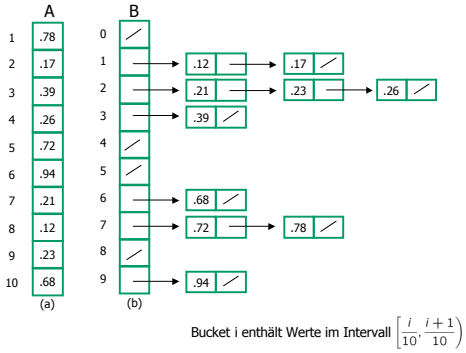
Bucketsort

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

Beispiel



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

Programm

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

```
import math
n = len(A)
B = n * [[]] # array of n empty lists
for i in range(0,n):
    B[ floor(n*A[i]) ].append( A[i] )
for i in range(0,n-1):
    B[i].sort()       # irgendein Algo
i = 0
for k in range(0,n-1):
    for j in range(0, len(B[k])):
        A[i] = B[k][j]
        i += 1
```


Korrektheit

- Betrachte A_i und A_j . Sei o.B.d.A. $A_i \le A_j$.
- Dann gilt $\lfloor n \cdot A_i \rfloor \le \lfloor n \cdot A_j \rfloor$
- Somit wird A_i zu dem Bucket, in dem A_j ist, oder zu einem mit kleinerem Index hinzugefügt:
  - derselbe Bucket → interne Sortierung
  - ein vorheriger Bucket → Zusammenfügen der Buckets

Laufzeit

- Angewiesen darauf, daß kein Bucket "zu viele Werte" beinhaltet
- Alle Zeilen außer der Bucket-Sortierung benötigen eine Zeitkomplexität von $O(n)$
- Intuitiv: wenn jeder Bucket eine konstante Anzahl an Elementen bekommt, braucht man $O(1)$ Zeit, um jeden Bucket zu sortieren → $O(n)$ für Sortieren aller Buckets
- Annahme scheint plausibel, aber sorgfältigere Analyse folgt

- n_i = Anzahl der Elemente im Bucket B_i . Insertion-Sort benötigt quadratische Zeit. Damit ist die Zeitkomplexität von Bucketsort:

$$T(n) \in \Theta(n) + \sum_{i=1}^{n} O(n_i^2)$$

- Anwendung des Erwartungswerts auf beiden Seiten und die Linearität des Erwartungswerts ergibt

$$\begin{aligned} E[T(n)] &\in E\left[\Theta(n) + \sum_{i=1}^{n} O(n_i^2)\right] \\ &\in \Theta(n) + \sum_{i=1}^{n} E\left[O(n_i^2)\right] \qquad \text{Linearität} \\ &\in \Theta(n) + \sum_{i=1}^{n} O(E\left[n_i^2\right]) \qquad (E[aX] = aE[X]) \end{aligned}$$

- Behauptung: $E\left[n_i^2\right] = 2 - \frac{1}{n}$
- Beweis:
  - $X_{ij} := \begin{cases} 1 & \text{falls } A_j \text{ in Bucket } B_i \\ 0 & \text{sonst} \end{cases}$
  - X_{ij} ist eine Zufallsvariable
  - $P\left[X_{ij} = 1\right] = \frac{1}{n}$
  - $n_i := \sum_{j=1}^{n} X_{ij}$

- $$\begin{aligned} E\left[n_i^2\right] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{k=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{\substack{k=1 \\ j\neq k}}^{n} X_{ij}X_{ik}\right] \\ &= \sum_{k=1}^{n} E\left[X_{ij}^2\right] + \sum_{j=1}^{n}\sum_{\substack{k=1 \\ j\neq k}}^{n} E\left[X_{ij}X_{ik}\right] \end{aligned}$$

- $E\left[X_{ij}^2\right] = 0^2 \cdot P\left[X_{ij}=0\right] + 1^2 \cdot P\left[X_{ij}=1\right]$
  $= 1 \cdot \frac{1}{n} = \frac{1}{n}$
- $E\left[X_{ij}X_{ik}\right]$ für $j \neq k$:
  - wegen j ≠ k sind X_{ij} und X_{ik} unabhängige Zufallsvariablen

  $E\left[X_{ij}X_{ik}\right] = E\left[X_{ij}\right] E\left[X_{ik}\right]$
  $= \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$

- Daraus folgt: $E\left[n_i^2\right] = \sum_{k=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ j \neq k}} \frac{1}{n^2}$
  $= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$
  $= 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$
- Einsetzen liefert: $E\left[T(n)\right] \in \Theta(n) + \sum_{i=1}^{n} O(2 - \frac{1}{n})$
  $\in \Theta(n) + O(n)$
  $\in \Theta(n)$

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)
- Naïve (intuitive) Idee:
  - Sortiere alle Daten gemäß erster (höchstwertiger) Ziffer in Fächer
  - Sortiere Fach 0 mittels Radix-Sort rekursiv
    - arbeite dabei auf dem Teil-Array, das Fach 0 entspricht
  - Sortiere Fach 1, etc. …
- Problem: bei jeder Rekursion muß man *r*-1 viele Fächer "aufbewahren" (mittels Marker-Array wie bei Counting-Sort)
  → erfordert rel. viel Zwischenspeicher: $O(r^d)$,
  r = Radix, d = Anzahl Stellen der Keys

- Lösung: sortiere zuerst nach letzter (niederwertigster) Stelle, dann nach zweitletzter, etc. (→ *Backward Radix-Sort*)
- Sei d Anzahl Digits, 0 = niederwertigstes Digit
- Ann. (oBdA): alle Keys haben gleiche Anzahl Stellen
- Definiere $z_r(t,a)$ = t-te Stelle der Zahl a dargestellt zur Basis r, t=0 ist niederwertigste Stelle
- Algo-Skizze:

```
def radix_sort( A ):
    for i in range(0, d):
        führe stabilen Sort auf A durch
         mit z_r(i,A[*]) als Key
```

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

Beispiel

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

Brief	1	nach	3	5	0	3	7	Marburg	Brief	11	nach	8	2	3	4	0	Feldafing
Brief	2	nach	7	1	6	7	2	Marbach	Brief	2	nach	7	1	6	7	2	Marbach
Brief	3	nach	3	5	2	8	8	Wohratal	Brief	4	nach	3	5	2	8	2	Rauschenberg
Brief	4	nach	3	5	2	8	2	Rauschenberg	Brief	5	nach	8	8	6	6	2	Überlingen
Brief	5	nach	8	8	6	6	2	Überlingen	Brief	1	nach	3	5	0	3	7	Marburg
Brief	6	nach	7	9	6	9	9	Zell	Brief	8	nach	8	0	6	3	7	München
Brief	7	nach	8	0	6	3	8	München	Brief	12	nach	8	2	3	2	7	Tutzing
Brief	8	nach	8	0	6	3	7	München	Brief	3	nach	3	5	2	8	8	Wohratal
Brief	9	nach	5	5	1	2	8	Mainz	Brief	7	nach	8	0	6	3	8	München
Brief	10	nach	5	5	4	6	9	Simmern	Brief	9	nach	5	5	1	2	8	Mainz
Brief	11	nach	8	2	3	4	0	Feldafing	Brief	6	nach	7	9	6	9	9	Zell
Brief	12	nach	8	2	3	2	7	Tutzing	Brief	10	nach	5	5	4	6	9	Simmern

Briefe vor dem Sortieren — *Briefe sortiert nach der letzten Ziffer*

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

Brief	11	nach	8	2	3	4	0	Feldafing	Brief	12	nach	8	2	3	2	7	Tutzing
Brief	2	nach	7	1	6	7	2	Marbach	Brief	9	nach	5	5	1	2	8	Mainz
Brief	4	nach	3	5	2	8	2	Rauschenberg	Brief	1	nach	3	5	0	3	7	Marburg
Brief	5	nach	8	8	6	6	2	Überlingen	Brief	8	nach	8	0	6	3	7	München
Brief	1	nach	3	5	0	3	7	Marburg	Brief	7	nach	8	0	6	3	8	München
Brief	8	nach	8	0	6	3	7	München	Brief	11	nach	8	2	3	4	0	Feldafing
Brief	12	nach	8	2	3	2	7	Tutzing	Brief	5	nach	8	8	6	6	2	Überlingen
Brief	3	nach	3	5	2	8	8	Wohratal	Brief	10	nach	5	5	4	6	9	Simmern
Brief	7	nach	8	0	6	3	8	München	Brief	2	nach	7	1	6	7	2	Marbach
Brief	9	nach	5	5	1	2	8	Mainz	Brief	4	nach	3	5	2	8	2	Rauschenberg
Brief	6	nach	7	9	6	9	9	Zell	Brief	3	nach	3	5	2	8	8	Wohratal
Brief	10	nach	5	5	4	6	9	Simmern	Brief	6	nach	7	9	6	9	9	Zell

Briefe sortiert nach der letzten Ziffer — *Briefe sortiert nach der vierten Ziffer*

Brief	12	nach	8	2	3	2	7	Tutzing	Brief	1	nach	3	5	0	3	7	Marburg
Brief	9	nach	5	5	1	2	8	Mainz	Brief	9	nach	5	5	1	2	8	Mainz
Brief	1	nach	3	5	0	3	7	Marburg	Brief	4	nach	3	5	2	8	2	Rauschenberg
Brief	8	nach	8	0	6	3	7	München	Brief	3	nach	3	5	2	8	8	Wohratal
Brief	7	nach	8	0	6	3	8	München	Brief	12	nach	8	2	3	2	7	Tutzing
Brief	11	nach	8	2	3	4	0	Feldafing	Brief	11	nach	8	2	3	4	0	Feldafing
Brief	5	nach	8	8	6	6	2	Überlingen	Brief	10	nach	5	5	4	6	9	Simmern
Brief	10	nach	5	5	4	6	9	Simmern	Brief	8	nach	8	0	6	3	7	München
Brief	2	nach	7	1	6	7	2	Marbach	Brief	7	nach	8	0	6	3	8	München
Brief	4	nach	3	5	2	8	2	Rauschenberg	Brief	5	nach	8	8	6	6	2	Überlingen
Brief	3	nach	3	5	2	8	8	Wohratal	Brief	2	nach	7	1	6	7	2	Marbach
Brief	6	nach	7	9	6	9	9	Zell	Brief	6	nach	7	9	6	9	9	Zell

Briefe sortiert nach der vierten Ziffer — *Briefe sortiert nach der dritten Ziffer*

Brief	8	nach	8	0	6	3	7	München	Brief	1	nach	3	5	0	3	7	Marburg
Brief	7	nach	8	0	6	3	8	München	Brief	4	nach	3	5	2	8	2	Rauschenberg
Brief	2	nach	7	1	6	7	2	Marbach	Brief	3	nach	3	5	2	8	8	Wohratal
Brief	12	nach	8	2	3	2	7	Tutzing	Brief	9	nach	5	5	1	2	8	Mainz
Brief	11	nach	8	2	3	4	0	Feldafing	Brief	10	nach	5	5	4	6	9	Simmern
Brief	1	nach	3	5	0	3	7	Marburg	Brief	2	nach	7	1	6	7	2	Marbach
Brief	9	nach	5	5	1	2	8	Mainz	Brief	6	nach	7	9	6	9	9	Zell
Brief	4	nach	3	5	2	8	2	Rauschenberg	Brief	8	nach	8	0	6	3	7	München
Brief	3	nach	3	5	2	8	8	Wohratal	Brief	7	nach	8	0	6	3	8	München
Brief	10	nach	5	5	4	6	9	Simmern	Brief	12	nach	8	2	3	2	7	Tutzing
Brief	5	nach	8	8	6	6	2	Überlingen	Brief	11	nach	8	2	3	4	0	Feldafing
Brief	6	nach	7	9	6	9	9	Zell	Brief	5	nach	8	8	6	6	2	Überlingen

Briefe sortiert nach der zweiten Ziffer — *Briefe sortiert nach der ersten Ziffer*

Ausführlicher Algorithmus

```
def radix_sort( A, d ):
    # init bins
    bin = r * [[]]
    # now bin = [ [], [], [], … ]
    for i in range( 0, d ):
        # distribute a[i] on bin according to z(t,.)
        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 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:
  - Element A_j kommt in *Bin* $z_i(A_j)$
  - für alle A_k, $k < j$, gilt:
    - $z(A_k) = z(A_j) \rightarrow A_k$ steht im selbem Bin wie A_j, aber an früherer Stelle innerhalb dieses Bins → Reihenfolge von A_k & A_j bzgl. $\langle z_{i-1}(.), \ldots, z_0(.)\rangle$ bleibt gewahrt, damit ist Reihenfolge auch bzgl $\langle z_i(.), \ldots, z_0(.)\rangle$ ok
    - $z(A_k) \neq z(A_j) \rightarrow A_k$ steht in anderem Bin → Reihenfolge bzgl. $\langle z_i(.), \ldots, z_0(.)\rangle$ ist sowieso korrekt, da nur i-tes Digit relevant
  - nach Zusammenfassen der Bins ist Reihenfolge aller Schlüssel bzgl. Digits i-1,…,0 immer noch korrekt

Analyse

- d viele 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: $d(n+r)$
- Spezialfall $n \ll r^d$ (z.B. d = 32-Bit Zahlen):
  - *d* und *r* konstant → Aufwand linear, d.h., $c*n$
- Spezialfall möglichst "kurze" Keys:
  - D.h., wähle $d = \lceil \log_r(n) \rceil$
  - Worst-case Aufwand: $n \log n$

Optimale Wahl von r

- Beobachtung: wir haben die freie Wahl für r → ausnutzen
- Lemma:
  Gegeben n b-Bit Zahlen.
  Radix-Sort sortiert diese Zahlen in Zeit $\Theta(\frac{b}{r}(n+2^r))$
  für jedes beliebige $r \leq b$.
- Beweis: verwende Counting-Sort als stabilen Sortier-Algo
  - Setze $d = \lceil \frac{b}{r} \rceil$, d.h., Keys pro Durchlauf haben r Bits
  - Zahlenbereich für Digits ist $k = 2^r - 1$
  - Beispiel: b = 32, r = 8, k = 255, d = 4
  - Pro Durchlauf von Counting-Sort: $\Theta(n+k)$
  - Zusammen: $\Theta(d(n+k)) = \Theta(\frac{b}{r}(n+2^r))$

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

1. Fall: $b \leq \lfloor \log n \rfloor$
   - Dann gilt $\forall r \leq b : (n+2^r) \in \Theta(n)$
   - Wähle also r=b, d.h., 1x Counting-Sort ist optimal
2. Fall: $b > \lfloor \log n \rfloor$
   - Wähle $r = \lfloor \log n \rfloor$ liefert Laufzeit $\Theta(bn/\log n)$
   - Beh.: diese Wahl ist optimal
   - Wenn $r > \lfloor \log n \rfloor$ gewählt wird, wächst 2^r im Zähler schneller als r im Nenner, ergibt längere Laufzeit
   - Wenn $r < \lfloor \log n \rfloor$ gewählt wird, bleibt $(n+2^r) \in \Theta(n)$, aber $\frac{b}{r} > \frac{b}{\log n}$

Umsetzung

- normalerweise hat man keinen dezimal ausgedrückten Schlüssel
- Lösung z.B. durch Betrachtung des Sortierschlüssels als Folge von Bytes
- 256 Sortierfächer werden benötigt
- Anzahl der Durchgänge entspricht Anzahl der Bytes
- Achtung: Bytesortierung muß mit Ordnung des Schlüssels übereinstimmen (kleinere Probleme bei Zweierkomplementzahlen: .. FFFE FFFF 0000 0001 ..)

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
- ist also kein allgemein anwendbares Sortierverfahren

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, Heapsort (evtl. auch Shellsort)

Table 2 Strengths and Weaknesses

Sort	Order	Worst Case	Memory	Stable	DataTypes	Complexity
MSD Radix	N	N	Nk + Np + R + stack	yes	strings	hi
Ternary Quick	N log N	??	Nk + Np + stack	yes	strings	hi
Quick	N log N	N^2	Nk + Np + stack	no	all	hi
Merge	N log N	N log N	Nk + 2Np + stack	yes	all	medium
Heap	N log N	N log N	Nk + Np	no	all	medium
Comb	N log N	??	Nk + Np	no	all	low
Shell	N (log N)^2	??	Nk + Np	no	all	low
Insertion	N^2	N^2	Nk + Np	yes	all	very low
Selection	N^2	N^2	Nk + Np	yes	all	very low