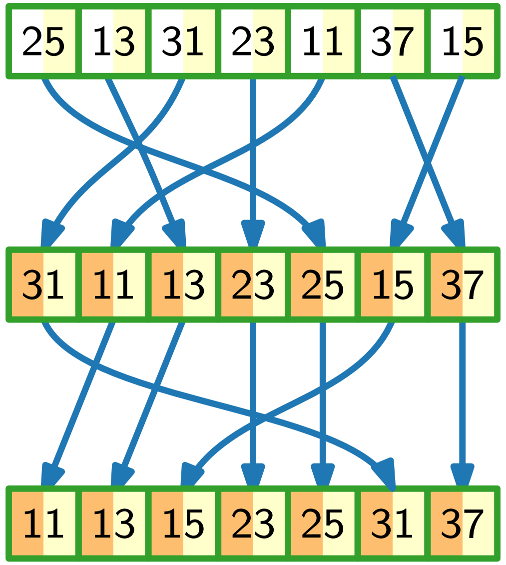
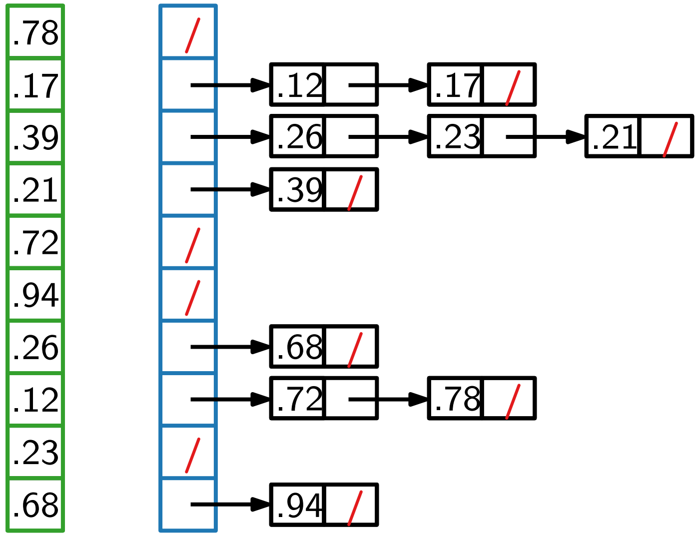
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



Vorlesung 9:
Sortieren in Linearzeit



Alexander Wolff

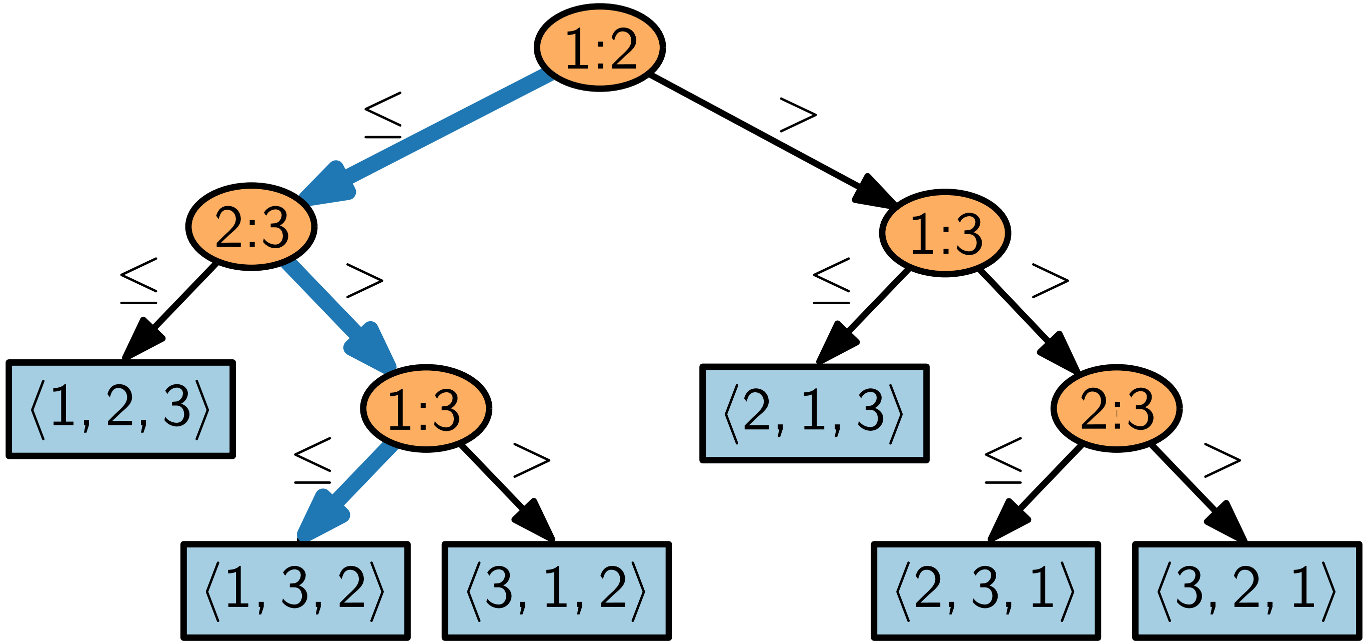
Wintersemester 2025

Sortieren durch Vergleichen

Eingabefolge $\langle a_1, a_2, \ldots, a_n \rangle$ $\xrightarrow[\text{Schlüsselvergleiche}]{\text{Sortieralg.}}$ Ausgabe: sortierte Eingabe

Für festes n ist ein **vergleichsbasierter** Sortieralgorithmus charakterisiert durch seinen **Entscheidungsbaum**:

- innere Knoten = Vergleiche (o.B.d.A. immer $\leq$, z.B. „$a_1 \leq a_2$?“)
- Blätter = sortierte Permutationen der Eingabe
- Kanten = Ergebnisse ($\leq / >$) eines Vergleichs



Entscheidungsbaum für InsertionSort und $n = 3$

Anz. Vgl. im schlechtesten Fall
= Länge eines **längsten** Wurzel-Blatt-Pfads
=: Höhe des Baums
$=_{\text{hier}}$ 3

Eine untere Schranke

Frage: Wie viele Vergleiche braucht **jeder** vergleichsbasierte Sortieralgorithmus im schlechtesten Fall um n **verschiedene** Objekte zu sortieren?

M.a.W. Gegeben

- ein beliebiger vergleichsbasierter Sortieralgorithmus,
- eine Zahl n von verschiedenen Objekten, die man sortieren soll,

welche Höhe hat der Entscheidungsbaum **mindestens**?

Beob.: Die Höhe ist eine Funktion der Blätteranzahl.

Anz. Blätter = Anz. Permutationen von n Obj. = $n!$

Höhe Binärbaum mit B Blättern $\geq \lceil \log_2 B \rceil$

Höhe Entscheidungsbaum $\geq \log_2 n! = \log_2 \prod_{i=1}^{n} i = \sum_{i=1}^{n} \log_2 i$

$$\geq \int_1^n \log_2 x \, dx = \frac{1}{\ln 2} \int_1^n \ln x \, dx = \frac{1}{\ln 2} \int_1^n 1 \cdot \ln x \, dx$$

$$= \frac{1}{\ln 2}\left([x \cdot \ln x]_1^n - \int_1^n \underbrace{x \cdot \frac{1}{x}}_{1} dx \right) = \frac{(n \ln n - 0) - (n-1)}{\ln 2} \in \Omega(n \log n)$$

Partielle Integration. $\int_a^b u'v = [uv]_a^b - \int_a^b uv'$

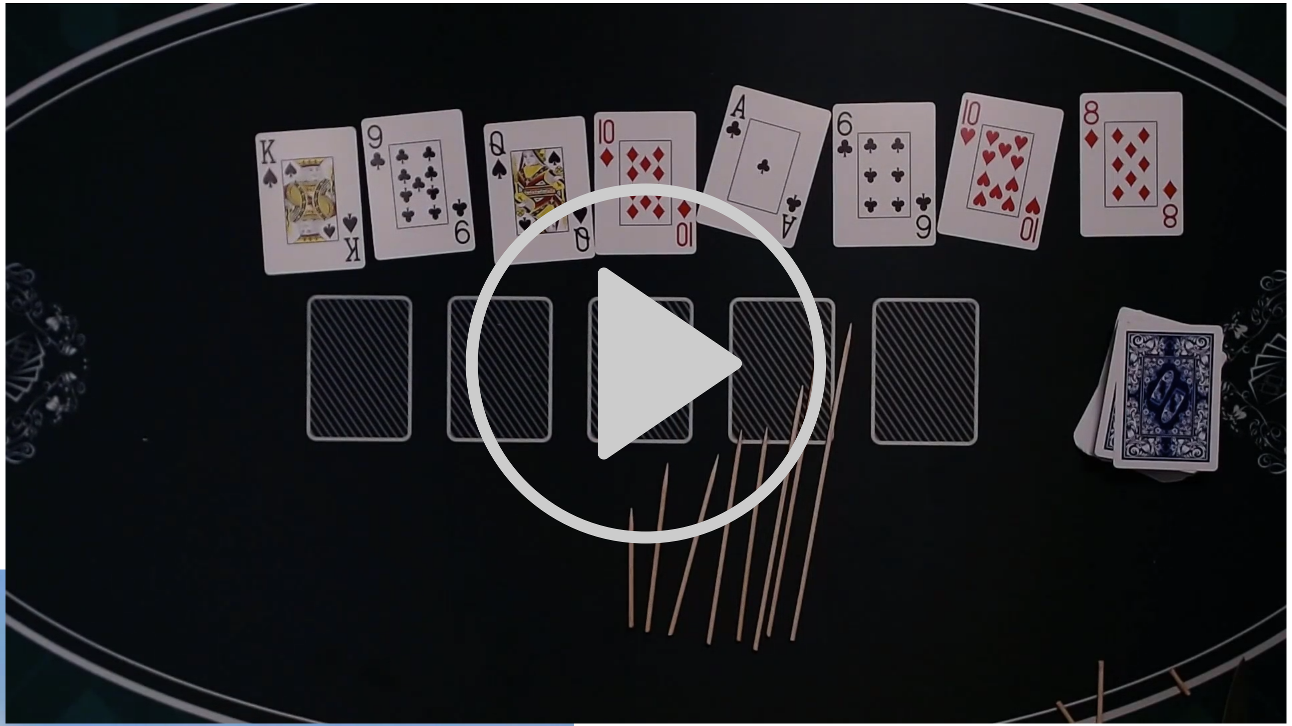
Resultat

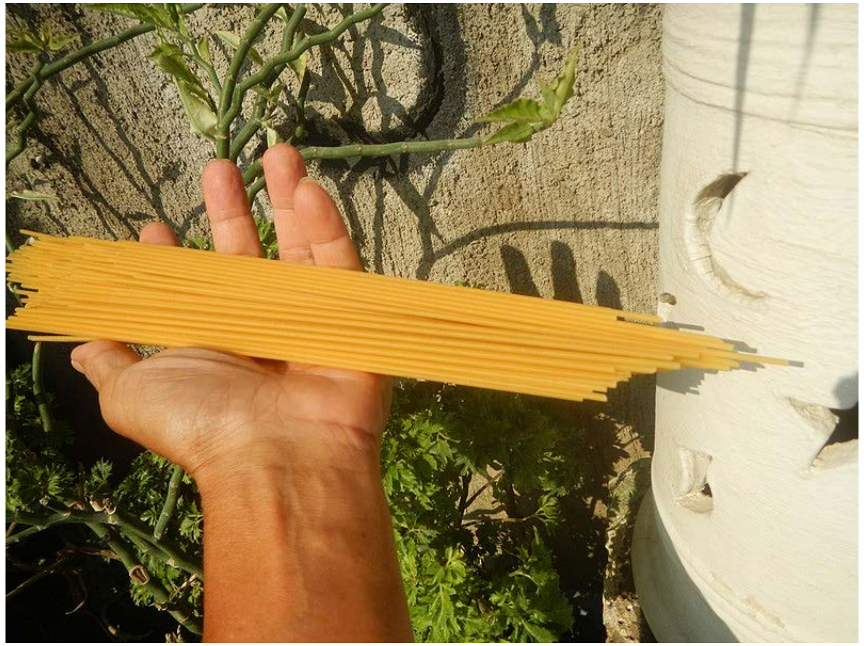
Satz. Jeder vergleichsbasierte Sortieralg. benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche um n Objekte zu sortieren.

Korollar. MergeSort und HeapSort sind **asymptotisch worst-case optimale** vergleichsbasierte Sortieralgorithmen.

Wir durchbrechen die Schallmauer

- SpaghettiSort



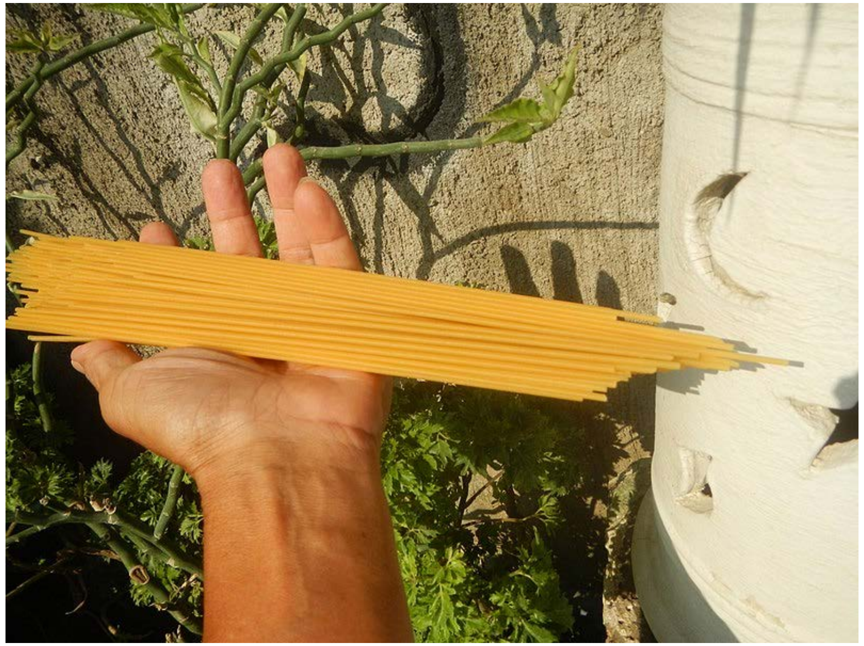


[JFVelasquez Floro, CC0, via Wikimedia Commons]

[Ensign John Gay, U.S. Navy, Public domain, via Wikimedia Commons]

Wir durchbrechen die Schallmauer

- (SpaghettiSort sortiert Spaghetti nach Länge)
- CountingSort sortiert Zahlen in $\{0, \ldots, k\}$
- RadixSort sortiert s-stellige b-adische Zahlen
- BucketSort sortiert gleichverteilte zufällige Zahlen



[JFVelasquez Floro, CC0, via Wikimedia Commons]

[Ensign John Gay, U.S. Navy, Public domain, via Wikimedia Commons]

CountingSort

Idee: 1) für jedes x in der Eingabe: **zähle** die Anzahl der Zahlen $\leq x$

2) benutze diese Information um x im Ausgabefeld direkt an die richtige Position zu **schreiben**

Variablen:

A	Eingabefeld	C	Rechenfeld
B	Ausgabefeld	k	begrenzt das **Universum:** $\{0, \ldots, k\}$

Beispiel: 1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

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

$\Rightarrow$

	0	1	2	3	4
C	1	2	0	2	3

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

	0	1	2	3	4
C	1	3			

CountingSort

Idee: 1) für jedes x in der Eingabe: **zähle** die Anzahl der Zahlen $\leq x$

2) benutze diese Information um x im Ausgabefeld direkt an die richtige Position zu **schreiben**

Variablen:

A	Eingabefeld	C	Rechenfeld
B	Ausgabefeld	k	begrenzt das **Universum:** $\{0, \ldots, k\}$

Beispiel: 1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

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

$\Rightarrow$

	0	1	2	3	4
C	1	2	0	2	3

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

	0	1	2	3	4
C	1	3	3	5	$\cancel{8}$

2) **Schreiben:** Schreibe jedes x in A direkt an die richtige Position in B

	1	2	3	4	5	6	7	8
B								4

CountingSort

Idee: 1) für jedes x in der Eingabe: **zähle** die Anzahl der Zahlen $\leq x$

2) benutze diese Information um x im Ausgabefeld direkt an die richtige Position zu **schreiben**

Variablen:

A	Eingabefeld	C	Rechenfeld
B	Ausgabefeld	k	begrenzt das **Universum:** $\{0, \ldots, k\}$

Beispiel:

1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

$$A \begin{array}{|c|c|c|c|c|c|c|c|} \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} & \multicolumn{1}{c}{7} & \multicolumn{1}{c}{8} \\ \hline 3 & 0 & 4 & 1 & 3 & 4 & 1 & 4 \\ \hline \end{array} \Rightarrow C \begin{array}{|c|c|c|c|c|} \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} \\ \hline 1 & 2 & 0 & 2 & 3 \\ \hline \end{array}$$

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

$$C \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 3 & 3 & 5 \\ \hline \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} \end{array}$$

2) **Schreiben:** Schreibe jedes x in A direkt an die richtige Position in B

$$B \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 3 & 3 & 4 & 4 & 4 \\ \hline \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} & \multicolumn{1}{c}{7} & \multicolumn{1}{c}{8} \end{array}$$

CountingSort ist **stabil!**

CountingSort

Plan: 1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

2) **Schreiben:** Schreibe jedes x in A direkt an die richtige Position in B

Eingabefeld Ausgabefeld beschränkt Universum $\{0, \ldots, k\}$

CountingSort(int[] A, int[] B, int k)

sei $C[0 \ldots k] = \langle 0, 0, \ldots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** 1a) Zählen

// $C[i]$ enthält jetzt die Anzahl der Elemente gleich i in A

for $i = 1$ **to** k **do** 1b) Berechnen

// $C[i]$ enthält jetzt die Anzahl der Elemente $\leq i$ in A

for $j = A.length$ **downto** 1 **do**

2) Schreiben

Aufgabe.

Füllen Sie die Felder mit Code, der obige Idee umsetzt!

CountingSort

Plan: 1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

2) **Schreiben:** Schreibe jedes x in A direkt an die richtige Position in B

Eingabefeld | Ausgabefeld | beschränkt Universum $\{0, \dots, k\}$

CountingSort(int[] A, int[] B, int k)

sei $C[0 \dots k] = \langle 0, 0, \dots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** $C[A[j]] = C[A[j]] + 1$ — 1a) Zählen

// $C[i]$ enthält jetzt die Anzahl der Elemente gleich i in A

for $i = 1$ **to** k **do** $C[i] = C[i] + C[i-1]$ — 1b) Berechnen

// $C[i]$ enthält jetzt die Anzahl der Elemente $\leq i$ in A

for $j = A.length$ **downto** 1 **do**

$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

Aufgabe.

Füllen Sie die Felder mit Code, der obige Idee umsetzt!

Demo.

https://algo.uni-trier.de/demos/sort.html

CountingSort

Plan: 1a) **Zählen:** Für jedes x in A, zähle die Anzahl der Zahlen gleich x (in C)

1b) **Berechnen:** Für jedes x in A, berechne die Anzahl der Zahlen $\leq x$ (in C)

2) **Schreiben:** Schreibe jedes x in A direkt an die richtige Position in B

Eingabefeld | Ausgabefeld | beschränkt Universum $\{0, \ldots, k\}$

CountingSort(int[] A, int[] B, int k)

sei $C[0 \ldots k] = \langle 0, 0, \ldots, 0 \rangle$ ein neues Feld — Initialisierung

for $j = 1$ **to** $A.length$ **do** $C[A[j]] = C[A[j]] + 1$ — 1a) Zählen

// $C[i]$ enthält jetzt die Anzahl der Elemente gleich i in A

for $i = 1$ **to** k **do** $C[i] = C[i] + C[i-1]$ — 1b) Berechnen

// $C[i]$ enthält jetzt die Anzahl der Elemente $\leq i$ in A

for $j = A.length$ **downto** 1 **do** — 2) Schreiben

    $B[C[A[j]]] = A[j]$

    $C[A[j]] = C[A[j]] - 1$

Laufzeit: $\mathcal{O}(n + k)$

1. Zwischentest am Do, 13.11., ab 8:15 Uhr

Bitte melden Sie sich jetzt in WueCampus für den Test an,
wenn Sie das noch nicht getan haben.

(Sonst drucken wir keinen Test für Sie aus. Frist: Mi, 10:00 Uhr.)

Zusammenfassung

- Jedes vergleichsbasierte Sortierverfahren braucht im Worst-Case $\Omega(n \log n)$ Vergleiche.

- COUNTINGSORT sortiert Zahlen in $\{0, \ldots, k\}$ **(stabil!)**
  Laufzeit für n Zahlen: $\mathcal{O}(n + k)$

- RADIXSORT sortiert s-stellige b-adische Zahlen

  max. s Stellen

  b mögliche unterschiedliche Ziffern

  z.B. Dezimalzahl: $b = 10$
  Binärzahl: $b = 2$
  Wörter: $b = 26$

- BUCKETSORT sortiert gleichverteilte zufällige Zahlen

RadixSort

(Jahr, Monat, Tag)

Frage: Gegeben Liste von Menschen mit deren Geburtstagen.
Wie würden Sie die Liste nach Alter sortieren?

Drei (?) Lösungen:

- Geburtstage in Anzahl Tage seit 1.1.1970 umrechnen, dann vergleichsbasiertes Sortierverfahren verwenden.
- Spezielle Vergleichsroutine schreiben und in vergleichsbasiertes Sortierverfahren einbauen.
- Liste 3× sortieren: je 1× nach Jahr, Monat, Tag. Aber in welcher Reihenfolge?

Anz. Stellen (hier: 3)

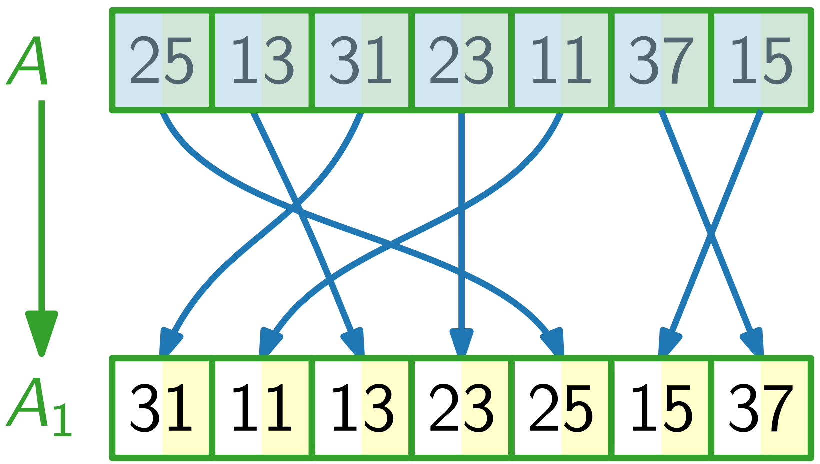
RadixSort(A, s)

for $i = 1$ **to** s **do** 1 = Index der **niederwertigsten** (!) Stelle
 sortiere A **stabil** nach der i-ten Stelle

z.B. mit CountingSort

Beispiel

Sortiere



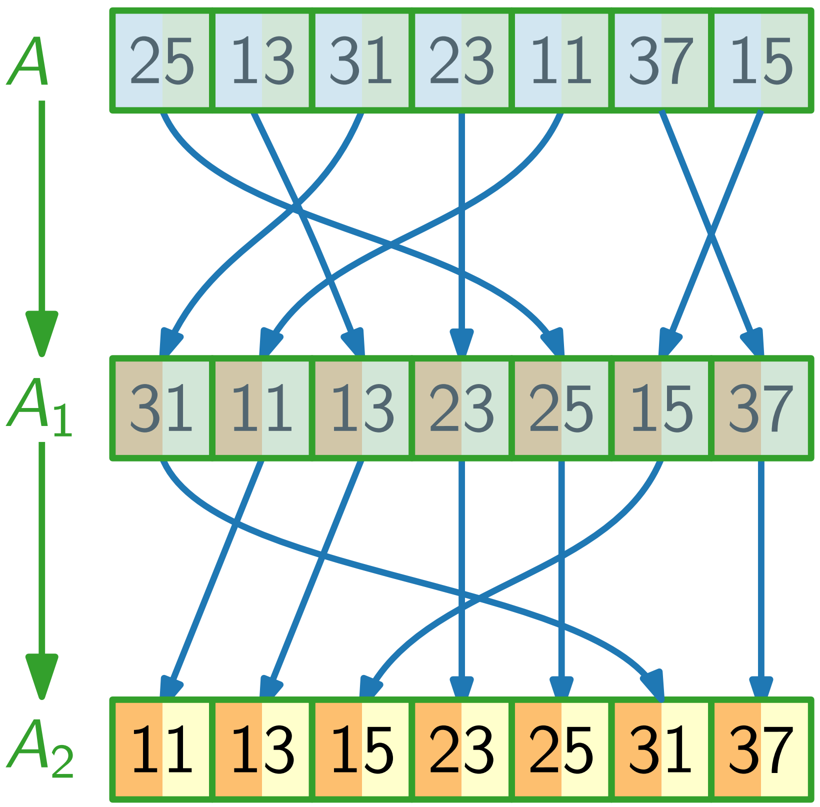
Gemäß RadixSort erst nach Einern, dann (stabil) nach Zehnern.

RadixSort(A, s)

for $i = 1$ **to** s **do**
  sortiere A **stabil** nach der i-ten Stelle

Beispiel

Sortiere



Gemäß RadixSort erst nach Einern, dann (stabil) nach Zehnern.

Demo.
`https://algo.uni-trier.de/demos/sort.html`

✓

RadixSort(A, s)

for $i = 1$ **to** s **do**
 sortiere A **stabil** nach der i-ten Stelle

Zusammenfassung

- Jedes vergleichsbasierte Sortierverfahren braucht im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

- CountingSort sortiert Zahlen in $\{0, \ldots, k\}$ **(stabil!)**

  Laufzeit für n Zahlen: $\mathcal{O}(n + k)$

- RadixSort sortiert s-stellige b-adische Zahlen

  max. s Stellen

  b mögliche unterschiedliche Ziffern

  z.B. Dezimalzahl: $b = 10$
  Binärzahl: $b = 2$
  Wörter: $b = 26$

  Laufzeit für n Zahlen: $\mathcal{O}(s \cdot (n + b))$

- BucketSort sortiert gleichverteilte zufällige Zahlen

BucketSort

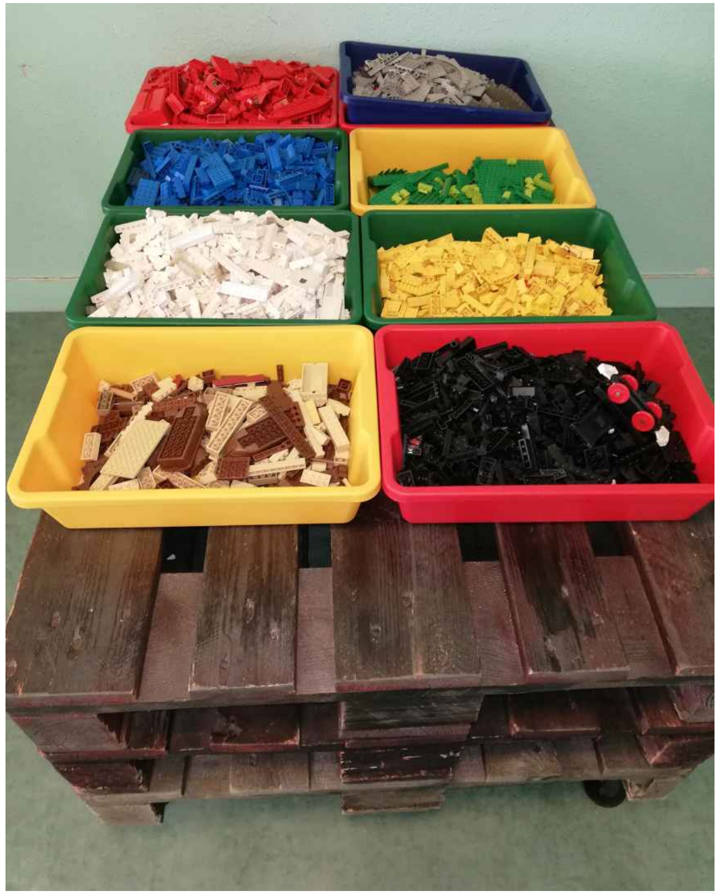
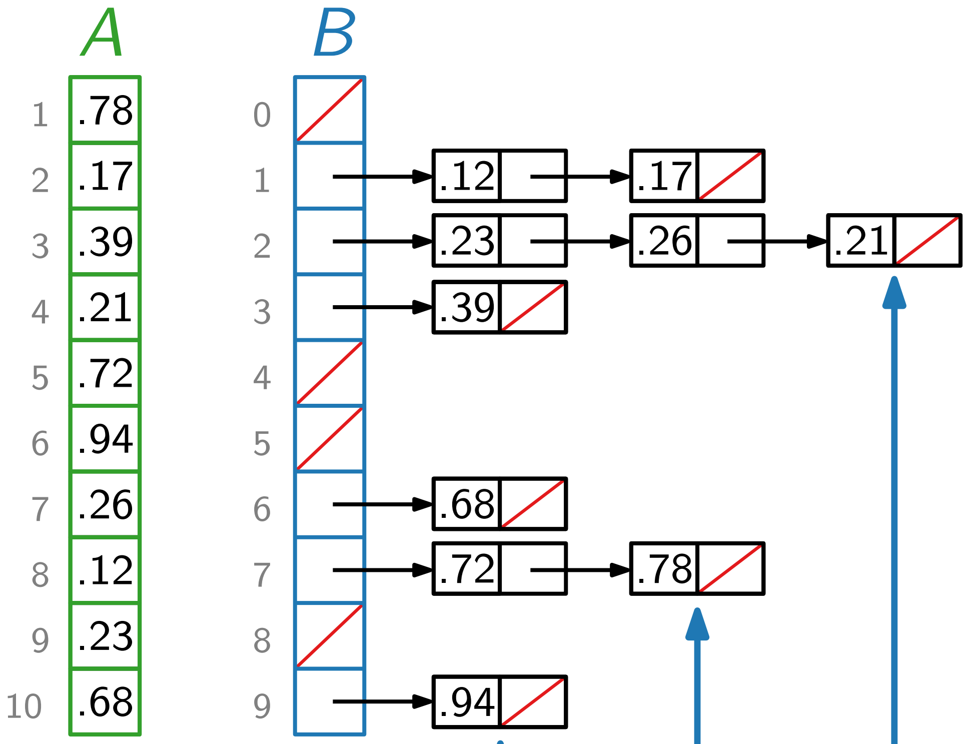


photo by JulienFou on reddit

BucketSort



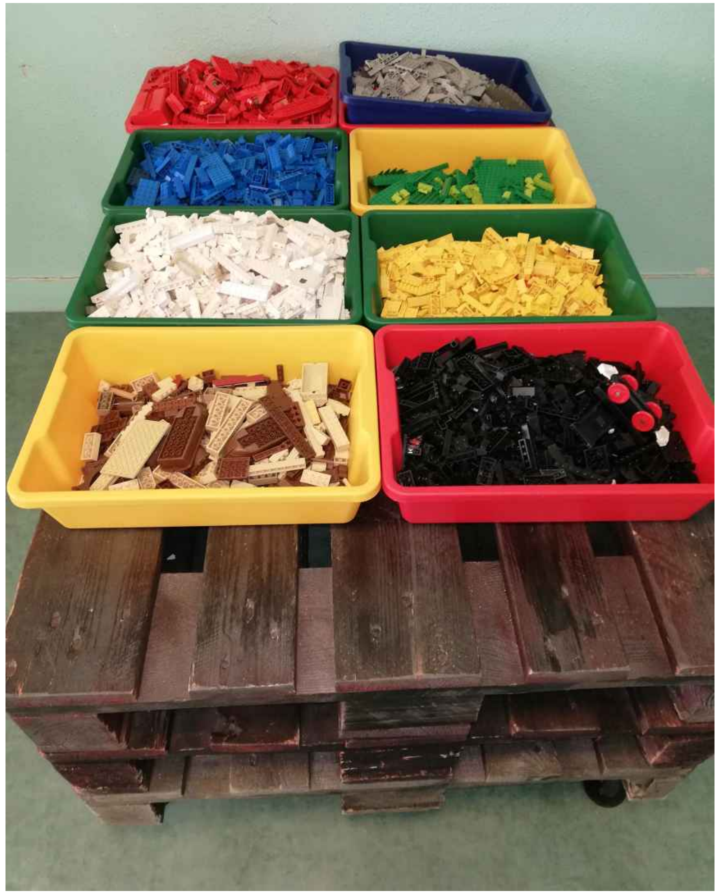


photo by JulienFou on reddit

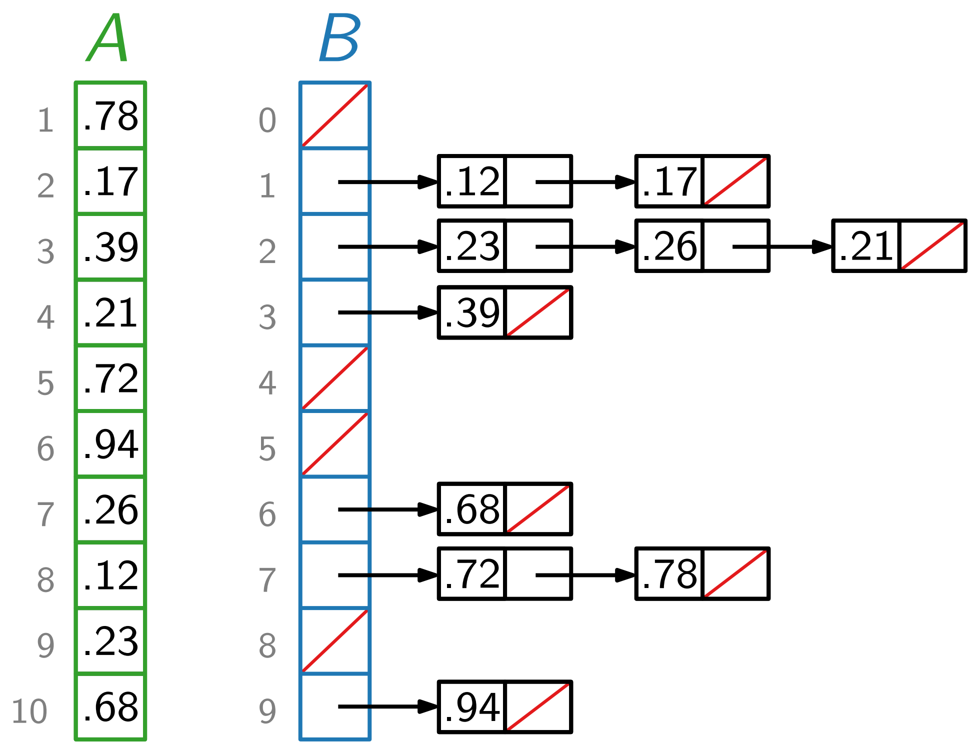
„Eimerinhalt“: **Verkettete Liste** von Elementen aus A.

Hilfsfeld $B[0 \dots n-1]$;
jeder Eintrag entspricht einem „Eimer“ der Weite $1/n$

Eingabefeld $A[1 \dots n]$ enthält Zahlen,
zufällig und gleichverteilt aus $[0, 1)$ gezogen

Im Beispielt auf 2 Nachkommastellen gerundet

BucketSort



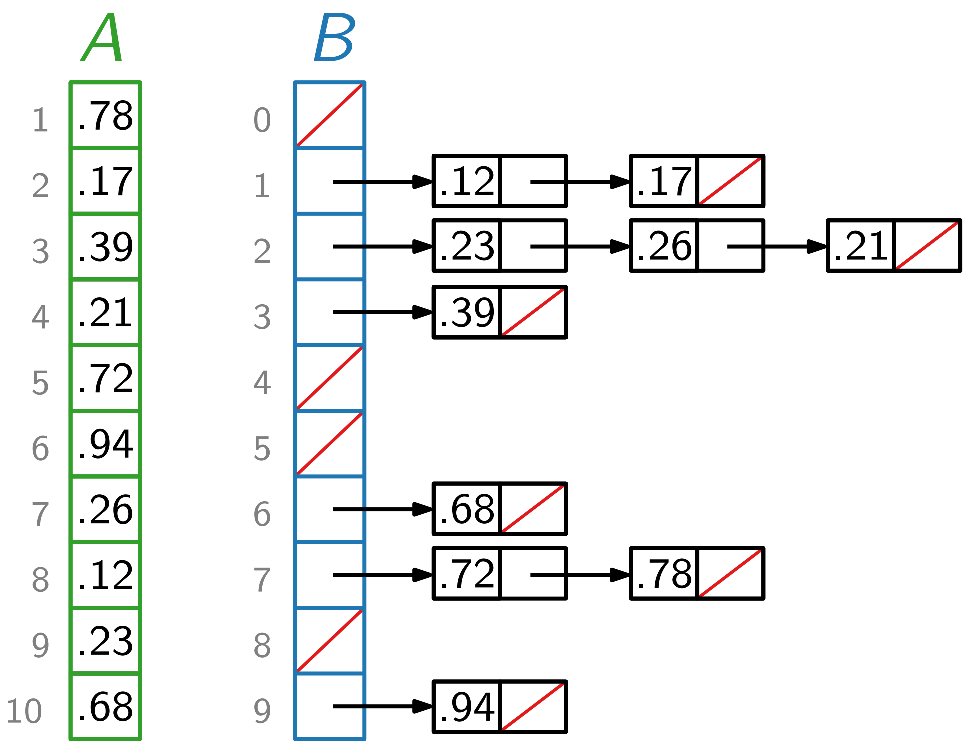
BucketSort(Feld A von Zahlen in $[0,1)$)

$n = A.length$
lege Feld $B[0 \ldots n-1]$ von Listen an
for $j = 1$ **to** n **do**
  füge $A[j]$ in Liste $B[\quad]$ ein
for $i = 0$ **to** $n-1$ **do**
  sortiere Liste $B[i]$

Aufgabe:

Füllen Sie die Felder mit Code, der BucketSort umsetzt!

BucketSort



BucketSort(Feld A von Zahlen in $[0,1)$)

$n = A.length$
lege Feld $B[0 \dots n-1]$ von Listen an
for $j = 1$ **to** n **do**
  füge $A[j]$ in Liste $B[\lfloor n \cdot A[j] \rfloor]$ ein
for $i = 0$ **to** $n-1$ **do**
  sortiere Liste $B[i]$
hänge $B[0], \dots, B[n-1]$ aneinander
kopiere das Ergebnis nach $A[1 \dots n]$

$= [\frac{i}{n}, \frac{i+1}{n}) \cap A$

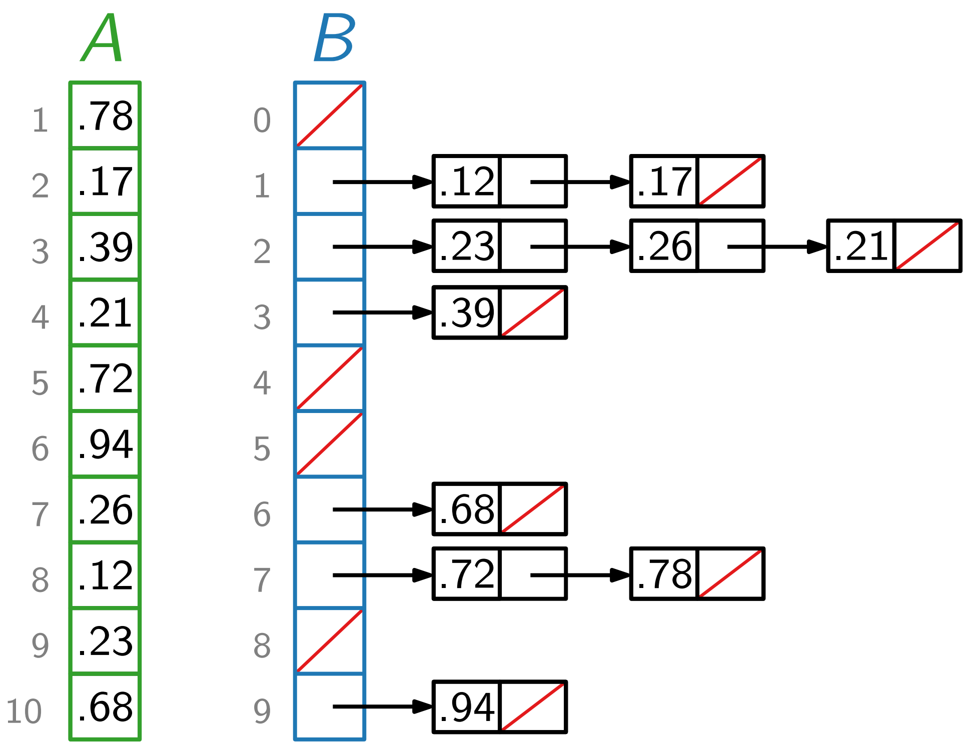
Korrektheit? 2 Fälle:

- $A[i]$ und $A[j]$ in der gleichen Liste
- $A[i]$ und $A[j]$ in verschiedenen Listen

Laufzeit?

- **erwartet**, hängt von den zufälligen Zahlen in A ab
- hängt vom Sortieralgorithmus in Zeile 6 ab;
  wir nehmen InsertionSort: schnell auf kurzen Listen!

BucketSort



BucketSort(Feld A von Zahlen in $[0, 1)$)

$n = A.length$
lege Feld $B[0 \ldots n-1]$ von Listen an
for $j = 1$ **to** n **do**
  füge $A[j]$ in Liste $B[\lfloor n \cdot A[j] \rfloor]$ ein
for $i = 0$ **to** $n-1$ **do**
  sortiere Liste $B[i]$
hänge $B[0], \ldots, B[n-1]$ aneinander
kopiere das Ergebnis nach $A[1 \ldots n]$

Korrektheit? 2 Fälle:

- $A[i]$ und $A[j]$ in der gleichen Liste
- $A[i]$ und $A[j]$ in verschiedenen Listen

Laufzeit? $T_{\mathrm{BS}}(n) = \Theta(n) + \sum_{i=0}^{n-1} T_{\mathrm{IS}}(n_i)$,

wobei $T_{\mathrm{IS}}(\cdot)$ Laufzeit von InsertionSort
n_i Zufallsvariable für Länge der Liste $B[i]$

Erwartete Laufzeit von BucketSort

$$T_{\mathrm{BS}}(n) = \Theta(n) + \sum_{i=0}^{n-1} T_{\mathrm{IS}}(n_i) \;=\; \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)$$

$$E[T_{\mathrm{BS}}(n)] = E[\Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(n_i^2)]$$

Linearität des Erwartungswerts: $E[X+Y] = E[X] + E[Y]$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[\mathcal{O}(n_i^2)]$$

Für $a \in \mathbb{R}$: $E[aX] = a \cdot E[X]$

$$= \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(E[n_i^2]) \;=\; \Theta(n)$$

Behauptung: $E[n_i^2] \;\leq\; 2 - \frac{1}{n}$

fest!

Beweis. Definiere Indikator-Zufallsvariable $X_j := 1$, falls $A[j]$ in Eimer i fällt.

$$\Rightarrow\; n_i \;=\; \sum_{j=1}^{n} X_j \qquad E[X_j] = \Pr[X_j = 1] = 1/n$$

$$\Rightarrow\; n_i^2 = \left(\sum_{j=1}^{n} X_j\right)^2 = \sum_{j=1}^{n}\sum_{k=1}^{n} X_j X_k$$

$$= \sum_{j=1}^{n} X_j^2 + \sum_{j=1}^{n}\sum_{k\neq j} X_j X_k$$

Erwartete Laufzeit von BucketSort

Es gilt $n_i^2 = \sum_{j=1}^n X_j^2 + \sum_{j=1}^n \sum_{k \neq j} X_j X_k$

$$\Rightarrow E[n_i^2] = \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k]$$

Behauptung:
$E[n_i^2] \leq 2 - \frac{1}{n}$

Behandle die beiden Typen von Erwartungswerten getrennt:

$$E[X_j^2] = 1 \cdot \Pr[X_j^2 = 1] + 0 \cdot \Pr[X_j^2 = 0]$$
$$= 1 \cdot \Pr[X_j = 1] + 0 \cdot \Pr[X_j = 0] = 1 \cdot \frac{1}{n} + 0 = \frac{1}{n}$$

unabhängig von j

$$E[X_j X_k] = E[X_j] \cdot E[X_k] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

unabhängig von j und k

für $j \neq k$ sind X_j und X_k **unabhängig**

Fasse die Zwischenergebnisse zusammen:

$$E[n_i^2] = \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k]$$
$$= n \cdot \frac{1}{n} + n \cdot (n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \qquad \square$$

Zusammenfassung

- Jedes vergleichsbasierte Sortierverfahren braucht im Worst-Case $\Omega(n \log n)$ Vergleiche.

- CountingSort sortiert Zahlen in $\{0, \ldots, k\}$. (**stabil!**)
  Laufzeit für n Zahlen: $\mathcal{O}(n + k)$

- RadixSort sortiert s-stellige b-adische Zahlen.
  Laufzeit für n Zahlen: $\mathcal{O}(s \cdot (n + b))$

- BucketSort sortiert gleichverteilte zufällige Zahlen.
  Erwartete Laufzeit für n Zahlen: $\mathcal{O}(n)$

Bemerkung. Die Idee mit den (gleichgroßen) Eimern ist natürlich nicht nur auf Zufallszahlen beschränkt, aber hier lässt sie sich hübsch analysieren.

Vergleich Sortieralgorithmen

	Bester Fall	Erw. Fall	Schl. Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✗
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	✓	✗
CountingSort	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	✗	✓
RadixSort	$\mathcal{O}(s \cdot (n + b))$	$\mathcal{O}(s \cdot (n + b))$	$\mathcal{O}(s \cdot (n + b))$	✗	✓
BucketSort	$\mathcal{O}(n)$	$\mathcal{O}(n)$ wenn Eingabe zufällig und gleichverteilt	$\mathcal{O}(n^2)$	✗	✓ wenn verwendeter Sortieralg. stabil