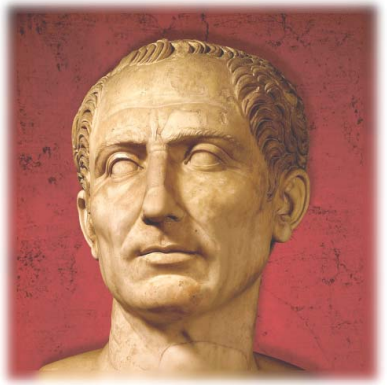
Sortierverfahren 2

- Sie kennen das Prinzip: "Teile und Herrsche"
- Sie kennen zwei schnelle Sortierverfahren: Quick Sort, 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 Sortieralgorithmen

School of Engineering

- 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;
      }
```

- 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
       }
}
```

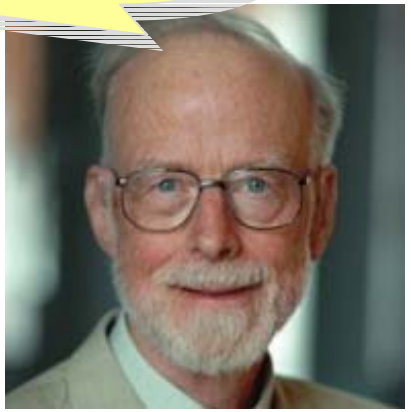
Bei der Zerlegung sollten die Teile möglichst gleich gross sein.

Idee des Quicksort

Quick-Sort

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

> 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.



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

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 $\mathbf{A}_1$ und $\mathbf{A}_2$ mit folgenden Eigenschaften:

  - $A = \mathbf{A}_1 \cup \mathbf{A}_2 \cup \{W\}$
  - Alle Elemente von $\mathbf{A}_1$ sind $\leq W$ (aber noch evtl. unsortiert).
  - Alle Elemente von $\mathbf{A}_2$ sind $\geq W$ (aber noch evtl. unsortiert).

Wenn jetzt $\mathbf{A}_1$ und $\mathbf{A}_2$ sortiert werden, ist das Problem gelöst.

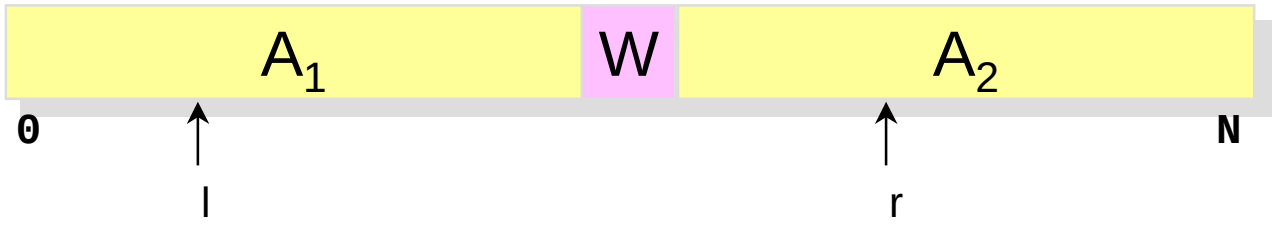
Quick-Sort

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

noch zu lösendes Problem:

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

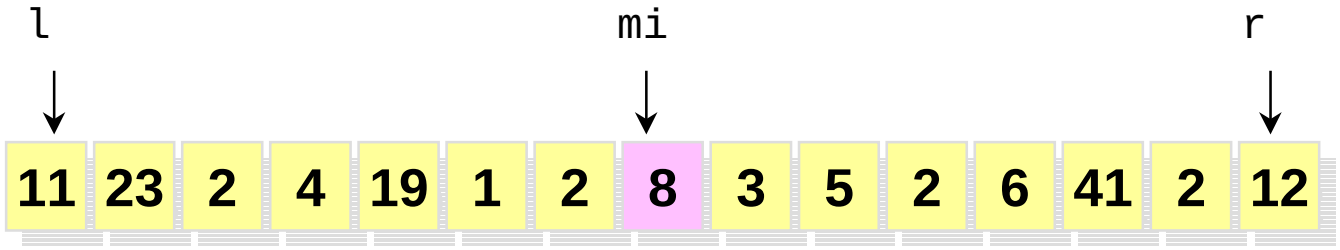
Quick-Sort



- Die Konstruktion der Partition erfolgt durch:
  - wähle ein Element W
  - suche von links ein Element, das auf falscher Seite ist, d.h. A[l] ≥ W
  - suche von rechts ein Element, das auf falscher Seite ist, A[r] ≤ W
  - vertausche von A[l] und A[r]
  - wiederhole obige Schritte bis l und r sich kreuzen, d.h. l ≥ r.

Aufgabe

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



Quick-Sort: Wahl des Pivots

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

W wird lediglich geschätzt

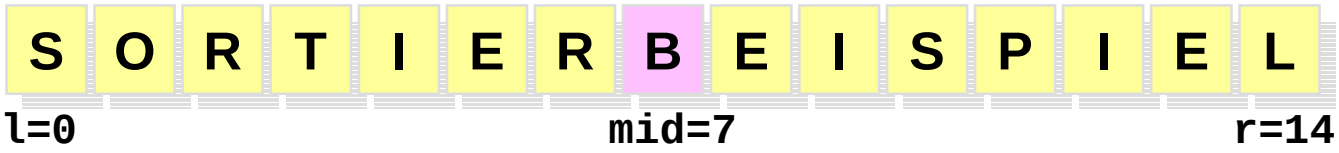
- folgende Pivotwahlen sind möglich
  - A[l] das (der Position nach) linke Element von A;
  - A[r] das (der Position nach) rechte Element von A;
  - A[mid] das (der Position nach) mittlere Element von A mit mid = (l+r)/2

- 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

Schlechte Pivotwahl

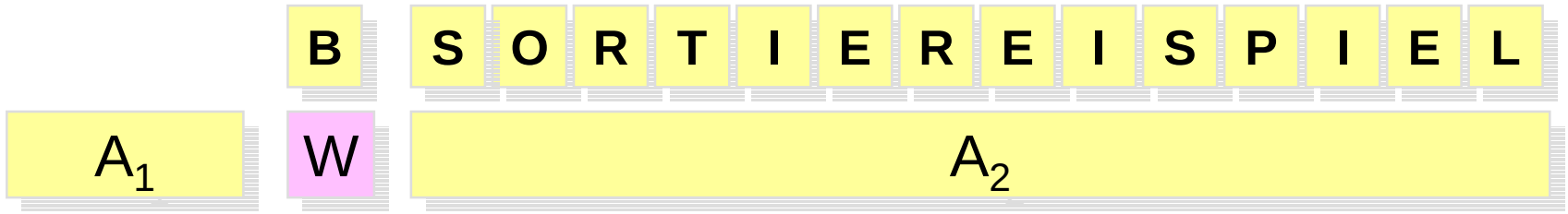
- Gutes Pivot-Element ist nicht unbedingt das Element, welches der Position nach in der Mitte liegt.

- 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.

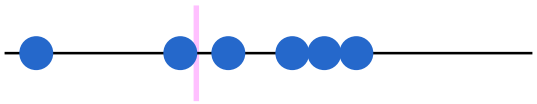


Quick-Sort: Wahl des Pivots

- W wird auch Pivot genannt
  - von entscheidender Bedeutung für die Effizienz von QuickSort.
  - optimal wäre ein Element, das A in **zwei gleich grosse Teile** partitioniert.

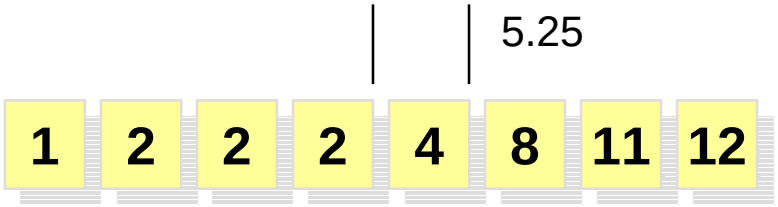
- W muss so bestimmt werden, dass gleich viele Werte grösser wie kleiner als W sind.

- 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.

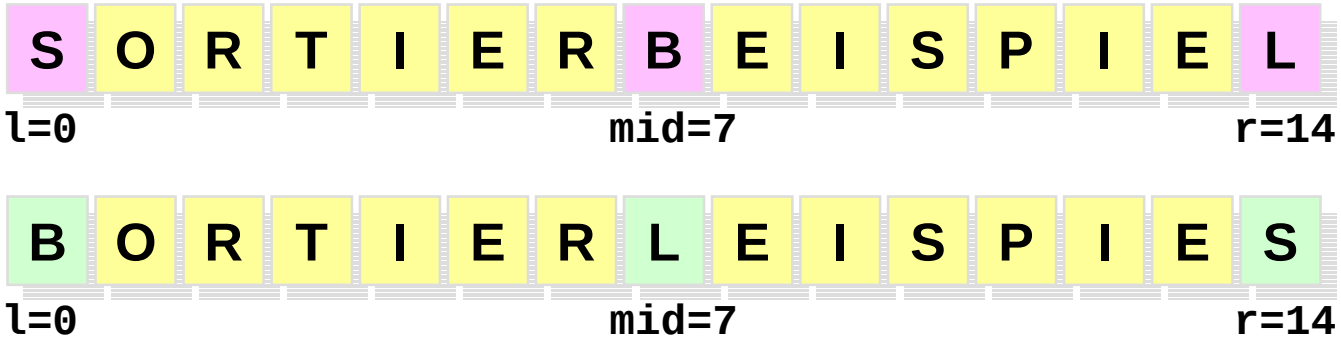
** spätestens jetzt dürfte klar sein, dass das Beispiel völlig aus der Luft gegriffen ist und absolut keinen Bezug zur Realität hat.

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**.

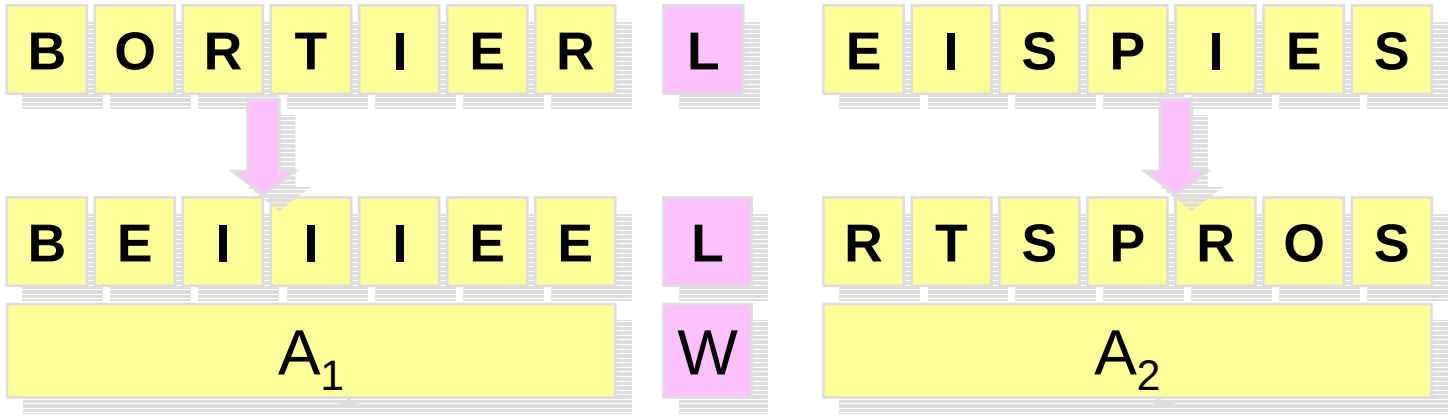
vorsortieren:

```
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) {...
```



Partition mit Median Methode

- Dadurch ergibt sich eine bessere Pivot Wahl und ausgeglichenere Hälften



`mid=7`

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

Partition Methode

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

```
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

Quick-Sort

- die eigentliche Sort Methode ist rekursiv

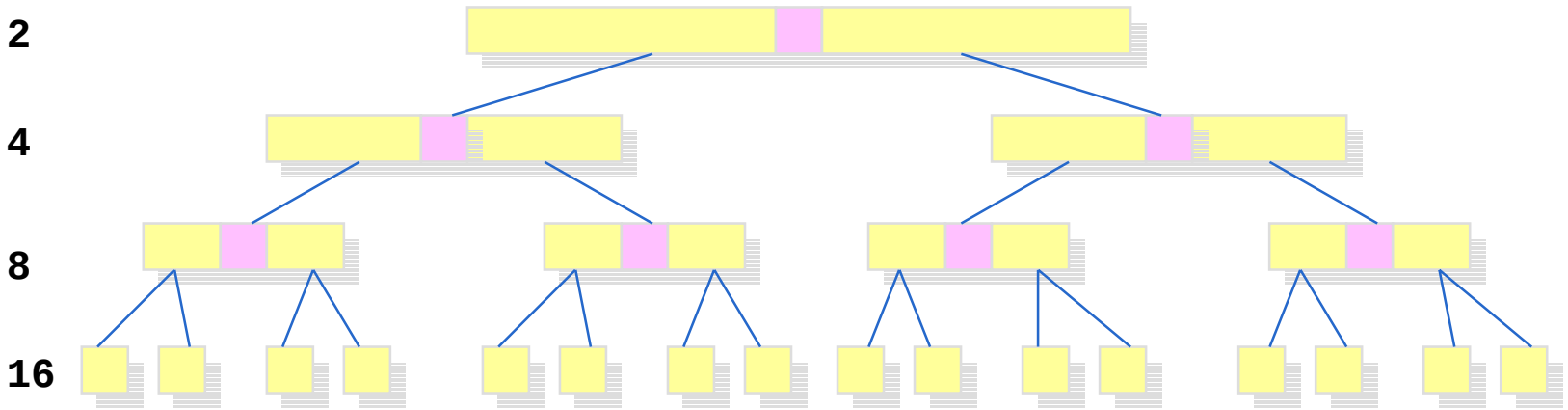
```
void quickSort(int[] a, int left, int right) {
    if (left < right) {
        int l = partition (a,left,right);
        quickSort(arr, left, l-1);
        quickSort(arr, l , right);
    }
}

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

Quick-Sort: Aufwand

- 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



Quick-Sort: Aufwand

- 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 $O(N^2)$.

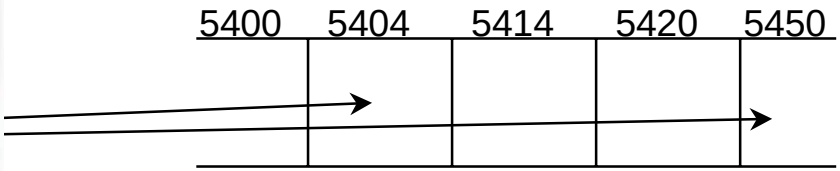
- Dies tritt jedoch nur in extra konstruierten Fällen auf
- Im Normalfall ist die Partitionierung bei QuickSort nahezu optimal

QuickSort immer die **erste Wahl**, wenn grösserer Mengen ungeordnete Daten sortiert werden müssen.

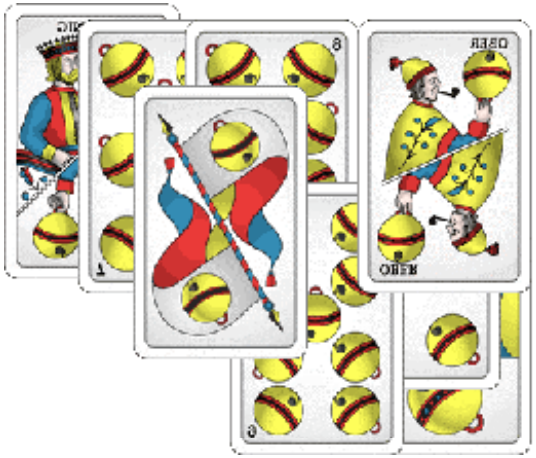
Distribution Sort

Distribution-Sort

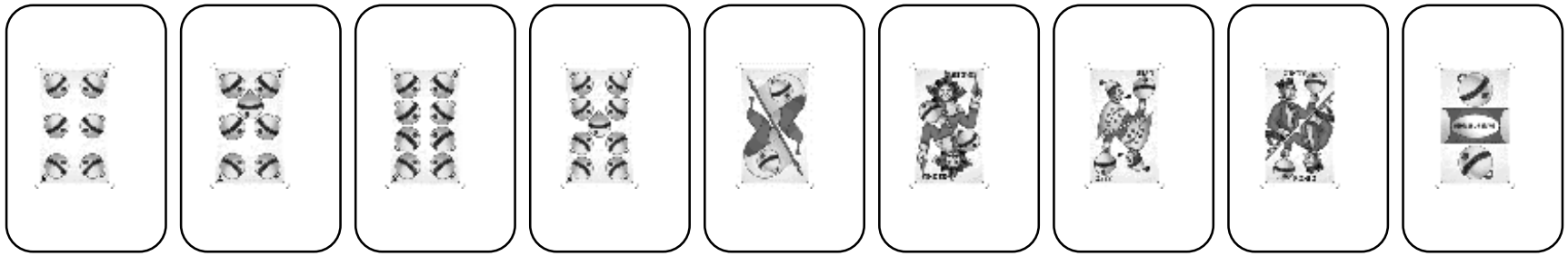
- Die bisher diskutierten Sortieralgorithmen basieren auf den Operationen: Vergleichen zweier Elemente und ev. Vertauschen zweier Elemente (swap).
- Im Gegensatz dazu kommt Distribution-Sort **ohne Vergleiche** aus.
- Die zu sortierenden Elemente werden
  - entsprechend dem Sortierschlüssel in Fächer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



Distribution-Sort Beispiel

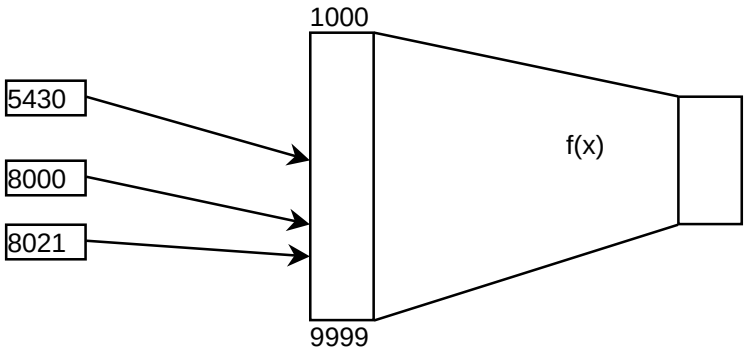


direktes Einfügen



Distribution-Sort

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



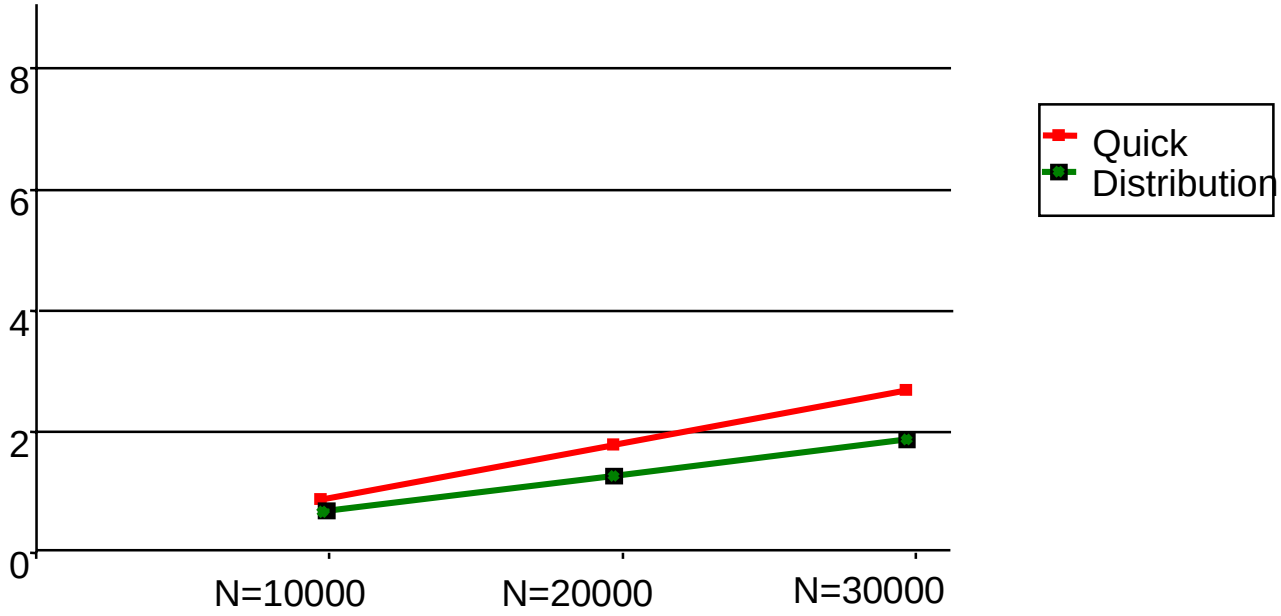
- Vorteile
  - schneller geht's nicht
  - linearer Algorithmus: die Komplexität ist also O(N)
- Nachteile
  - Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
  - Geht nur bei Schlüsseln, die einen kleinen Wertebereich haben oder auf einen solchen abgebildet werden können, ohne dass 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 Sortierverfahren.

Laufzeitvergleich schneller Sortieralgorithmen

Laufzeitvergleich schneller Sortieralgorithmen

Laufzeitmessung bei zufällig sortierten Daten (in Sekunden):

	N=10000	N=20000	N=30000	N=40000
Quick	1.57	3.38	5.21	7.19
Distribution	1.2	2.4	3.6	4.8



Vorsortierte Daten

- Bei vorsortierten Daten:

	N=10000	N=20000	N=30000	N=40000
Quick	0.48	1.02	1.43	2.16
Distribution	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

Übung

- Ein O(n*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.

Optimalitätssatz für Sortieralgorithmen

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 Sortierverfahen

Externen Sortierverfahren: 2 Phasen Sortieren

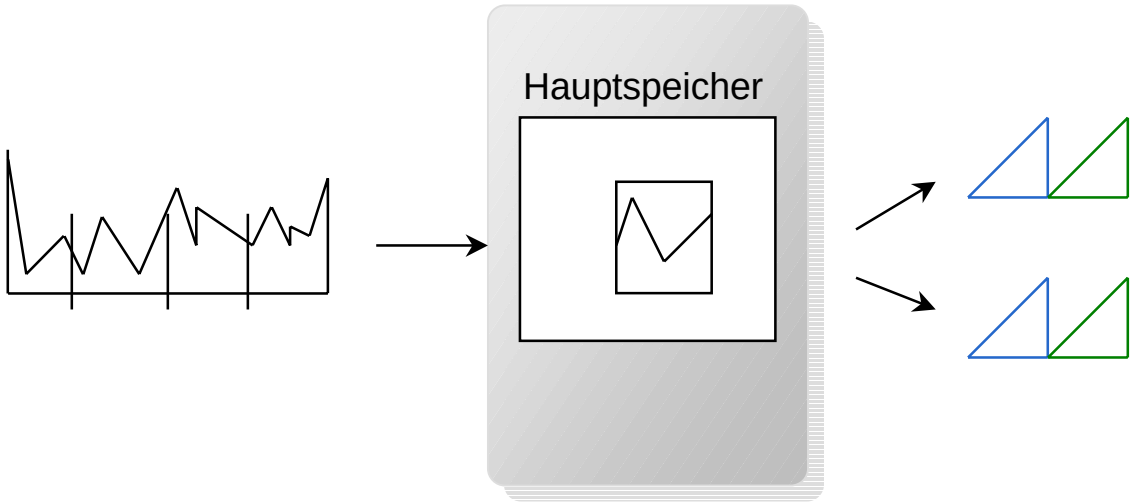
School of Engineering

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

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

1. Phase: Sortieren-Verteilen

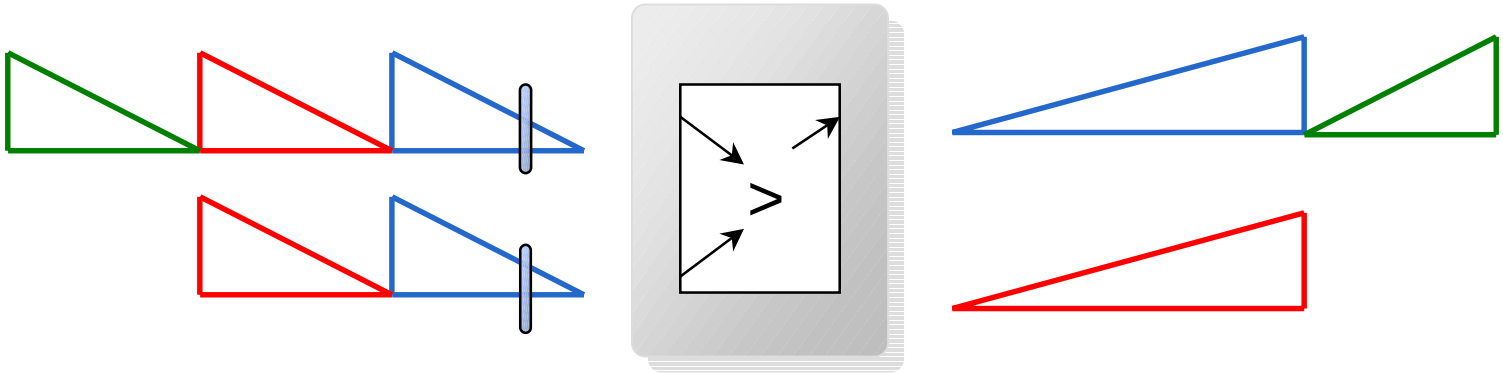
- lade jeweils einen Teil des Datei in den Speicher
  - sortiere diesen Teil mit schnellem internem Verfahren
  - schreibe sortiere Abschnitte in mehrere (mindestens zwei) Ausgabedateien
  - es entstehen Folgen von sortierten Abschnitten in den Ausgabedateien



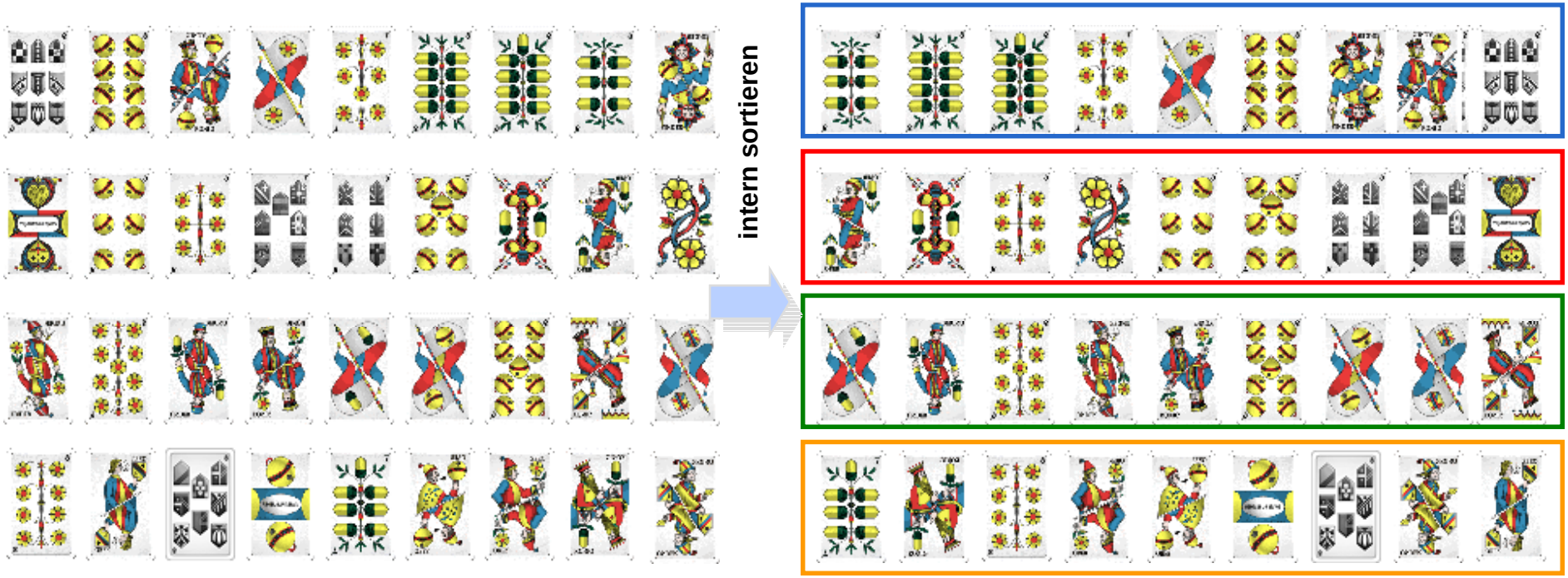
2. Phase: Mischen

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

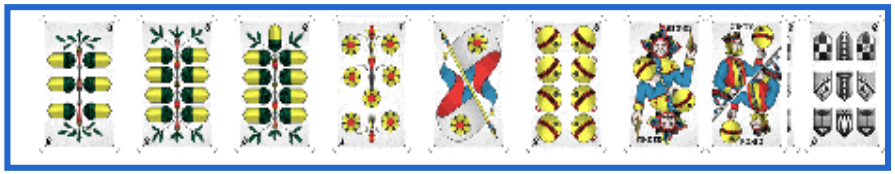
- -> Länge der geordneten Abschnitte hat sich verdoppelt
  - ouput -> input
  - solange wiederholen, bis vollständig sortiert



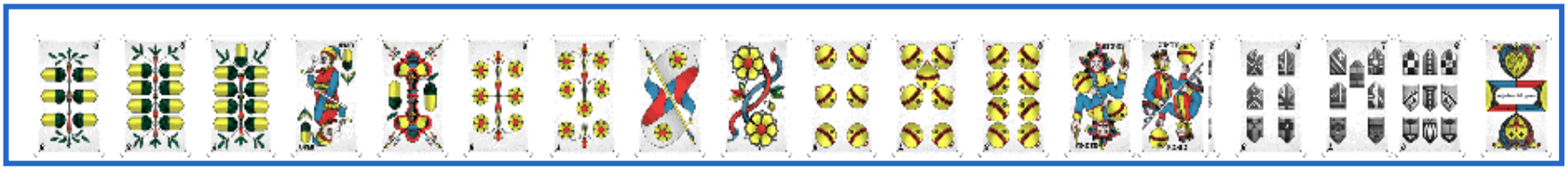
Beispiel Jasskarten



Misch Durchgang 1

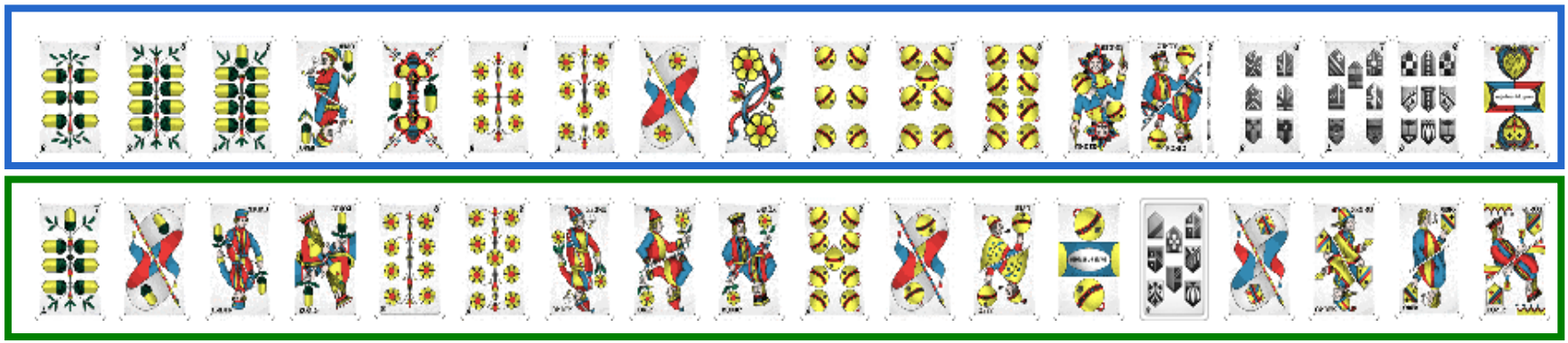


mischen

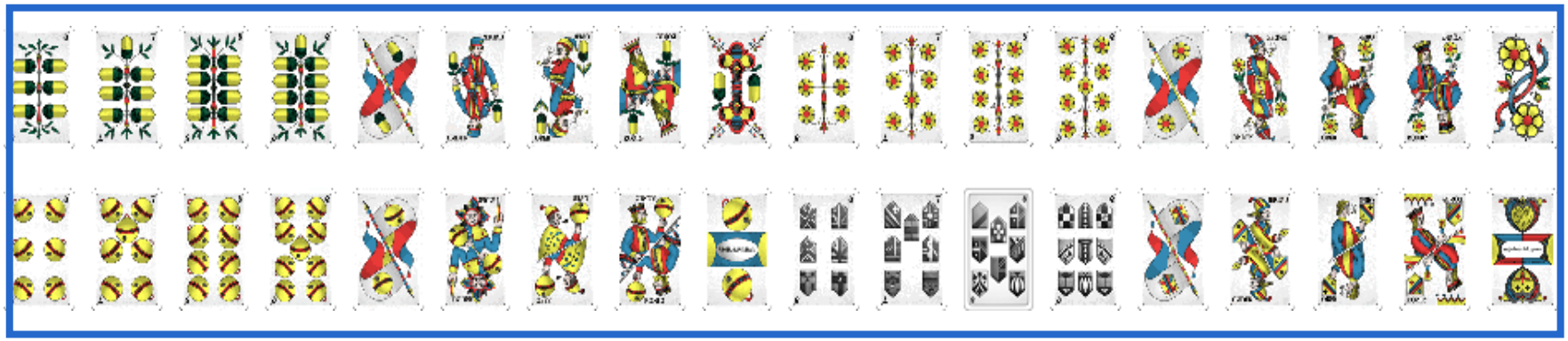


mischen

Misch Durchgang 2

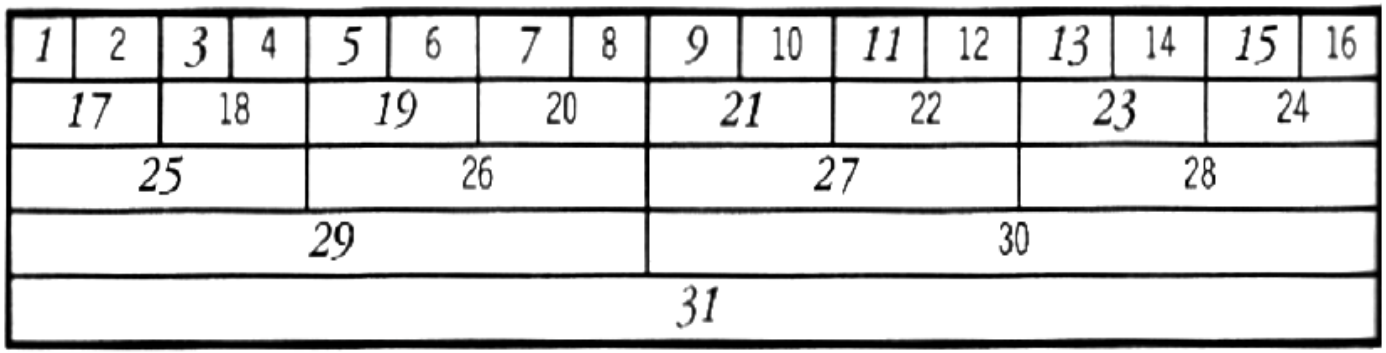


mischen



Laufzeit

- Annahme:
  - Zeit für internes Sortieren kann vernachlässigt werden
  - Beispiel 16 Sequenzen, 2 Eingabe und 2 Ausgabe Dateien



- Generell:
  - *bei n Sequenzen und m Eingabedateien*
  - *# Mischphasen* $\lfloor log_m\, n \rfloor$
  - *Aufwand von Mischen* $n * \lfloor log_m\, n \rfloor$

Wahl des Sortierverfahrens

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

Wahl des Sortierverfahrens

- 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.

Wahl des Sortierverfahrens

- 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 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

Optimierung durch Kombination von Algorithmen

Optimierungen von Quicksort

- 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);    }
}
```

z.B. 50

Einschub Zufallszahlen

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
    - *3.14159265358979323846264338327 95*

Einschub: Zufallszahlen

- 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: `Math.random()` für Bereich [0..1[
- für beliebigen anderen Bereich: r = (int)(k*Math.random())

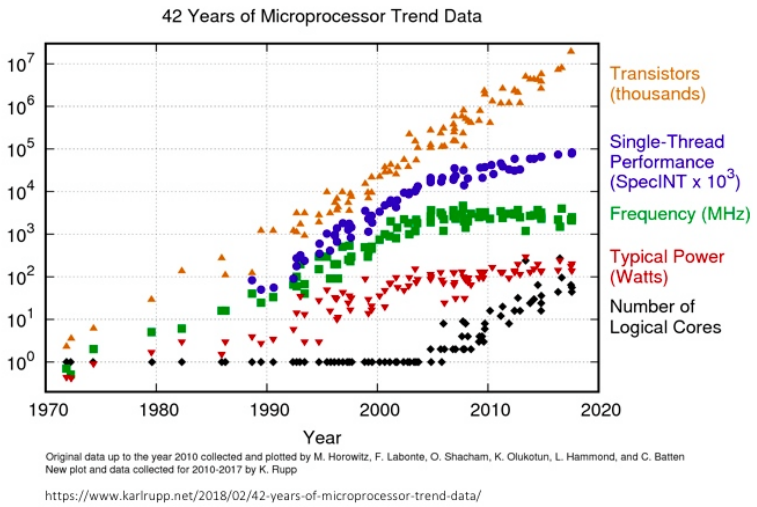
Einschub Zufallszahlen: Zufallszahlen mit Java

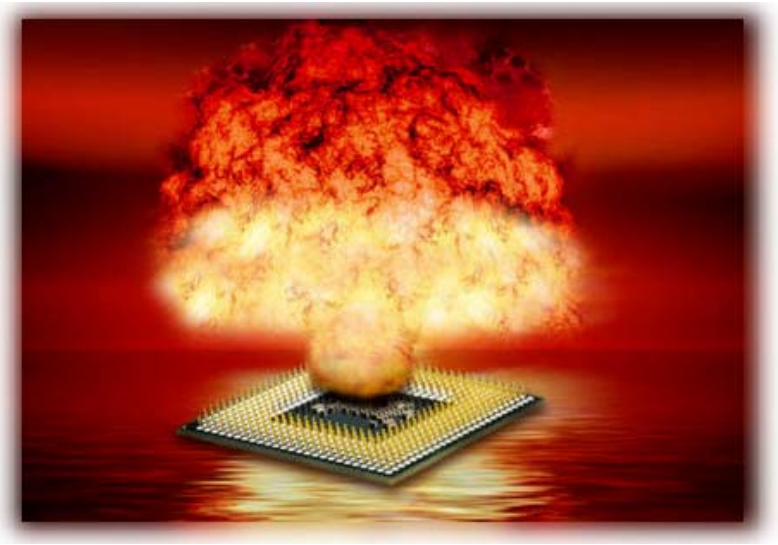
- Die Klasse `Random` erlaubt grössere Flexibilität.
- `Random(long seed)`
  Erzeugt einen Zufallszahlengenerator. *seed* ist der Startwert für die erzeugten Zufallszahlen. Damit kann mehrmals die gleiche Zufallssequenz erzeugt werden.
- `Random()`
  Erzeugt einen Zufallszahlengenerator. Der Startwert 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.

Optimierung durch Parallelisierung

Speed vs. Power Consumption

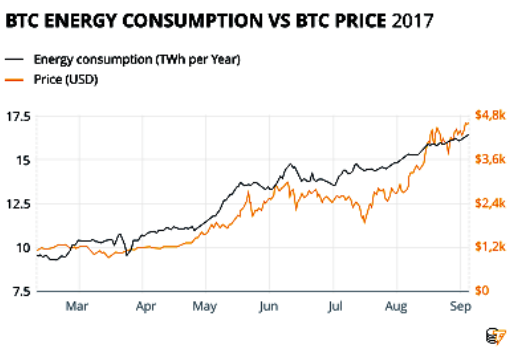
- Das grösste Problem heute ist die Wärmeabgabe der Chips
- Jeder CMOS Schaltvorgang braucht Energie, Verkleinerungen erhöht die Leckströme
- Die Energiedichte innerhalb einer CPU ist grösser als diejenige im Kern eines Kernreaktors: berechnete W'keit eines GAUs alle 250'000 Reaktorjahre
- Computer Industrie ist für 2% der weltweiten CO_2 Emissionen verantwortlich
- Rechenzentren: Stromkosten p.a. ~ HW Kosten

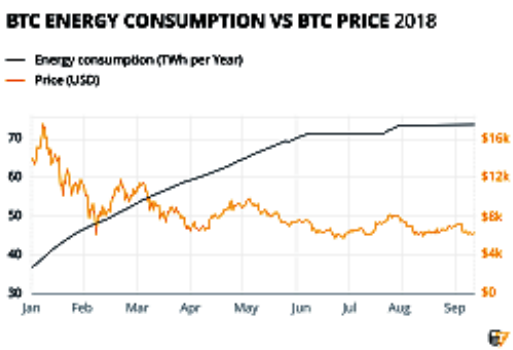




Bitcoin

- Bitcoin Miners verbrauchen mehr elektrische Energie als die Schweiz
  - Stromverbrauch der Schweiz ca. 50 TWh p.a.





- Braunkohle Kraftwerke Mongolei, Ulan Bator (Most Polluted Capital)

https://seekingalpha.com/article/4132836-winners-whiners-bitcoins-path-value

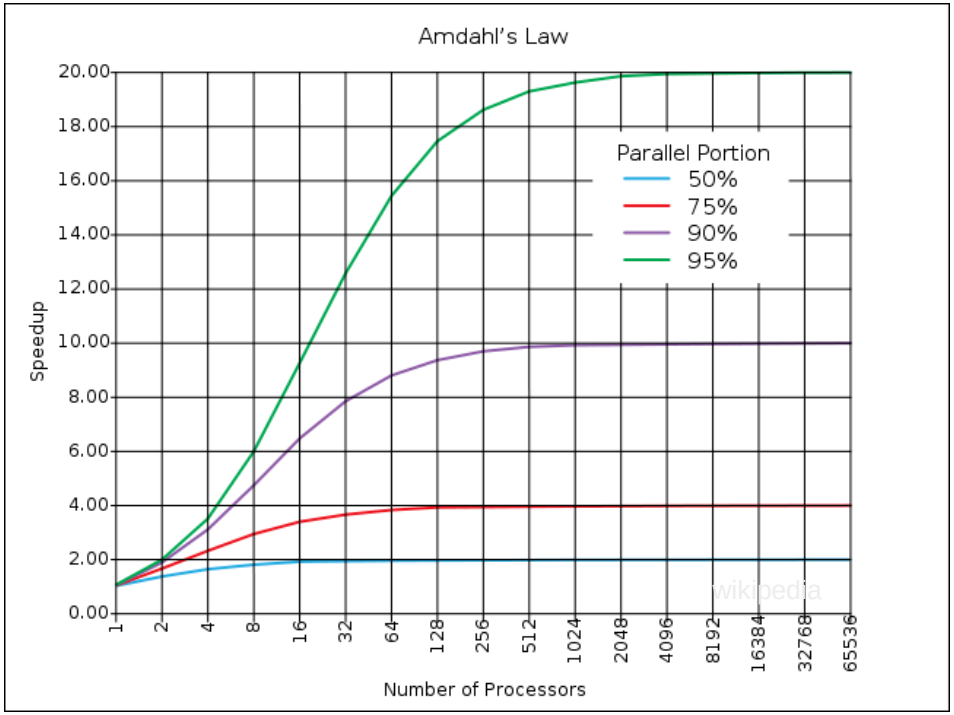
Beschleunigung durch Parallelisierung

- 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

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



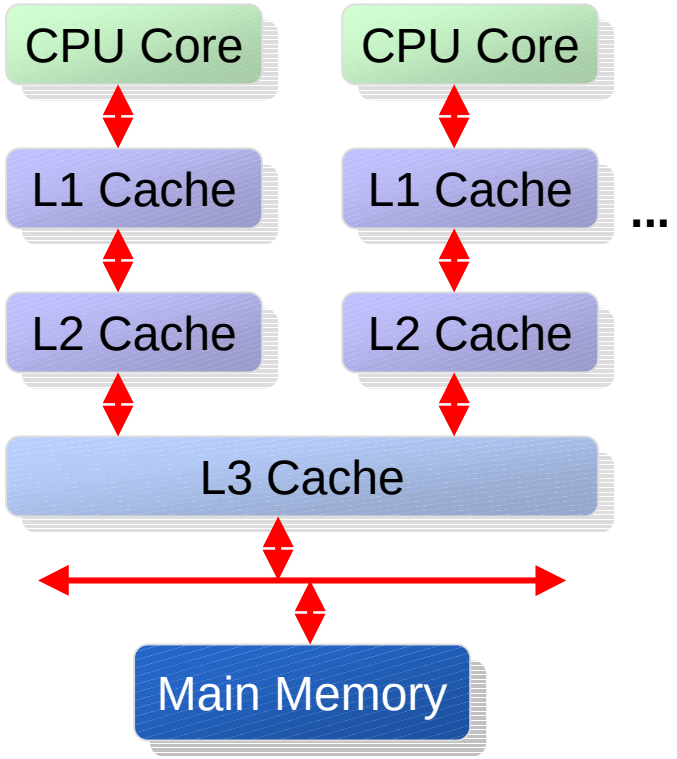
- Es kommt aber noch ein Overhead durch die Steuerung der Threads

Zugriff auf den Hauptspeicher

- Zugriffszeiten

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

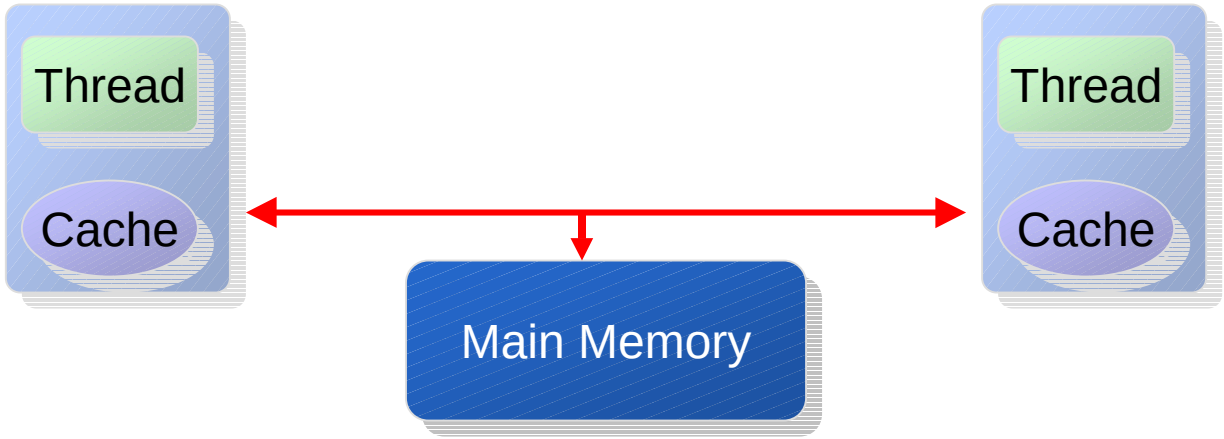
- Hauptspeicherzugriff ist im Vergleich zum Cache sehr teuer



© Angelika Langer

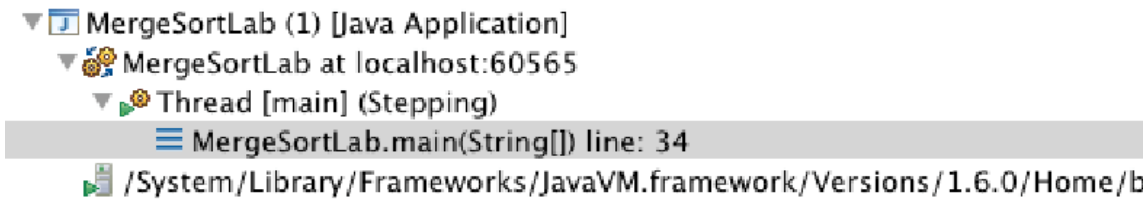
Java Memory Modell

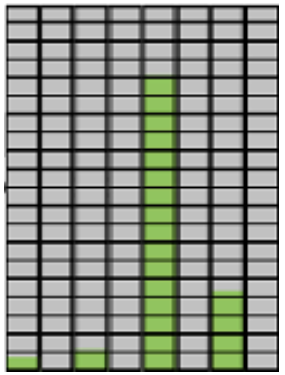
- JMM abstrahiert von physikalischem Memory Modell
- Alle Threads teilen Hauptspeicher
- Jeder Thread hat einen lokalen Arbeitsspeicher
- Veränderungen aus lokalem (Thread-) Arbeitsspeicher werden zurückgeschrieben
  - Bei expliziter Synchronisation
  - Thread Start und Beendigung
  - Lesen/Schreiben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



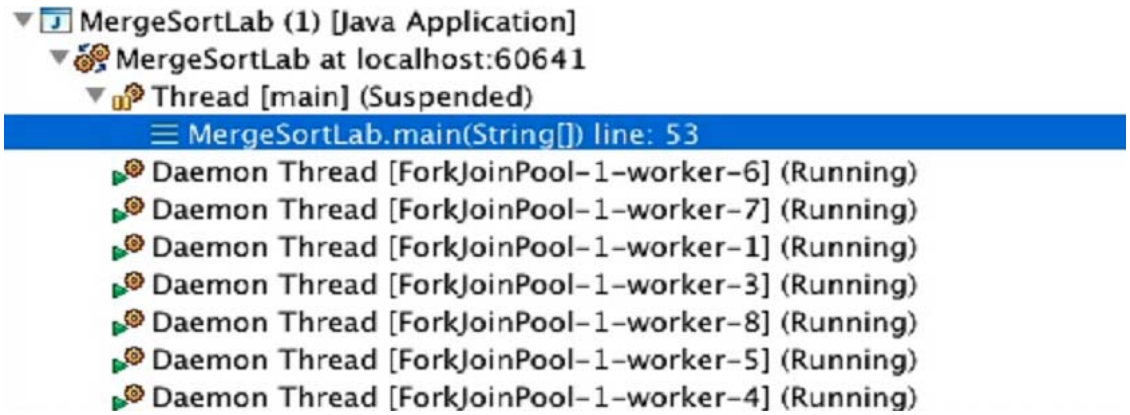
Prozessor Auslastung

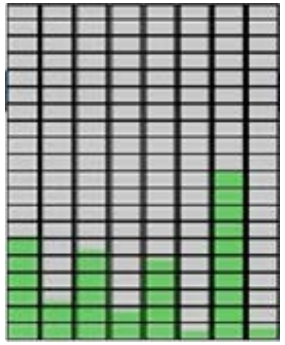
- Single Threaded





- Multithreaded





NaiveParallelQuickSort

Als separater Thread

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

Setzen der Parameter

Parallele Threads starten

Warte bis beendet

```
class NaiveParallelQuickSort extends Thread {
.....

     public NaiveParallelQuickSort(int[] a, int left, int right) {
        thia.a = a; ...

     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();
        }
     }
```

Ein neuer Thread pro Task

- Je ein neuer Thread für jede Partitionsaufgabe
- Java Threads werden auf Betriebssystems Threads abgebildet: "Kernel Level Threads"
- Vorteile
  - Die Rechenzeitzuteilung 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
  - Instanziierte 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

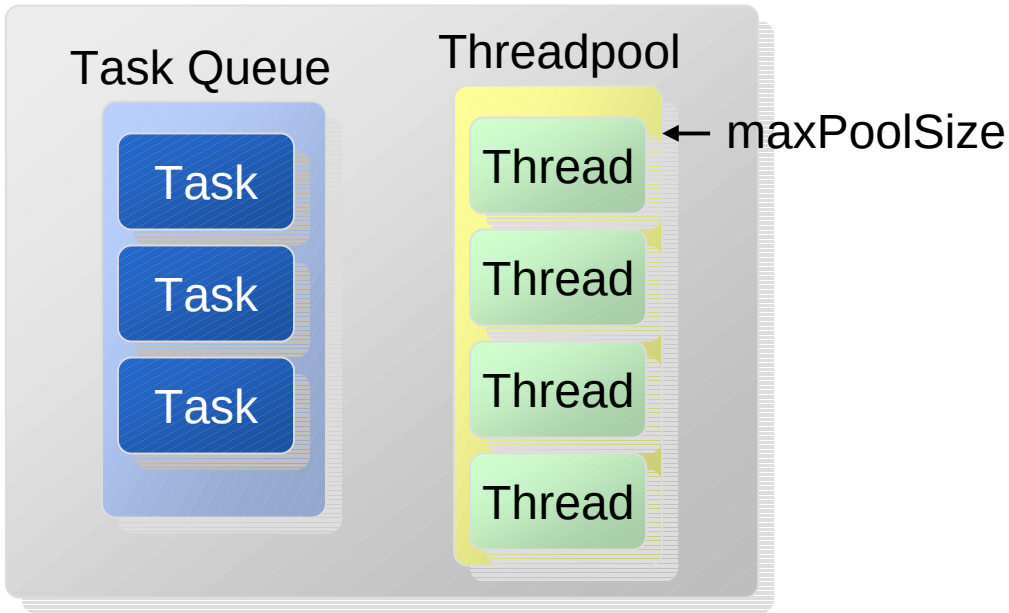
Threadpools

Idee der Thread Pools

- 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 mehr, da immer ein Teil wartend/blockiert ist.

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

- Die Reihenfolge der Abarbeitung der Tasks wird durch eine Ausführungs-strategie bestimmt



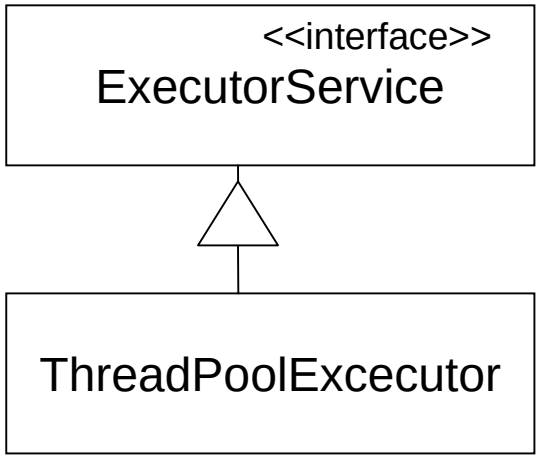
Ausführungsstrategien

School of Engineering

- 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 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

Java Implementation

- ExecutorService Interface für ThreadPools



- Java Executor (**`java.util.concurrency`**) stellt Framework zur Thread-Ausführung bereit, das verschiedene Ausführungsstrategien realisiert
  - verschiedene Varianten werden durch Factory-Methoden der Klasse **`Executors`** bereitgestellt
- FixedThreadPool
  - Pool mit fester Menge von Threads, die zur Ausführung eingereichter Tasks eingesetzt werden
- CachedThreadPool
  - Pool mit flexibler Menge von Threads, die sich an die Zahl der eingereichten Tasks anpasst
- SingleThreadExecutor
  - einzelner Thread, der eingereichte Tasks nacheinander (gemäß FIFO- oder LIFO-Prinzip oder nach Priorität) abarbeitet
- ScheduledThreadPool
  - Pool mit fester Menge von Threads, die zur zeitgesteuerten bzw. wiederkehrenden Ausführung eingereichter Tasks eingesetzt werden

Lebenszyklus des Threadpools

School of Engineering

- Erzeugung via Fabrikmuster Methoden

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

- Drei Zustände:
  - **running**
    - Threadpool nimmt Tasks entgegen und führt sie aus, sobald Threads verfügbar sind
  - **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
  - **terminated**
    - keine Tasks werden mehr ausgeführt oder angenommen

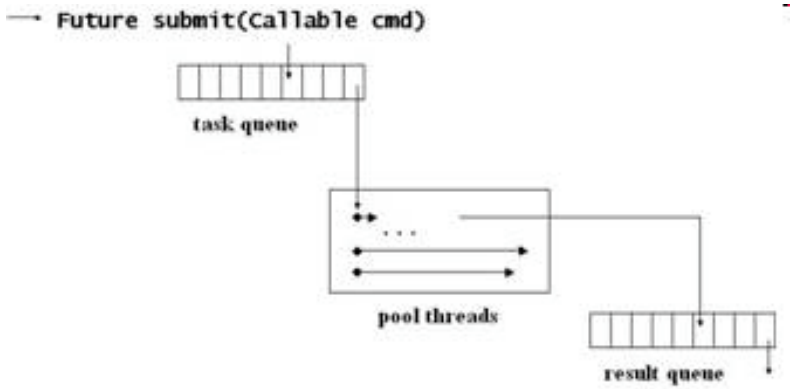
Einstellen eines Threads

- 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

Resultat der Ausführung

- **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

Tasks mit Rückgabewerten

- Realisierung der Anwendungslogik mit Rückgabetyp **`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;
    }
}
```

Thread Safe Collections

School of Engineering

- im Paket `java.util.concurrent` gibt es einige Thread Safe Collections
- Im Fall von erwarteter hoher Parallelität sollten diese genommen 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

Terminiere Threadpool

füge neuen Task in Threadpool

Fork Join

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

- Erzeuge ein ForkJoinPool “thread-manager” Object
- Definiere ein Task Object das von RecursiveTask erbt
  - Kann generisch sein, wenn Rückgabewerte vorhanden sind.
  - Instanziere den FJ Pool
  - Instanziere (Haupt-)Task Objekt und rufe `fjpool.invoke(task)` auf

- In der compute Methode
  - 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	Fork/Join
Do subclass **Thread**	Do subclass **RecursiveTask<V>**
Do override **run**	Do override **compute**
Do call **start**	Do call **invoke, invokeAll, fork**
Do call **join**	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);
     }


     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 Task FJ Queue und warte auf Terminierung

übergebe FJ Queue und warte auf Terminierung

Beispiel QuickSortForkJoin - fork/invoke/join

School of Engineering

```
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

Lösung Funktional

School of Engineering

- 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)
```

Zusammenfassung

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