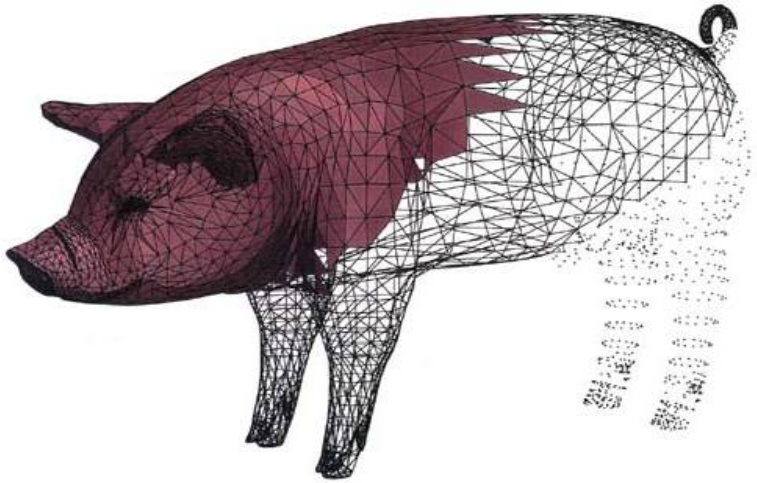
Komplexität

Aufwand, Zeitbedarf, O-Notation, Komplexität rekursiver Algorithmen, Komplexitätsklassen, Sortieren ist n*log(n), DistributionSort, linear – aber …

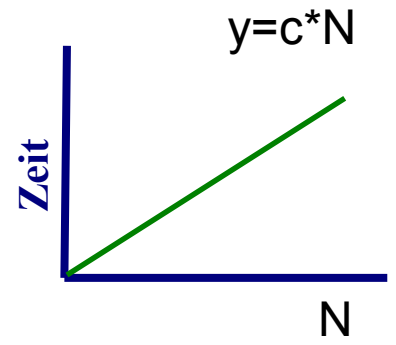
Welcher Algorithmus ist besser ?

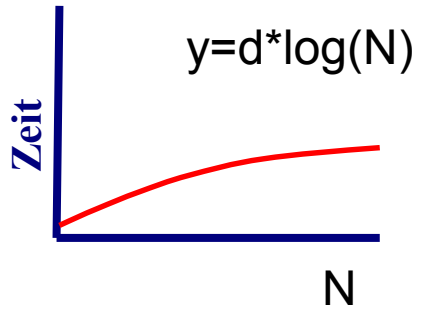
- Kriterien für Vergleich von Algorithmen:
  - Effizienz/Aufwand
    - Schnelligkeit
    - Ressourcenverbrauch
  - Einfachheit
    - Einfach zu implementieren
    - Fehleranfälligkeit
  - Allgemeinheit
    - vielseitig verwendbar
    - für viele Datentypen
      - z.B. nicht nur für int, sondern alle geordneten Typen

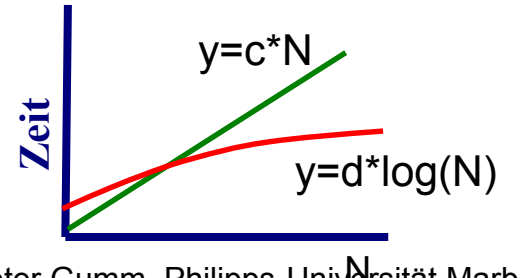


Aufwand

- Offensichtlich ist lineare Suche aufwändiger als binäre Suche
  - insbesondere bei großen Datenmengen
  - Umgekehrt ausgedrückt: binäre Suche ist effizienter
- Wie kann man Effizienz (bzw. Aufwand)
  - definieren
  - messen
- Aufwand hängt von der Größe N der Datenmenge ab
  - Aufwand = f(N)
  - Lineare Suche : f(N) = a+c*N (linear)
  - Binäre Suche : f(N) = c+d*$\log_2$(N) (logarithmisch)
- Wir interessieren uns nur für das asymptotische Verhalten
  - für sehr große N
  - Unabhängig von den Konstanten c und d wird ab einem gewissen Zeitpunkt c*N schneller wachsen als d*$\log_2$(N)







Effizienzabschätzung

- Wir vergleichen Algorithmen bezüglich der Anzahl der Schritte
  - Schritte sind:
    - Zuweisung
      - k++ , temp[k] = daten[re], ...
    - Vergleich
      - daten[li] < daten[re], daten[li] < pivot , liste[k]==x, ...
    - Funktionsaufruf mit konstantem Zeitbedarf
      - swap(daten,i,j),
  - Anzahl der Schritte abhängig von Datengröße N
    - Wir interessieren uns nur für große N
  - Das ergibt nur eine grobe Abschätzung
    - Asymptotisch aber genau genug

Zeitbedarf von linearer Suche

Datengrösse N = liste.length

```
public static int linSearch(int[] liste, int x){
  int k=0;                                          } 1 Schritt
  do
    if ( liste[k] == x ) return k;                  } 2 Schritte
    else k++;
  while (k<=liste.length-1);                        } 1 Schritt
                                                    } maximal N*3 Schritte
  return -1;                                        } 1 Schritt
}
```

Σ= 2+3*N Schritte maximal

Zeitbedarf von binärer Suche

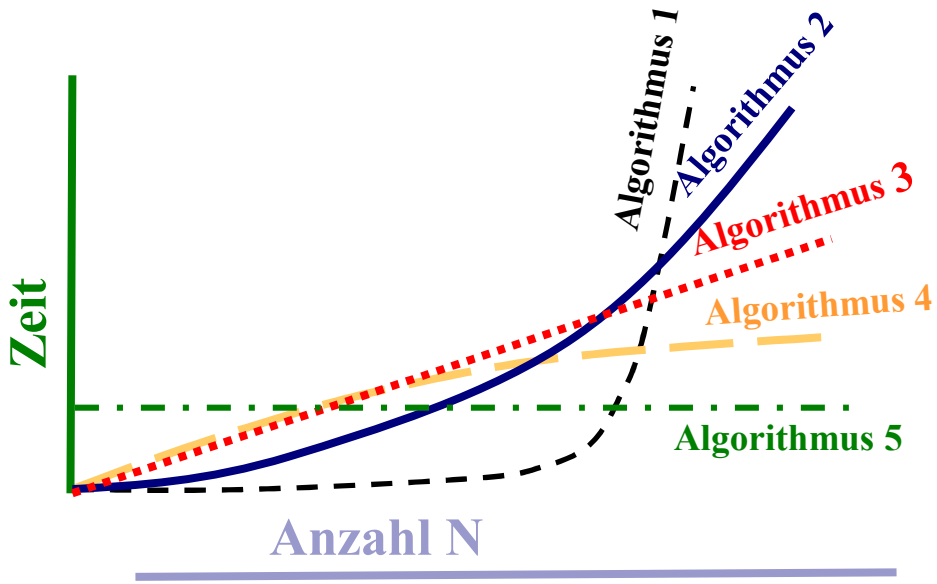
Datengrösse N = liste.length

```
public static int binSearch(int[] liste,int x){
  int lo = 0,mid=0;                                   } 2 Schritte
  int hi=liste.length-1;                              } 1 Schritt
  while(lo <= hi){                                    } maximal 4 * log2(N) Schritte
    mid=(lo+hi)/2;                                    } 1 Schritt
    if (x==liste[mid]) return mid;                    } max 3 Schritte
    else if (x < liste[mid]) hi=mid-1;
    else lo=mid+1;
  }
  return -1;                                          } 1 Schritt
}
```

$\Sigma = 4+4*\log_2(N)$ Schritte maximal

Welcher Algorithmus ist besser

- Aufwand in Abhängigkeit von der Größe N der Eingangsdaten
- Wir interessieren uns nur für große N
- **Exponentiell** wachsende Algorithmen sind nur für kleine N praktikabel
- **Quadratisch** wachsende Algorithmen sind noch akzeptabel
- **Lineare** Algorithmen (linSearch): Aufwand proportional der Datengröße
- **Logarithmisch** wachsende Algorithmen (z.B. binSearch): Datengröße fällt kaum ins Gewicht
- **Konstante** Algorithmen: Laufzeit unabhängig von der Datengröße



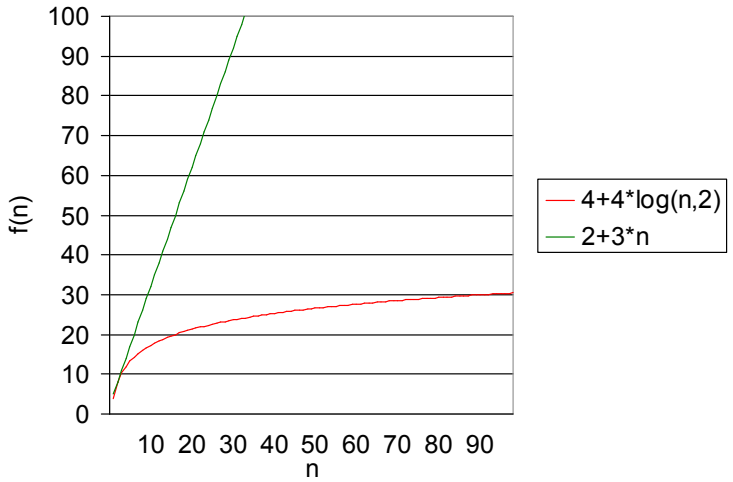
Showdown – nur für große N:

- Was wächst schneller ?
  - Binäre Suche : f(N) = 4+4*$\log_2$(N)
  - Lineare Suche : g(N) = 2+3*N
- Für große N wird f(N) durch g(N) begrenzt
  - Mathematisch:

$$\exists n_0 \in IN, c \in IR^+ \text{ mit : } \forall n \geq n_0 \text{ ist } f(n) \leq c \cdot g(n).$$

  - Beweis: 4+4*$\log_2$(N) $\leq$ 4+6*N = 2*g(N)
- Wir benutzen dafür die *O-Notation*:
  - f(n) = O(g(n))
- Vorsicht:
  - f(n) = O(g(n)) ist nur eine Notation. Es folgt nicht, dass g=O(f(n)).

$$f = O(g) \not\Rightarrow g = O(f)$$



O(g(n)) als Klasse von Funktionen

- *O(g(n))* kann man als Klasse von Funktionen auffassen

$$O(g(n)) = \{f : IN \rightarrow IR^+ \mid \exists n_0 \in IN . \exists c \in IR^+ . \forall n \geq n_0 . f(n) \leq c \cdot g(n)\}$$

- Damit wäre die folgende Notation besser:

$$f(n) \in O(g(n))$$

- Allerdings hat sich *f(n) = O(g(n))* eingebürgert
  - Äquivalent dazu:
    - *O(f(n)) ≤ O(g(n))* statt *O(f(n)) ⊆ O(g(n))*

- Aus der Definition erhalten wir sofort:
  - *O(f(n)) ≤ O(g(n)) ⇒ O(f(n) + g(n)) = O(g(n))*
  - *Insbesondere z.B.*
    - *O(a_k*n^k + …+ a_1*n+a_0) = O(n^k)*

Wichtiges Kriterium: Dominierung

- *f(n) dominiert g(n)*, falls $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ existiert
- Beispiele:
  - 2^n *dominiert* n^k *für jedes feste k*
  - n^k dominiert n^r, *falls* $k \geq r$
  - *n* dominiert $log_b(n)$ *für jede Basis b*
  - *n*log(n) dominiert n*
  - n^3+n^2 dominiert n^3 und n^3 dominiert n^3+n^2
- Wird *g(n) von f(n)* dominiert, dann gilt *g(n) = O(f(n))* und

$$O(f(n)+g(n)) = O(f(n))$$

Beispiele

- $O(n^2 - n) = O(5*n^2 + 88*n+1) = O(n^2)$
- $O(c*f(n)) = O(f(n))$
- $O(\log_{10}(n)) = O(\log_2(n))$
- $O(789) = O(1)$
- $O(n^2+3^n) = O(3^n)$

f dominiert g $\Rightarrow$ g=O(f)

$$f \text{ dominiert } g$$

$$\Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} \text{ existiert}$$

Definition „dominiert“

$$\Leftrightarrow \exists k \geq 0.\ \lim_{n\to\infty} \frac{g(n)}{f(n)} = k$$

$$\Leftrightarrow \exists k \geq 0.\ \forall \varepsilon > 0. \exists n_0 . \forall n \geq n_0 . \left| \frac{g(n)}{f(n)} - k \right| < \varepsilon$$

insbes. für ε=1

$$\Rightarrow \exists k \geq 0.\ \exists n_0 . \forall n \geq n_0 . \left| \frac{g(n)}{f(n)} - k \right| < 1$$

$|x| < 1 \Rightarrow x < 1$

$$\Rightarrow \exists k \geq 0.\ \exists n_0 . \forall n \geq n_0 . \frac{g(n)}{f(n)} - k < 1$$

$$\Leftrightarrow \exists k \geq 0.\ \exists n_0 . \forall n \geq n_0 . g(n) < (1+k) \cdot f(n)$$

Wähle c=(1+k)

$$\Rightarrow \exists c \geq 0.\ \exists n_0 . \forall n \geq n_0 . g(n) < c \cdot f(n)$$

$$\Leftrightarrow g(n) = O(f(n))$$

Definition O(g(n))

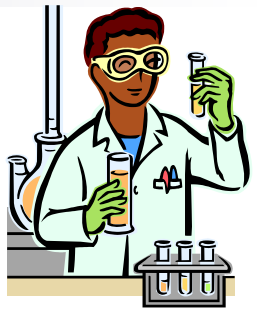
Bestimmung der O-Klasse

- Elementare Aktionen
  - Zuweisungen, Vergleiche, …
  - O(Zuweisung) = O(Vergleich) = O(1)
- Sequenz (Hintereinanderausführung)
  - O(S_1 ; S_2) = dom(O(S_1),O(S_2))
- Bedingte Anweisungen
  - O(if (B) S_1 ; else S_2) = dom(O(B),O(S_1),O(S_2))
- Schleifen
  - O(while(B) S) = O(S) * #Schleifendurchläufe
  - O(for(..) S) = O(S) * #Iterationen

muss man abschätzen

Mit *dom(f(n),g(n))* bezeichnen wir die dominierende Funktion – sofern eine solche existiert – ansonsten einfach *f(n)+g(n)*

Analyse von BubbleSort



```
static void bubbleSort(int[] daten){
  for(int j=daten.length-1; j >0; j--){
    for(int i=0; i< j;i++){
      if(daten[i]>daten[i+1])
        swap(daten,i,i+1);
    }
  }
}
```

O(1)

O(1)

O(1)

n*O(1) = O(n)

(n-1)*O(n) = $O(n^2)$

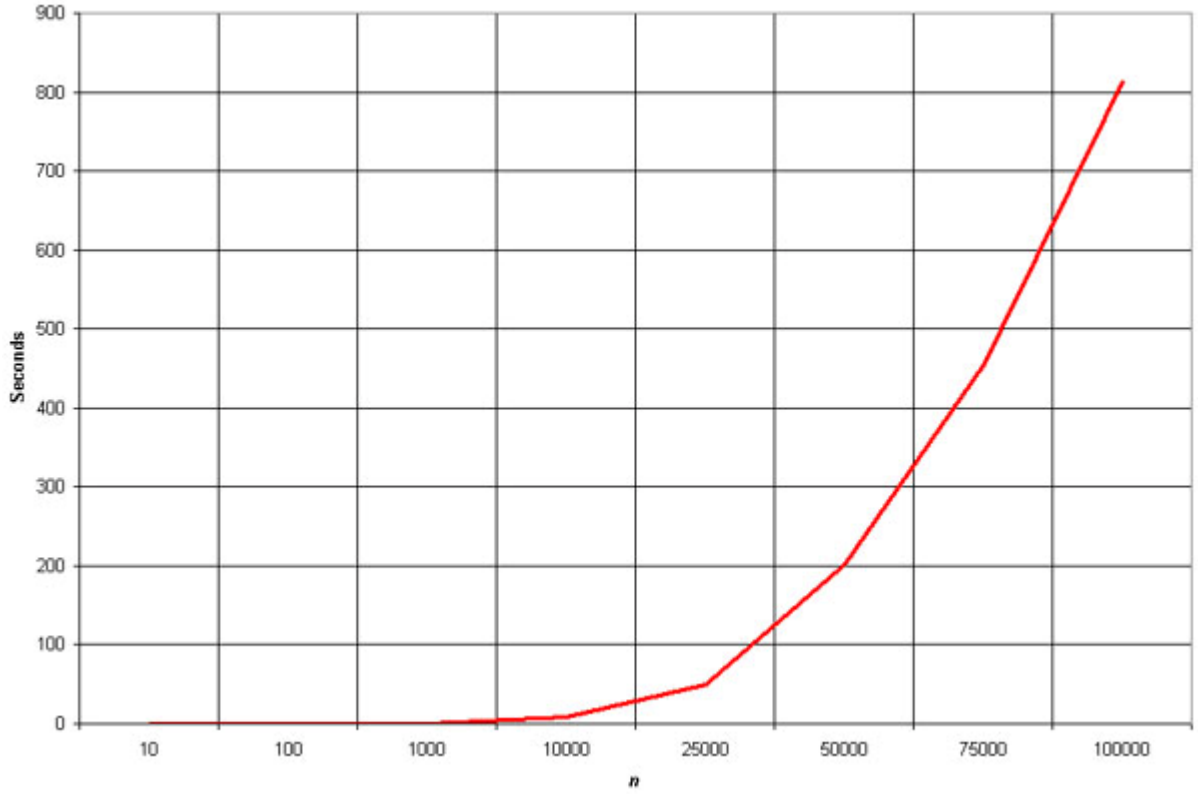
BubbleSort hat Komplexität $O(n^2)$

Eine genauere Analyse ergibt auch nur

$(n-1)*O(1) + (n-2)*O(1) + \ldots + 1*O(1) = O(n*(n-1)/2) = O(½*n^2 - ½*n) = O(n^2)$

BubbleSort in der Praxis

- Die empirisch gewonnene Kurve belegt die quadratische Komplexität von BubbleSort



Beachte die „gestauchte“ Skala

Quelle: http://linux.wku.edu/~lamonml/algor/sort

Analyse rekursiver Algorithmen

- Sei T(n) die Zeitkomplexität von auxBinSearchRec, der rekursiven Variante von binSearch
- n = hi-lo

```
static int auxBinSearchRec(int[] a, int lo, int hi,int s)
                                        throws NixDaException{
  if (hi < lo) throw new NixDaException();
  else{
      int mitte=(hi+lo)/2;
      if( s == a[mitte]) return a[mitte];
      else if (s < a[mitte])
            return auxBinSearchRec(a,lo,mitte-1,s);
      else return auxBinSearchRec(a,mitte+1,hi,s);
    }
}
```

O(1)

O(1)

O(1)

b

T((n-1)/2)

T((n-1)/2)

alternativ

T(n) = b+T((n-1)/2)

Wir erhalten die Gleichungen

T(1) = a, T(n) = b+T((n-1)/2)

Lösung der rekursiven Gleichung

- Ausgangspunkt: $T(1)=a$, $T(n) = b+T((n-1)/2)$
- Gegeben n, wähle k mit $2^{k-1} \leq n \leq 2^k-1$,
- $T(n) \leq T(2^k-1)$

$$
\begin{aligned}
&= b + T(((2^k-1)-1)/2) \\
&= b + T(2^{k-1}-1) \\
&= b + b+T(2^{k-2}-1) \\
&= \ldots \\
&= b + b + \ldots + b + T(2^1-1) \\
&= b(k-1)+T(1) \\
&\leq b \log_2(n) + a
\end{aligned}
$$

- Folglich: $T(n) = O(log(n))$
- *BinSearch hat Komplexität O(log(n)).*

MergeSort ist O(n*log(n))

```
public static void mergeSort(int[] daten){
  mSort(daten,0,daten.length-1);
}

private static void mSort(int[] daten, int lo, int hi){
  if (hi - lo >= 1){
     int mitte = (lo+hi+1)/2;
     mSort(daten,lo,mitte-1);
     mSort(daten,mitte,hi);
     merge(daten,lo,mitte,hi);
  }
}
```

O(1)

O(T(n/2))

O(T(n/2))

O(n)

O(n)+2*O(T(n/2))

Gleichung: $T(1)=a, \quad T(n) = b*n + 2*T(n/2))$

Lösung der rekursiven Gleichung

- Wir haben: $T(1)=a$, $T(n) \leq b*n+2*T(n/2)$
- Gegeben n, wähle k mit $2^k \leq n < 2^{k+1}$, d.h. $k \leq \log_2(n) < k+1$.
- $T(n) \leq T(2^{k+1})$

$$\leq b*2^{k+1}+ 2*T(2^{k+1}/2)$$

$$= b*2^{k+1}+ 2*T(2^k)$$

$$\leq b*2^{k+1}+ 2*(b*2^k+ 2*T(2^k/2))$$

$$= \ldots$$

$$= \overbrace{b*2^{k+1} + 2*b*2^k + \ldots 2^k*b*2^1}^{K+1 \text{ Summanden}} + 2^{k+1}*b*T(2^0)$$

$$= (k+1)*b*2^{k+1} + T(1)*b*2^{k+1}$$

$$\leq (\log_2 n + 1)*b*(2*n) + a*b*2^{k+1}$$

$$\leq 2b*n*\log_2(n) + 2*b*n + a*b*2*n$$

$$= O(n*\log(n))$$

- Folglich: $T(n) = O(n*\log(n))$, *also: mergeSort hat Komplexität* $O(n*\log(n))$.

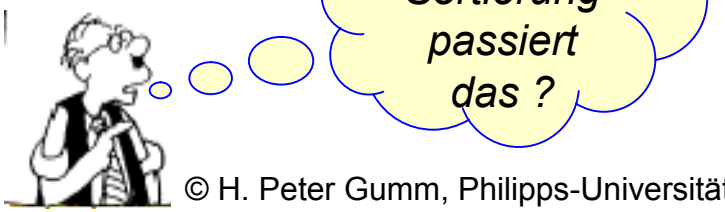
Best/average/worst case

- Bisher haben wir den Aufwand für einen Algorithmus pessimistisch abgeschätzt:
  - Wieviele Schritte braucht er im schlimmsten Fall (engl.: worst case)
- Man kann auch den besten Fall (best case) betrachten
  - linSearch bzw. binSearch finden sofort das gesuchte Element
    - Best case: O(1)
  - Die Daten bei Insertionsort können schon sortiert sein
    - Best case: O(n) (man muss jedes Element anschauen)
- Aussagekräftiger (aber oft schwerer abzuschätzen) ist der durchschnittliche Fall (average case)
  - Bei linSearch wird das Element im Schnitt nach n/2 Schritten gefunden
  - O(n/2) = O(n) , also worst case = average case
  - Für binSearch gilt ebenfalls O(worst case) = O(average case)

O(-) für best/worst/average case

- Komplexitäten können übereinstimmen
  - Best/average case
  - Average und worst case
- Average case ist oft nicht einfach zu bestimmen

Algorithmus	best case	average case	worst case
binäre Suche	1	log(n)	log(n)
lineare Suche	1	n	n
bubbleSort	n	n*n	n*n
selectionSort	n*n	n*n	n*n
quickSort	n*log(n)	n*log(n)	n*n



Quicksort im günstigsten Fall

Angenommen, wir könnten den pivot immer so wählen, dass er einen mittleren Datenwert liefert.

$$\left|\{\, i \mid a[i] \leq pivot \,\}\right| = \left|\{\, i \mid a[i] > pivot \,\}\right|$$

Dann zerlegt die Partitionierung den Array-Abschnitt a[lo .. hi] immer in zwei gleich große Teile

a[lo .. pivIndex-1] und a[pivIndex+1 .. hi]

Benötigt quickSort für die Sortierung eines Arrays mit n Elementen T(n) Schritte, so benötigt er für die Sortierung der so entstandenen Hälften höchstens 2*T(n/2) Schritte

Genaue Analyse des optimalen Falles

```
private static void quSort(int[] daten, int i, int j){
  // Trivialfälle: länge <= 1:
  if (j==i+1 && daten[j]< daten[i]) swap(daten,i,j);
  if (j-i<=1)return;

  // Wähle Index für einen pivot mit i < pivIndex < j:
  int pivIndex=(i+j)/2;

  // Partitioniere und gebe Index für Pivot zurück:
  pivIndex = partition(daten,i,j,pivIndex);

  // Sortiere untere Hälfte daten[i..pivIndex-1]
  quSort(daten,i,pivIndex-1);

  // Sortiere obere Hälfte daten[pivIndex..j]
  quSort(daten,pivIndex+1,j);
}
```

O(1)

O(n)

O(n)

O(T(n/2))

O(T(n/2))

Gleichung: $T(1)=a, \quad T(n) \leq b*n + 2*T((n-1)/2))$

Lösung der rekursiven Gleichung

- Wir haben: $T(1)=a$, $T(n) \leq b*n+2*T((n-1)/2)$
- Gegeben n, wähle k mit $2^{k-1} \leq n \leq 2^k-1$, d.h. $(k-1) \leq \log_2(n) < k$.
- $T(n) \leq T(2^k-1) \leq b(2^k-1)+ 2T(((2^k-1)-1)/2)$

  $= b(2^k-1)+ 2T(2^{k-1}-1)$

  $= b(2^k-1)+ 2(b*(2^{k-1}-1)+ 2*T(2^{k-2}-1))$

  $= \ldots$ (K-1) Summanden

  $= b(2^k-1) + 2b(2^{k-1}-1) + \ldots 2^{k-2}b(2^2-1) + 2^{k-1}T(2^1-1)$

  $= b(k-1)2^k - b(1+2+\ldots+2^{k-2}) + 2^{k-1}T(1)$

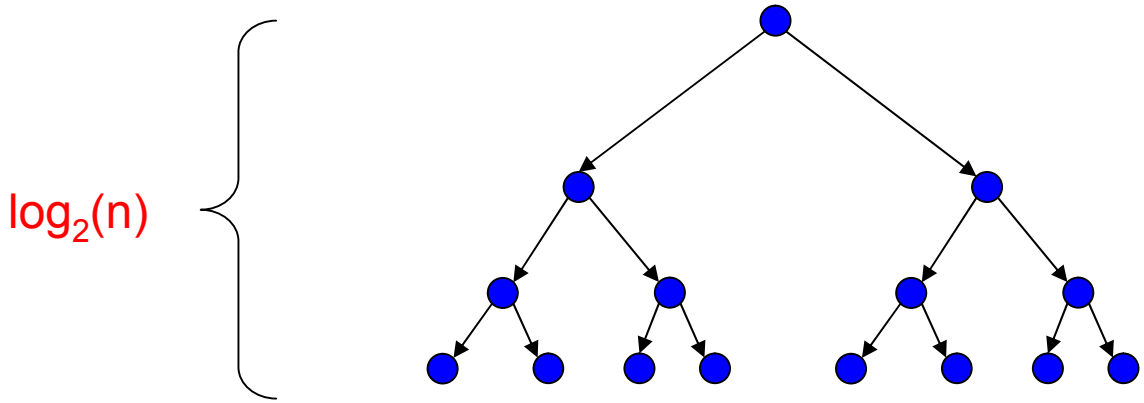
  $= 2b(k-1)2^{k-1} - b(2^{k-1}-1) + 2^{k-1}a$

  $= 2b(k-1)2^{k-1} +(a-b)2^{k-1} + b$

  $\leq 2b*\log_2(n)*n + (a-b)*n + b$

  $= O(n*\log(n))$
- Folglich: $T(n) = O(n*\log(n))$, *also:* *quickSort hat im Best Case Komplexität* $O(n*\log(n))$.

Graphische Aufwandsabschätzung



$$c \cdot n$$

$$2 \cdot (c \cdot n)/2 = c \cdot n$$

$$4 \cdot (c \cdot n)/4 = c \cdot n$$

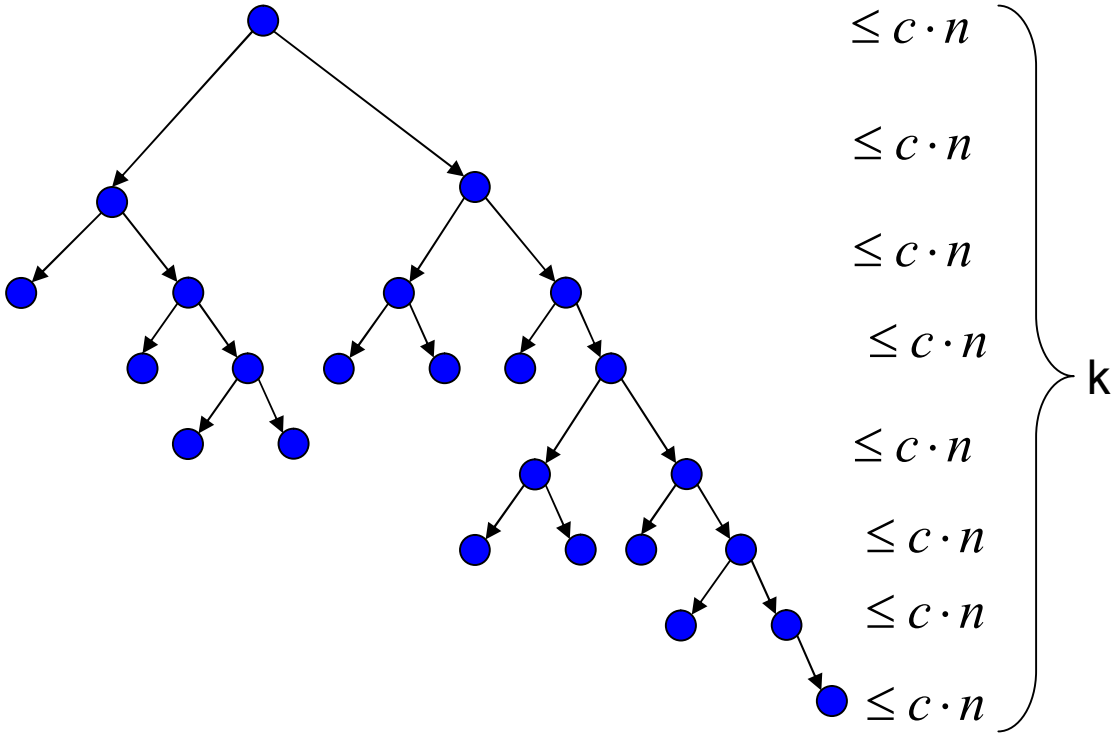
$$8 \cdot (c \cdot n)/8 = c \cdot n$$

$$\sum = c \cdot n \cdot \log_2 n$$

- Knoten: Aufteilung der Arbeit in zwei Teilaufgaben
  - Pivot wählen und partitionieren
- In jeder Ebene wird jeder Teil des Arrays partitioniert
  - Aufwand pro Ebene insgesamt $c \cdot n$
- Nach spätestens $log_2 n$ Aufteilungen Sind die Intervalle ein-elementig
- Gesamtaufwand daher: $c \cdot n \cdot log_2 n$

Quicksort mit schlechter Pivotwahl

- Optimale Pivotwahl i.A. unmöglich
- Angenommen, Aufteilung nie schlechter als 1/10 : 9/10, dann:
  - in jeder Ebene wird jeder Teil des Arrays partitioniert
    - Aufwand $\leq c \cdot n$
  - Größtes Intervall in Ebene k
    - ist höchstens $n \cdot (9/10)^k$
    - hat mehr als 1 Element wenn $n \cdot (9/10)^k > 1$
  - Logarithmieren:
    $k < \log_2(n) / \log_2(10/9)$
  - Der längste Ast im Baum hat also Länge $\log_2(n)/\log_2(10/9)$
- Gesamtaufwand wieder:
  $c/\log_2(10/9) \cdot \log_2(n) \cdot n = O(n \cdot \log n)$



$$\sum = k \cdot c \cdot n \cdot \log_2 n$$

QuickSort – average Case

- Es lässt sich zeigen, dass bei zufällig verteilten Daten Quicksort *O(n log n)* ist.
- Sei *C(n)* die Anzahl der Vergleiche (*Comparisons*), die Quicksort für das Sortieren von n zufällig verteilten Datenelementen braucht.
  - n -1 Vergleiche benötigt das Partitionieren
  - Ist der gewählte Pivot gerade das i-größte Element, dann gilt

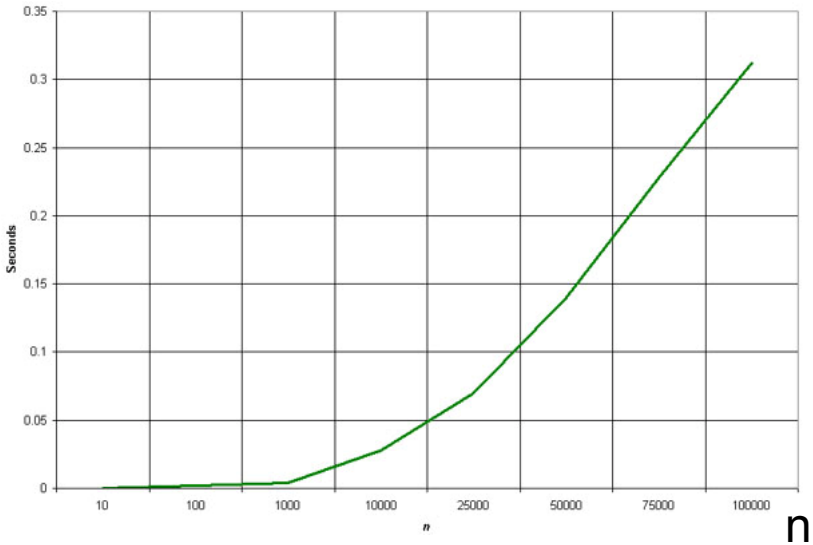
    C(n) = (n -1) + C(i) + C(n-i-1)
- Da alle i zwischen 1 und n gleich wahrscheinlich sind, folgt:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1))$$

- Die Lösung dieser Gleichung führt auf
  - $C(n) = O(n*\log_2(n))$
- Das bestätigt sich auch in der Praxis

Man beachte die gegenüber der BubbleSort-Folie geänderte Skala

t

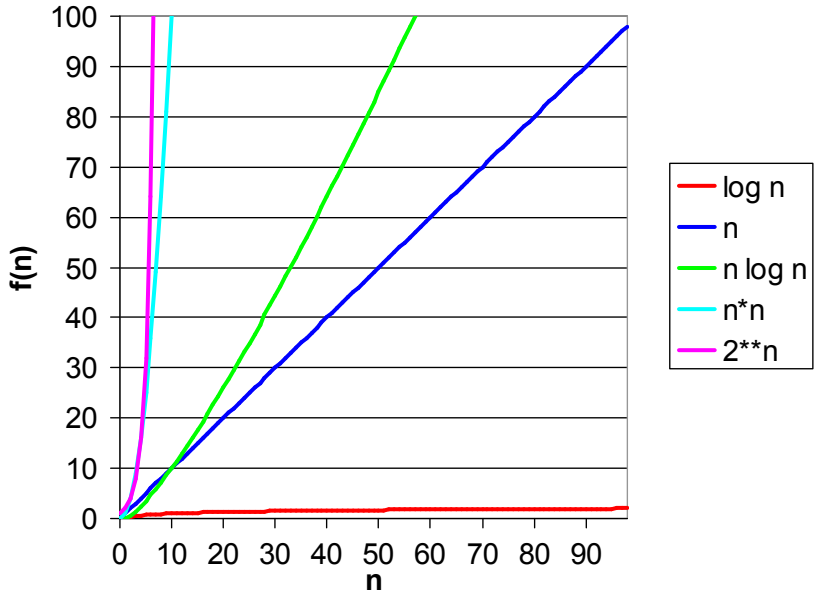


n

Quelle: http://linux.wku.edu/~lamonml/algor/sort/

Wichtige O-Klassen

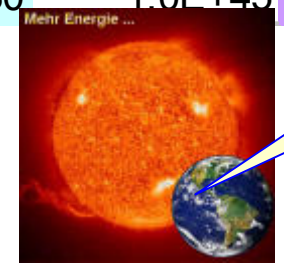
- **$O(2^n)$: Klasse aller exponentiellen Algorithmen**
  - Algorithmen, die ihre Lösung durch systematisches Ausprobieren finden
  - Beispiel: Wie packt man möglichst viele verschieden große Quader in einen Waggon?
  - Hoffnungslos ineffizient für große n
- **$\bigcup_{k \in IN} O(n^k)$: Klasse aller polynomialen Algorithmen**
  - Gelten als „noch praktikable“ Algorithmen
- **$O(n^2)$: Klasse aller quadratischen Algorithmen**
  - Einfache Sortieralgorithmen sind quadratisch (bubbleSort, insertionSort, selectionSort)
- **O(n*log(n)) : loglineare Algorithmen**
  - Gute Sortieralgorithmen sind loglinear (quickSort)
- **O(n) : Klasse aller linearen Algorithman**
  - sehr gut behandelbare Algorithmen
  - Beispiel: lineare Suche
- **O(log(n)) : logarithmische Algorithmen**
  - Extrem effizient
  - Beispiel binäre Suche
- **O(1) : Klasse aller konstanten Algorithmen**
  - Laufzeit unabhängig von Datengrösse



Wachstum einiger Funktionen

n	log n	n	n log n	n*n	n*n*n	2**n
1	0,0	1	0,0	1	1	2
10	3,3	10	33,2	100	1000	1024
100	6,6	100	664,4	10000	1000000	1,E+30
1.000	10,0	1.000	9965,8	1000000	1000000000	1,E+301
10.000	13,3	10.000	132877,1	100000000	1,0E+12	
100.000	16,6	100.000	1660964,0	1,0E+10	1,0E+15	
1.000.000	19,9	1.000.000	19931568,6	1,0E+12	1,0E+18	
10.000.000	23,3	10.000.000	232534966,6	1,0E+14	1,0E+21	
100.000.000	26,6	100.000.000	2657542475,9	1,0E+16	1,0E+24	
1.000.000.000	29,9	1.000.000.000	3,0E+10	1,0E+18	1,0E+27	
1,0E+10	33,2	1,0E+10	3,3E+11	1,0E+20	1,0E+30	
1,0E+11	36,5	1,0E+11	3,7E+12	1,0E+22	1,0E+33	
1,0E+12	39,9	1,0E+12	4,0E+13	1,0E+24	1,0E+36	
1,0E+13	43,2	1,0E+13	4,3E+14	1,0E+26	1,0E+39	
1,0E+14	46,5	1,0E+14	4,7E+15	1,0E+28	1,0E+42	
1,0E+15	49,8	1,0E+15	5,0E+16	1,0E+30	1,0E+45	

Vor ca 1,0 E+14 msec habe ich meine Pyramide gebaut



Die Sonne wiegt ca. 3,0E+35 g

Machbarkeitsüberlegungen

- Angenommen:
  - im Test zeigt sich, dass ein Programm für 10 Datenwerte 1 sec benötigt
- Wenn der Algorithmus Komplexität O(f(n)) hat, wieviele Eingabedaten kann er in 1 Tag, 1 Jahr, 10 Jahren, 1000 Jahren verarbeiten?

Komplexität O(-)	Gegeben: 1 sec	Verarbeitbare Datengrösse 1 Tag	1 Jahr	10 Jahre	1000 Jahre
n	10	864000	315360000	3,E+14	9,9E+18
n*log(n)	10	165551	4724923	41402010	3,3E+09
n*n	10	930	17758	16506697	3153600000
n*n*n	10	95	681	64830	2150492
2**n	10	20	35	38	45

- Folgerung
  - Polynomiale Algorithmen skalieren auch auf große Datenmengen
  - Exponentielle Algorithmen sind für große Datenmengen wertlos

Gibt es schnellere Sortieralgorithmen ?

- Satz: Ein Sortieralgorithmus, der
  - auf Vergleichen von Datenelementen beruht, und
  - mit einer unbeschränkten Anzahl verschiedener Daten umgehen kann,

  hat mindestens Komplexität O(n*log((n)).

Beweisidee: Sortieren bedeutet umordnen, d.h. *permutieren* einer ungeordneten Folge zu einer geordneten Folge.

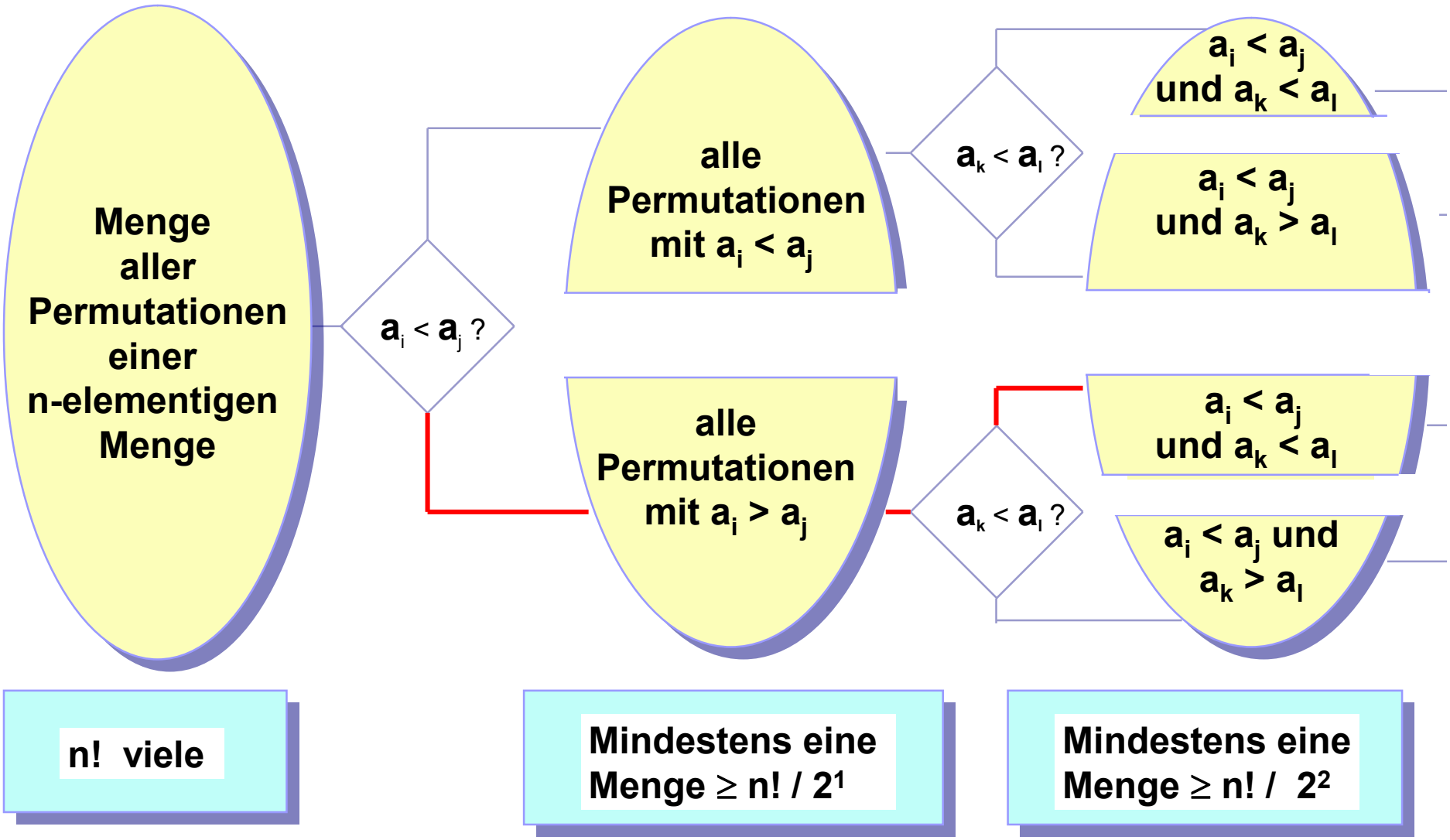
Jede der n! denkbare Permutation σ einer n-elementigen Folge kann in Frage kommen um die ungeordnete Folge S_0 zur geordneten Folge $S_1 = \sigma(S_0)$ zu ordnen. ($S_1 = \sigma(S_0) \Leftrightarrow S_0 = \sigma^{-1}(S_1)$) .

Sortieren durch Vergleichen heißt, durch Fragen der Art

$$a_i < a_j \quad ?$$

die Permutation zu finden, die die Daten in die richtige Reihenfolge bringt.

Jede solche Frage zerlegt die Menge aller noch möglichen Permutationen in maximal zwei Teile :

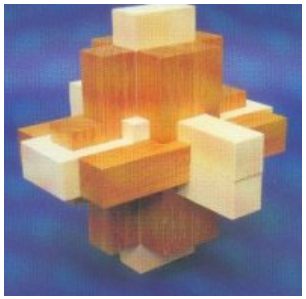


Ende des Beweises

- Frühestens nach $\log_2(n!)$ vielen Zerlegungen, also $\log_2(n!)$ Vergleichen, ist in jeder der Teilmengen höchstens ein Element übrig.
- Für große n kann man n! durch die Stirlingsche Formel approximieren:

$$n! \approx (n/e)^n \cdot \sqrt{2\pi \cdot n}$$

- Es folgt: $O(\log_2(n!)) = O(n*\log_2(n))$.
- Es werden also mindestens O(N * lg N) viele Vergleiche benötigt.



Stabile Sortierverfahren

- Ein Sortierverfahren heißt stabil, falls Elemente mit gleichem Sortierschlüssel nicht vertauscht werden
  - BubbleSort ist stabil
  - SelectionSort, InsertionSort kann man stabil implementieren
  - QuickSort ist nicht stabil

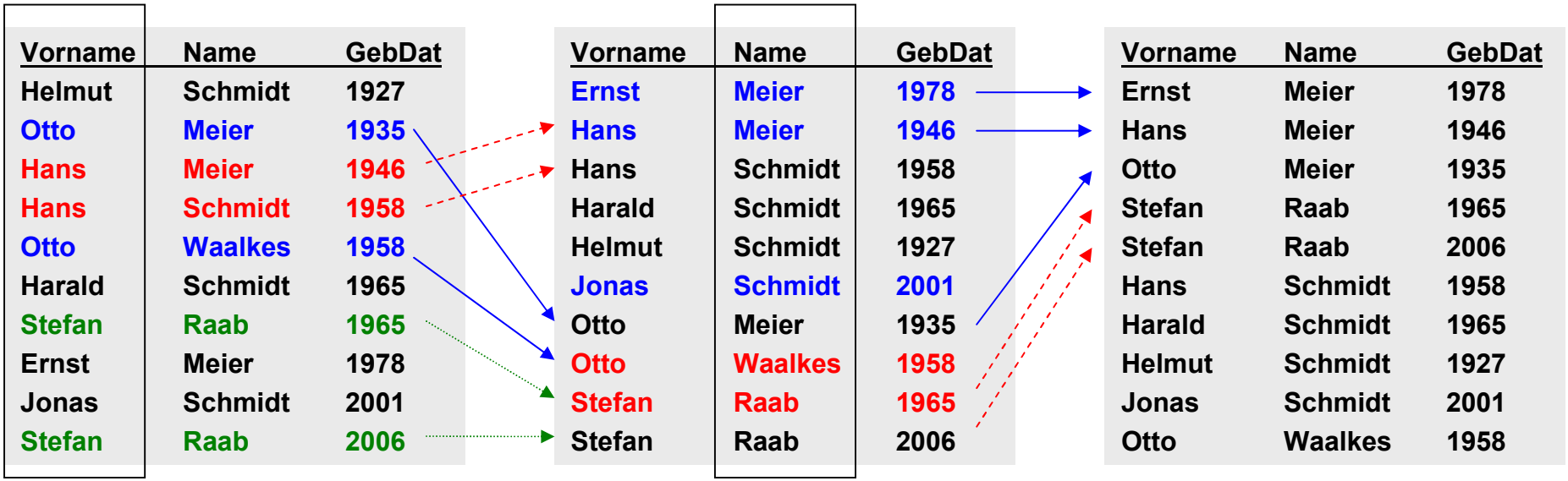
Vorname	Name	GebDat
Hans	Meier	1946
Hans	Schmidt	1958
Harald	Schmidt	1965
Helmut	Schmidt	1927
Otto	Waalkes	1958
Stefan	Raab	1965
Ernst	Meier	1978
Jonas	Schmidt	2001
Otto	Meier	1935
Stefan	Raab	2006

stabiles Sortieren
nach Geburtsdatum

Vorname	Name	GebDat
Helmut	Schmidt	1927
Otto	Meier	1935
Hans	Meier	1946
Hans	Schmidt	1958
Otto	Waalkes	1958
Harald	Schmidt	1965
Stefan	Raab	1965
Ernst	Meier	1978
Jonas	Schmidt	2001
Stefan	Raab	2006

Weitere stabile Sortierschritte

- Wir sortieren anschließend stabil nach
  - Vornamen, dann nach
  - Namen



Vorname	Name	GebDat
Helmut	Schmidt	1927
Otto	Meier	1935
Hans	Meier	1946
Hans	Schmidt	1958
Otto	Waalkes	1958
Harald	Schmidt	1965
Stefan	Raab	1965
Ernst	Meier	1978
Jonas	Schmidt	2001
Stefan	Raab	2006

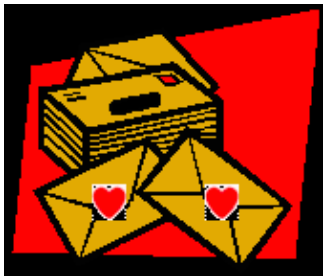
Vorname	Name	GebDat
Ernst	Meier	1978
Hans	Meier	1946
Hans	Schmidt	1958
Harald	Schmidt	1965
Helmut	Schmidt	1927
Jonas	Schmidt	2001
Otto	Meier	1935
Otto	Waalkes	1958
Stefan	Raab	1965
Stefan	Raab	2006

Vorname	Name	GebDat
Ernst	Meier	1978
Hans	Meier	1946
Otto	Meier	1935
Stefan	Raab	1965
Stefan	Raab	2006
Hans	Schmidt	1958
Harald	Schmidt	1965
Helmut	Schmidt	1927
Jonas	Schmidt	2001
Otto	Waalkes	1958

- Ergebnis:
  - Sortierung nach Namen
  - bei Namensgleichheit nach Vornamen
  - bei Gleichheit von Namen und Vornamen, nach Geburtsdatum

RadixSort

- Gegeben ein Array von Daten mit Sortierschlüsseln $k_1,\ldots,k_n$
  - k_n habe höchste Wertigkeit, k_1 niedrigste



- Gewünscht :
  - Sortierung nach k_n
  - bei Gleichheit, nach k_{n-1}
  - etc..
- Algorithmus

```
for i = 1 to n do
        sortiere stabil nach k_i
```

- Korrektheit
  - nach dem i-ten Durchlauf sind die Daten nach den Schlüsseln $k_i,\ldots,k_1$ geordnet
  - stabiles Sortieren im i+1-ten Schritt garantiert:
    - nach dem i+1-ten Durchlauf sind Daten nach k_{i+1} geordnet und bei gleichem k_{i+1} immer noch nach $k_i,\ldots,k_1$

So machte es die Post …

… als Postleitzahlen noch vierstellig waren

… auch Lochkarten wurden auf diese Weise geordnet:

Sortieren durch Verteilen in Fächer (DistributionSort)

Verteile alle Briefe in 10 Fächer anhand der letzten Ziffer

Zusammenlegen unter Beibehaltung der bisherigen Ordnung

Verteile alle Briefe in 10 Fächern anhand der vorletzten Ziffer

Zusammenlegen unter Beibehaltung der Ordnung

... etc. ...

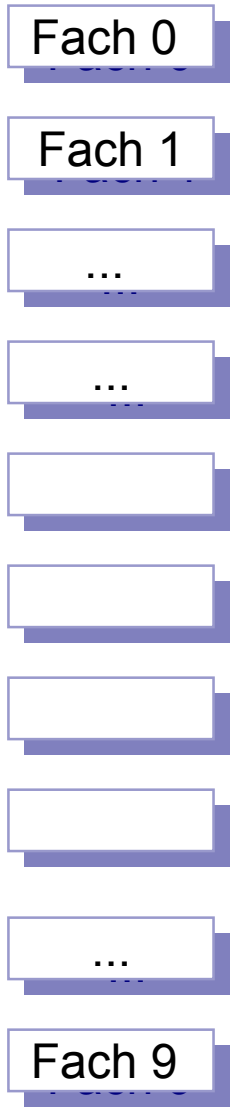
DistributionSort ist eine Variante von RadixSort – daher ist Korrektheit offensichtlich

DistributionSort

Die folgenden Briefe sollen nach Postleitzahlen sortiert werden.

Es stehen uns 10 Behälter (Fächer) zur Verfügung.

Brief 1	nach	3 5 5 0	Marburg
Brief 2	nach	7 1 4 2	Marbach
Brief 3	nach	8 6 5 1	Mainroth
Brief 4	nach	7 3 2 1	Zell
Brief 5	nach	5 5 8 3	Zell
Brief 6	nach	7 8 6 3	Zell
Brief 7	nach	8 0 0 0	München
Brief 8	nach	7 0 0 0	Stuttgart
Brief 9	nach	7 4 7 1	Schwenningen
Brief 10	nach	8 0 4 3	Unterföhring
Brief 11	nach	8 9 1 1	Unterfinning
Brief 12	nach	7 9 3 4	Untermarchtal
Brief 13	nach	7 7 7 0	Überlingen
Brief 14	nach	8 1 3 2	Tutzing
Brief 15	nach	8 1 2 1	Obersöchering



Sortieren nach der letzten Ziffer

Sortiere nach der letzten Ziffer in die Fächer:

Brief 1	nach	3 5 5	0	Marburg	Fach 0
Brief 7	nach	8 0 0	0	München	
Brief 8	nach	7 0 0	0	Stuttgart	
Brief 13	nach	7 7 7	0	Überlingen	
Brief 3	nach	8 6 5	1	Mainroth	Fach 1
Brief 4	nach	7 3 2	1	Zell	
Brief 9	nach	7 4 7	1	Schwenningen	
Brief 11	nach	8 9 1	1	Unterfinning	
Brief 15	nach	8 1 2	1	Obersöchering	
Brief 2	nach	7 1 4	2	Marbach	Fach 2
Brief 14	nach	8 1 3	2	Tutzing	
Brief 5	nach	5 5 8	3	Zell	Fach 3
Brief 6	nach	7 8 6	3	Zell	
Brief 10	nach	8 0 4	3	Unterföhring	
Brief 12	nach	7 9 3	4	Untermarchtal	Fach 4

3. Ziffer

Unter Beibehaltung der bisherigen Reihenfolge sortiere nach 3. Ziffer:

Brief 7	nach	8	0	0	0	München	Fach 0
Brief 8	nach	7	0	0	0	Stuttgart	
Brief 11	nach	8	9	1	1	Unterfinning	Fach 1
Brief 4	nach	7	3	2	1	Zell	Fach 2
Brief 15	nach	8	1	2	1	Obersöchering	
Brief 14	nach	8	1	3	2	Tutzing	Fach 3
Brief 12	nach	7	9	3	4	Untermarchtal	
Brief 2	nach	7	1	4	2	Marbach	Fach 4
Brief 10	nach	8	0	4	3	Unterföhring	
Brief 1	nach	3	5	5	0	Marburg	Fach 5
Brief 3	nach	8	6	5	1	Mainroth	
Brief 6	nach	7	8	6	3	Zell	Fach 6
Brief 13	nach	7	7	7	0	Überlingen	Fach 7
Brief 9	nach	7	4	7	1	Schwenningen	
Brief 5	nach	5	5	8	3	Zell	Fach 8

2. Ziffer

Unter Beibehaltung der bisherigen Reihenfolge sortiere nach 2. Ziffer:

Brief 7	nach	8	0	0	0	München	Fach 0
Brief 8	nach	7	0	0	0	Stuttgart	
Brief 10	nach	8	0	4	3	Unterföhring	
Brief 15	nach	8	1	2	1	Obersöchering	Fach 1
Brief 14	nach	8	1	3	2	Tutzing	
Brief 2	nach	7	1	4	2	Marbach	
Brief 4	nach	7	3	2	1	Zell	Fach 3
Brief 9	nach	7	4	7	1	Schwenningen	Fach 4
Brief 1	nach	3	5	5	0	Marburg	Fach 5
Brief 5	nach	5	5	8	3	Zell	
Brief 3	nach	8	6	5	1	Mainroth	Fach 6
Brief 13	nach	7	7	7	0	Überlingen	Fach 7
Brief 6	nach	7	8	6	3	Zell	Fach 8
Brief 11	nach	8	9	1	1	Unterfinning	
Brief12	nach	7	9	3	4	Untermarchtal	Fach 9

1. Ziffer

Unter Beibehaltung der bisherigen Reihenfolge sortiere nach 1. Ziffer:

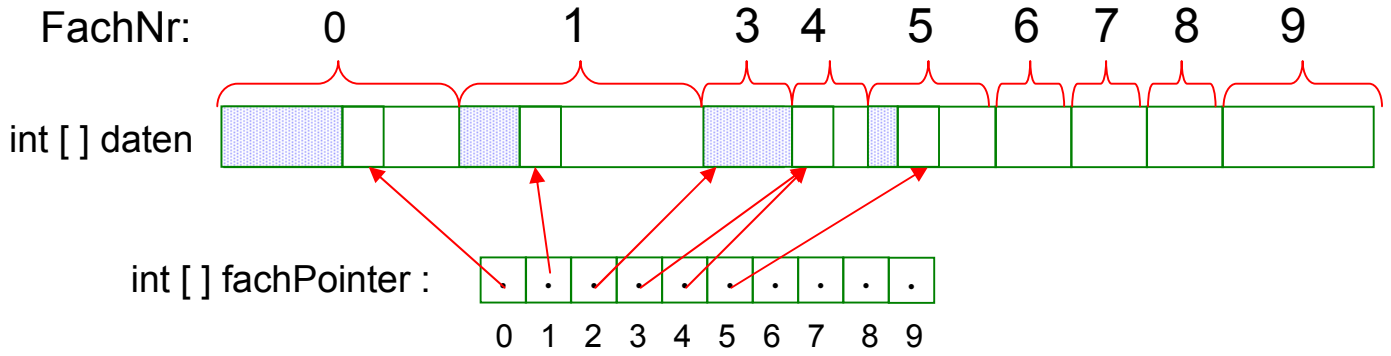
Brief 1	nach	3 5 5 0	Marburg	Fach 3
Brief 5	nach	5 5 8 3	Zell	Fach 5
Brief 8	nach	7 0 0 0	Stuttgart	
Brief 2	nach	7 1 4 2	Marbach	
Brief 4	nach	7 3 2 1	Zell	
Brief 9	nach	7 4 7 1	Schwenningen	Fach 7
Brief 13	nach	7 7 7 0	Überlingen	
Brief 6	nach	7 8 6 3	Zell	
Brief 12	nach	7 9 3 4	Untermarchtal	
Brief 7	nach	8 0 0 0	München	
Brief 10	nach	8 0 4 3	Unterföhring	
Brief 15	nach	8 1 2 1	Obersöchering	Fach 8
Brief 14	nach	8 1 3 2	Tutzing	
Brief 3	nach	8 6 5 1	Mainroth	
Brief 11	nach	8 9 1 1	Unterfinning	

Verteilung auf einen Blick

	0	1	2	3	4	5	6	7	8	9
Sortiert nach 4. Ziffer	3550 8000 7000 7770	8651 7321 7471 8911 8121	7142 8132	5583 7863 8043	7934					
Sortiert nach 3.und 4.Ziffer	8000 7000	8911	7321 8121	8132 7934	7142 8043	3550 8651	7863	7770 7471	5583	
Sortiert nach 2, 3. und 4. .Ziffer	8000 7000 8043	8121 8132 7142		7321	7471	3550 5583	8651	7770	7863	8911 7934
Sortiert nach 1, 2,3, und 4. .Ziffer				3550		5583		7000 7142 7321 7471 7770 7863 7934	8000 8043 8121 8651 8911	

DistributionSort

- funktioniert auch mit Schlüsseln, die Zahlen in anderen Zahlensystemen repräsentieren
  - Binärzahlen (Basis 2)
  - Dezimalzahlen (Basis 10)
  - Strings (Basis 26 oder 128 oder 256)
- Für Schlüssel zur Basis d benötigt man d Fächer.
- Jedes der d Fächer muss potentiell alle n Datenelemente aufnehmen können
- Die Summe der Größe aller Fächer braucht aber nur n zu sein.
- Realisiere die Fächer als entsprechend große Abschnitte in einem Array der Länge n.
  - Einige Fächer können leer sein
- Berechne in einem ersten Daten-Durchlauf die benötigte Fachgröße.
  - In einem Hilfsarry **fachZeiger** speichere in **fachZeiger[i]** einen Zeiger auf die Position im Fach **i**, an die das nächste Element gespeichert werden muss.

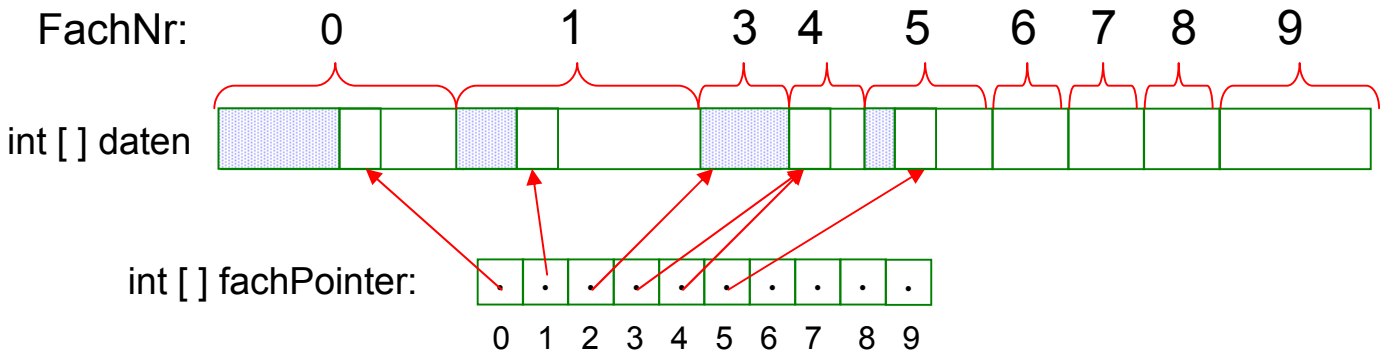


Schnappschuss:
(die grauen Felder sind bereits besetzt)
- Fach 2 ist leer
- Fach 3 bereits gefüllt

DistributionSort: Fächerorganisation

- **fachPointer**[] zerlegt **daten**[] virtuell in Fächer
- Daten werden in Hilfsarray **temp**[] zwischen-gespeichert
- Gleichzeitig benötigte Fächergröße bestimmen
- Fächeranfänge berechnen

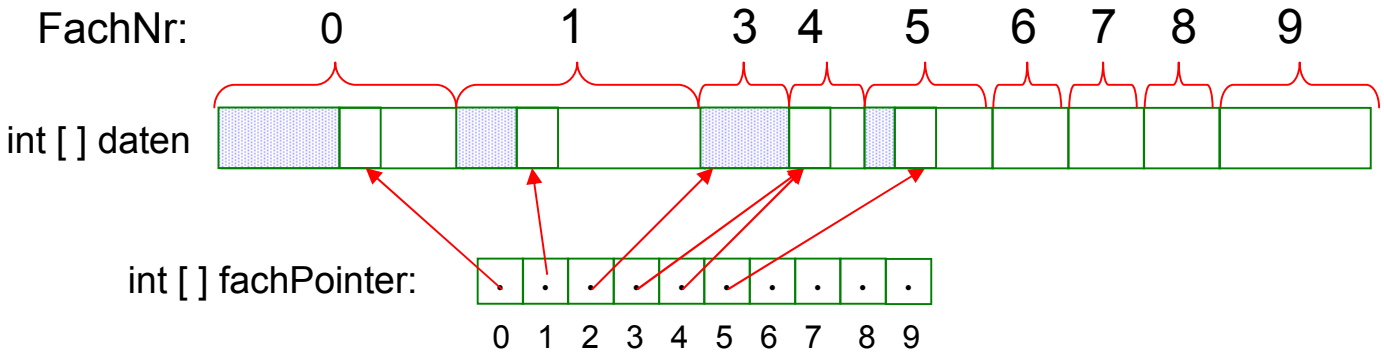
```
public static void radixSort(int[] daten, int keyLaenge){
  int d=10; // d : Basis des Zahlensystems
  int[] temp = new int[daten.length];
  for(int runde=0; runde<keyLaenge; runde++){
     int[] fachPointer = new int[d];
     // Kopiere daten[] nach temp[]
     // und bestimme gleichzeitig die Fachgrößen
     for(int i=0; i<daten.length;i++){
       temp[i]=daten[i];
       fachPointer[getZiffer(daten[i],runde)]++;
     }
     // Setze fachPointer[i] an die erste Position in Fach i
     for(int i=d-1; i>0; i--)fachPointer[i]=fachPointer[i-1];
     fachPointer[0]=0;
     for(int i=1; i<d; i++) fachPointer[i] += fachPointer[i-1];
```



Ein Schnappschuss:
(die grauen Felder sind bereits besetzt)

- Fach 2 ist leer
- Fach 3 bereits gefüllt

DistributionSort: Einsortieren



Ein Schnappschuss:
(die grauen Felder sind bereits besetzt)

- Fach 2 ist leer
- Fach 3 bereits gefüllt

- Einsortieren, dabei immer die Fächerindizes erhöhen

```
      // Sortiere die Daten in die Fächer
      int wert, fach;
      for (int i=0; i<temp.length; i++){
        wert=temp[i];                    // einzuordnender Wert
        fach=getZiffer(wert,runde);      // Richtiges Fach
        daten[fachPointer[fach]++]=wert; // einordnen,inkrementieren
      }
    }//for runde = ...
  }//end radixSort
```

DistributionSort: Komplett

- **fachPointer**[] zerlegt **daten**[] virtuell in Fächer
- Daten werden in Hilfsarray **temp**[] zwischen-gespeichert
- Gleichzeitig benötigte Fächergröße bestimmen
- Fächeranfänge berechnen
- Einsortieren, dabei immer die Fächerindizes erhöhen

```
public static void radixSort(int[] daten, int keyLaenge){
  int d=10; // d : Basis des Zahlensystems
  int[] temp = new int[daten.length];
  for(int runde=0; runde<keyLaenge; runde++){
     int[] fachPointer = new int[d];
     // Kopiere daten[] nach temp[]
     // und bestimme gleichzeitig die Fachgrößen
     for(int i=0; i<daten.length;i++){
       temp[i]=daten[i];
       fachPointer[getZiffer(daten[i],runde)]++;
     }
     // Setze fachPointer[i] an die erste Position in Fach i
     for(int i=d-1; i>0; i--)fachPointer[i]=fachPointer[i-1];
     fachPointer[0]=0;
     for(int i=1; i<d; i++) fachPointer[i] += fachPointer[i-1];
     // Sortiere die Daten in die Fächer
     int wert, fach;
     for (int i=0; i<temp.length; i++){
       wert=temp[i];                    // einzuordnender Wert
       fach=getZiffer(wert,runde);      // Richtiges Fach
       daten[fachPointer[fach]++]=wert; // einordnen,inkrementieren
     }
  }//for runde = ...
}//end radixSort
```

DistributionSort ist linear ? ! ?

- Genauer
  - Bei fester Schlüssellänge k ist der Aufwand T(n) = k*n, also

    $$T(n) = O(n).$$

  - Allerdings kann man bei fester Schlüssellänge k nur 10^k verschiedene Schlüssel haben
- Beschränkt man die Schlüssellänge nicht, so benötigt man für n verschiedene Datenelemente einen Schlüssel von Länge

  $$k = \log_{10}(n).$$

  - Dann hat Distributionsort in Abhängigkeit von der Anzahl n der Daten auch wieder die Komplexität

    $$T(n) = k*n = \log_{10}(n)*n = O(n*\log(n))$$

- In der Komplexitätstheorie berücksichtigt man allerdings auch noch die Größe (Anzahl der Bits) der einzelnen Datenelemente. Mit dieser Definition ist dann RadixSort wieder linear.

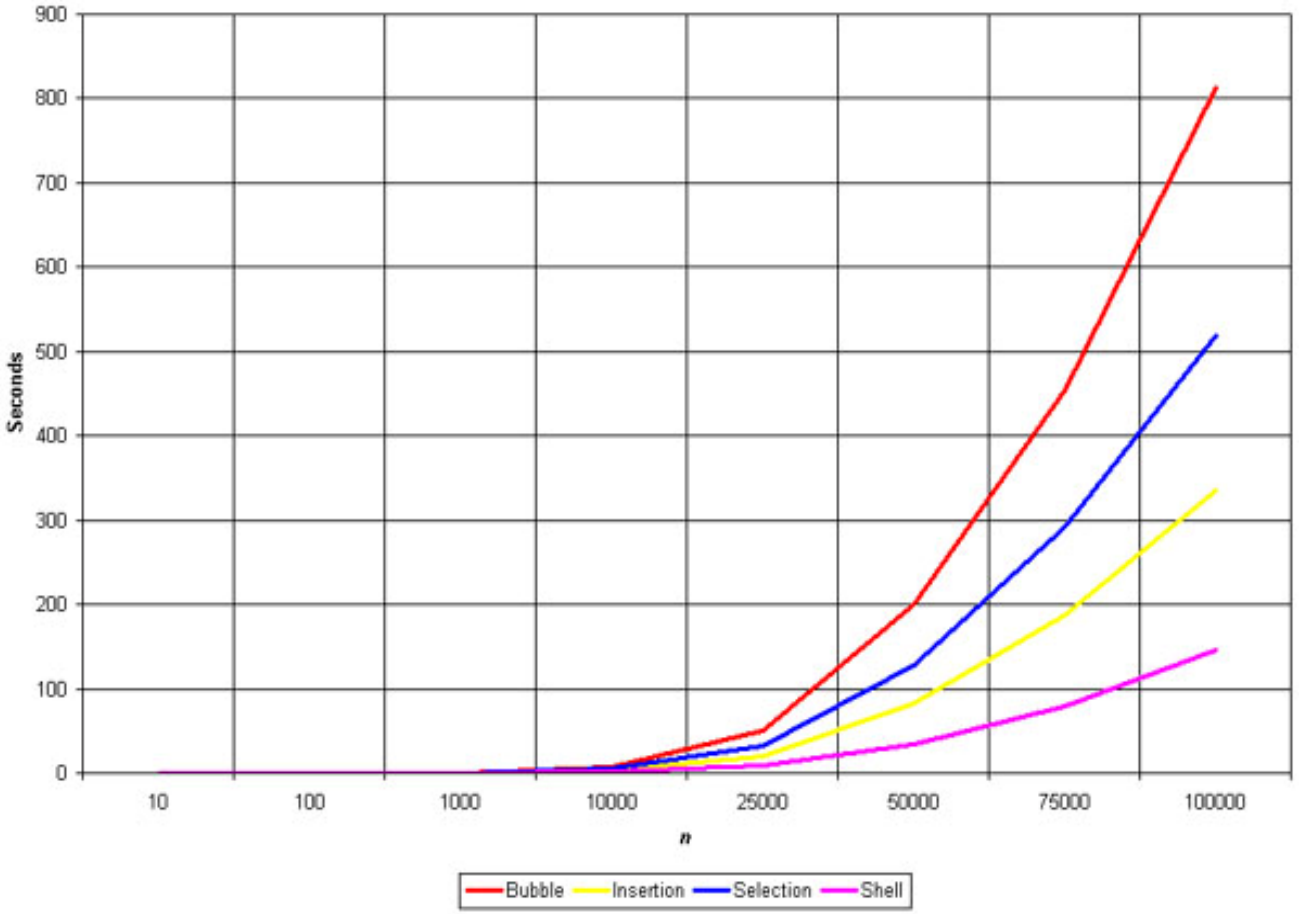
Theorie und Praxis

- Zwei Messungen mit zufällig erzeugten int-arrays
- Früher scheute man sich vor Rekursion
  - Rekursive Programme galten als ineffizient
- Man beachte wie gut sich rekursive Programme schlagen
  - mergeSort
  - quickSort
- Der Unterschied zwischen n*log(n) und n*n wird sehr deutlich
- Für kleine Datenmengen sollte man den am einfachsten zu implementierend Algorithmus verwenden

Anzahl	10000	15000	25000	50000
radixSort	78	47	63	156
mergeSort	16	31	62	63
quickSort	31	31	79	62
bubbleSort	500	985	2703	10703
selectionSort	282	391	1032	3921
insertionSort	203	438	1000	4203

Anzahl	10000	15000	25000	50000
radixSort	47	47	109	94
mergeSort	31	62	63	63
quickSort	16	15	31	47
bubbleSort	453	1000	2750	10375
selectionSort	188	375	1015	3891
insertionSort	203	359	1094	4110

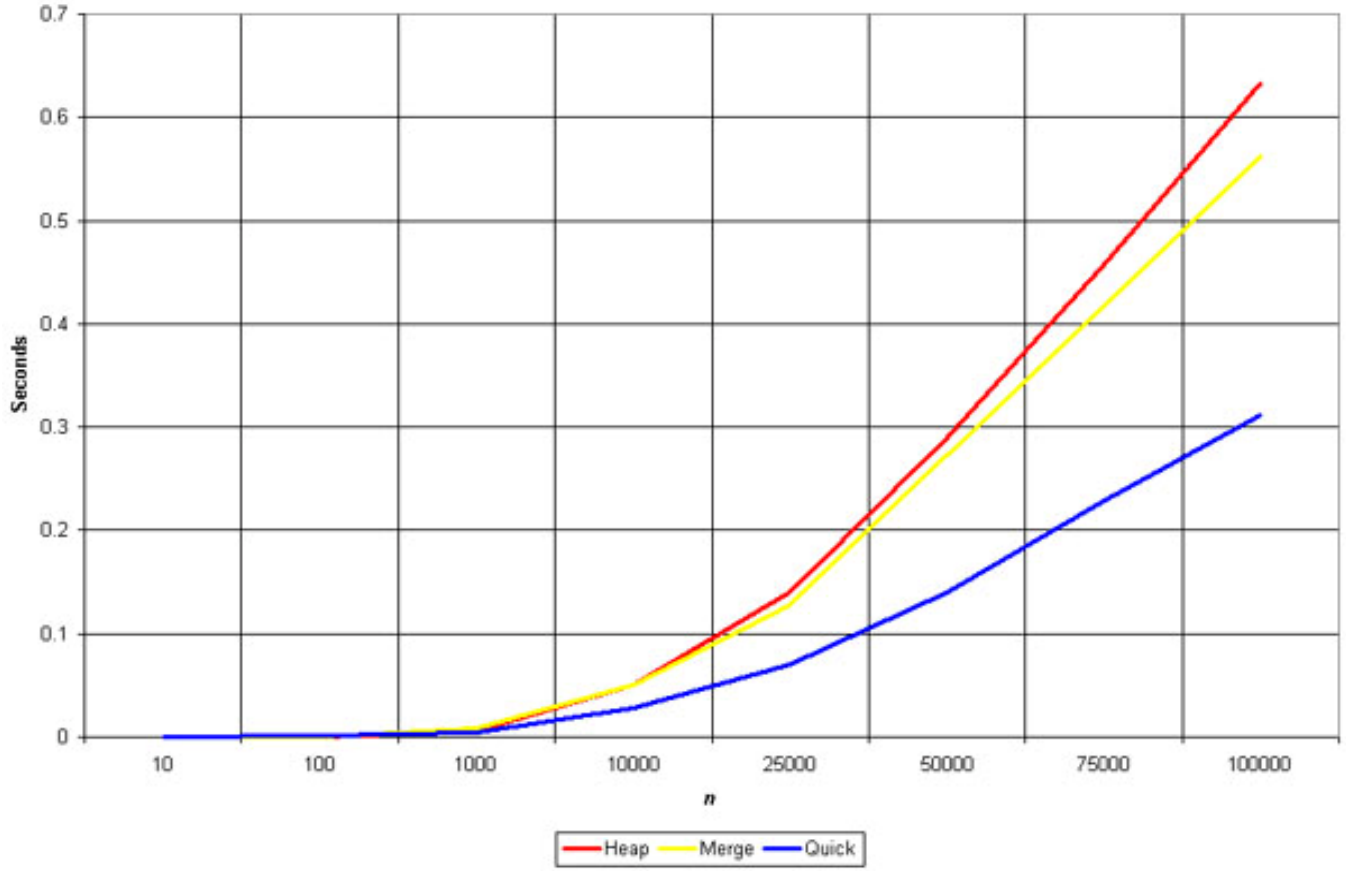
Die quadratischen Sortieralgorithmen



Quelle: http://linux.wku.edu/~lamonml/algor/sort

Die loglinearen Sortieralgorithmen

Man beachte die bzgl. der vorigen Folie geänderte Skala



Quelle: http://linux.wku.edu/~lamonml/algor/sort

Theorie und Praxis

- Komplexität beschreibt nur das asymptotische Verhalten
  - Für „kleine“ Datenmengen kann ein theoretisch schlechterer Algorithmus besser abschneiden
  - Abhängig von den Konstanten n_0, c in der Definiton von Komplexität
    - Für kleine Datenmengen kann ein quadratischer Sortieralgorithmus (bubbleSort, selectionSort) besser sein als quickSort.
    - Ab einer gewissen Größe der Daten wird quickSort garantiert schneller sein.
  - Analogie:
    - Flugzeug ist schneller als Auto
    - Für kleine Strecken kann es aber umgekehrt sein