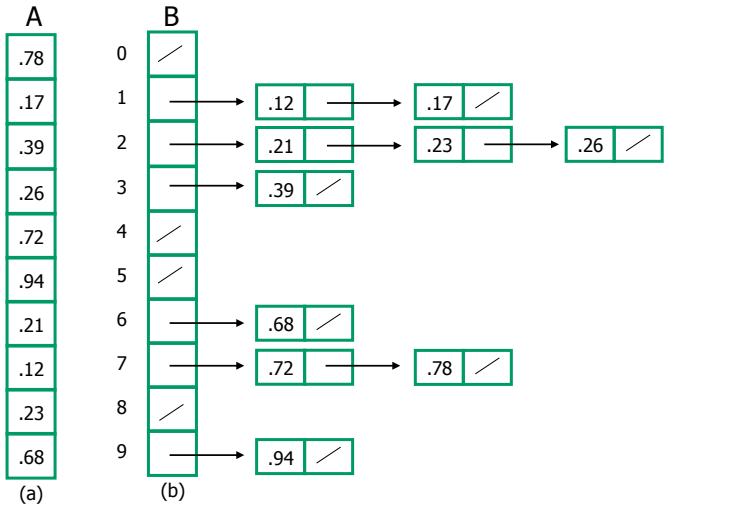
# Bucketsort

- Eingabe: Array *A* mit *n* Elementen im Bereich [0,1)
- Annahme: die Elemente sind in [0,1) gleichverteilt
  - Sonst: Skalieren ( Aufwand O(n) ), oder Algo etwas umschreiben
- Idee:
  - Teile [0, 1) in *k* gleich große *Buckets, k* konstant
  - Verteile die *n* Eingabewerte in diese *k* Buckets
  - Sortiere jedes Bucket
  - Gehe durch die Buckets der Reihe nach, hänge die Elemente an eine gesamte Liste



# Beispiel




$k = n$ Bucket *i* enthält Werte im Intervall $\left[\frac{i}{10}, \frac{i+1}{10}\right)$

## Der Algorithmus

- Eingabe: $A[0\ldots n-1]$, mit $0 \le A[i] < 1$ für alle $i$
- Hilfsarray: $B[0\ldots k-1]$ der verketteten Listen, jede am Anfang leer

```
import math
n = len(A)
B = k * [ [] ]        # array of k empty lists
for i in range(0,n):
    B[ floor(k*A[i]) ].append( A[i] )
for i in range(0,k):
    B[i].sort()       # irgendein Algo
A = []
for i in range(0,k):
    # append list B[k] to end of A
    A.extend( B[k] )
```



## Korrektheit

- Betrachte $A_i$ und $A_j$ mit $A_i \le A_j$
- Dann gilt $\lfloor k \cdot A_i \rfloor \le \lfloor k \cdot A_j \rfloor$
- Somit wird $A_i$ zu dem Bucket, in dem $A_j$ ist, oder zu einem mit kleinerem Index hinzugefügt:
  - Dasselbe Bucket → interne Sortierung liefert korrekte Reihenfolge zwischen mit $A_i$ und $A_j$
  - Ein vorheriger Bucket → nach dem Zusammenfügen der Buckets steht $A_i$ vor $A_j$

## Laufzeit

- Alle Zeilen außer der Bucket-Sortierung benötigen eine Zeitkomplexität von $O(n)$
- Wie wählt man $k$?
- Intuitiv ist klar: wähle $k=n \rightarrow$ jeder Bucket bekommt eine konstante Anzahl an Elementen, d.h., O(1) viele Elemente
- Folge: man braucht O(1) Zeit, um jedes Bucket zu sortieren $\rightarrow O(n)$ für das Sortieren aller Buckets
- Annahme scheint plausibel, aber sorgfältigere Analyse folgt



## Radix-Sort

- Vorbild: Sortieranlagen für Briefe entsprechend ihrer Postleitzahl
- Nachteile:
  - Verwendet eine konkrete Zahlen-repräsentation (typ. als Byte-Folge)
  - Verfahren muß in jedem Fall an den konkreten Sortierschlüssel angepasst werden
  - Ist also kein allgemeines Sortierverfahren

- Beobachtung: nutze aus, daß Integers zu beliebiger Basis *r* dargestellt werden können (daher der Name, "radix" = Wurzel)
- Naive (intuitive) Idee:
  - Sortiere alle Daten gemäß erster (höchstwertiger) Ziffer in Bins
  - Sortiere Bin 0 mittels Radix-Sort rekursiv
  - Sortiere Bin 1 rekursiv mittels Radix-Sort, etc. …
- Nennt man *MSD radix sort* (MSD = *most significant digit*)

- Sei im Folgenden der Radix *r* einmal fest gewählt
- Definiere $z(t,a)$ = *t*-te Stelle der Zahl *a* dargestellt zur Basis *r*, *t*=0 ist niederwertigste Stelle



## Der Algorithmus

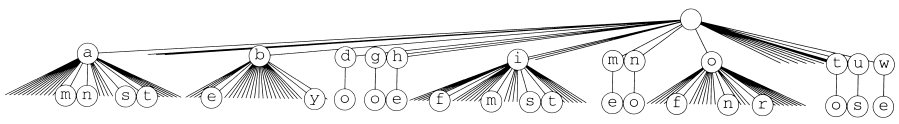
```
A = array of numbers
i = current digit used for sorting ( 0 <= i <= d-1 )
d = total number of digits (same for all keys)
def msd_radix_sort( A, i, d ):
    # init array of r empty lists = [ [], [], [], … ]
    bin = r * [[]]
    # distribute all A's in bins according to z(i,.)
    for j in range(0, len(A) ):
        bin[ z(i, A[j]) ].append( A[j] )
    # sort bins
    if i >= 0:
        for j in range(0, r):
            msd_radix_sort( bin[j], i-1, d )
    # gather bins
    A = []
    for j in range(0, r):
        A.extend( bin[j] )
        bin[j] = []
```

## Beispiel

- Keys = Integers mit 64 Bits
- Arraygröße = $2^{24}$ (ca. 16 Mio)
- Wir wählen r = $2^{16}$ (damit das Bins-Array nicht zu groß wird)
- Der Algo checkt also auf dem ersten Rekursionslevel die vorderen 16 Bits und verteilt alle Elemente auf $2^{16}$ Bins
- Durchschnittliche Arraygröße auf dem zweiten Rekursionslevel = $2^{24} / 2^{16} = 2^8 = 256$
- Radix r = $2^{16}$ ist für diese kleinen Bins völlig Overkill



- Problem: bei jeder Rekursion muß man *r*-1 viele Bins "aufbewahren"
  - Entweder mittels Marker-Array wie bei Counting-Sort
  - Oder mittels eines Arrays von Listen, wobei jeder Liste = ein Bin
- Folge: erfordert viel Zwischenspeicher, nämlich O(*d*·*r*), *d* = Anzahl Digits der Keys, *r* = Radix (= Anzahl möglicher Werte pro Digit)
- Zweites Problem:
  - Viele leere Bins
  - Tritt besonders beim Sortieren von Strings mittels Radix-Sort auf



- Lösung: sortiere zuerst nach letzter (niederwertigster) Stelle, dann nach zweitletzter, etc. (*LSD Radix-Sort* / *Backward Radix-Sort* )
- Sei *d* = Anzahl Digits, Digit 0 = niederwertigstes Digit
- Annahme (oBdA): alle Keys haben gleiche Anzahl Stellen
- Algo-Skizze:

```
def lsd_radix_sort( A ):
    for i in range(0, d):
        führe stabilen Sort auf A durch
         mit i-ter Ziffer der Elem. als Key
```

- Da Digits in [0,r-1] sind, verwende i.A. Counting-Sort



## Beispiel

- 12 Briefe anhand der Postleitzahl sortieren
- Beginne mit der letzten Ziffer

| | | | | |
|---|---|---|---|---|
| Brief | 1 | nach | 35037 | Marburg |
| Brief | 2 | nach | 71672 | Marbach |
| Brief | 3 | nach | 35288 | Wohratal |
| Brief | 4 | nach | 35282 | Rauschenberg |
| Brief | 5 | nach | 88662 | Überlingen |
| Brief | 6 | nach | 79699 | Zell |
| Brief | 7 | nach | 80638 | München |
| Brief | 8 | nach | 80637 | München |
| Brief | 9 | nach | 55128 | Mainz |
| Brief | 10 | nach | 55469 | Simmern |
| Brief | 11 | nach | 82340 | Feldafing |
| Brief | 12 | nach | 82327 | Tutzing |

*Briefe vor dem Sortieren*

| | | | | |
|---|---|---|---|---|
| Brief | 11 | nach | 82340 | Feldafing |
| Brief | 2 | nach | 71672 | Marbach |
| Brief | 4 | nach | 35282 | Rauschenberg |
| Brief | 5 | nach | 88662 | Überlingen |
| Brief | 1 | nach | 35037 | Marburg |
| Brief | 8 | nach | 80637 | München |
| Brief | 12 | nach | 82327 | Tutzing |
| Brief | 3 | nach | 35288 | Wohratal |
| Brief | 7 | nach | 80638 | München |
| Brief | 9 | nach | 55128 | Mainz |
| Brief | 6 | nach | 79699 | Zell |
| Brief | 10 | nach | 55469 | Simmern |

*Briefe sortiert nach der letzten Ziffer*

- Briefe unter Beibehaltung der Ordnung wieder zusammenlegen
- Nach vorletzter Ziffer sortieren, etc.

| | | | | |
|---|---|---|---|---|
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 5 | nach | 8866 | 2 Überlingen |
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 6 | nach | 7969 | 9 Zell |
| Brief | 10 | nach | 5546 | 9 Simmern |

*Briefe sortiert nach der letzten Ziffer*

| | | | | |
|---|---|---|---|---|
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 5 | nach | 8866 | 2 Überlingen |
| Brief | 10 | nach | 5546 | 9 Simmern |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 6 | nach | 7969 | 9 Zell |

*Briefe sortiert nach der vierten Ziffer*



| | | | | |
|---|---|---|---|---|
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 5 | nach | 8866 | 2 Überlingen |
| Brief | 10 | nach | 5546 | 9 Simmern |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 6 | nach | 7969 | 9 Zell |

*Briefe sortiert nach der vierten Ziffer*

| | | | | |
|---|---|---|---|---|
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 10 | nach | 5546 | 9 Simmern |
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 5 | nach | 8866 | 2 Überlingen |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 6 | nach | 7969 | 9 Zell |

*Briefe sortiert nach der dritten Ziffer*

| | | | | |
|---|---|---|---|---|
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 10 | nach | 5546 | 9 Simmern |
| Brief | 5 | nach | 8866 | 2 Überlingen |
| Brief | 6 | nach | 7969 | 9 Zell |

*Briefe sortiert nach der zweiten Ziffer*

| | | | | |
|---|---|---|---|---|
| Brief | 1 | nach | 3503 | 7 Marburg |
| Brief | 4 | nach | 3528 | 2 Rauschenberg |
| Brief | 3 | nach | 3528 | 8 Wohratal |
| Brief | 9 | nach | 5512 | 8 Mainz |
| Brief | 10 | nach | 5546 | 9 Simmern |
| Brief | 2 | nach | 7167 | 2 Marbach |
| Brief | 6 | nach | 7969 | 9 Zell |
| Brief | 8 | nach | 8063 | 7 München |
| Brief | 7 | nach | 8063 | 8 München |
| Brief | 12 | nach | 8232 | 7 Tutzing |
| Brief | 11 | nach | 8234 | 0 Feldafing |
| Brief | 5 | nach | 8866 | 2 Überlingen |

*Briefe sortiert nach der ersten Ziffer*

## Ausführlicher Algorithmus

- Zunächst mit Listen statt Counting-Sort (der Einfachheit halber):

```
A = array of numbers
d = number of digits
def lsd_radix_sort( A, d ):

    # init array of r empty lists = [ [], [], [], … ]
    bin = r * [[]]

    for i in range( 0, d ):

        # distribute all A's in bins according to z(i,.)
        for j in range(0, len(A) ):
            bin[ z(i, A[j]) ].append( A[j] )

        # gather bins
        A = []
        for j in range(0, r):
            A.extend( bin[j] )
            bin[j] = []
```



## Korrektheit

- Zunächst "*counter-intuitive*", daß dieser Algorithmus tatsächlich funktioniert
  - Funktioniert tatsächlich auch nur, wenn Sort im Inneren der Schleife stabil ist!
- Schleifeninvariante: Nach dem *i*-ten Durchlauf ist *A* bzgl. des Schlüssels $\langle z_i(.), \ldots, z_0(.) \rangle$ sortiert
  - Erinnerung: $z_0(.)$ liefert das LSD [least significant digit] einer Zahl

- Vor dem ersten Durchlauf: *A* ist unsortiert
- Nach dem ersten Durchlauf: *A* ist gemäß letztem Digit sortiert
- Im i-ten Durchlauf [Notation $z_i(A_j) := z(i,A_j)$] :
  - Betrachte Element $A_j$ → kommt in Bin $z_i(A_j)$
  - Für alle $A_k$, $k < j$, gilt:
    - Falls $z_i(A_k) = z_i(A_j)$ → $A_k$ steht im selben Bin wie $A_j$, aber an früherer Stelle innerhalb dieses Bins → Reihenfolge von $A_k$ & $A_j$ bzgl. $\langle z_{i-1}(.), \ldots, z_0(.)\rangle$ blieb erhalten (Ind.annahme), damit ist Reihenfolge auch bzgl $\langle z_i(.), \ldots, z_0(.)\rangle$ ok
    - $z_i(A_k) \neq z_i(A_j)$ → $A_k$ steht in anderem Bin → Reihenfolge bzgl. $\langle z_i(.), \ldots, z_0(.)\rangle$ ist sowieso korrekt
  - Nach dem Zusammenfassen der Bins ist Reihenfolge aller Schlüssel bzgl. Digits i,…,0 immer noch korrekt



## Analyse

- Keys haben *d* Digits, also *d* äußere Schleifendurchläufe
- Pro äußerem Schleifendurchlauf:
  - *Distribute*: n Elemente in A, für jedes Element A[i] wird konstanter Zeitaufwand betrieben (Digit t extrahieren, A[i] kopieren, …)
  - *Gather*: r Bins, alle Bins zusammen haben n Elemente
- Zusammen: Aufwand $O(d \cdot n)$
- Spezialfall $d = const$ (z.B. $d$ = 32-Bit Zahlen)
  → Aufwand ist linear in $n$
- Spezialfall möglichst "kurze" Keys:
  - D.h., wähle $d = \lceil \log_r(n) \rceil$
  - Aufwand = $n \log n$

## Der "wahre" Algorithmus mit Counting-Sort

- Listen zu verwalten ist sehr teuer → Counting-Sort verwenden
- Damit bekommt man:

```
A = array of numbers
d = number of digits
def lsd_radix_sort( A, d ):
    C = [0] * r     # empty array for storing bin ranges
    B = [0] * len(A) # auxiliary array for counting sort
    # now for the sorting
    for i in range( 0, d ):
        # perorm one counting sort with z(i, A[j]) as
        # sorting key and with B as auxiliary array
        A = counting_sort( A, i, B )
    return A
```



## Analyse des "wahren" Radix-Sort

- Wieder *d* äußere Schleifendurchläufe
- Jeder Counting-Sort benötigt $\Theta(n+r)$
- Zusammen: LSD-Radix-Sort hat die Laufzeit-Komplexität von

$$\Theta(d(n+r))$$

## Die optimale Wahl von *r*

- Beobachtung: wir haben die freie Wahl für $r$ → ausnutzen
- Lemma:

  Gegeben seien $n$ $b$-Bit Zahlen.
  Radix-Sort sortiert diese Zahlen in Zeit $\Theta(\frac{b}{m}(n+2^m))$
  für jedes beliebige $m \leq b$.
- Beweis: verwende Counting-Sort als stabilen Sortier-Algo
  - Verwende als "Digits" jeweils $m$ Bits, d.h., die Keys haben $m$ Bits in jedem Durchlauf, d.h. $r = 2^m$
  - Anzahl Digits $d = \frac{b}{m}$
  - Einsetzen liefert: $\Theta(d(n+r)) = \Theta(\frac{b}{m}(n+2^m))$
  - Beispiel: $b$ = 32, $m$ = 8, $r$ = 256, $d$ = 4



- Frage: für welches $m$ ($m \leq b$) wird $\frac{b}{m}(n+2^m)$ minimal?

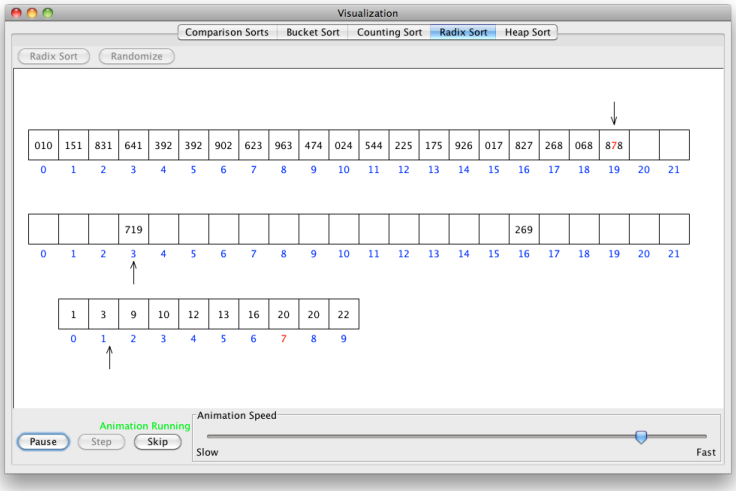
1. Fall: $b \leq \lfloor \log n \rfloor$
   - Dann gilt $\forall m \leq b : (n+2^m) \in \Theta(n)$
   - Wähle also $m$=$b$ → 1 Durchlauf (= 1x Counting Sort) reicht und ist optimal
     - Aber vielleicht nicht praktikabel …
2. Fall: $b > \lfloor \log n \rfloor$
   - Wähle $m = \lfloor \log n \rfloor$; das liefert die Laufzeit $\Theta(b\frac{n}{\log n})$
   - Behauptung: diese Wahl ist optimal
   - $2^m$ (im Zähler) wächst schneller als $m$ (im Nenner) und $\forall m \geq \log n : \frac{2^m}{m} \geq \frac{n}{\log n}$ → man sollte $m \leq \log n$ wählen
   - Wenn $m < \lfloor \log n \rfloor$ gewählt wird, bleibt $(n+2^m) \in \Theta(n)$, aber $\frac{b}{m} > \frac{b}{\log n}$

## Umsetzung

- Oftmals durch Betrachtung des Sortierschlüssels als Folge von Bytes = String (s. Beispiel 2 Folien zurück)
- 256 Sortierfächer werden benötigt
  - Bei Verwendung von Counting-Sort: Array C besteht aus 256 Zählern
- Anzahl der Durchgänge entspricht Anzahl der Bytes (Zeichen)
- Achtung: Bytesortierung muß mit Ordnung des Schlüssels übereinstimmen (kleinere Probleme bei Zweierkomplementzahlen: .. FFFE FFFF 0000 0001 ..)
- Denkaufgabe: kann man Radix-Sort für Float-Zahlen anpassen?



## Visualisierung des Algorithmus'




David Galles, Univerisity of San Francisco,
http://www.cs.usfca.edu/~galles/visualization/

## Absch(l)ießende Bemerkungen

- Lineare Verfahren sind O(N), es kann im Sinne der Komplexität keine schnelleren Verfahren geben
- Aber Achtung: die "verborgene" Konstante zählt in der Praxis!
- Verfahren muß in jedem Fall an den konkreten Sortierschlüssel angepaßt werden
- LSD-Radix-Sort ist gut geeignet für ganze Zahlen und Strings fester Länge
- MSD-Radix-Sort läßt sich besser für Strings mit variabler Länge anpassen
  - Benötigt ein paar Tricks zur effizienten Behandlung der vielen leeren Bins



## Implementierungen

- Die meisten Programmiersprachen liefern heute wenigstens einen Sortieralgorithmus in einer Library mit
- In C: `qsort()` in der C-Library (wird automatisch dazugelinkt)
- In C++: `std::sort` in der Standard-Template-Library
  - Beides hoch optimierte Varianten von Quicksort
- In Java: `Arrays.sort` in `java.util.Arrays`
- In Python: alle Listen haben eine eingebaute `sort()`-Methode

# Welcher Sortieralgorithmus ist der beste?

- Nicht leicht zu beantworten
- Wenige Datensätze (z.B. unter 100)
  - → möglichst einfachen Algorithmus verwenden ⇒ Insertion-, Selection- oder Bubblesort
  - Datenbestand bereits fast sortiert ⇒ Insertion- oder Bubblesort
- Viele Daten, zufällig angeordnet, die man häufig sortieren muß
  - Radix-Sort an das spezielle Problem anpassen (Achtung: Neuprogrammierung = Fehlerquelle)
- Will man flexibel sein und einen Standardalgorithmus verwenden
  - Scheut man nicht das Risiko, eine ungünstige Verteilung der Eingabedaten zu erwischen ⇒ Quicksort
  - Will man sicher gehen ⇒ Mergesort oder Heapsort