Algorithmen und Datenstrukturen

TECHNISCHE UNIVERSITÄT DARMSTADT

vi visual inference SYSTEMS

Stefan Roth, SS 2025

02

Sortieren

Folien beruhen auf der Veranstaltung von Prof. Marc Fischlin und Christian Janson aus dem SS 2024

Laufzeitanalyse Quicksort: Worst-Case, Average-Case und erwartete Laufzeit

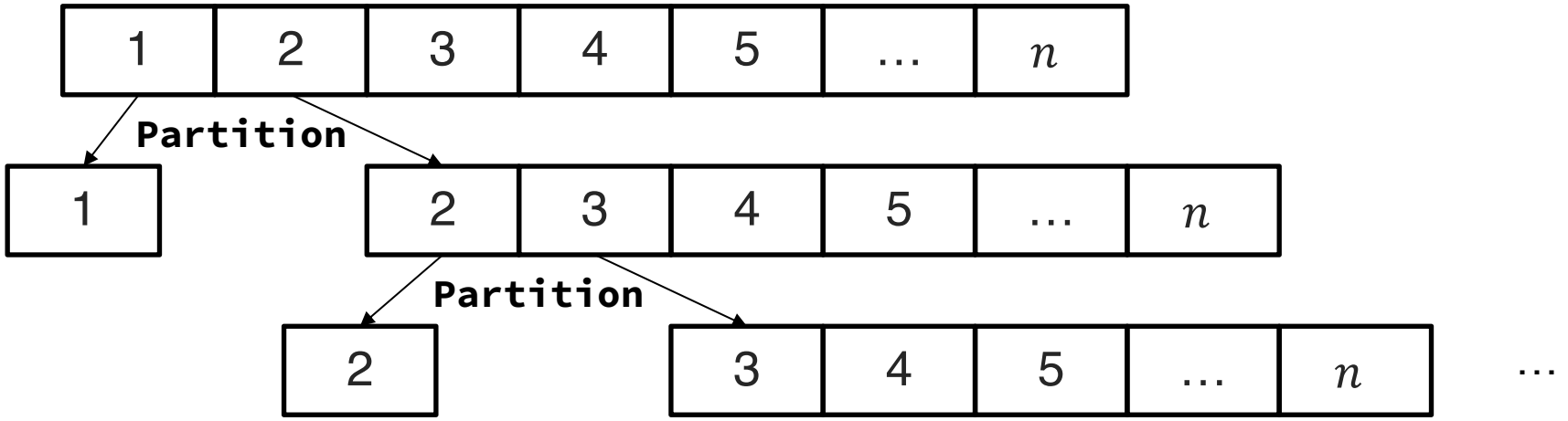
Laufzeit im Worst-Case: Untere Schranke

```
quicksort(A,left,right) //initial left=0,right=A.length-1

1  IF left<right THEN  //more than one el
2    q=partition(A,left,right);   // q pa
3    quicksort(A,left,q);         // sort
4    quicksort(A,q+1,right);      // sort right part
```

$(n-1)$-mal **Partition** ergibt Quicksort Gesamtlaufzeit $\Omega(n^2)$

Ungünstiges Array: **Partition** spaltet immer nur ein Element ab (Bsp #2)



Laufzeit im Worst-Case: Obere Schranke (I)

```
quicksort(A,left,right) //initial left=0,right=A.length-1

1  IF left<right THEN  //more than one element
2    q=partition(A,left,right);   // q partition index
3    quicksort(A,left,q);         // sort left part
4    quicksort(A,q+1,right);      // sort right part
```

Intuition: nur ein Element abzuspalten ist auch schlechtester Fall und daher

$$T(n) \leq T(n-1) + dn$$

Aufwand für $T(1)$, **IF** und ggf. **Partition**

$$\leq T(n-2) + d(n-1) + dn$$

$$\vdots$$

$$\leq \sum_{i=1}^{n} di = \mathcal{O}(n^2)$$

Laufzeit im Worst-Case: Obere Schranke (II)

```
quicksort(A,left,right) //initial left=0,right=A.length-1

1  IF left<right THEN  //more than one el
2    q=partition(A,left,right);   // q pa
3    quicksort(A,left,q);         // sort
4    quicksort(A,q+1,right);      // sort right part
```

Quicksort
(Worst-Case-)Laufzeit
$\Theta(n^2)$

Formal per Induktion: Behauptung $T(n) \leq dn^2$

Basisfall: gilt für $n = 1$, da $T(1) \leq dn$

i **nicht-trivialer Partitionsindex (daher Induktion anwendbar)**

Induktions-schritt:

$$
\begin{aligned}
T(n) &\leq \max_{i=1,\ldots,n-1} \big(T(n-i) + T(i)\big) + dn \\
&\leq \max_{i=1,\ldots,n-1} (d(n-i)^2 + di^2) + dn \\
&\leq d(n-1)^2 + d + dn \\
&\leq dn^2 - 2dn + d + d + dn \\
&\leq dn^2 \quad \text{für } n \geq 2
\end{aligned}
$$

maximal für $i = 1$

Best-Case für Quicksort (I)

Im besten Fall Aufteilung in gleichgroße Arrays wie bei Merge Sort:

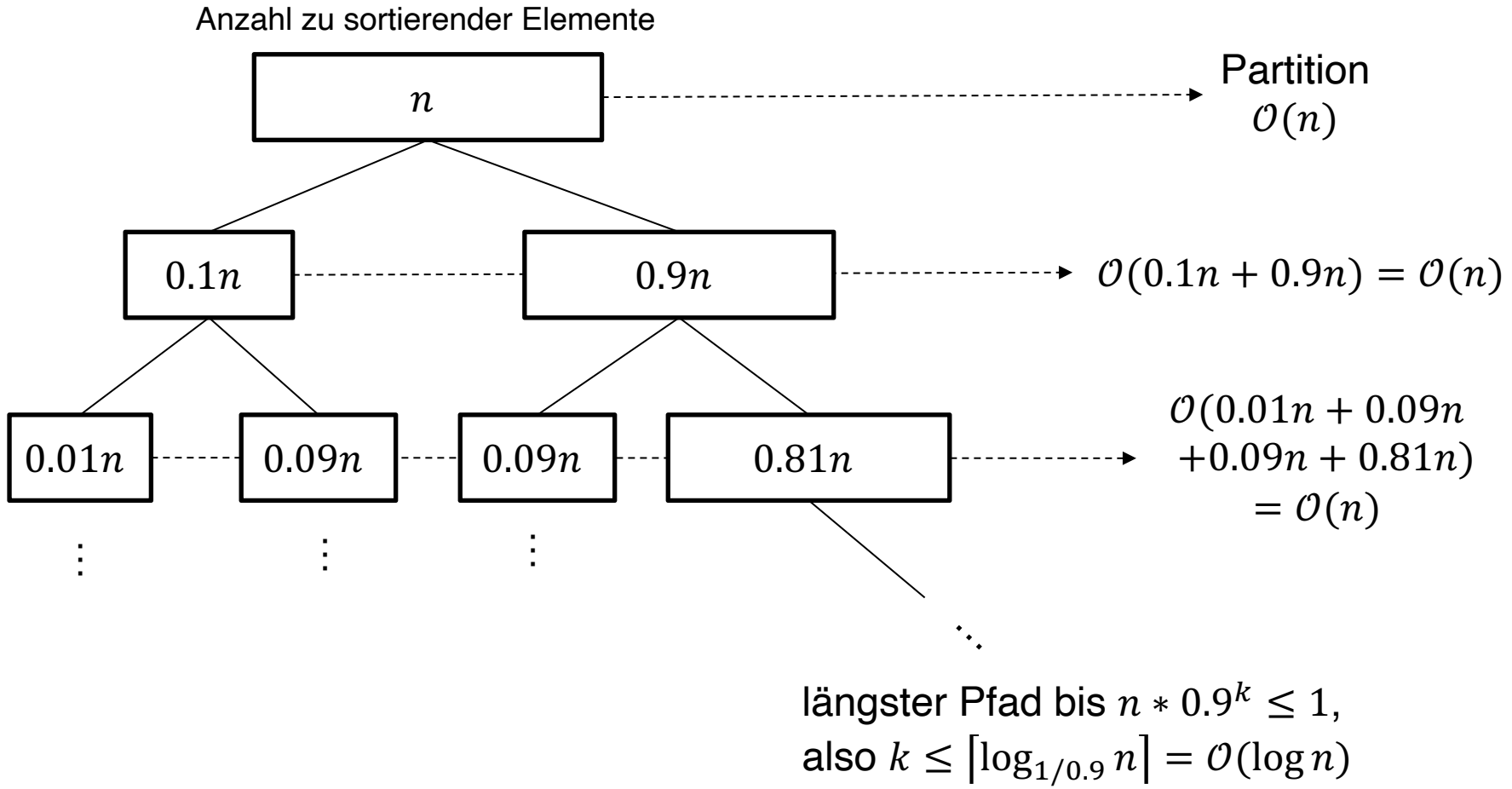
$$T(n) = 2T(n/2) + \Theta(n) \implies \text{Laufzeit } \Theta(n \log n)$$

Gilt auch, solange beide Arrays in Größenordnung $\Omega(n)$,
z.B. stets 10% der n Elemente links und 90% rechts:

$$T(n) = T(0.1n) + T(0.9n) + \Theta(n) \implies \text{Laufzeit } \Theta(n \log n)$$

Best-Case für Quicksort (II)

„Rekursionsbaum“



auf jeder Ebene Aufwand $\mathcal{O}(n) \Rightarrow$ Gesamtaufwand $\mathcal{O}(n \log n)$

Average-Case-Laufzeiten?

(Worst-Case-)Laufzeit
$T(n)$ = max { #Schritte für x }

Achtung: übliche Definition ist komplizierter

Intuitiver Ansatz:
$T(n) = E_{D(n)}$ [#Schritte für x]

erwartete Anzahl von Schritten über Verteilung $D(n)$ auf Eingabedaten der Komplexität n

Wie verhält sich Quicksort im Durchschnitt auf „zufälliger“ Eingabe?

Für zufällige Permutation $D(n)$ eines fixen Arrays von n Elementen benötigt Quicksort $E_{D(n)}[\,Anzahl\ Schritte] = \mathcal{O}(n \log n)$

Aber was ist eine realistische Verteilung auf Eingaben???

Randomisierte Variante Quicksort

Ansatz: betrachte statt zufälliger Eingabe randomisierte Variante **`randomizedQuicksort`**

wählt als Pivot-Element immer uniform eines der Elemente
(und tauscht es an Anfang des Arrays, um wie zuvor fortzufahren)

```
partition(A,left,right) //req.left<right,ret.left..right-1
0 j=RANDOM(left,right); swap(A[left],A[j]);
                              //j uniform in [left…right]
1  pivot=A[left];
2  p=left-1; q=right+1;  //move from left resp. right
3  WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7  return q          // A[left..q], A[q+1..right]
```

Erwartete Laufzeit

Achtung: manchmal auch als Average-Case-Laufzeit bezeichnet

(Worst-Case-)Laufzeit
$T(n)$ = max { #Schritte für x }

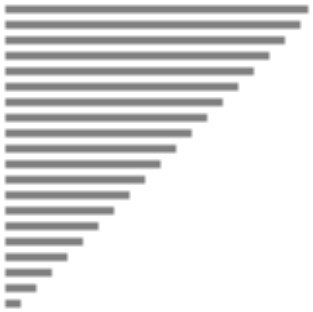
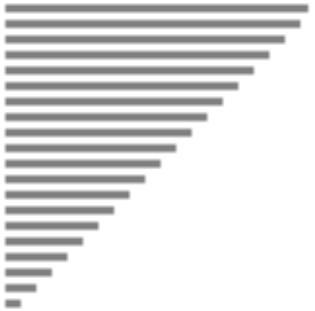
Erwartete Laufzeit
$T(n)$ = max { E_A[#Schritte für x] }

erwartete Anzahl von Schritten
(über zufällige Wahl des Algorithmus A)
für „schlechteste“ Eingabe
der Komplexität n

Erwartete Laufzeit von Randomized-Quicksort ist $\mathcal{O}(n \log n)$

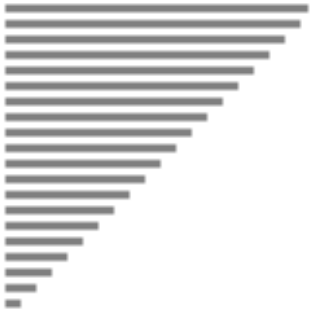
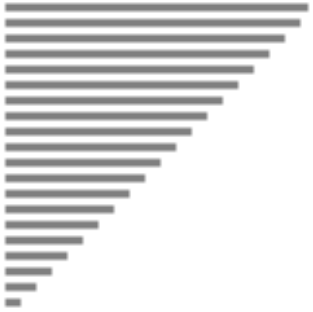
Intuition:
zufällige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig, unabhängig davon, wie Array aussieht

Merge Sort vs. Randomized-Quicksort (I)

	Merge Sort	Randomized-Quicksort
„unsortierte Eingabe“		
„schlechte Eingabe“		

Quelle: **https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/**

Merge Sort vs. Randomized-Quicksort (II)

	Merge Sort	Randomized-Quicksort
„unsortierte Eingabe“		
„schlechte Eingabe“		

Quelle: **https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/**

Vergleich: Insertion, Merge, und Quicksort

Insertion **Sort**	Merge **Sort**	Quicksort
Laufzeit $\Theta(n^2)$	Beste asymptotische Laufzeit $\Theta(n \log n)$	Wort-Case-Laufzeit $\Theta(n^2)$, randomisiert mit erwarteter Laufzeit $\Theta(n \log n)$
einfache Implementierung		in Praxis meist schneller als Merge Sort, da weniger Kopieroperationen
für kleine $n \leq 50$ oft beste Wahl		Implementierungen schalten für kleine n meist auf Insertion Sort

Sortieren in Java

sort

```
public static void sort(byte[] a)
```

Sorts the specified array into ascending numerical order.

Implementation Note:

The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

Quelle: **docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Arrays.html**

Dual-Pivot-Quicksort: Drittelt Array gemäß zweier Pivot-Elemente

Ist **`randomizedQuicksort`** im engeren Sinne überhaupt ein Algorithmus?

Wieviel Schritte braucht **`randomizedQuicksort`** im schlechtesten Fall?

Untere Schranke für vergleichsbasiertes Sortieren

(hier nur für deterministische Algorithmen,
gilt aber auch für randomisierte Algorithmen im Durchschnitt)

Vergleichsbasiertes Sortieren

```
sortByComp(n)
// returns array I with sorted indexes:
// A[ I[i] ] <= A[ I[i+1] ] for i=0,…,n-1

1   done=false;
2   WHILE !done DO
3      determine (i,j); // arbitrarily
4      comp(i,j); // returns A[i] <= A[j]?
5      set done; // true or false
6   compute I from comp-information only;
7   return I
```

kennt nur Größe **n** des Eingabe-Arrays **A**

Informationen über **A** nur durch Vergleichsresultate (Ja/Nein) für gewählte Indizes **i,j**

Alle Sortieralgorithmen bisher sind vergleichsbasiert

Untere Schranke

```
sortByComp(n)
// returns array I with sorted indexes:
// A[ I[i] ] <= A[ I[i+1] ] for i=0,…,n-1

1   done=false;
2   WHILE !done DO
3      determine (i,j); // arbitrarily
4      comp(i,j); // returns A[i] <= A[j]?
5      set done; // true or false
6   compute I from comp-information only;
7   return I
```

Theorem: Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens $\Omega(n \cdot \log n)$ viele Vergleiche machen

Untere Schranke: Eingabemenge

Betrachte Ausgangs-Array **`A`** mit **`A[i]=i`**

und jede Permutation davon, π**`(A)`**, für alle π:

Insgesamt also $n!$ viele mögliche Eingabe-Arrays

Jedes Eingabe-Array erfordert andere Ausgabe **`I`**$=\pi^{-1}$

Mögliche Ausgaben

```
sortByComp(n)
// returns array I with sorted indexes:
// A[ I[i] ] <= A[ I[i+1] ] for i=0,…,n-1

1   done=false;
2   WHILE !done DO
3      determine (i,j); // arbitrarily
4      comp(i,j); // returns A[i] <= A[j]?
5      set done; // true or false
6   compute I from comp-information only;
7   return I
```

Annahme:
macht m Vergleiche

es gibt maximal
2^m mögliche
Antwortsequenzen
Ja/Nein

einzige Info
über **A**
(außer n)

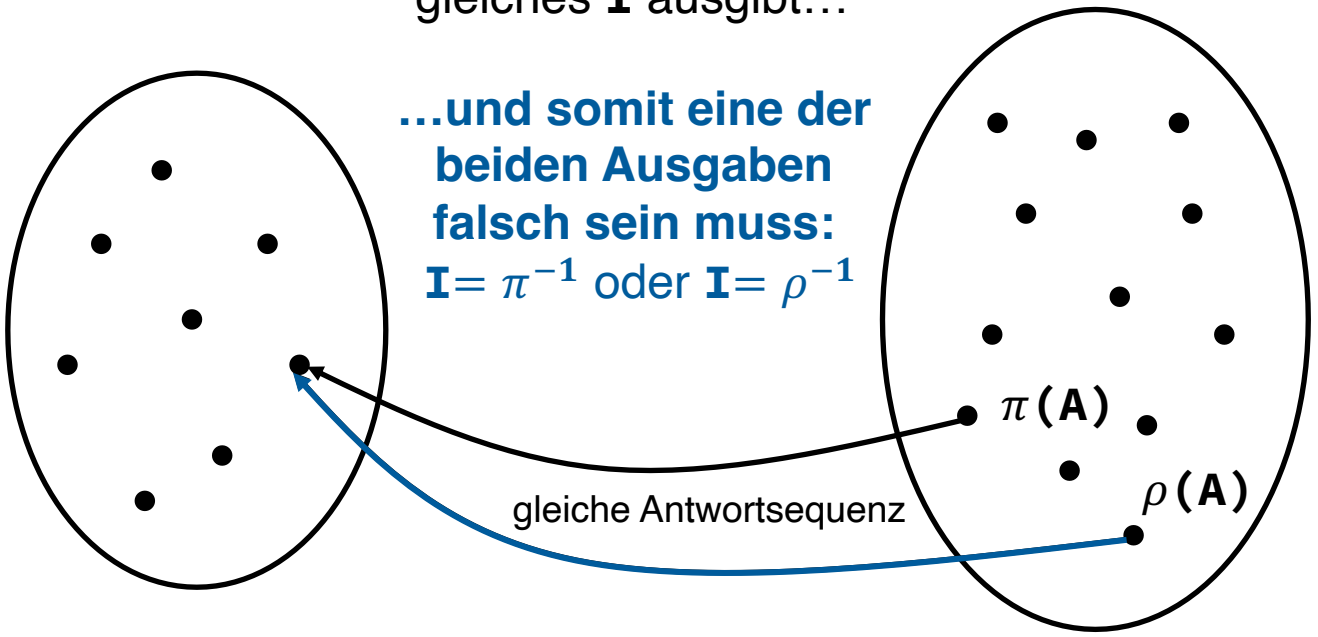
Determiniertheit:
gleiche Antwortsequenzen
(selbst für verschiedene Arrays)
führen zu gleicher Ausgabe

Algorithmus gibt
maximal 2^m verschiedene
Ausgaben **I**

Ein- und Ausgabeverhältnis

Wenn $2^m < n!$, dann gibt es verschiedene Arrays π**(A)** und ρ**(A)** für die der Algorithmus gleiches **I** ausgibt…

…und somit eine der beiden Ausgaben falsch sein muss:
I$= \pi^{-1}$ oder **I**$= \rho^{-1}$



Algorithmus gibt maximal 2^m verschiedene Ausgaben **I**

Es gibt $n!$ verschiedene Eingabe-Arrays π**(A)**

Schranke ausrechnen

Wenn $2^m < n!$, dann gibt es verschiedene Arrays π`(A)` und ρ`(A)` für die der Algorithmus gleiches `I` ausgibt…

Es muss also $2^m \geq n!$ gelten bzw. $m \geq \log_2(n!)$

↓ Stirling-Approximation: $n! \geq \left(\frac{n}{e}\right)^n$

bzw. $m = \Omega\left(n \cdot \log(n)\right)$

Theorem: Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens $\Omega(n \cdot \log n)$ viele Vergleiche machen

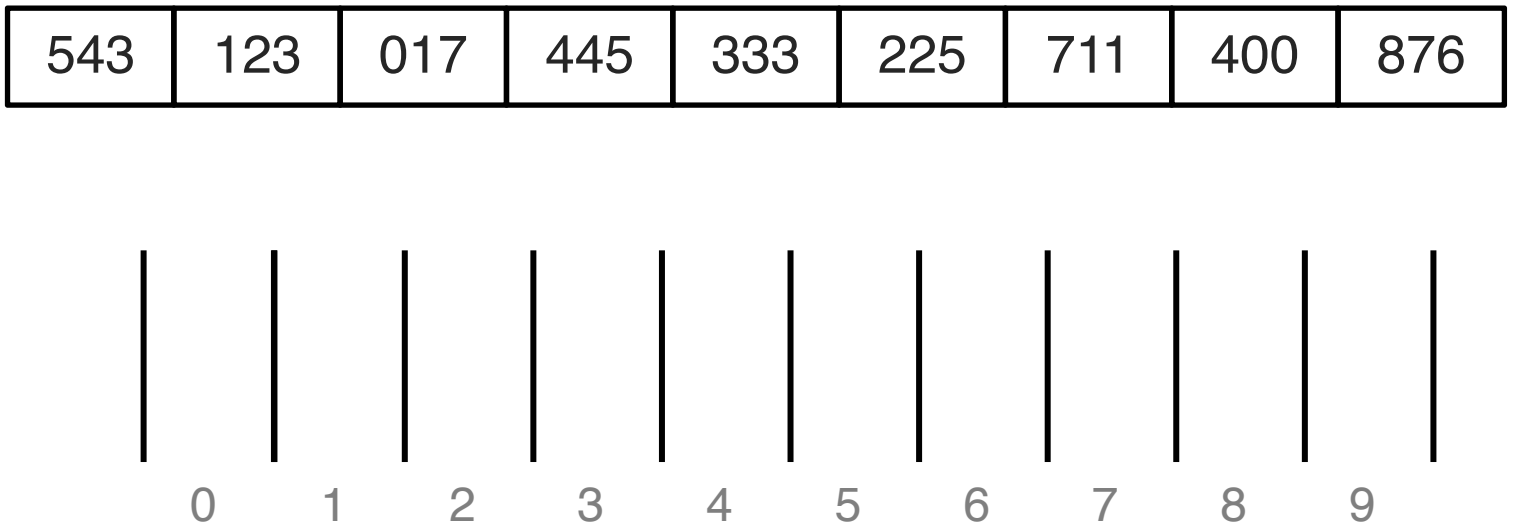
Warum ist Quicksort mit seinen Vertauschungen vergleichsbasiert?

Können Sie sich eine Operation in einem Algorithmus vorstellen, die nicht kompatibel mit der Eigenschaft „vergleichsbasiert“ ist?

RadixSort:
Sortieren in linearer Zeit (?)

Ansatz

Schlüssel sind d-stellige Werte in D-närem Zahlensystem:

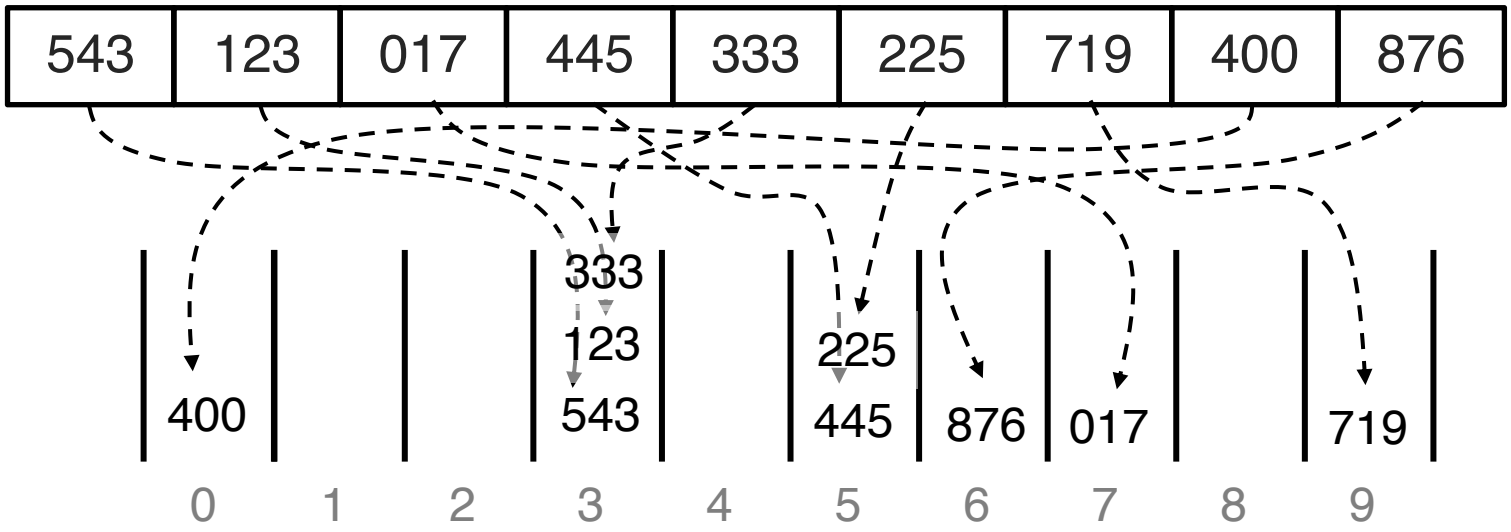


„Buckets“ erlauben
Einfügen und dann Entnehmen (in eingefügter Reihenfolge)
in konstanter Zeit

Queues
(FIFO-Systeme)
→ Abschnitt 3

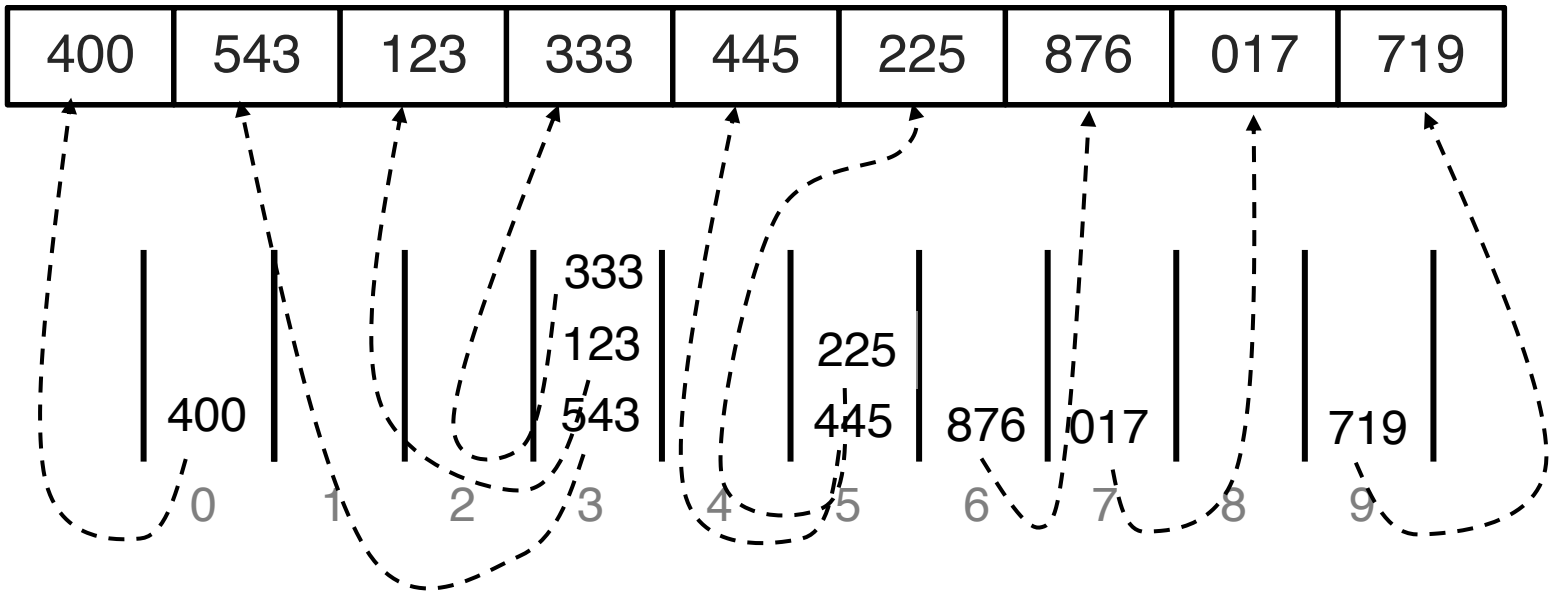
Erste Iteration (I)

Gehe Array von links nach rechts durch und
füge Werte entsprechend **kleinstwertigster** Ziffer in Bucket ein:



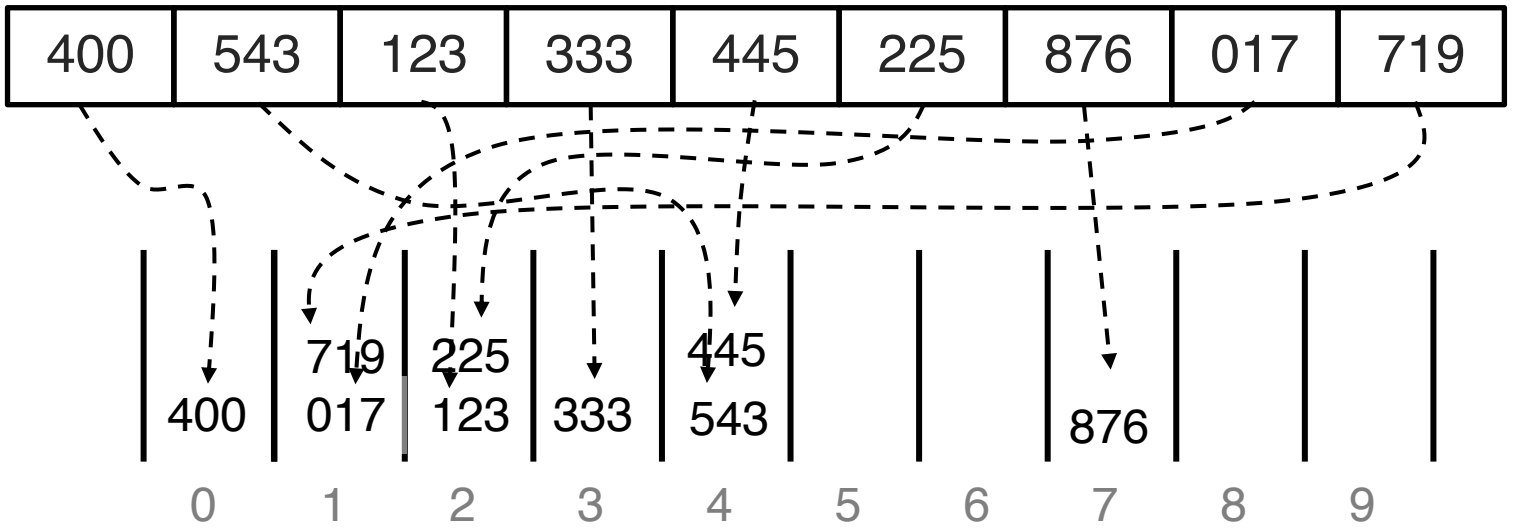
Erste Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



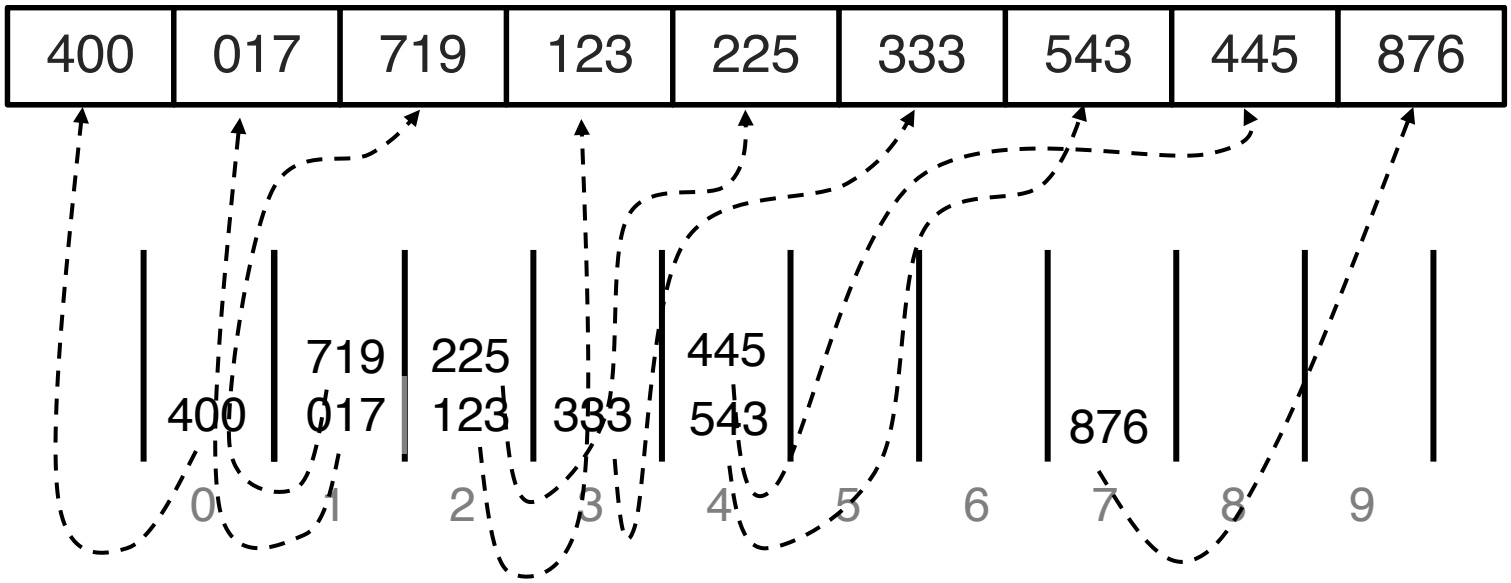
Zweite Iteration (I)

Gehe Array von links nach rechts durch und
füge Werte entsprechend **zweitkleinstwertigster** Ziffer in Bucket ein:



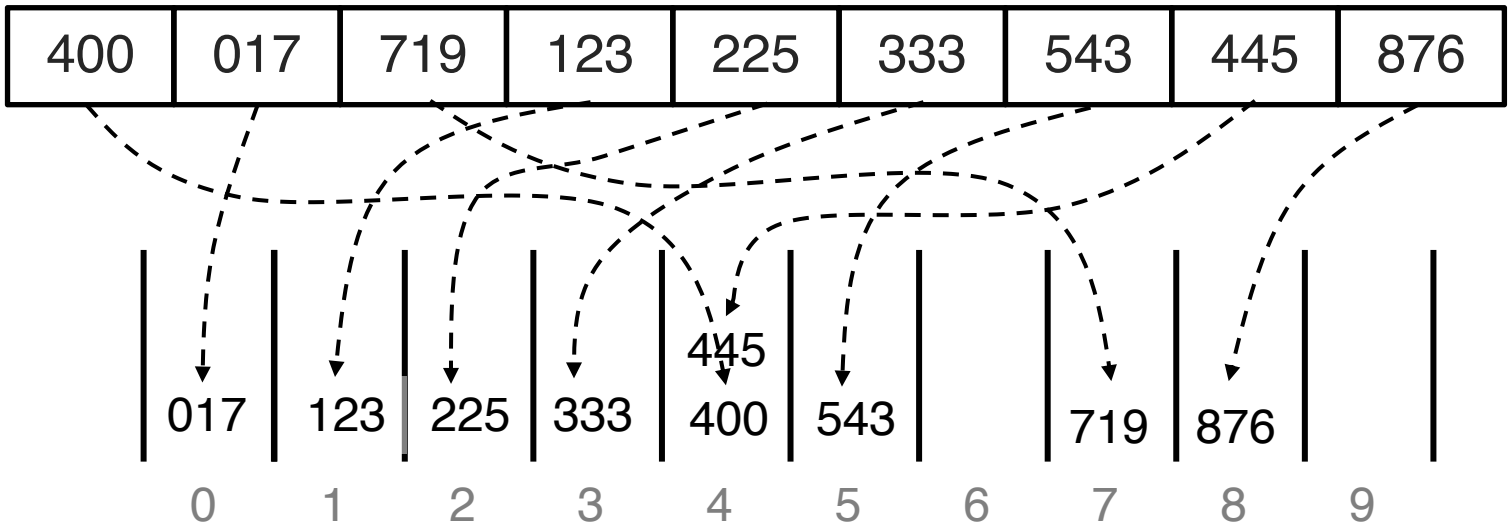
Zweite Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



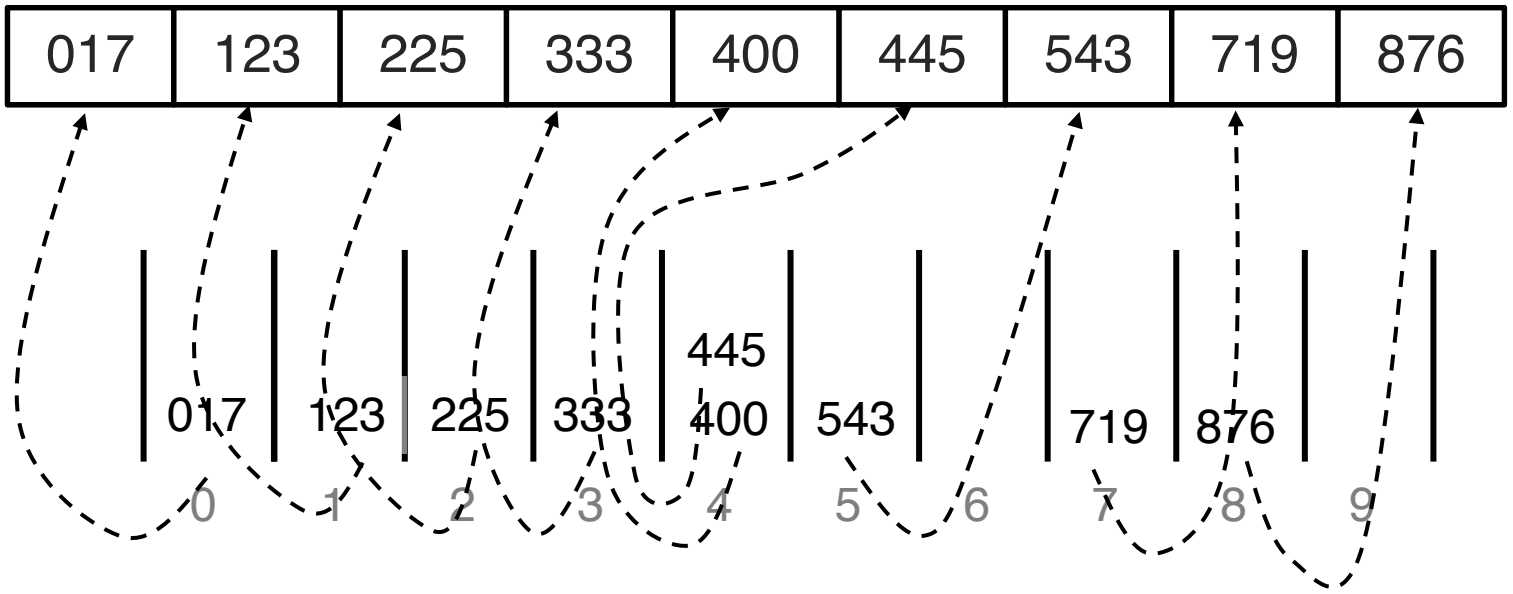
Dritte Iteration (I)

Gehe Array von links nach rechts durch und
füge Werte entsprechend **größtwertigster** Ziffer in Bucket ein:



Dritte Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



Algorithmus

(Terminierung klar)

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][],…, B[D-1][] buckets (init: B[k].size=0)

1   FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
2      FOR j=0 TO n-1 DO putBucket(A,B,i,j);
3      a=0;
4      FOR k=0 TO D-1 DO     //rewrite to array
5          FOR b=0 TO B[k].size-1 DO
6              A[a]=B[k][b]; //read out bucket in order
7              a=a+1;
8          B[k].size=0;      //clear bucket again
9   return A
```

A.size=Anzahl eingetragender Elemente in Array **A**

```
putBucket(A,B,i,j) // call-by-reference
1   z=A[j].digit[i]; // i-th digit of A[j]
2   b=B[z].size;     // next free spot
3   B[z][b]=A[j];
4   B[z].size=B[z].size+1;
```

Korrektheit (I)

Achtung: erste Iteration (i=1) ist für Schleifenwert `i=0`

Per Induktion: Nach i-ter Iteration ist Array gemäß letzten i Ziffern sortiert

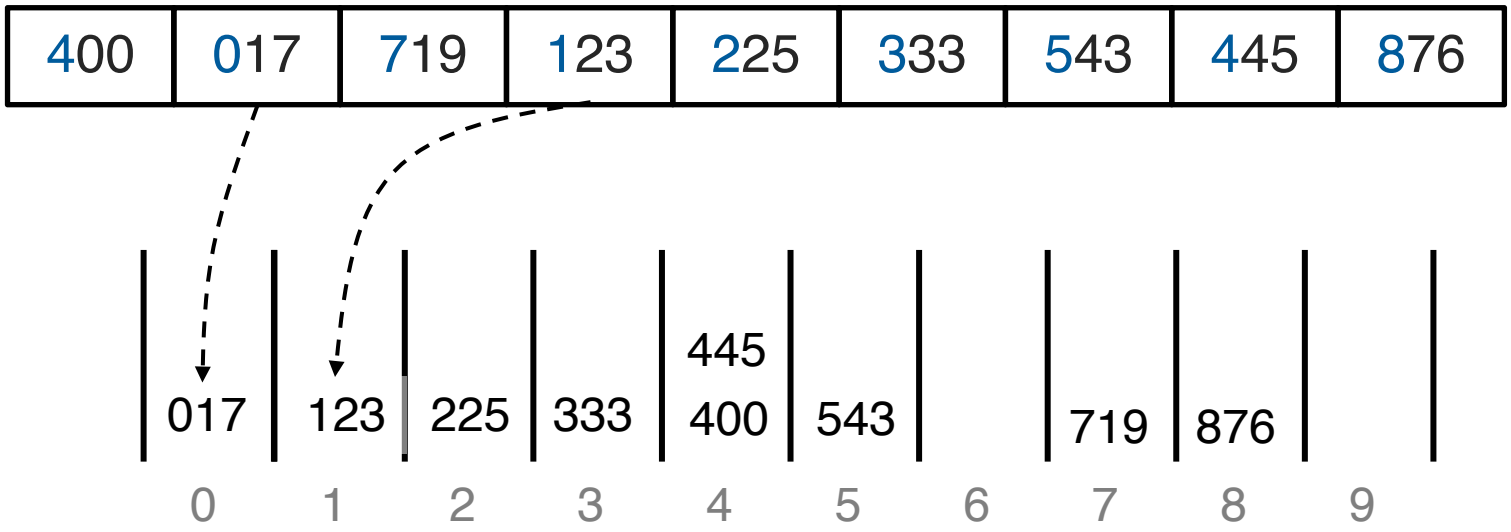
400	543	123	333	445	225	876	017	719

0	1	2	3	4	5	6	7	8	9
400			333 123 543		225 445	876	017		719

Induktionsanfang i=1:
Gilt nach erster Iteration, weil nach letzter Ziffer sortiert wird

Korrektheit (II)

Per Induktion: Nach i-ter Iteration ist Array gemäß letzten i Ziffern sortiert

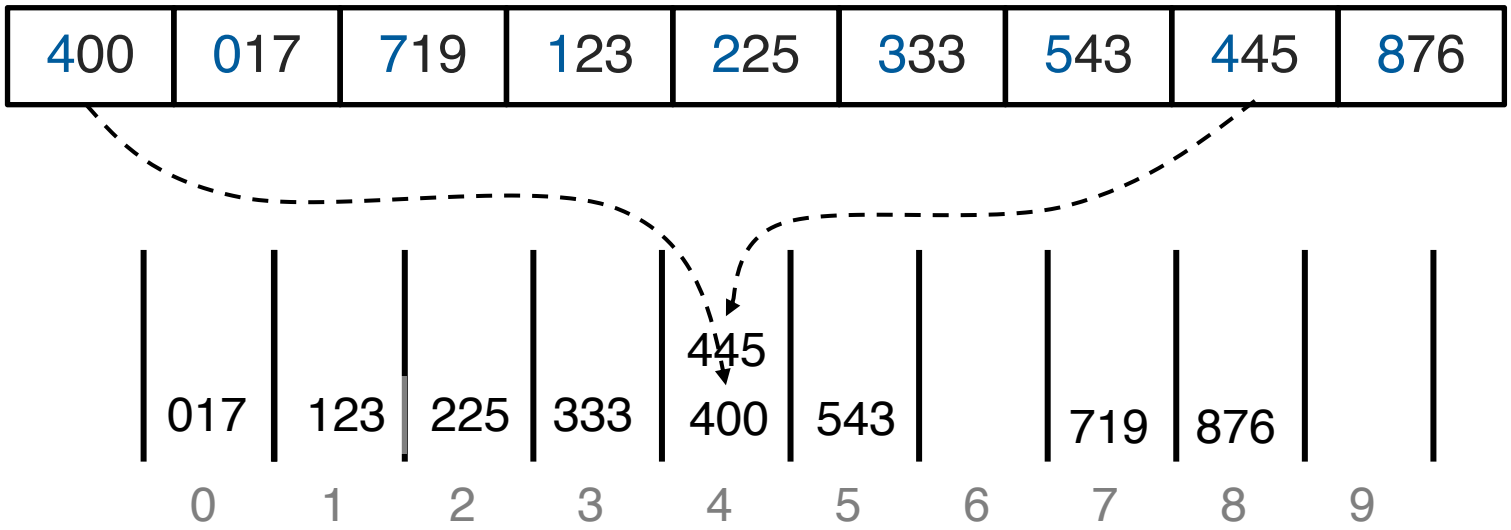


Induktionsschritt i → i+1 (1.Fall):
Wenn (i+1)-te Ziffer zweier beliebiger Werte verschieden,
dann wird Wert mit kleinerer (i+1)-ten Ziffer weiter vorne einsortiert,
zuerst wieder ausgelesen und steht somit auch im Array vorher

Korrektheit (III)

Per Induktion: Nach i-ter Iteration ist Array gemäß letzten i Ziffern sortiert

(und damit nach d-ter Iteration das Array sortiert)



Induktionsschritt i →i+1 (2.Fall):
Wenn (i+1)-te Ziffer gleich, dann steht nach Induktionsvoraussetzung der auf letzten i Ziffern kleinere Wert weiter links, wird zuerst einsortiert und auch zuerst wieder ausgelesen

Laufzeit

Gesamtlaufzeit
$\mathcal{O}(d \cdot (n + D))$

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][],…, B[D-1][] buckets (init: B[k].size=0)

1   FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
2      FOR j=0 TO n-1 DO putBucket(A,B,i,j);
3      a=0;
4      FOR k=0 TO D-1 DO      //rewrite to array
5          FOR b=0 TO B[k].size-1 DO
6              A[a]=B[k][b]; //read out bucket in order
7              a=a+1;
8          B[k].size=0;       //clear bucket again
9   return A
```

$\mathcal{O}(n)$

$\mathcal{O}(n)$ Schritte (alles in **A** kopieren)

$\mathcal{O}(D)$

```
putBucket(A,B,i,j) // call-by-reference
1   z=A[j].digit[i]; // i-th digit of A[j]
2   b=B[z].size;     // next free spot
3   B[z][b]=A[j];
4   B[z].size=B[z].size+1;
```

O(1)

Laufzeit – Interpretation

Gesamtlaufzeit
$\mathcal{O}(d \cdot (n + D))$

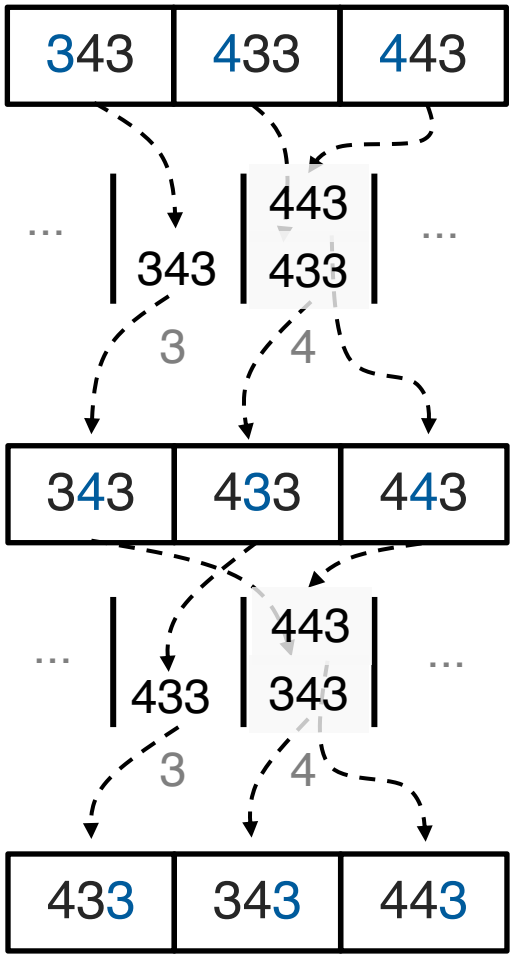
Größe Zifferbereich D oft als konstant angesehen: Laufzeit $\mathcal{O}(dn)$

Wenn auch d als konstant angesehen: Laufzeit $\mathcal{O}(n)$

linear!

Aber:
eindeutige Schlüssel für n Elemente
benötigen $d = \Theta(\log_D n)$ Ziffern! Laufzeit $\mathcal{O}(n \cdot \log n)$

Mit höchstwertiger Ziffer beginnen?



…folgende Iteration ändert Reihenfolge nicht mehr

Wo scheitert der Korrektheitsbeweis beim Sortieren beginnend mit der höchstwertigen Ziffer?

Was sind Beispiele von Schlüsselwerten, die sich nicht per RadixSort sortieren lassen?

Ein Sortieralgorithmus ist stabil, wenn es die relative Ordnung von gleichen Werten beibehält.
Überlegen Sie sich, dass RadixSort stabil ist.