2.3 Sortieren

Stabilität von Sortieralgorithmen

- Ein Sortieralgorithmus heißt stabil, wenn sich die relative Reihenfolge von gleichen Elementen während des Sortierens nicht ändert.
- Beispiel:

3	1		1		1	
1	1	1			3	stabil
1	1	1			3	nicht stabil

Beispiel: Sortieren von Namen. Erst Sortieren nach Vorname, dann Nachname. Ein stabiles Verfahren behält die Reihenfolge der Vornamen bei demselben Nachnamen bei.
Ein nicht-stabiles Verfahren nicht.

Counting-Sort

- Annahme: $R_1, R_2, \ldots, R_n \in \{1,\ldots,k\}$
- Idee: Bestimme zu jedem R_i die Zahl der Elemente $\leqslant R_i$ und sortiere R_i an die entsprechende Stelle

Counting-Sort funktioniert nur, wenn man weiß, dass die zu sortierenden Elemente in einem endlichen Wertebereich liegen.

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
   B[ i ] ← 0
for i ← 1 to n do
   B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
   B[ i ] ← B[ i ] + B[ i–1 ]
for i ← n downto 1 do
   C[ B[ A[ i ] ] ] ← A[ i ]
   B[ A[ i ] ] ← B[ A[ i ] ]–1
```

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
    B[ i ] ← 0
for i ← 1 to n do
    B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
    B[ i ] ← B[ i ] + B[ i–1 ]
for i ← n downto 1 do
    C[ B[ A[ i ] ] ] ← A[ i ]
    B[ A[ i ] ] ← B[ A[ i ] ]–1
```

Eingabe: A[1..n]
Ausgabe: C[1..n]

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
    B[ i ] ← 0
for i ← 1 to n do
    B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
    B[ i ] ← B[ i ] + B[ i–1 ]
for i ← n downto 1 do
    C[ B[ A[ i ] ] ] ← A[ i ]
    B[ A[ i ] ] ← B[ A[ i ] ]–1
```

B ist mit 0en initialisiert

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
   B[ i ] ← 0
for i ← 1 to n do
   B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
   B[ i ] ← B[ i ] + B[ i−1 ]
for i ← n downto 1 do
   C[ B[ A[ i ] ] ] ← A[ i ]
   B[ A[ i ] ] ← B[ A[ i ] ]−1
```

B[j] enthält die Anzahl der Elemente = j

B ist das Array, in dem gezählt wird, wieviele Elemente es jeweils in A gibt.

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
   B[ i ] ← 0
for i ← 1 to n do
   B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
   B[ i ] ← B[ i ] + B[ i−1 ]
for i ← n downto 1 do
   C[ B[ A[ i ] ] ] ← A[ i ]
   B[ A[ i ] ] ← B[ A[ i ] ]−1
```

B[j] enthält die Anzahl der Elemente ≤ j

Counting-Sort: Algorithmus

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do
    B[ i ] ← 0
for i ← 1 to n do
    B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do
    B[ i ] ← B[ i ] + B[ i−1 ]
for i ← n downto 1 do
    C[ B[ A[ i ] ] ] ← A[ i ]
    B[ A[ i ] ] ← B[ A[ i ] ]−1
```

B[j] enthält das j-te Element

Letzte Zeile: Dekrementierung. Wieviele Elemente in A wurden schon abgearbeitet?

Counting-Sort: Beispiel

A	3	6	4	1	3	4	1	4

n=8, k=6

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
0	0	0	0	0	0

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
0	0	1	0	0	0

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
0	0	1	0	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
0	0	1	1	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
1	0	1	1	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
1	0	2	1	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	0	2	3	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	$\boxed{0}$	2	3	0	1

+

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	$\boxed{2}$	2	3	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	$\boxed{2}$	3	0	1

+

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	3	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	3	0	1

+

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	$\boxed{7}$	0	1

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	7	7	8

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	7	7	8

C

1	2	3	4	5	6	7	8

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	7	7	8

C

1	2	3	4	5	6	7	8

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	7	7	8

C

1	2	3	4	5	6	7	8
						4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	6	7	8

C

1	2	3	4	5	6	7	8
						4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
2	2	4	6	7	8

C

1	2	3	4	5	6	7	8
						4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

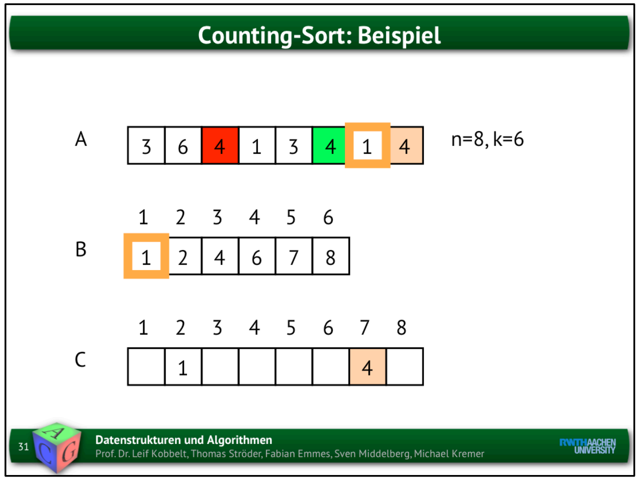
n=8, k=6

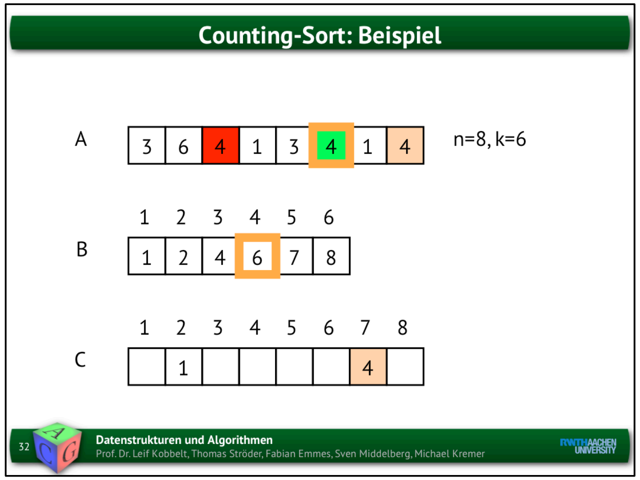
B

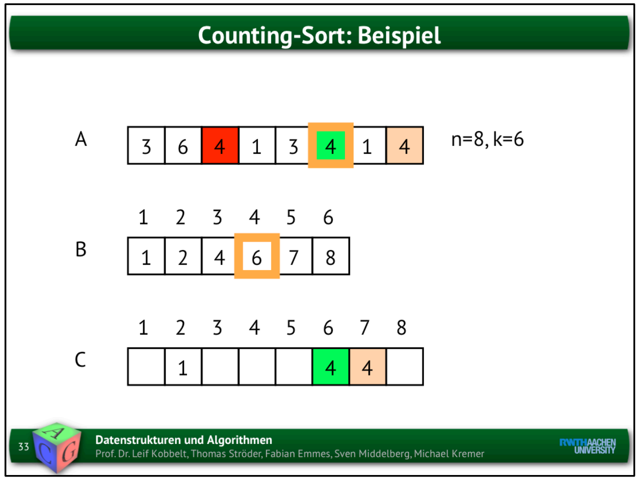
1	2	3	4	5	6
2	2	4	6	7	8

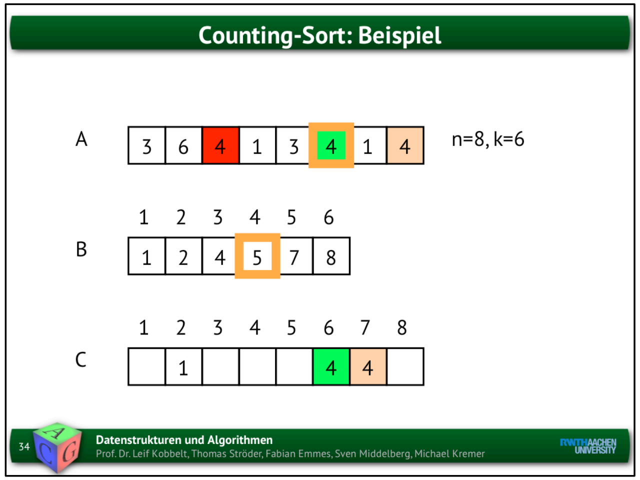
C

1	2	3	4	5	6	7	8
	1					4	


Counting-Sort: Beispiel
A
3 6 4 1 3 4 1 4
n=8, k=6
1 2 3 4 5 6
B
1 2 4 6 7 8
1 2 3 4 5 6 7 8
C
1
4
31
Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY


Counting-Sort: Beispiel
A
3 6 4 1 3 4 1 4
n=8, k=6
1 2 3 4 5 6
B
1 2 4 6 7 8
1 2 3 4 5 6 7 8
C
1
4
32
Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY


Counting-Sort: Beispiel
A
3 6 4 1 3 4 1 4
n=8, k=6
B
1 2 3 4 5 6
1 2 4 6 7 8
C
1 2 3 4 5 6 7 8
1 4 4
33
Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY


Counting-Sort: Beispiel
A
3 6 4 1 3 4 1 4
n=8, k=6
B
1 2 3 4 5 6
1 2 4 5 7 8
C
1 2 3 4 5 6 7 8
1 4 4
34
Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY

Counting-Sort: Beispiel

A n=8, k=6

3	6	4	1	3	4	1	4

B

1	2	3	4	5	6
1	2	4	5	7	8

C

1	2	3	4	5	6	7	8
	1				4	4	

Counting-Sort: Beispiel

A n=8, k=6

3	6	4	1	3	4	1	4

B

1	2	3	4	5	6
1	2	4	5	7	8

C

1	2	3	4	5	6	7	8
	1		3		4	4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
1	2	3	5	7	8

C

1	2	3	4	5	6	7	8
	1		3		4	4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
1	2	3	5	7	8

C

1	2	3	4	5	6	7	8
1	1		3		4	4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

n=8, k=6

B

1	2	3	4	5	6
0	2	3	5	7	8

C

1	2	3	4	5	6	7	8
1	1		3	4	4	4	

Counting-Sort: Beispiel

A

3	6	4	1	3	4	1	4

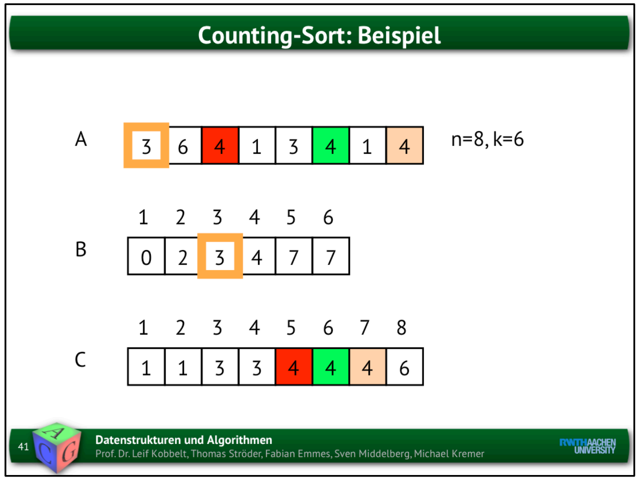
n=8, k=6

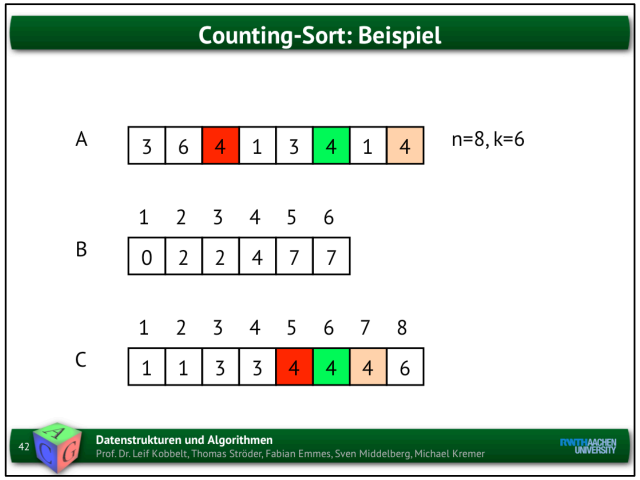
B

1	2	3	4	5	6
0	2	3	4	7	8

C

1	2	3	4	5	6	7	8
1	1		3	4	4	4	6





Nun ist die komplette Folge sortiert. Es gab vier Schleifen, die jeweils die Arrays von vorne nach hinten durchgehen, aber es werden nie zwei Elemente miteinander verglichen.

Counting-Sort: Aufwand

- CountingSort(A[1..n],C[1..n])

```
for i ← 1 to k do                        O(k)
   B[ i ] ← 0
for i ← 1 to n do                        O(n)
   B[ A[ i ] ] ← B[ A[ i ] ]+1
for i ← 2 to k do                        O(k)
   B[ i ] ← B[ i ] + B[ i–1 ]
for i ← n downto 1 do                    O(n)
   C[ B[ A[ i ] ] ] ← A[ i ]             ______
   B[ A[ i ] ] ← B[ A[ i ] ]–1           O(k+n)
```


Gesamtaufwand ist O(k + n). k ist eine Konstante und in den meisten Fällen sehr viel kleiner als n. Das heißt, dass wir das Sortierproblem für diesen Spezialfall in O(n) gelöst haben.
Dieses Sortierverfahren funktioniert natürlich nur, wenn das k bekannt und endlich ist. Wenn das k sehr groß ist, hat das Sortierverfahren keinen Effizienzvorteil mehr.

Counting-Sort: Aufwand

- Counting-Sort ist nur sinnvoll, wenn $k = O(n)$ und damit
  $T(n) = O(n)$
- Counting-Sort verwendet keine Vergleiche
- Counting-Sort ist stabil

Algorithmus ist stabil, weil die zu sortierenden Elemente von hinten nach vorne abgearbeitet werden und jeweils an die hinterste mögliche Stelle geschrieben werden. Wie ist das mit größeren Zahlenräumen?

Radix-Sort

- Annahme: $R_1, R_2, \ldots, R_n \in \{0,\ldots,k^d-1\}$
- „d-stellige Zahlen zur Basis k“
- „Wörter der Länge d aus einem Alphabet der Größe k“

Beispiel: Matrikelnummern.

Radix-Sort

- RadixSort(A[1..n])
  for i ← 1 to d do
  „sortiere A stabil nach Stelle i“

Radix-Sort: Es wird z.B. Counting-Sort mehrere Male angewendet. Wie funktioniert das? Stabilität spielt Schlüsselrolle. Beispiel: Erst nach Tausender-, dann nach Hunderter-, dann nach Zehnerstelle sortieren. Das geht nur, wenn das Sortierverfahren stabil ist.

Radix-Sort: Beispiel

F	C	T
H	L	P
N	L	P
R	F	T
H	F	N
P	C	A
F	L	L

Beispiel: Drei-Tupel eines Alphabets.

Radix-Sort: Beispiel

F	C	T
H	L	P
N	L	P
R	F	T
H	F	N
P	C	A
F	L	L

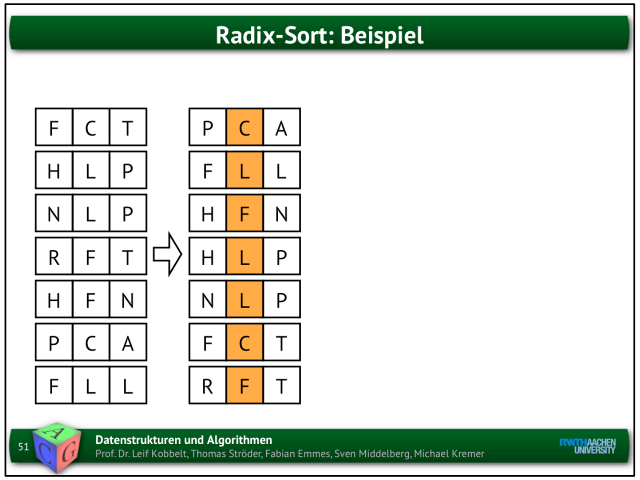
Fange an mit letzter Stelle.

Radix-Sort: Beispiel

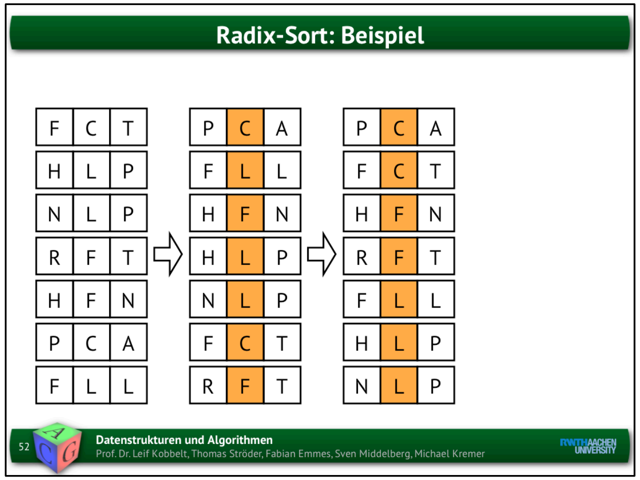
			⇨			
F	C	T		P	C	A
H	L	P		F	L	L
N	L	P		H	F	N
R	F	T	⇨	H	L	P
H	F	N		N	L	P
P	C	A		F	C	T
F	L	L		R	F	T

Radix-Sort: Beispiel

			⇨			
F	C	T		P	C	A
H	L	P		F	L	L
N	L	P		H	F	N
R	F	T	⇨	H	L	P
H	F	N		N	L	P
P	C	A		F	C	T
F	L	L		R	F	T



Dann die mittlere Stelle



Radix-Sort: Beispiel

F	C	T		P	C	A		P	C	A
H	L	P		F	L	L		F	C	T
N	L	P		H	F	N		H	F	N
R	F	T	⇒	H	L	P	⇒	R	F	T
H	F	N		N	L	P		F	L	L
P	C	A		F	C	T		H	L	P
F	L	L		R	F	T		N	L	P

Radix-Sort: Beispiel

F	C	T		P	C	A		P	C	A		F	C	T
H	L	P		F	L	L		F	C	T		F	L	L
N	L	P		H	F	N		H	F	N		H	F	N
R	F	T	⇒	H	L	P	⇒	R	F	T	⇒	H	L	P
H	F	N		N	L	P		F	L	L		N	L	P
P	C	A		F	C	T		H	L	P		P	C	A
F	L	L		R	F	T		N	L	P		R	F	T

Radix-Sort: Beispiel

F	C	T		P	C	A		P	C	A		F	C	T
H	L	P		F	L	L		F	C	T		F	L	L
N	L	P		H	F	N		H	F	N		H	F	N
R	F	T	⇨	H	L	P	⇨	R	F	T	⇨	H	L	P
H	F	N		N	L	P		F	L	L		N	L	P
P	C	A		F	C	T		H	L	P		P	C	A
F	L	L		R	F	T		N	L	P		R	F	T

Radix-Sort: Aufwand

- RadixSort(A[1..n])
  for i ← 1 to d do
  „sortiere A stabil nach Stelle i“

Radix-Sort: Aufwand

- RadixSort(A[1..n])
  for i ← 1 to d do
  „sortiere A stabil mit Counting-Sort nach Stelle i“

Radix-Sort: Aufwand

- RadixSort(A[1..n])
  for i ← 1 to d do
  „sortiere A stabil mit Counting-Sort nach Stelle i“

$T(k,d,n)=O(d\times(n+k))$

n = Anzahl der Elemente, d = Anzahl der zu sortierenden Schlüssel (Länge der Wörter), k Größe des möglichen Zahlenbereichs.

Radix-Sort: Aufwand

- Beachte: Ist k gegeben, so benötigt man zur Darstellung von n verschiedenen Elementen mindestens $d \geqslant \log_k(n)$ Stellen

$\longrightarrow$ $T(n) = \Omega(n \times \log n)$

59 Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY

Bucket-Sort

- Annahme:
  $R_1, R_2, \ldots, R_n$ gleichverteilt aus $[0,1)$
- Idee:
  1. Unterteile $[0,1)$ in n „Buckets"
     $[0,1/n), [1/n,2/n), \ldots, [(n-1)/n,1)$
  2. Füge die R_i in die Buckets ein (erwartet: ein Element pro Bucket)
  3. Sortiere die Buckets
  4. Hänge die Buckets aneinander

60 Datenstrukturen und Algorithmen
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY

Bucket-Sort hat Worst-Case-Komplexität von O(n log n), aber Average-Case-Komplexität von O(n). Dies gilt allerdings nur,
wenn die Elemente gleichverteilt sind, d.h. jedes Element statistisch gesehen gleich oft vorkommt. Erwartungswert: Ein Element pro "Eimer".

Bucket-Sort: Algorithmus

- BucketSort(A[1..n])

```
new B[0..n−1]
for i ← 1 to n do
    push(B[⌊n×A[ i ]⌋], A[ i ])
for i ← 0 to n−1 do
    sort(B[ i ])
c ← 1
for i ← 0 to n−1 do
    for j ← 1 to size(B[ i ]) do
        A[c] ← B[ i ][ j ]
        c ← c+1
```


Laufzeit linear, wenn die Größe jedes Buckets eins ist.

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6)
6 [0.6,0.7)
7 [0.7,0.8)
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8)
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	**0.73**	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3)
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	**0.24**	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0.24
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	**0.75**	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0.24
3 [0.3,0.4)
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	**0.31**	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	**0.21**	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2)
2 [0.2,0.3) → 0.24 → 0.21
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	**0.12**	0.82	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9)
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	**0.82**	0.20	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	**0.20**	0.05

n=10

0 [0.0,0.1)
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	**0.05**

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59		0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → $\boxed{}$ → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24		0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 →
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.24 → 0.21 → 0.20
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12	0.82	0.20	0.05

n=10

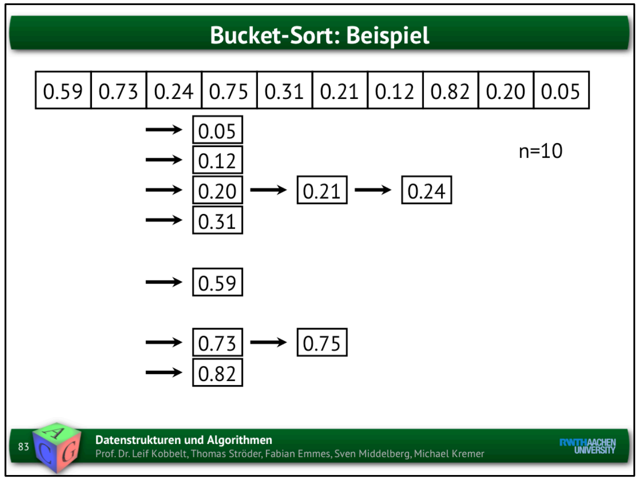
0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) → 0.82
9 [0.9,1.0)

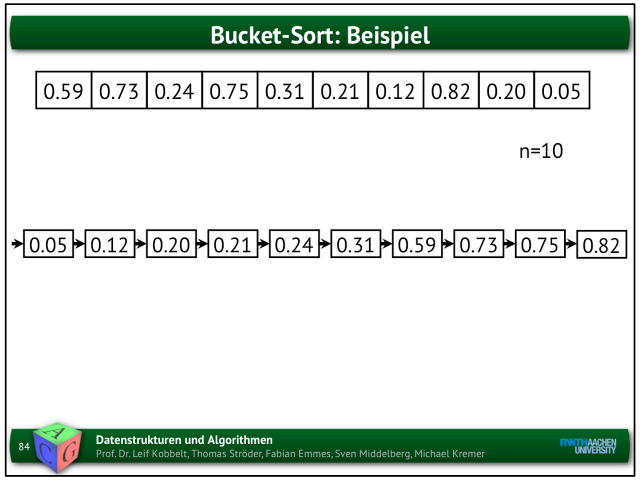
Bucket-Sort: Beispiel

0.59	0.73	0.24	0.75	0.31	0.21	0.12		0.20	0.05

n=10

0 [0.0,0.1) → 0.05
1 [0.1,0.2) → 0.12
2 [0.2,0.3) → 0.20 → 0.21 → 0.24
3 [0.3,0.4) → 0.31
4 [0.4,0.5)
5 [0.5,0.6) → 0.59
6 [0.6,0.7)
7 [0.7,0.8) → 0.73 → 0.75
8 [0.8,0.9) →
9 [0.9,1.0)





Wenn man zeigen kann, dass die größeren Buckets (mit mehreren Elementen) selten sind, ändert das an der asymptotischen Laufzeit nichts.

Bucket-Sort: Analyse

- Diskrete Zufallsvariable
  $X \in \{x_1, x_2, x_3, \ldots\}$
- Erwartungswert

$$E(X) = \mu = \sum_{i=1}^{\infty} x_i P(X = x_i)$$

- Varianz

$$\begin{aligned} V(X) = \sigma^2 &= E((X-\mu)^2) \\ &= E(X^2) - E^2(X) \end{aligned}$$

Erwartungswert: Welcher Füllgrad wird pro Eimer erwartet? Varianz: Erwartete Abweichung davon.

Bucket-Sort: Analyse

- X=1 mit 50 % Wahrscheinlichkeit
  X=3 mit 50 % Wahrscheinlichkeit
- Erwartungswert:
  E(X) = 1×0.5 + 3×0.5 = 2.0
- X=1 mit 75 % Wahrscheinlichkeit
  X=3 mit 25 % Wahrscheinlichkeit
- Erwartungswert:
  E(X) = 1×0.75 + 3×0.25 = 1.5

Bucket-Sort: Analyse

- X=1 mit 50 % Wahrscheinlichkeit
  X=3 mit 50 % Wahrscheinlichkeit
- Varianz:
  $V(X) = (1-2)^2 \times 0.5 + (3-2)^2 \times 0.5 = 1.0$
- X=1 mit 75 % Wahrscheinlichkeit
  X=3 mit 25 % Wahrscheinlichkeit
- Varianz:
  $V(X) = (1-1.5)^2 \times 0.75 + (3-1.5)^2 \times 0.25 = 0.75$

Die erwartete Abweichung vom Erwartungswert 2 (wie eben ermittelt) ist 1, da wir zwei Elemente mit den Werten 1 und 3 betrachten.

Bucket-Sort: Analyse

- Sei n_i die Zahl der Elemente in Bucket i
- Wahrscheinlichkeit, dass ein Element in Bucket i fällt ist $p=1/n$
- Wahrscheinlichkeit, dass genau k Elemente in Bucket i fallen ist

$$P(n_i = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

"Binomialverteilung"

Bucket-Sort: Analyse

- Erwartungswert

$$E(n_i) = np = 1$$

- Varianz

$$V(n_i) = np(1-p) = 1 - \frac{1}{n}$$

- Sortieraufwand, z.B. Insertion-Sort

$$T(l) = O(l^2) \Rightarrow \exists c > 0 : T(l) \leq cl^2$$

Der Erwartungswert ist 1, wenn man 1/n Eimer wählt, deren (bei einer gleichverteilten Eingabefolge) Werteintervalle gleich groß sind.

Bucket-Sort: Analyse

- Erwarteter Sortieraufwand für Bucket i

$$\begin{aligned} E(T(n_i)) &= \textstyle\sum_{k=0}^{n} T(k)P(n_i = k) \\ &= \textstyle\sum_{k=0}^{n} ck^2 P(n_i = k) \\ &= c\textstyle\sum_{k=0}^{n} k^2 P(n_i = k) \\ &= cE(n_i^2) \end{aligned}$$

Der Sortieraufwand ist quadratisch, wenn man erwartet, dass n Elemente im Bucket liegen.

Bucket-Sort: Analyse

- Erwarteter Sortieraufwand für Bucket i

$$\begin{aligned} E(T(n_i)) &= \cdots \\ &= cE(n_i^2) \\ &= c\left(V(n_i) + E^2(n_i)\right) \\ &= c\left(1 - \frac{1}{n} + 1\right) \\ &\leq 2c \\ &= O(1) \end{aligned}$$

Bucket-Sort: Analyse

- Erwarteter Sortieraufwand für alle Buckets:

$$\begin{aligned} E\left(\sum_{i=0}^{n-1} T(n_i)\right) &= \sum_{i=0}^{n-1} \underbrace{E(T(n_i))}_{O(1)} \\ &= O(n) \end{aligned}$$

Bucket-Sort: Analyse

- BucketSort(A[1..n])

```
new B[0..n−1]
for i ← 1 to n do                          O(n)
   push(B[⌊n×A[ i ]⌋], A[ i ])
for i ← 0 to n−1 do                        O(n)
   sort(B[ i ])
c ← 1
for i ← 0 to n−1 do                        O(n) erwartet
   for j ← 1 to size(B[ i ]) do            O(n) (!)
      A[c] ← B[ i ][ j ]                   ────────────
      c ← c+1                              O(n) erwartet
```


Spezielle Sortierverfahren

- Counting-Sort: O(n+k)
- Radix-Sort: O(d×(n+k))
- Bucket-Sort: O(n) erwartet

Die drei vorgestellten speziellen Sortieralgorithmen basieren alle auf der Annahme, dass die zu sortierenden Elemente alle aus einem endlichen Zahlenintervall sind.