Klassen und Objekte, Dynamische Speicherallokation, Überladen, Eine Klasse für rationale Zahlen, Datenkapselung, Klassen, Verstecken der Daten, Fallstudie Online Statistik, Komplexität

3. KLASSEN

Klassen und Objekte

- An die Stelle von RECORDs in Pascal treten bei Java die *Klassen*
- Hauptunterschiede

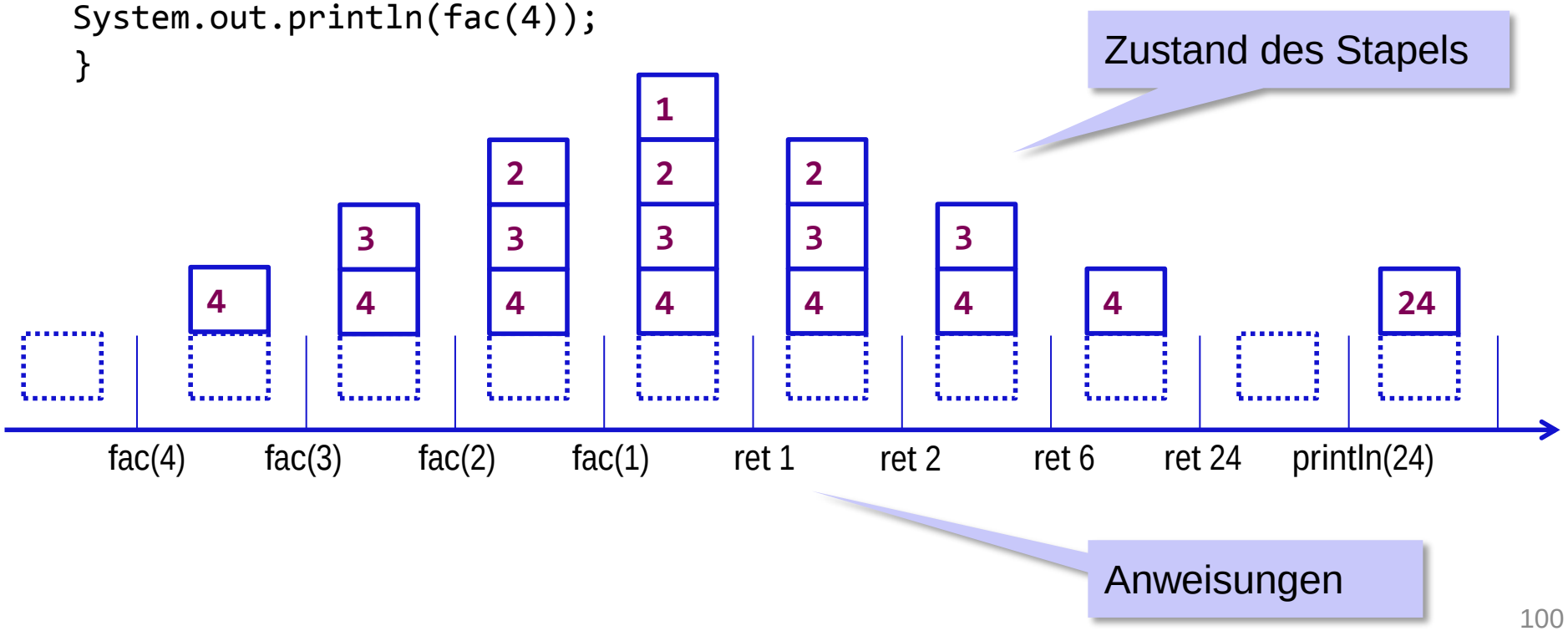
Pascal	Java
RECORDs in Pascal sind reine Datenobjekte. Auf ihnen wird mit Prozeduren operiert.	Klassen in Java beherbergen Daten *und* Code. Sie stellen einen Teil des Codes bereit, mit dem auf ihnen operiert wird.
RECORDs sind wertesemantische Typen. Instanzen werden "in place" alloziert.	Klassen in Java haben Referenzsemantik. Instanzen müssen mit "new" alloziert werden.

Speicherallokation: Der Stack

- Sie kennen bereits einen Teil der dynamischen Speicherallokation: den Stack (Aufrufstapel).
  - Beim Aufruf von Prozeduren / Funktionen / Methoden wird automatisch Speicher für lokale Variablen und Parameter angelegt, welcher beim Rücksprung wieder freigegeben wird.
  - Die Struktur dieses Speichers ist wegen der Unmöglichkeit des *gleichzeitigen* Aufrufs mehrerer Prozeduren besonders einfach: ein Stapel
  - Um Verwaltung und Aufbau des Aufrufstapels müssen wir uns glücklicherweise nicht kümmern.

Der Stack: Beispiel

```
static int fac(int n)
{   if (n>1)
        return n*fac(n-1);
    else
        return 1;
}
public static void main(String[] arg)
{
System.out.println(fac(4));
}
```



Dynamische Speicherallokation

- Für die Implementation von dynamischen Datenstrukturen (später!) im allgemeinen und
- für die Nutzung von Klassen in Java im speziellen

benötigt man *dynamischen Speicher,* also Speicher den man *explizit* anfordern muss.

Bei Java wird der Speicher, sobald nicht mehr verwendet, von einem *Garbage Collector* abgeräumt.

- Bei C++ ist das zum Beispiel im Allgemeinen nicht so, d.h. man muss den Speicher "manuell" abräumen. (Dort gibt es aber auch andere Speicherallokationsmodelle.)

Allokation mit **new**

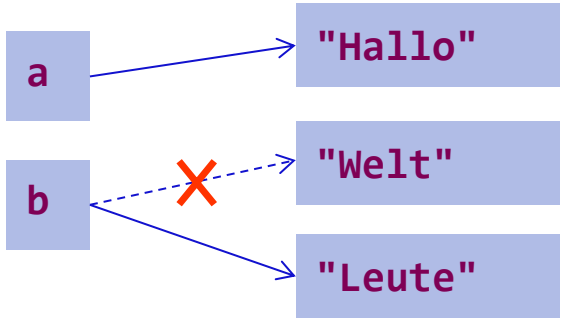
new *T* (par_0, ..., par_n)

Ausdruck vom Typ *T*

- Semantik: neuer Speicher für ein Objekt vom Typ *T* wird angelegt
- Wert des Ausdrucks ist die Adresse des Objekts
- (Optionale) Parameter werden dem *Konstruktor* des Typs weitergegeben

Dynamische Speicherallokation

- Mit new erzeugte Objekte haben dynamische Speicherdauer. Sie leben, bis sie nicht mehr erreichbar sind.




```
String a = new String("Hallo");
String b = new String ("Welt");
System.out.println(a + " " + b); // "Hallo Welt"
b = new String("Leute"); // alter String "Welt" nicht mehr erreichbar
System.out.println(a + " " + b); // "Hallo Leute"
```

- Zum Glück müssen wir uns um freigegebene Objekte nicht kümmern. Der Garbage Collector räumt "hinter uns" auf.

Überladen (Overloading)

- Methoden sind durch ihren Namen im Gültigkeitsbereich ansprechbar
- Es ist sogar möglich, mehrere Methoden des gleichen Namens zu definieren
- Die richtige Version wird aufgrund der *Signatur* der Funktion ausgewählt

Überladen (Overloading)

Die *Signatur* einer Methode ist bestimmt durch Art, Anzahl und Reihenfolge der Argumente

```
static double sq(double x) { ... }                  // fkt 1
static int sq(int x) { ... }                        // fkt 2
static double log(double x, double b) { ... }       // fkt 3
static double log(double x) { return log(x,2); }       // fkt 4
```

Der Compiler wählt beim Funktionsaufruf automatisch die Funktion, welche "am besten passt" (wir vertiefen das nicht)

```
System.out.println(sq(3.3));                        // fkt 1
System.out.println(sq(3));                          // fkt 2
System.out.println(log(4,10));                      // fkt 3
System.out.println(log(3));                         // fkt 4
```

Überladen (Overloading)

- Mit dem Überladen von Funktionen lassen sich also verschiedene Aufrufvarianten des gleichen Algorithmus realisieren und / oder
- verschiedene Algorithmen für verschiedene Datentypen mit dem gleichen Namen verbinden.
- Funktioniert ein Algorithmus für verschiedene Datentypen identisch, so verwendet man in Java *Generics.*

Überladen wird auch verwendet bei der Auswahl des geeigneten *Konstruktors* beim Aufruf von `new`

Operator Overloading?

- Das Überladen von *Operatoren* durch den Programmierer ist in Java nicht vorgesehen.
- Aber die eingebauten Operatoren sind natürlich überladen.
  - Das "+" Symbol, z.B., operiert auf Fliesskommazahlen anders als auf Ganzzahlen
  - und wieder anders bei Strings:

```
int i=3;
System.out.println("i= " + i);      // "i= 3"
System.out.println("s= " + i + i);  // ?
System.out.println("S= " + (i + i));// "S= 6"
```

Bei gleicher *Präzedenz* sind Operatoren *linksassoziativ*

Eine Klasse für rationale Zahlen

```
class Rational {
    int numerator, denominator;

    public Rational(int num, int denom){
      assert denominator != 0;
      numerator= num;
      denominator = denom;
    }

    public Rational(){
      numerator= 0;
      denominator = 1;
    }

    ...
}
...
Rational x = new Rational(2,3); // Initialisierung mit 2/3
Rational y = new Rational();    // Initialisierung mit 0
Rational z = new Rational(1);   // Fehler: (noch) nicht definiert
```

Sicherung von Invarianten!

Konstruktoren:

spezielle Methoden mit dem Namen der Klasse

und ohne Rückgabewert

Konstruktoren

- sind spezielle Methoden, die den Namen der Klasse tragen
- können überladen werden, also in der Klasse mehrfach, aber mit verschiedener *Signatur* vorkommen
- werden beim Aufruf von `new` wie eine Funktion aufgerufen. Der Compiler sucht die naheliegendste passende Funktion aus.
- wird kein passender Konstruktor gefunden, so gibt der Compiler eine Fehlermeldung aus.

Prozeduraler Ansatz?

Warum macht man es in Java nicht wie folgt?

```
public class Rational {                          // Eine Klasse mit
public int numerator, denominator;               // öffentlichen Daten

    // ... Konstruktoren etc.
}

public class RationalTest {

  static Rational Add(Rational x,y) { ... }      // Funktionen, die
  static void Print(Rational x) { ... }          // auf Rational operieren

  public static void main(String[] arg)
  {
    Rational x= new Rational(1,2); // 1/2
    Rational y= new Rational(2,3); // 2/3
    Print(Add(x,y));
  }
}
```

Datenkapselung: Motivation

Gedankenexperiment

- Erstellung des Datentyps Rational so wie oben, mit öffentlich zugänglicher Strukturierung der Daten
- Entwicklung und Verkauf einer Bibliothek an einen Kunden: *Rationales Denken AG* (RAT)
- RAT entwickelt Anwendung unter Benutzung der Bibliothek

Datenkapselung: Motivation, Problem 1

Invarianten beim Initialisieren mit Konstruktoren sichergestellt.

Aber: Insgesamt sind Invarianten noch nicht garantiert. Ein Programmierer bei RAT könnte z.B. schreiben:

```
Rational r = new Rational();  // Konstruktor stellt Invariante sicher, ok
r.numerator = 1;
r.denominator = 0;                    // Verletzung der Integrität von r
```

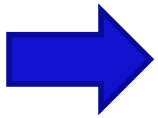
Selbst wenn der Programmierer bei RAT nicht so doof ist, kann durch felhlerhafte Berechnungen anderswo ein Nenner von 0 aus Versehen zustandekomen

Motivation, Problem 2

Interne Repräsentation ist nicht änderbar.

- RAT fragt an: können wir rationale Zahlen mit erweitertem Wertebereich haben?
- Klar, kein Problem, z.B.

```
public class Rational {
  public int numerator;
  public int denominator;
  ...
}
```



```
public class Rational {
  public long numerator;
  public long denominator;
  public long integer;
  ...
}
```

- Anpassen der Bibliothek und Lieferung der neuen Version an RAT

Motivation, Problem 2

RAT ruft an: *nichts geht mehr!*

Problem:

- Anwendungscode enthält jede Menge `.numerator's` und `.denominator's`
- ...aber die bedeuten jetzt etwas anderes (Werte von Zähler und Nenner vom Rest auf die nächste Ganzzahl)
- RAT's Anwendung kompiliert zwar noch, berechnet aber Unsinn

Motivation, Problem 2

Wenn man dem Kunden erlaubt, auf die interne Repräsentation zuzugreifen, kann man sie in Zukunft *nur mit Zustimmung* des Kunden ändern.

Die bekommt man nicht, wenn der Kunde (oder Arbeitskollege) dafür seinen Anwendungscode umschreiben muss.

Idee der Datenkapselung

- Eine komplexe Funktionalität wird auf einer möglichst hohen Abstraktionsebene semantisch definiert und durch ein vereinbartes minimales *Interface* zugänglich gemacht
- *Wie* der Zustand durch die Datenfelder einer Klasse repräsentiert werden, sollte für den Kunden nicht sichtbar sein
- Dem Kunden wird eine repräsentations-unabhängige Funktionalität angeboten

Klassen

- beherbergen das Konzept zur Datenkapselung
- sind Bestandteil jeder "objektorientierten Programmiersprache"
- Ersetzen in Java auch die Konzepte der Units oder Module von anderen Sprachen
- Können in Paketen als gemeinsamen Namensraum zusammengefasst werden.

Verstecken der Daten (und Methoden)

```
class Rational {
    int nominator;
    int denominator;
    ...
}
```

So sind die Daten nach aussen nicht mehr sichtbar

Sichtbarkeiten nach Modifizierer

Modifizierer	Klasse	Paket	Sub-Klasse	Global
public	✓	✓	✓	✓
protected	✓	✓	✓	
(keiner)	✓	✓		
private	✓			

Aber wie gewährleistet man nun den Zugriff zu den Daten?

Methoden (Member-Funktionen)

erlauben den Zugriff auf versteckte Daten

```
class Rational {
     int numerator, denominator;

     public Rational(int num, int denom){ ... }
     public Rational(){ ... }

     public int Numerator(){
          return numerator;
     }

     public int Denominator(){
     return denominator;
     }

     ...
}

int n = r.Numerator();
int d = r.Denominator();
```

Methoden haben immer Zugriff auf die Klassenvariblen

Üblich: Zugriff über sogenannte *"Getter" und "Setter" Methoden*

Zugriff auf eine Methode (Member-Funktion) von r

Der implizite Parameter von Methoden

```
public int Numerator(){
  return numerator;
}
..
```

- Eine Methode wird für einen Ausdruck mit Typ der Klasse aufgerufen, z.B. `r.Numerator()` Dieser Ausdruck (hier: `r`) ist implizites Argument des Aufrufs und wird in der aufgerufenen Methode mit **`this`** bezeichnet.

- Im Rumpf einer Methode bezieht sich der Name einer Datenvariable auf das entsprechende Feld des impliziten Arguments **`this`**

Der implizite Parameter von Methoden

Auf das implizite Argument `this` kann auch explizit zugegriffen werden.

```
class Rational {
    int numerator, denominator;
    ...
    public Rational Mul(Rational x){
        int numerator = x.numerator * this.numerator;
        int denominator = x.denominator * this.denominator;
        return new Rational (numerator, denominator);
    }

}
```

Bezug auf die Klassenvariable

Bezug auf die lokale Variable dieser Methode

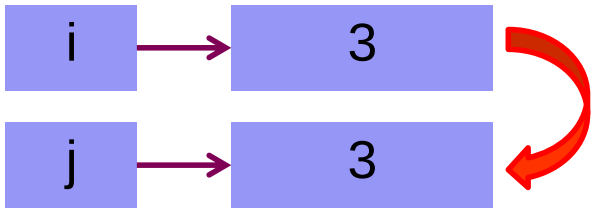
Lokale Variable verdeckt die Klassenvariable mit gleichem Namen

Das ist keine gute Idee und führt schnell zu Programmierfehlern, es dient hier nur der Illustration

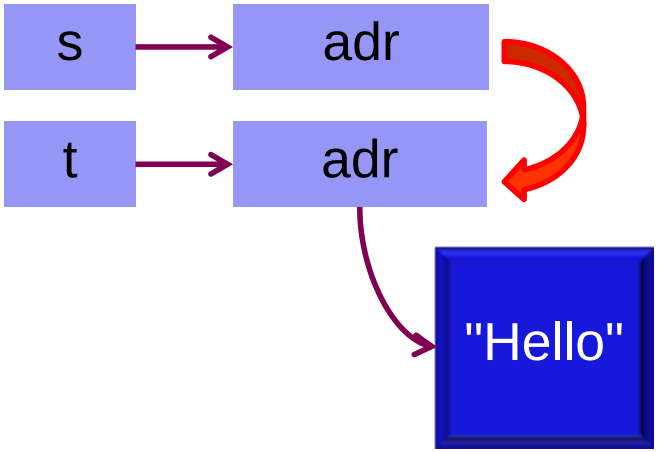
Werte und Referenzen

- Die eingebauten Datentypen von Java haben *Wertesemantik*, d.h. Zuweisung und Parameterübergabe kopieren den *Wert (Inhalt)*
- Klassen und Arrays haben *Referenzsemantik*, d.h. Zuweisungen und Parameterübergabe kopieren die *Referenz* (den Zeiger) auf das Objekt.
  - Anders gesagt: Klassen- und Arrayvariablen sind eigentlich Zeiger.
  - Es gibt kein Pass-By-Reference, Zeiger werden Pass-By-Value übergeben und stellen ihrerseits eine Referenz bereit.

```
int i = 3;
int j = i;
```




```
String s=new String("Hello");
String t=s;
```



Fallstudie: Online-Statistik

Ziel: Klasse / Objekt, welches Daten konsumiert und zu jeder Zeit Statistiken, z.B. Mittelwert, Varianz, Median (etc.) ausgeben kann

```
Statistics s = new Statistics(maxSize);

...
  // Beobachten von Daten
  s.Put(data);

...
System.out.println(s.Mean());
System.out.println(s.Variance());
System.out.println(s.Median());
```

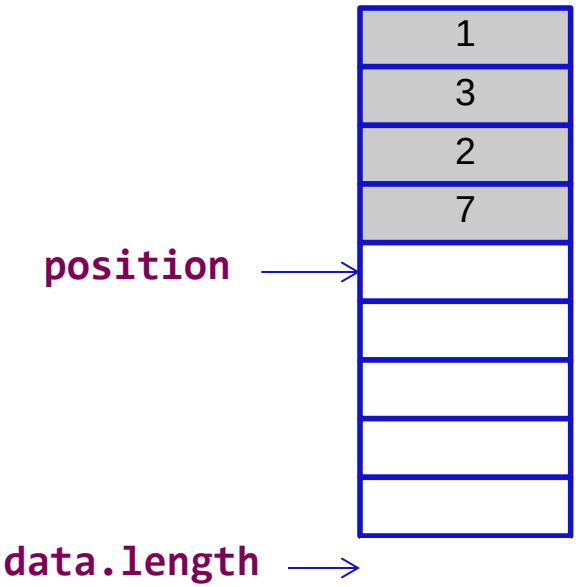
Erstellen der Klasse

```
public class Statistics {
  double data[];       // Zirkulärer Puffer
  int numberElements; // Füllgrad
  int position;        // momentane Position

  // Konstruktor
  Statistics(int maxSize) {
    numberElements = 0;
    data = new double[maxSize];
  }

  // füge Wert in die den zirkulären Puffer ein
  public void Put(double value){
    data[position] = value;
    if (numberElements < data.length)
      numberElements++;
    position = (position + 1) % data.length;
  }

...
```



Erstellen der Klasse

```
  public double Mean() {
    double sum = 0;
    for (int i=0; i<numberElements;++i)
      sum += data[i];
    return sum / numberElements;
  }

  public double Variance() {
    if (numberElements > 1)
    {
      double ssq = 0;
      double mean = Mean();
      for (int i=0; i<numberElements; ++i)
        ssq += (data[i]-mean)*(data[i]-mean);
       return ssq / (numberElements-1);
     }
     else
        return 0;
  }
}
```

numberElements Schritte

2*numberElements Schritte

Fragen

- Können wir die Beschränkung der Pufferlänge mit akzeptablem Overhead aufheben?
  - Antwort: ja, Übungsaufgabe
- Ist der Algorithmus oben für sehr grosse Datensätze im Sinne eines online-Algorithmus geeignet?
  - Antwort: nein, Online-Algorithmen halten nützliche Zwischenergebnisse und berechnen nicht jedes mal neu.

Effizienz

- Obige Implementation ist nicht sehr effizient.
- Wenn `Mean` oder `Variance` oft, z.B. nach jedem Eintrag eines Wertes, abgefragt wird, dann lohnt sich der *Provisional Means* Algorithmus
  - Wenn $m_n = \frac{1}{n}\sum_{i=1}^{n} x_i$, dann gilt

$$m_{n+1} = m_n + \frac{x_{n+1} - m_n}{n+1}$$

  - Für $s_n = \sum_{i=1}^{n}(x_i - m_n)^2$ gilt analog:

$$s_{n+1} = s_n + (s_n - m_n) \cdot (x_{n+1} - m_{n+1})$$

- Damit fällt sogar die Notwendigkeit zum Speichern des Arrays weg

Provisional Means

```
public class Statistics {
  int n = 0;
  double mean = 0;
  double ssq = 0;

  // Einfügen und Update
  public void Put(double value){
      n++;
     double oldMean = mean;
      mean = oldMean + (value - oldMean) / n;
      ssq = ssq + (ssq - oldMean) * (value - mean);
  }

  public double Mean(){
    return mean;
  }

  public double Variance() {
    if (n>1)
         return ssq / (n-1);
     else
         return 0;
  }
}
```

Fragen

- Können wir das auch mit dem Median machen?
  - Antwort: nein, deutlich komplizierter
- Was ist dann ein guter (On-line) Algorithmus für die Berechnung des Medians?

Wenn $s(x,k)$ denn k-grössten Wert des Datensatzes x mit Länge n bezeichnet $(1 \leq k \leq n)$, so ist der Median definiert als

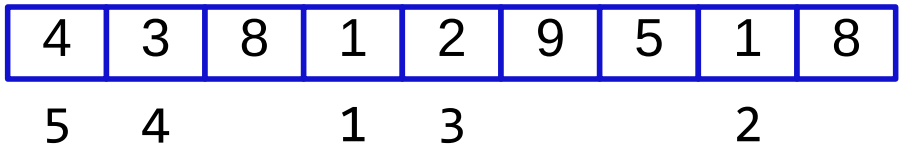
$s\left(x, \frac{n+1}{2}\right)$, falls n ungerade und

$\frac{1}{2} \cdot \left(s\left(x, \frac{n}{2}\right) + s\left(x, \frac{n+1}{2}\right)\right)$, falls n gerade

Median

Bestimmung des Medians ist ein *Auswahlproblem*

- Einfach: Bestimmung des grössten / kleinsten Elements in n Schritten durch Traversieren aller Elemente.



- Iterative Anwendung dieses Verfahrens unter Streichung des vorherigen Minimums liefert Median in $\frac{n+1}{2} \cdot n$ Schritten (wenn n ungerade)
- Bessere Ideen?

Median

Wir wissen schon aus Informatik I, dass *Mergesort* $n \log n$ Schritte benötigt.

Schneller als naive Methode:

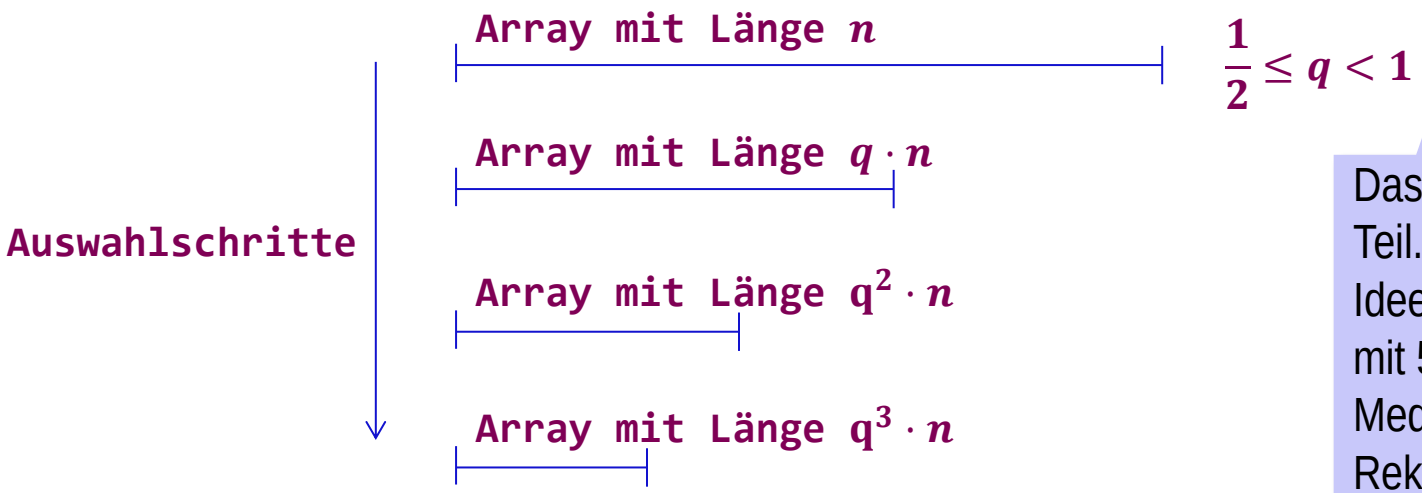
- Sortieren des Arrays (Mergesort) und nachfolgend
- Auswahl des mittleren Elements (Array-Zugriff)

- Nachteil
  - Verändert die Daten oder benötigt Kopie der Daten
- Vorteil
  - Ist generisch: direkt für Auswahl des k-kleinsten Elements verwendbar

Median*

Geht es *noch* schneller als $n \log n$?

- Ja: *Median der Mediane*. Algorithmus mit n Schritten, relativ kompliziert und eher von theoretischem Interesse.
- Idee: Verfahren ähnlich zu Quicksort
  - Es muss jedoch jeweils nur für eine Hälfte weitergemacht werden, sortiert wird letztlich nicht wirklich
  - Und es wird garantiert, dass für den Pivot "ein genügend guter" Fall eintritt



Das ist der komplizierte Teil.
Idee: Aufteilung in Töpfe mit 5 Elementen, jeweils Median im Topf nehmen.
Rekursiv: Median vom Median

On-line Median?

- Beobachtung: für zukünftige Updates des Medians sind immer alle Daten relevant.
  - Es gibt keine Abkürzung wie beim Mittelwert.
  - "Beweis": jeder Datenpunkt kann irgendwann zum Median werden.
- Idee
  - Daten werden grundsätzlich sortiert in das Array eingefügt. Geht das effizient?
  - Wir kommen darauf im nächsten Kapitel zurück