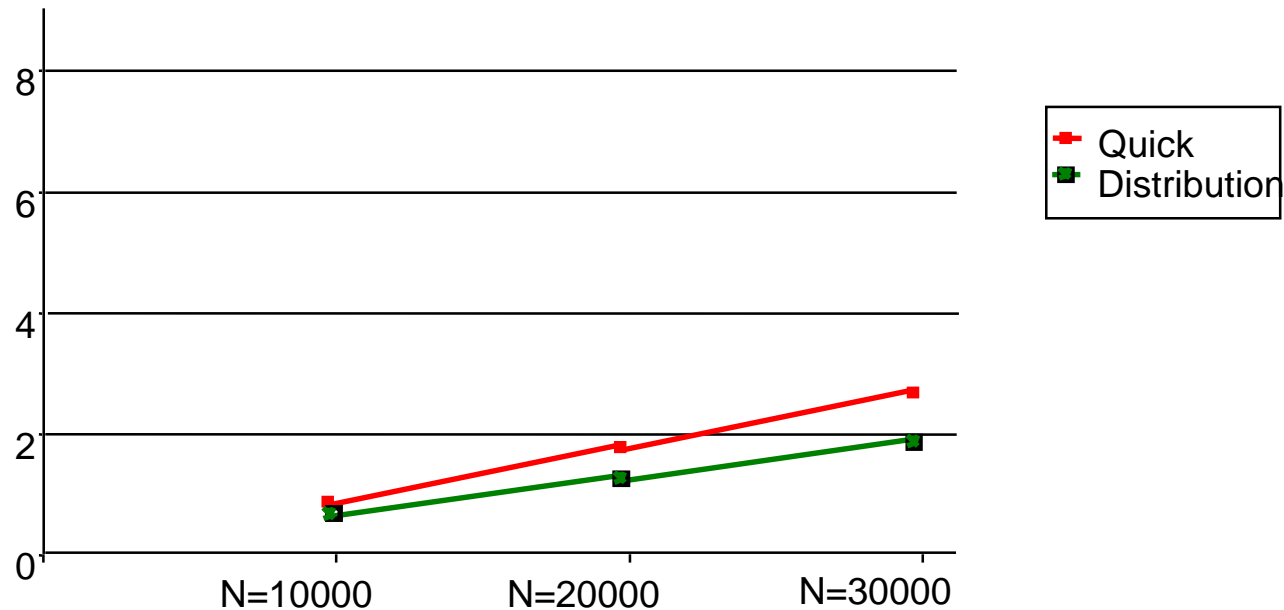
- Vorteile
  - schneller geht's nicht
  - linearer Algorithmus: die Komplexität ist also  $O(N)$
- Nachteile
  - Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
  - Geht nur bei Schlüsseln, die einen kleinen Wertebereich haben oder auf einen solchen abgebildet werden können, ohne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



# Laufzeitvergleich schneller Sortieralgorithmen

Laufzeitmessung bei zufällig sortierten Daten (in Sekunden):

	N=10 000	N=20 000	N=30 000	N=40 000
<b>Quick</b>	1.57	3.38	5.21	7.19
<b>Distribution</b>	1.2	2.4	3.6	4.8



- Bei vorsortierten Daten:

	N=10000	N=20000	N=30000	N=40000
<b>Quick</b>	0.48	1.02	1.43	2.16
<b>Distribution</b>	1.2	2.4	3.6	4.8

- 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 (wird z.B. im JDK-Sort gemacht).
- -> Insertionsort verwenden

- Ein  $O(n \cdot \log n)$  Sortieralgorithmus brauche 1 Sekunde für 10'000 Elemente; wie lange braucht er für 100'000 Elemente?
  - es soll nur der erste Koeffizient betrachtet werden.

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

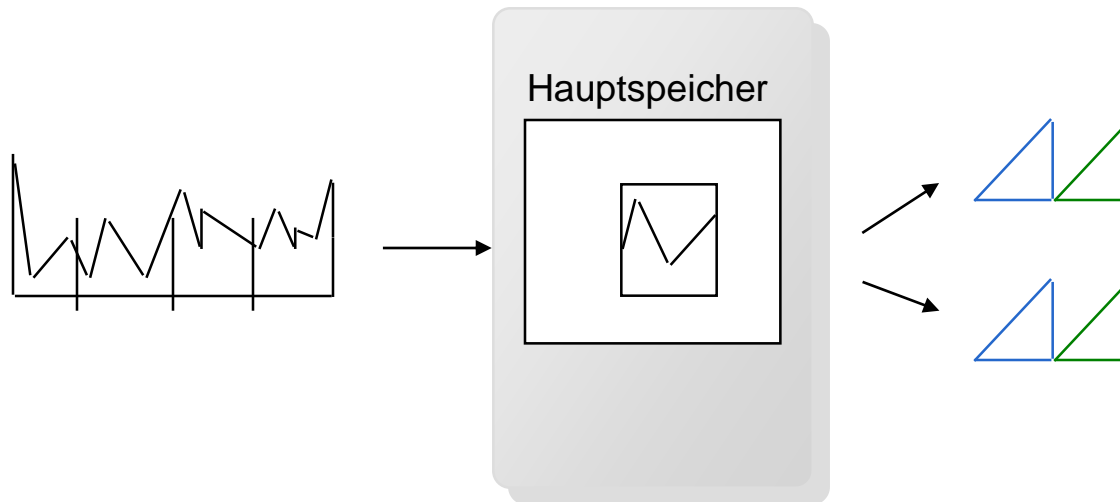
- **Distribution-Sort** fällt nicht darunter, da er
  - **nicht** auf dem Vergleich von Elementen untereinander beruht.
  - auf **Verteilen** und **Zusammentragen** von Datensätzen mit Hilfe von Fächern basiert.

# Externe Sortierverfahren

- 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 Element wäre zwar möglich, ergibt aber einen grossen Effizienzverlust (siehe Vorlesung BS)
  
- Annahmen:
  - Datenstrom der sequentiell gelesen wird
  - Jeweils nur ein Teil der Daten passt in den Hauptspeicher

# 1. Phase: Sortieren-Verteilen (Sort)

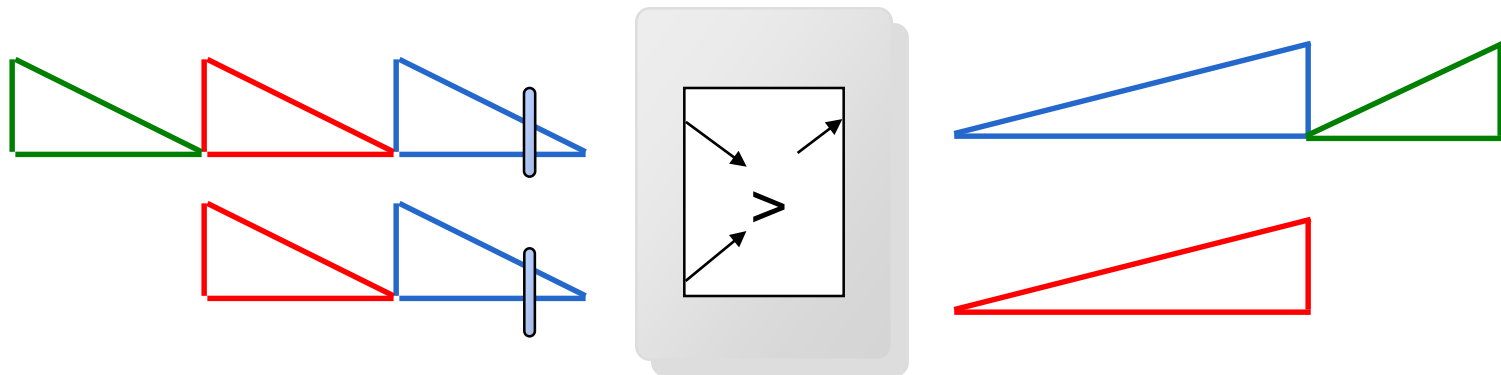
- Lade jeweils einen Teil der Datei in den Speicher
  - Sortiere diesen Teil mit schnellem internem Verfahren
  - Schreibe sortierte Abschnitte in mehrere (mindestens zwei) Ausgabedateien
  - Es entstehen Folgen von sortierten Abschnitten in den Ausgabedateien





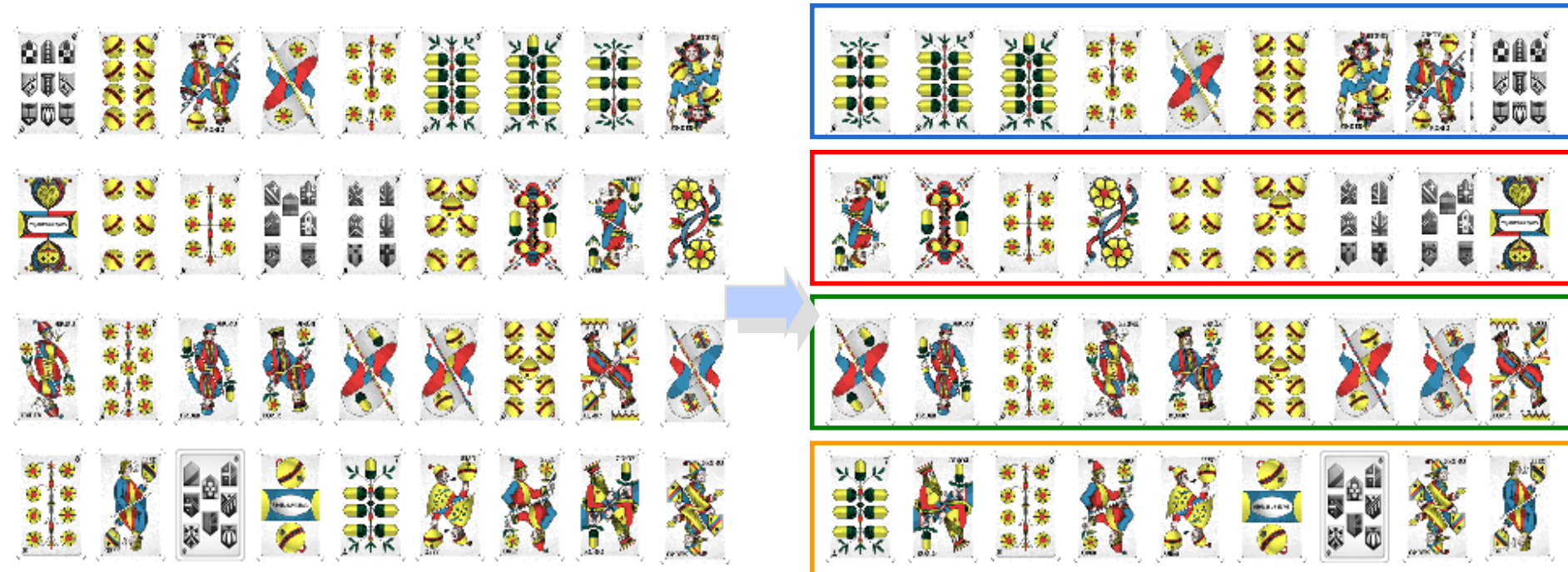
## 2. Phase: Mischen (Merge)

- 2. Phase: Mischen
  - Lese von beiden Dateien das erste Element
  - Schreibe das Kleinere und lese das Nächste von der gleichen Datei
- -> Länge der geordneten Abschnitte hat sich verdoppelt
  - output -> input
  - solange wiederholen, bis vollständig sortiert

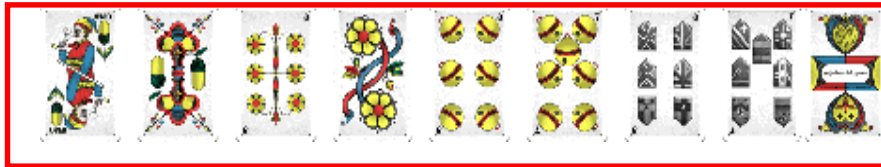


# Beispiel Jasskarten

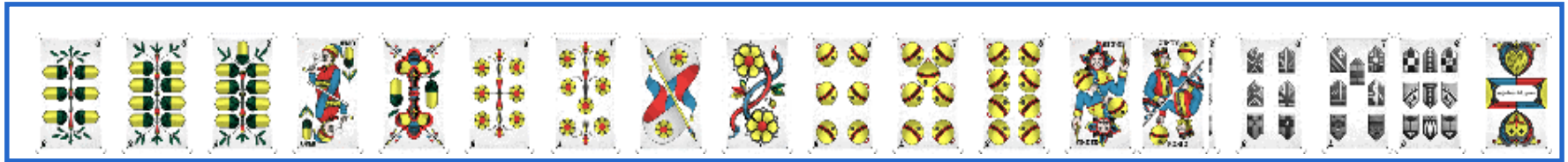
intern sortieren  
(nach Farbe und Punkte)



# Misch Durchgang 1



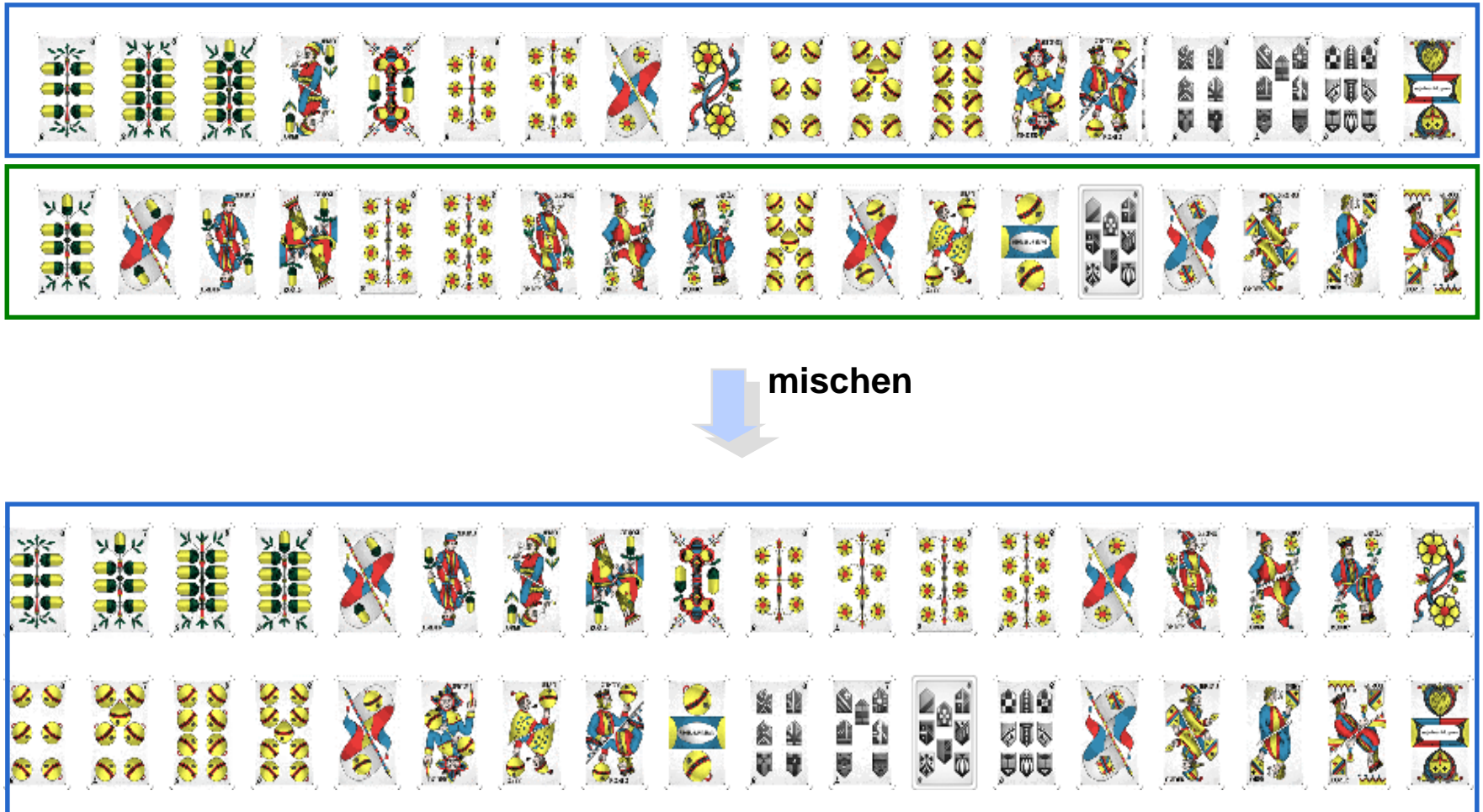
mischen



mischen



# Misch Durchgang 2





## ■ Annahme:

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17		18		19		20		21		22		23		24	
25				26				27				28			
29								30							
31															

## ■ Generell, bei n Sequenzen und m Eingabedateien:

- Aufwand Sortierphase:  $n$
- Aufwand je Mischphase:  $n$
- Anzahl Mischphasen:  $\lfloor \log_m n \rfloor$
- Gesamtaufwand Sortieren und Mischen:  $n + n * \lfloor \log_m n \rfloor \Rightarrow O(n \log_m n)$

# Wahl des Sortierverfahrens

- **Internes oder externes Verfahren**
  - Sortieren im Hauptspeicher
  - Sortieren im Hauptspeicher und Sekundärspeicher (Platte)
  
- **Methode des Algorithmus:**
  - Vertauschen
  - Auswählen
  - Einfügen
  - Rekursion
  - Verteilen
  - Mehrphasen: Sortieren-Mischen

- Nach Effizienz:
  - Laufzeit:  $O(n^2)$  oder  $O(n \times \log n)$  oder sogar  $O(n)$
  - Speicherbedarf: Wieviel Speicher wird zusätzlich zu dem für die zu sortierenden Daten benötigt?
  
- Allgemein / mit speziellen Voraussetzungen:
  - allgemein: Benötigt nur Schlüsselvergleiche
  - allgemein: Ändern der Reihenfolge oder Angabe der Reihenfolge
  - allgemein: Wird die Reihenfolge von Datensätzen mit gleichem Sortierkriterium durch den Algorithmus geändert? → Stabilität
  - **speziell**: Der Algorithmus setzt eine bestimmte Struktur der Schlüssel voraus.



- Bibliotheksfunktion -> Collections.sort oder Arrays.sort
- wenige Datensätze (weniger als 1000),
  - 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 [Quicksort](#) bevorzugen.
- viele Daten, ungeordnet, sehr oft zu sortieren
  - [Distribution-Sort](#) an das spezielle Problem angepasst
- sehr viele Daten
  - externes Sortiervfahren in Kombination mit schnellem Internem

# Optimierung durch Kombination von Algorithmen

- Quicksort ist schnell, aber wegen der Rekursion hat er einen relativ grossen initialen Aufwand: *"Footprint"*
- Für wenige (~10..100) zu sortierende Daten ist ein einfaches Sortierverfahren schneller
- Idee: ab einer bestimmten Länge des Intervalls nicht mehr Quicksort verwenden, sondern z.B. InsertionSort; bringt ca.10%

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

# Einschub Zufallszahlen

- **Gleichverteilt:** Jede Zahl innerhalb eines Intervalls kommt gleich häufig vor.  
Beispiel Zahlen eines Würfels
- **Die Wahl der nächste Zahl einer Folge muss zufällig (ohne Bildungsgesetz) erfolgen.**
- -> Es ist schwierig oder sogar unmöglich, echte Zufallszahlen auf einem Computern zu erzeugen
- echt zufällige physikalische Prozesse
  - Würfeln
  - radioaktiver Zerfall
  - Elektronen-Rauschen in einer Diode
  - Braunsche Bewegung
- echter Zufall in der Mathematik
  - nächste Zahl an einer beliebigen Stelle der Zahl PI
    - 3.1415926535897932384626433832795

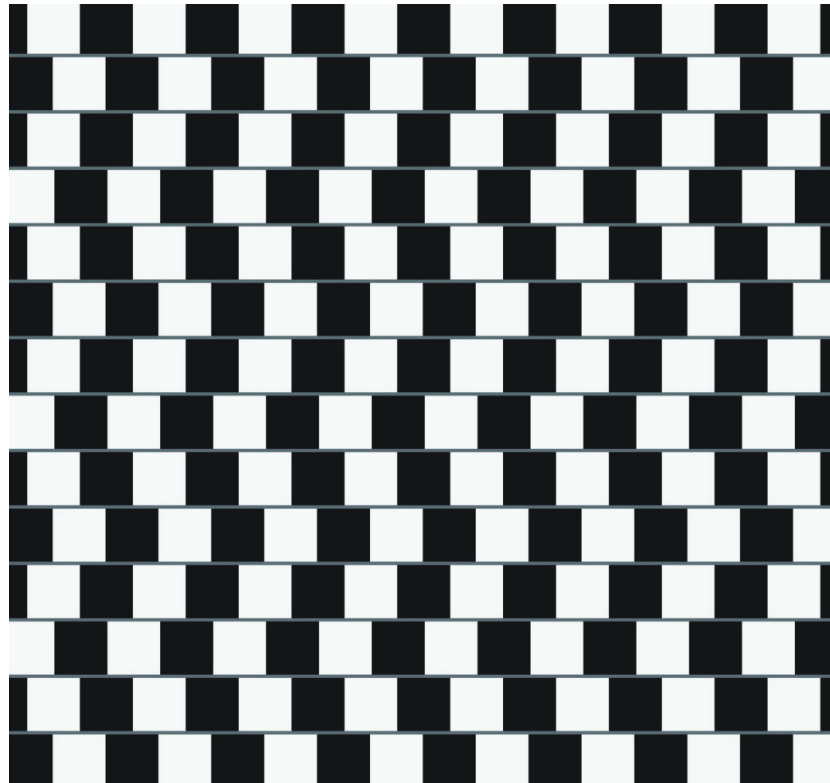
- Es werden **Folgen von Zahlen** erzeugt, die möglichst viele Eigenschaften von Zufallszahlen besitzen.
- Es wird eine Funktion bestimmt, bei der die Auswahl der nächsten Zahl scheinbar zufällig erfolgt: es kommt erst nach langer Zeit zu einer Wiederholung der Folge -> **Pseudozufallszahlen**

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

- gute Wahl von a,c und p ist schwierig; kann nur durch statische Analysen überprüft werden
- Nachteil: keine "echte" Zufallszahl
- Vorteil: es kann mehrmals die gleiche Folge erzeugt werden.
- in Java-Klasse Math: `Math.random()` für Bereich `[0..1[`
- für beliebigen anderen Bereich: `r = (int)(k*Math.random())`

- Die Java-Klasse `Random` erlaubt grössere Flexibilität (als `Math`).
- `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 `[0..n[`
- `nextDouble()` liefert eine pseudozufällige, gleichverteilte Double-Zahl im Bereich `[0..1[`
- `nextGaussian()` liefert eine pseudozufällige, Gauss-verteilte Double-Zahl mit Mittelwert 0.0 und Standard-Abweichung 1.0.

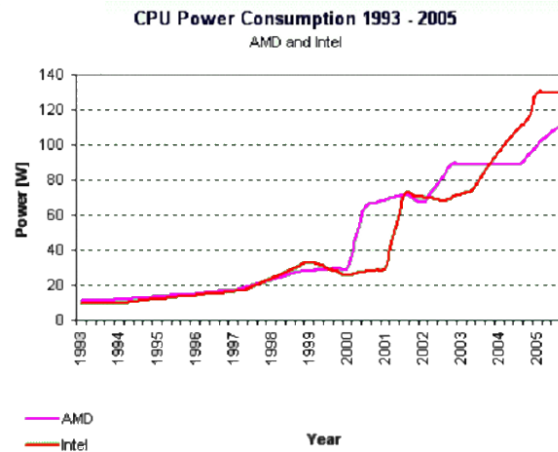
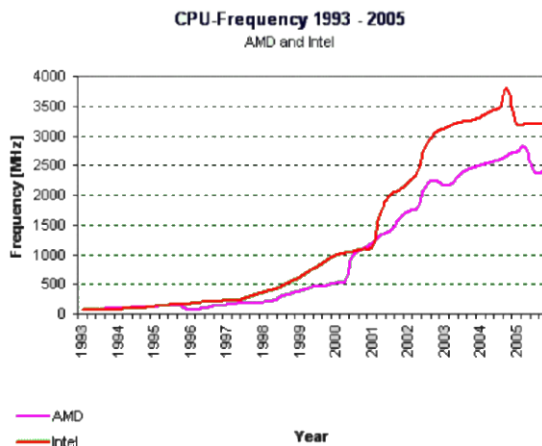
# Optimierung durch Parallelisierung





# 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 2% der weltweiten CO<sub>2</sub> Emissionen verantwortlich
- Rechenzentren: Stromkosten p.a. ~ HW Kosten



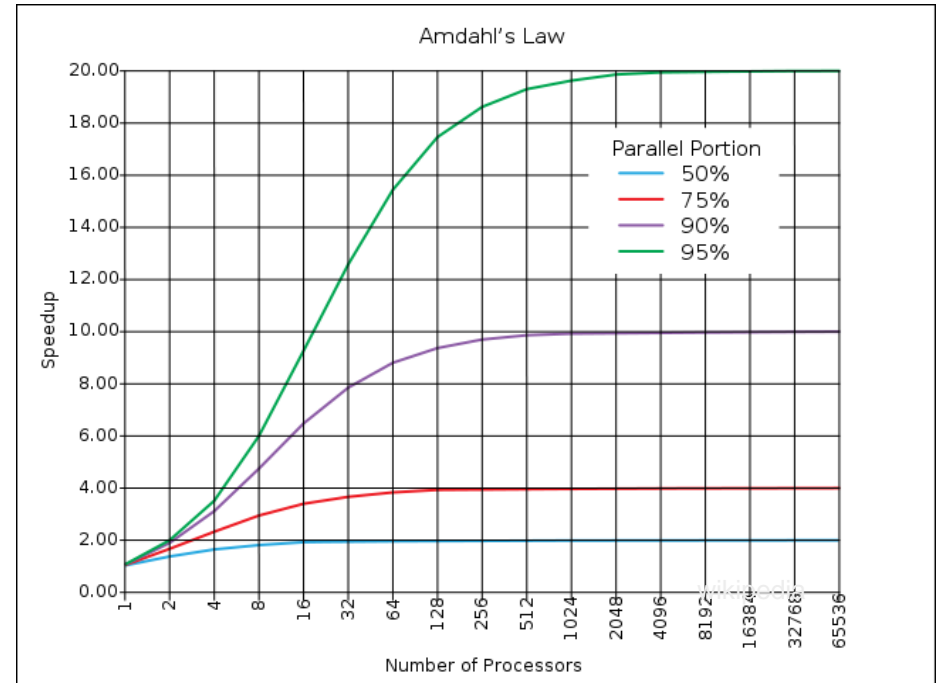
- Modern multicore CPUs sprechen eigentlich dafür aber
- Amdahl's Law

- Die Geschwindigkeitssteigerung ist durch den sequentiellen Anteil im Programm limitiert

P: ist der Anteil des Programms,  
der parallelisiert werden kann

N: Zahl der Prozessoren

$$\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{N}}$$



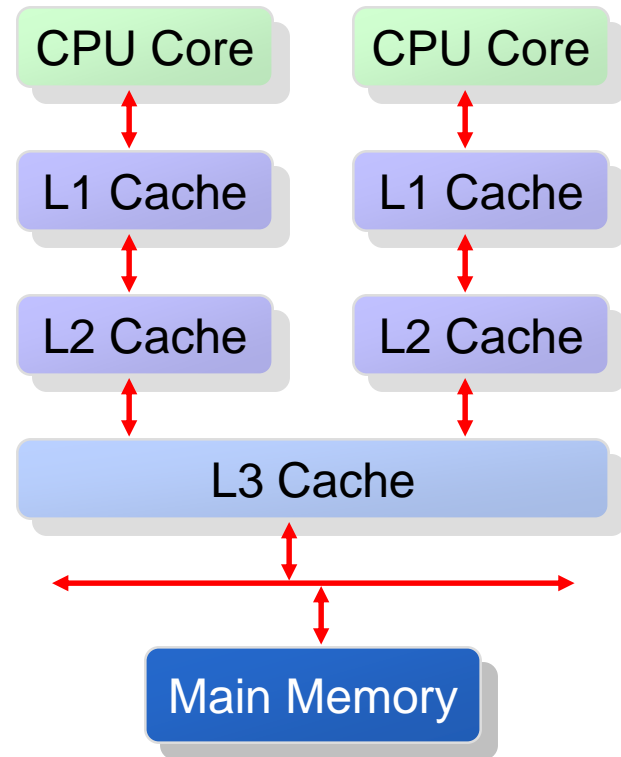
- Es kommt aber noch ein Overhead durch die Steuerung der Threads

# Zugriff auf den Hauptspeicher

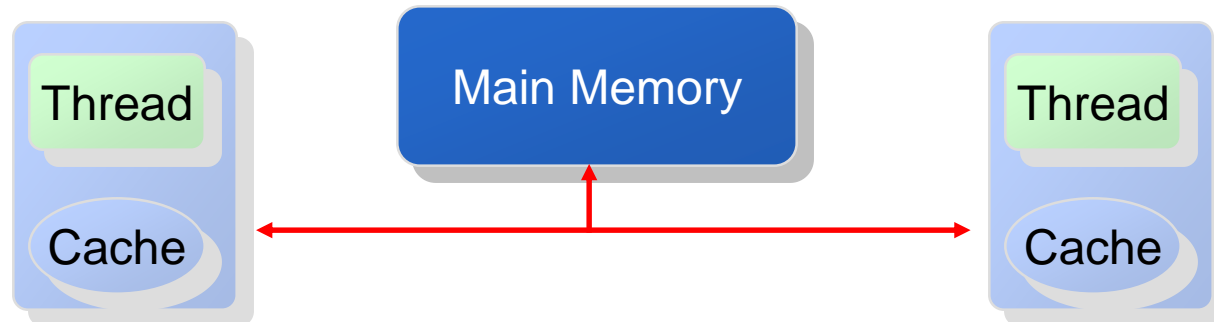
- Zugriffszeiten

register:	<	1 clk
L1:	~	4 clks
L2:	~	11 clks
L3:	~	40 clks
main memory:	~	150-300 clks

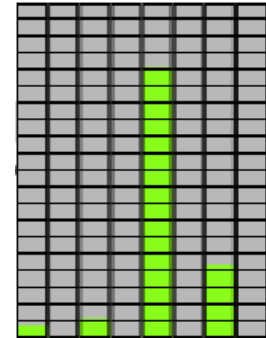
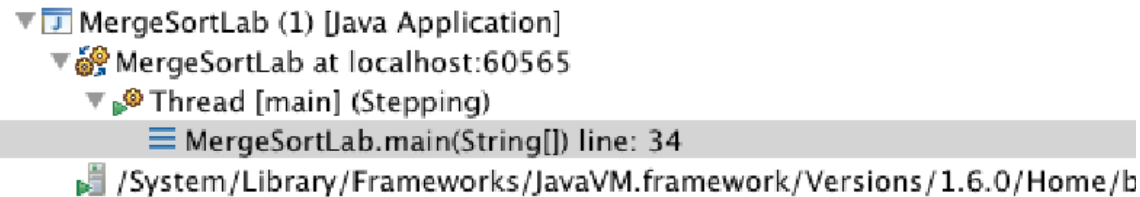
- Hauptspeicherzugriff ist im Vergleich zum Cache sehr teuer



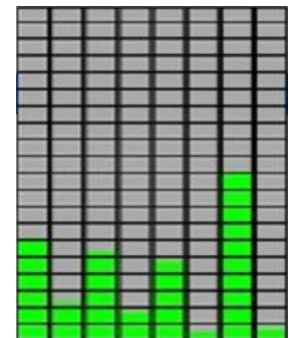
- JMM abstrahiert von physikalischem Memory Modell
- Alle Threads teilen Hauptspeicher
- Jeder Thread hat einen lokalen Arbeitsspeicher
- Veränderungen aus lokalem (Thread-) Arbeitsspeicher werden zurückgeschrieben
  - Bei expliziter Synchronisation
  - Thread Start und Beendigung
  - Lesen/Schreiben von *volatiles*
  - Erstes Lesen von *finals*



## ■ Single Threaded



## ■ Multithreaded



# NaiveParallelQuicksort

Als separater Thread

private int[] a;  
private int left;  
private int right;

```
class NaiveParallelQuicksort extends Thread {  
    ....
```

```
    public NaiveParallelQuicksort(int[] a, int left, int right) {  
        this.a = a; ...
```

Setzen der  
Parameter

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

```
    public void run() {  
        int l = 0;  
        Thread t1 = null;  
        Thread t2 = null;  
  
        if (left < right) {  
            l = partition(a, left, right);  
            t1 = new NaiveParallelQuicksort(a, left, l-1);  
            t1.start();  
            t2 = new NaiveParallelQuicksort(a, l, right);  
            t2.start();  
            if (t1 != null) t1.join();  
            if (t2 != null) t2.join();  
        }  
    }
```

Parallele Threads  
starten

Warte bis  
beendet

- Je ein neuer Thread für jede Partitionsaufgabe
- Java Threads werden auf Betriebssystems Threads abgebildet: "Kernel Level Threads"
- Vorteile
  - Die Rechenzeituteilung kann besser (von BS) verwaltet werden
  - Sind mehrere CPU-Kerne/Proz. im Rechner vorhanden, können diese ausgenutzt werden
- Aber
  - Erzeugung und Zerstörung von Threads kostet (viel) Zeit
  - Instanzierte Threads belegen Speicher
  - Ineffektiv, mehr Threads zu erzeugen, als die Prozessoren gleichzeitig handhaben können, da immer einige inaktiv (idle) sein werden

# NaiveParallelQuicksort erste Korrektur

```
class NaiveParallelQuicksort extends Thread {  
    .....  
    private final int SPLIT_THRESHOLD = 100000;  
  
    public void run() {  
        int l = 0;  
        Thread t1 = null;  
        Thread t2 = null;  
  
        if (left < right) {  
            l = partition(a, left, right);  
            if (l - left > SPLIT_THRESHOLD) {  
                t1 = new NaiveParallelQuicksort(a, left, l-1);  
                t1.start();  
            } else {  
                Quicksort.sort(a, left, l-1);  
            }  
            if (right - l > SPLIT_THRESHOLD) {  
                t2 = new NaiveParallelQuicksort(a, l , right);  
                t2.start();  
            } else {  
                Quicksort.sort(a, l , right);  
            }  
            if (t1 != null) t1.join();  
            if (t2 != null) t2.join();  
        }  
    }  
}
```

Nur wenn Task  
genügend gross  
parallel ausführen

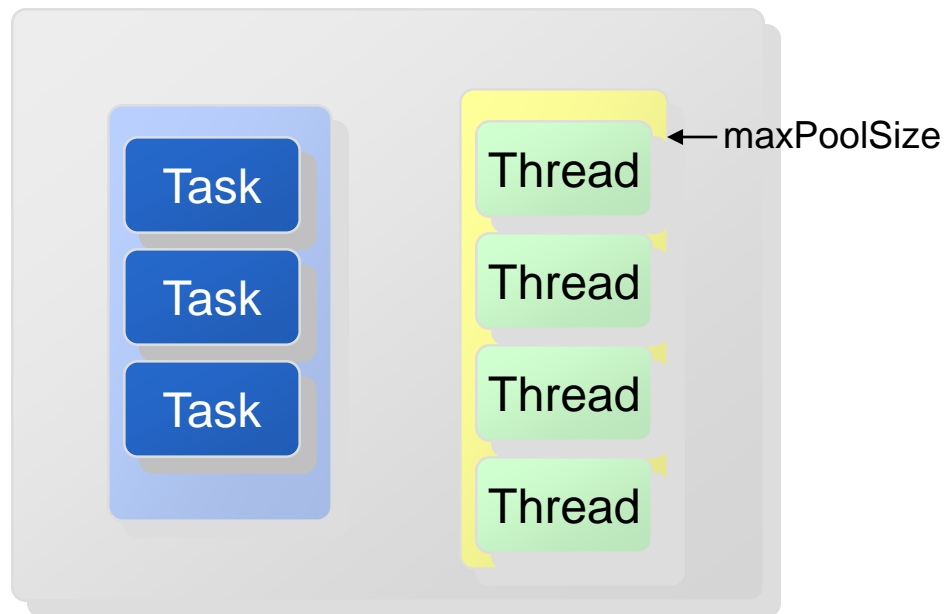
Sonst sequentiell



# Threadpools

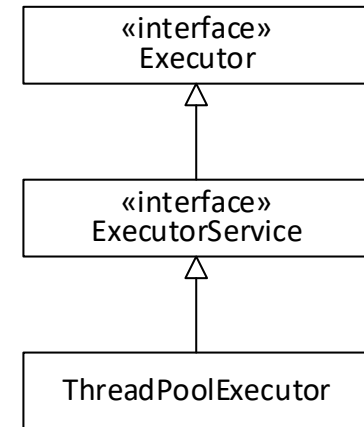
- Es wird einer gewissen Anzahl von einmal gestarteten Threads (Pool von Threads) immer wieder eine neue Aufgabe (Tasks) übergeben.
- Pool Size entspricht in etwa der Anzahl Rechnerkerne
  - bei I/O intensiven Threads mehr, da immer ein Teil wartend/blockiert ist
- Die Reihenfolge der Abarbeitung der Tasks wird durch eine Ausführungsstrategie bestimmt

```
java.lang.Runtime.getRuntime().availableProcessors()
```



- bessere Kontrolle über Ressourcen durch Einhaltung einer Ausführungsstrategie
  - in welchem Thread wird ein Task ausgeführt
  - in welcher Reihenfolge werden die Tasks ausgeführt (LIFO, FIFO, priorisiert, ...)
  - Anzahl der Tasks die gleichzeitig ausgeführt werden dürfen
  - Anzahl der Tasks die auf Ausführung warten dürfen
  
- Mögliche Ausführungsstrategien:
  - Begrenzung der Thread-Anzahl
  - Caching von Threads
  - Queueing von Tasks
  
- Auswahl effektivster Policy hängt vom Anwendungsfall ab

- Interface `Executor` (`java.util.concurrent`) stellt Framework zur Thread-Ausführung bereit, das verschiedene Ausführungsstrategien realisiert
- Interface `ExecutorService` für ThreadPools
- verschiedene Varianten des `ThreadPoolExecutors` werden durch Factory-Methoden der Klasse `Executors` bereitgestellt:
  - `FixedThreadPool`, (`public static ExecutorService newFixedThreadPool(int nThreads)`) Pool mit fester Menge von Threads, die zur Ausführung eingereicher Tasks eingesetzt werden
  - `CachedThreadPool`, (`public static ExecutorService newCachedThreadPool()`): Pool mit flexibler Menge von Threads, die sich an die Zahl der eingereichten Tasks anpasst
  - `SingleThreadExecutor`, (`public static ExecutorService newSingleThreadExecutor()`): einzelner Thread, der eingereichte Tasks nacheinander (gemäß FIFO- oder LIFO-Prinzip oder nach Priorität) abarbeitet
  - `ScheduledThreadPool`, (`public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`): Pool mit fester Menge von Threads, die zur zeitgesteuerten bzw. wiederkehrenden Ausführung eingereicher Tasks eingesetzt werden
  - Und andere...



- Erzeugung via Fabrikmuster Methoden

```
threadPool = Executors.newFixedThreadPool(parallelism);
```

- Drei Zustände:

1. **running**

- Threadpool nimmt Tasks entgegen und führt sie aus, sobald Threads verfügbar sind

2. **shutting down** – Varianten:

```
threadPool.shutdown();
```

- graceful shutdown
- Threadpool führt laufende und bereits angenommene, aber nicht begonnene Tasks noch aus, nimmt jedoch keine neuen Tasks mehr an
- abrupt shutdown
- Threadpool führt laufende Tasks noch aus, verwirft aber bereits angenommene, noch nicht laufende Tasks, und nimmt keine neuen Tasks mehr an

3. **terminated**

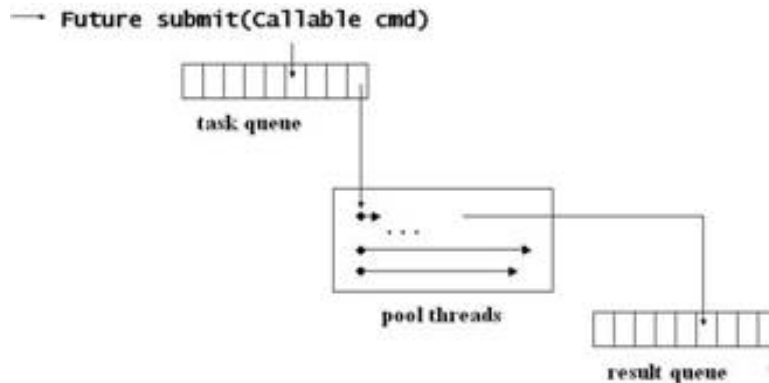
- keine Tasks werden mehr ausgeführt oder angenommen

- Einstellen eines Tasks in einen Threadpool

```
Future<V> submit(Runnable task)  
Future<V> submit(Callable task)
```

Keine Rückgabewerte

Mit Rückgabewerten



- Verwendung von Callable-Interface wenn Rückgabewerte/Resultate benötigt werden (Runnable liefert NULL-Wert bei Erfolg)
- Ein Future (Interface) stellt das Ergebnis einer asynchronen Berechnung dar

- **Submit** des **Runnable** (oder **Callable**) in Threadpool liefert **Future**
  - mit **get**-Methode zur Abfrage des Rückgabewerts vom Typ **V**
    - wartet, bis Rückgabewert feststeht
    - ggf. nur solange, bis angegebener Timeout erreicht
  - mit **cancel**-Methode zur Stornierung der Aufgabe
    - unmittelbare Streichung, wenn Bearbeitung noch nicht begonnen
    - Abbruchversuch, wenn schon in Bearbeitung

```
public interface Future<V> {  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
    V get() throws InterruptedException, ExecutionException,  
           CancellationException;  
    V get(long timeout, TimeUnit unit)  
        throws InterruptedException, ExecutionException,  
           CancellationException, TimeoutException;  
}
```

warte auf Beendigung  
des Tasks

- Realisierung der Anwendungslogik mit Rückgabebetyp **V** in der Methode **call** einer Implementierung des Interfaces **Callable** statt **Runnable**:

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

- Beispiel:

```
class Foo implements Callable<Integer> {  
    ...  
    public Integer call() {  
        int result = ...;  
  
        return result;  
    }  
}
```



- im Paket `java.util.concurrent` gibt es einige Thread Safe Collections
- Im Fall von erwarteten Race Conditions (oder ähnlichem) sollten diese verwendet werden

**`ConcurrentHashMap<K,V>`**

// A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates

**`ConcurrentLinkedQueue<E>`**

// An unbounded thread-safe queue based on linked nodes.

**`SynchronousQueue<E>`**

// A blocking queue in which each put must wait for a take, and vice versa.

# Beispiel QuicksortTask

```
public class QuicksortTask implements Runnable {

    static ExecutorService threadPool;
    static ConcurrentLinkedQueue<Future> futureList;

    public void sort(int[] a) {
        int parallelism = java.lang.Runtime.getRuntime().availableProcessors()*2;
        threadPool = Executors.newFixedThreadPool(parallelism);

        futureList = new ConcurrentLinkedQueue<Future>();
        QuicksortTask rootTask = new QuicksortTask(a, 0, a.length - 1);
        futureList.add(threadPool.submit(rootTask));
        while (!futureList.isEmpty()) {
            futureList.poll().get();
        }
        threadPool.shutdown();
    }

    public void run() {
        int l = 0;
        if (left < right) {
            l = partition(a, left, right);
            if (l - left > SPLIT_THRESHOLD) {
                futureList.add(threadPool.submit(new QuicksortTask(a, left, l-1)));
            }
            ...
        }
    }
}
```

Initialisiere Threadpool

starte ersten Task

warte auf Beendigung  
aller Tasks

Terminierte Threadpool

füge neuen Task in  
Threadpool

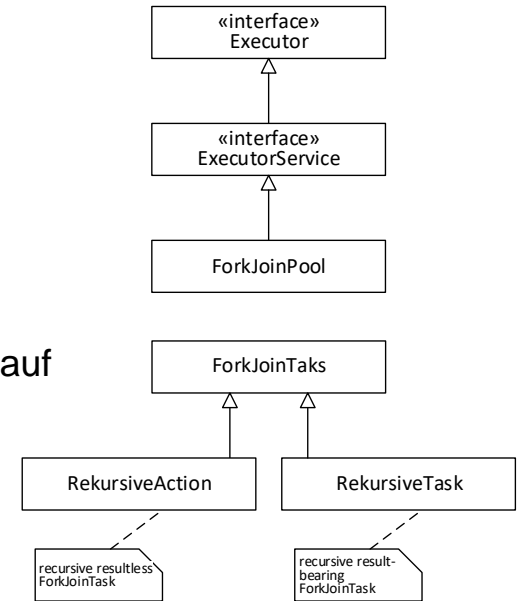
# Fork Join

- Neues Framework für die parallele Ausführung ab Java 7
- Macht grundsätzlich dasselbe wie Threadpools
- Threads manchmal immer noch zu schwergewichtig
- "Teile und Herrsche" Prinzip auf Parallelität angewandt

```
Result solve(Problem problem) {  
    if (problem is small)  
        directly solve problem  
    else {  
        split problem into independent parts  
        fork new subtasks to solve each part  
        join all subtasks  
        compose result from subresults  
    }  
}
```

# So wird es gemacht

- Definiere ein ForkJoinPool-"thread-manager"-Object
- Definiere ein Task-Object das von der Klasse RecursiveTask erbt
  - Kann generisch sein, wenn Rückgabewerte vorhanden sind.
  - Instanziere den ForkJoinPool
  - Instanziere (Haupt-)Task Objekt und rufe `fjpool.invoke(task)` auf
- In der compute-Methode (innerhalb Task-Object)
  - Code mit der Aufteilung des Problems
  - Löse Problem direkt falls klein/einfach genug
  - Sonst teile Problem auf:
    - rufe `invokeAll(task1, task2, ...)`. dieses wartet bis alle beendet oder
    - starte task `fork()` auf dem ersten und `invoke()` mit dem zweiten mit anschliessendem `join()`



# Threads vs. Fork/Join

## Threads

Do **subclass** Thread

Do **override** run

Do **call** start

Do **call** join (je Thread)

## Fork/Join

Do **subclass** RecursiveTask<V>

Do **override** compute

Do **call** invoke, invokeAll, fork

Do **call** join which returns answer  
or

Do **call** invokeAll on multiple tasks

# Beispiel QuicksortForkJoin - invokeAll

```
public class QuicksortForkJoin extends RecursiveAction {
```

```
    public static void sort(int[] a) {  
        int parallelism = java.lang.Runtime.getRuntime().availableProcessors()*2;  
        forkJoinPool = new ForkJoinPool(parallelism);  
        QuicksortForkJoin rootTask = new QuicksortForkJoin(a, 0, a.length - 1);  
        forkJoinPool.invoke(rootTask);  
    }
```

übergebe Task FJ Queue und  
warte auf Terminierung

```
    public void compute() {  
        int l = 0;  
  
        if (left < right) {  
            l = partition(a, left, right);  
            ForkJoinTask t1 = null, t2 = null;
```

```
            if (l - left > SPLIT_THRESHOLD && right - l > SPLIT_THRESHOLD) {  
                t1 = new QuicksortForkJoin(a, left, l-1);  
                t2 = new QuicksortForkJoin(a, l, right);  
                invokeAll(t1, t2);
```

```
            } else {  
                Quicksort.sort(a, left, l-1);  
                Quicksort.sort(a, l, right);  
            }
```

übergebe FJ Queue und warte  
auf Terminierung

```
        }  
    }
```

# Beispiel QuicksortForkJoin - fork/invoke/join

```
public class QuicksortForkJoin extends RecursiveAction {

    public static void sort(int[] a) {
        int parallelism = java.lang.Runtime.getRuntime().availableProcessors();
        forkJoinPool = new ForkJoinPool(parallelism);
        QuicksortForkJoin rootTask = new QuicksortForkJoin(a, 0, a.length - 1);
        forkJoinPool.invoke(rootTask);
    }

    public void compute() {
        int l = 0;

        if (left < right) {
            l = partition(a, left, right);
            ForkJoinTask t1 = null;
            if (l - left > SPLIT_THRESHOLD) {
                t1 = new QuicksortForkJoin(a, left, l-1).fork();
            } else Quicksort.sort(a, left, l-1);
            if (right - l > SPLIT_THRESHOLD) {
                new QuicksortForkJoin(a, l, right).invoke();
            } else Quicksort.sort(a, l, right);
            if (t1 != null) {
                t1.join();
            }
        }
    }
}
```

übergebe Task FJ Queue und  
warte auf Terminierung



- Teile und Herrsche Prinzip
  
- Quicksort
  - Partitionierung
  - Wahl des Pivots
  - Sequentielle Optimierungen
  
- Optimierung durch Parallelisierung
  - Naive
  - Threads
  - Threadpools
  - Fork/Join

## Java 8 Streams:

```
import java.util.*

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));
    }
}
```

## Haskell:

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