

Algorithmen und Datenstrukturen

A7. Sortieren III

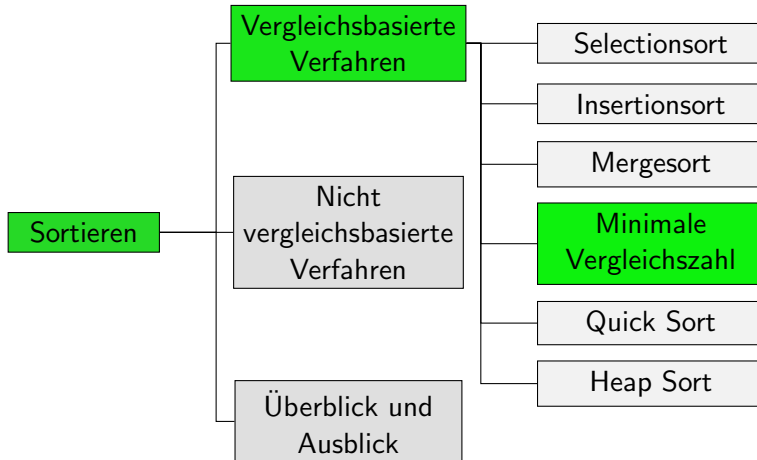
Marcel Lüthi and Gabriele Röger

Universität Basel

6. März 2019

Untere Schranke

Sortiervverfahren



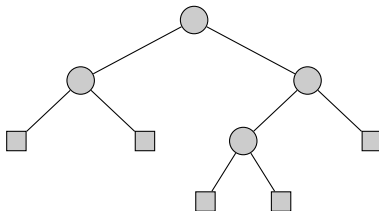
Untere Schranke I

- Mergesort hatte bisher mit $O(n \log_2 n)$ die beste (Worstcase-)Laufzeit.
- Geht es noch besser?
- Wir zeigen: Nicht mit vergleichsbasierten Verfahren!

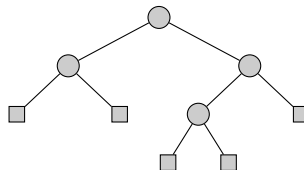
Untere Schranke II

Betrachte beliebigen vergleichsbasierten Sortialgorithmus A .

- Verhalten hängt nur vom Ergebnis der Schlüsselvergleiche ab.
- Bei jedem Schlüsselvergleich gibt es zwei Möglichkeiten, wie der Algorithmus weiter macht.
- Wir können das graphisch als Baum darstellen.

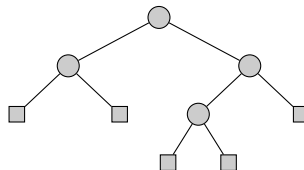


Untere Schranke III



- **Binärbaum**: jeder Knoten hat höchstens zwei Nachfolger
- Knoten ohne Nachfolger heissen **Blätter** (Bild: eckige Knoten).
- Der Knoten ganz oben ist die **Wurzel**.
- Die **Tiefe** eines Blattes entspricht der Anzahl von Kanten von der Wurzel zu dem Blatt.

Untere Schranke III



- **Binärbaum**: jeder Knoten hat höchstens zwei Nachfolger
- Knoten ohne Nachfolger heißen **Blätter** (Bild: eckige Knoten).
- Der Knoten ganz oben ist die **Wurzel**.
- Die **Tiefe** eines Blattes entspricht der Anzahl von Kanten von der Wurzel zu dem Blatt.

Die maximale Tiefe eines Blattes in einem Binärbaum mit k Blättern ist mindestens $\log_2 k$.

Untere Schranke IV

Was muss der Algorithmus können?

- **Annahme:** alle Elemente unterschiedlich
- Muss **alle Eingaben** der Grösse n **korrekt** sortieren.

Untere Schranke IV

Was muss der Algorithmus können?

- **Annahme:** alle Elemente unterschiedlich
- Muss **alle Eingaben** der Grösse n **korrekt** sortieren.
- Wir können alle Algorithmen so anpassen, dass sie verfolgen, von welcher Position zu welcher Position die Elemente bewegt werden müssen.
- Das Ergebnis ist dann nicht das sortierte Array, sondern die entsprechende **Permutation**.

Beispiel: $\text{pos0} \mapsto \text{pos2}$, $\text{pos1} \mapsto \text{pos1}$, $\text{pos2} \mapsto \text{pos0}$

Untere Schranke IV

Was muss der Algorithmus können?

- **Annahme:** alle Elemente unterschiedlich
- Muss **alle Eingaben** der Grösse n **korrekt** sortieren.
- Wir können alle Algorithmen so anpassen, dass sie verfolgen, von welcher Position zu welcher Position die Elemente bewegt werden müssen.
- Das Ergebnis ist dann nicht das sortierte Array, sondern die entsprechende **Permutation**.
Beispiel: $\text{pos0} \mapsto \text{pos2}$, $\text{pos1} \mapsto \text{pos1}$, $\text{pos2} \mapsto \text{pos0}$
- Da alle möglichen Eingaben der Grösse n korrekt gelöst werden müssen, muss der Algorithmus **alle $n!$ möglichen Permutationen** erzeugen können.

Untere Schranke V

- Jedes Blatt in der Baumdarstellung entspricht einer Permutation.
- Bei Eingabegrösse n muss der Baum also mindestens $n!$ Blätter haben.
- Die maximale Tiefe des entsprechenden Baumes ist demnach $\geq \log_2(n!)$.
- Es gibt also eine Eingabe der Grösse n mit $\geq \log_2(n!)$ Schlüsselvergleichen.

Untere Schranke VI

Abschätzung von $\log_2(n!)$

- Es gilt $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$$4! = 1 \cdot 2 \cdot \underset{\geq 2}{3} \cdot \underset{\geq 2}{4} \geq 2^2$$

Untere Schranke VI

Abschätzung von $\log_2(n!)$

- Es gilt $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$$4! = 1 \cdot 2 \cdot \underset{\geq 2}{3} \cdot \underset{\geq 2}{4} \geq 2^2$$

- $\log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log_2\left(\frac{n}{2}\right)$
 $= \frac{n}{2}(\log_2 n + \log_2 \frac{1}{2}) = \frac{n}{2}(\log_2 n - \log_2 2)$
 $= \frac{n}{2}(\log_2 n - 1)$

Untere Schranke VI

Abschätzung von $\log_2(n!)$

- Es gilt $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$$4! = 1 \cdot 2 \cdot \underset{\geq 2}{3} \cdot \underset{\geq 2}{4} \geq 2^2$$

- $\log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \log_2\left(\frac{n}{2}\right)$
 $= \frac{n}{2}(\log_2 n + \log_2 \frac{1}{2}) = \frac{n}{2}(\log_2 n - \log_2 2)$
 $= \frac{n}{2}(\log_2 n - 1)$

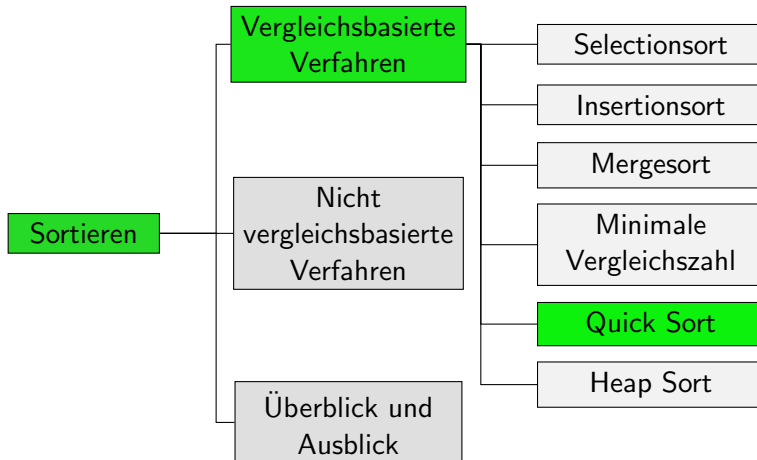
Theorem

Jeder *vergleichsbasierte Sortieralgorithmus* benötigt $\Omega(n \log n)$ viele Schlüsselvergleiche. Damit liegt auch die *Laufzeit* in $\Omega(n \log n)$.

Mergesort ist asymptotisch optimal.

Quicksort

Sortiervverfahren



Quicksort: Idee

- Wie Merge-Sort ein **Divide-and-Conquer-Verfahren**
- Die Sequenz wird nicht wie bei Mergesort nach Positionen aufgeteilt, sondern nach Werten.
- Hierfür wird ein Element P gewählt (das sogenannte **Pivotelement**).
- Dann wird so umsortiert, dass P an die endgültige Position kommt, vor P nur Elemente $\leq P$ stehen, und hinten nur Elemente $\geq P$.



- Macht man das rekursiv für den vorderen und den hinteren Teil, ist die Sequenz am Ende sortiert.

Quicksort: Algorithmus

```
1 def sort(array):
2     sort_aux(array, 0, len(array)-1)
3
4 def sort_aux(array, lo, hi):
5     if hi <= lo:
6         return
7     choose_pivot_and_swap_it_to_lo(array, lo, hi)
8     pivot_pos = partition(array, lo, hi)
9     sort_aux(array, lo, pivot_pos - 1)
10    sort_aux(array, pivot_pos + 1, hi)
```

Wie wählt man das Pivot-Element?

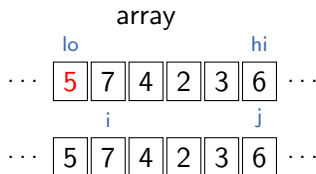
Für die Korrektheit des Verfahrens ist das egal. (Warum?)

Wir können zum Bsp. folgende Strategien wählen:

- **Naiv:** Nimm immer erstes Element
- **Median of Three:** Verwende Median aus erstem, mittlerem und letztem Element
- **Randomisiert:** Wähle zufällig ein Element aus

Gute Pivot-Elemente teilen Sequenz in etwa gleich grosse Bereiche.

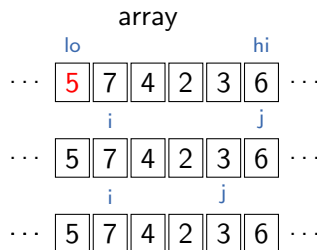
Wie macht man die Umsortierung?



Pivot ist an Pos 0.

Initialisiere $i = lo + 1, j = hi$

Wie macht man die Umsortierung?



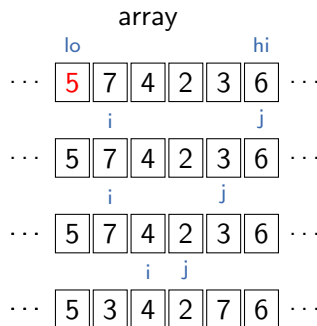
Pivot ist an Pos 0.

Initialisiere $i = lo + 1, j = hi$

i nach rechts bis zu Element $\geq$ Pivot,

j nach links bis Element $\leq$ Pivot

Wie macht man die Umsortierung?



Pivot ist an Pos 0.

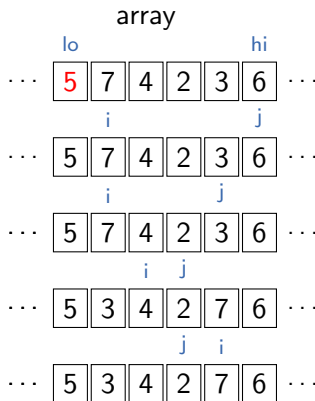
Initialisiere $i = lo + 1, j = hi$

i nach rechts bis zu Element $\geq$ Pivot,

j nach links bis Element $\leq$ Pivot

Falls $i < j$: Elemente tauschen, $i++$, $j--$

Wie macht man die Umsortierung?



Pivot ist an Pos 0.

Initialisiere $i = lo + 1, j = hi$

i nach rechts bis zu Element $\geq$ Pivot,

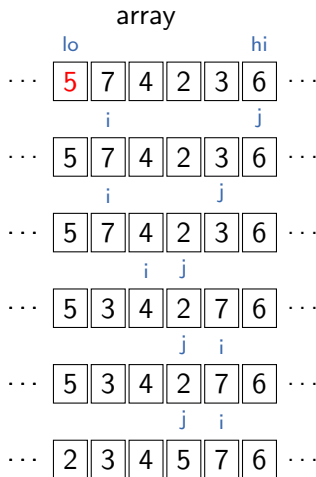
j nach links bis Element $\leq$ Pivot

Falls $i < j$: Elemente tauschen, $i++$, $j--$

i nach rechts bis zu Element $\geq$ Pivot,

j nach links bis Element $\leq$ Pivot

Wie macht man die Umsortierung?



Pivot ist an Pos 0.

Initialisiere $i = lo + 1, j = hi$

i nach rechts bis zu Element $\geq$ Pivot,

j nach links bis Element $\leq$ Pivot

Falls $i < j$: Elemente tauschen, $i++$, $j--$

i nach rechts bis zu Element $\geq$ Pivot,

j nach links bis Element $\leq$ Pivot

$i \geq j$: noch Pivot an Pos j tauschen

Fertig!

Quicksort: Partitionierung

```
1 def partition(array, lo, hi):
2     pivot = array[lo]
3     i = lo + 1
4     j = hi
5     while (True):
6         while i < hi and array[i] < pivot:
7             i += 1
8         while array[j] > pivot:
9             j -= 1
10        if i >= j:
11            break
12
13        array[i], array[j] = array[j], array[i]
14        i, j = i + 1, j - 1
15    array[lo], array[j] = array[j], array[lo]
16    return j
```

Quicksort: Laufzeit I

Best case: Pivot-Element teilt in gleich grosse Bereiche

- $O(\log_2 n)$ rekursive Aufrufe
- jeweils hi-lo Schlüsselvergleiche in Partitionierung
- auf einer Rekursionsebene insgesamt $O(n)$ Vergleiche in Partitionierung

→ $O(n \log n)$

Quicksort: Laufzeit I

Best case: Pivot-Element teilt in gleich grosse Bereiche

- $O(\log_2 n)$ rekursive Aufrufe
- jeweils hi-lo Schlüsselvergleiche in Partitionierung
- auf einer Rekursionsebene insgesamt $O(n)$ Vergleiche in Partitionierung

→ $O(n \log n)$

Worst case: Pivot-Element immer kleinstes oder grösstes Element

- insgesamt $n-1$ (nichttriviale) rekursive Aufrufe für Länge $n, n-1, \dots, 2$.
- jeweils hi-lo Schlüsselvergleiche in Partitionierung

→ $\Theta(n^2)$

Quicksort: Laufzeit II

Average case:

- Annahme: n verschiedene Elemente, jede der $n!$ Permutationen gleich wahrscheinlich, Pivotelement zufällig gewählt
- $O(\log n)$ rekursive Aufrufe
- insgesamt $O(n \log n)$
- etwa 39% langsamer als best case

Quicksort: Laufzeit II

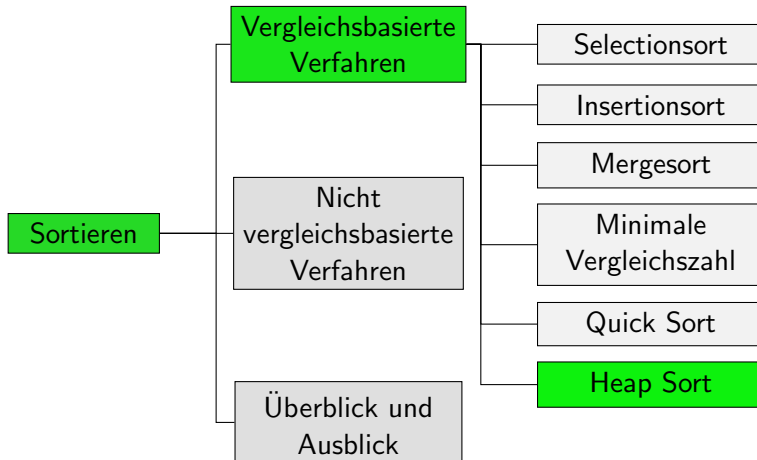
Average case:

- Annahme: n verschiedene Elemente, jede der $n!$ Permutationen gleich wahrscheinlich, Pivotelement zufällig gewählt
- $O(\log n)$ rekursive Aufrufe
- insgesamt $O(n \log n)$
- etwa 39% langsamer als best case

Bei randomisierter Pivotwahl tritt worst-case quasi nicht auf. Quicksort wird daher oft als $O(n \log n)$ -Verfahren betrachtet.

Heapsort

Sortiervverfahren

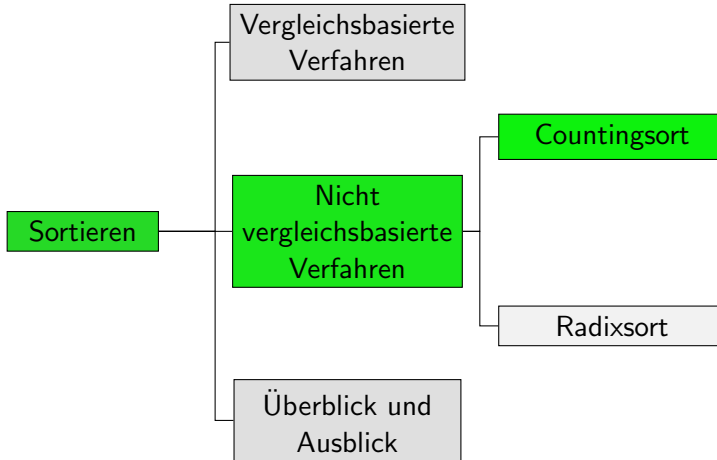


Heapsort

- **Heap:** Datenstruktur, die das Finden und Entnehmen des **grössten** Elements besonders effizient unterstützt
Finden: $\Theta(1)$, Entnehmen: $\Theta(\log n)$
- **Grundidee analog zu Selectionsort:** Setze sukzessive das grösste Element an das Ende des unsortierten Bereichs.
- Kann den **Heap direkt in der Eingabesequenz repräsentieren**, so dass Heapsort nur konstanten zusätzlichen Speicherplatz benötigt.
- Die Laufzeit von Heapsort ist leicht überlinear.
- Wir besprechen die Details später, wenn wir Heaps genauer kennengelernt haben.

Nicht vergleichsbasierte Verfahren

Sortierverfahren



Countingsort: Idee

„Sortieren durch Zählen“

- **Annahme:** Elemente sind aus Bereich $0, \dots, k - 1$.
- Laufe einmal über die Eingabesequenz und zähle dabei, wie oft jedes Element vorkommt.
- Sei $\#i$ die Anzahl der Vorkommen von Element i .
- Iteriere $i = 0, \dots, k - 1$ und schreibe jeweils $\#i$ -mal Element i in die Sequenz.

Countingsort: Algorithmus

```
1 def sort(array, k):
2     counts = [0] * k # list of k zeros
3     for elem in array:
4         counts[elem] += 1
5
6     pos = 0
7     for i in range(k):
8         occurrences_of_i = counts[i]
9         for j in range(occurrences_of_i):
10             array[pos + j] = i
11         pos += occurrences_of_i
```

Countingsort: Algorithmus

```
1 def sort(array, k):
2     counts = [0] * k # list of k zeros
3     for elem in array:
4         counts[elem] += 1
5
6     pos = 0
7     for i in range(k):
8         occurrences_of_i = counts[i]
9         for j in range(occurrences_of_i):
10             array[pos + j] = i
11         pos += occurrences_of_i
```

Laufzeit: $O(n + k)$ (n Grösse der Eingabesequenz)

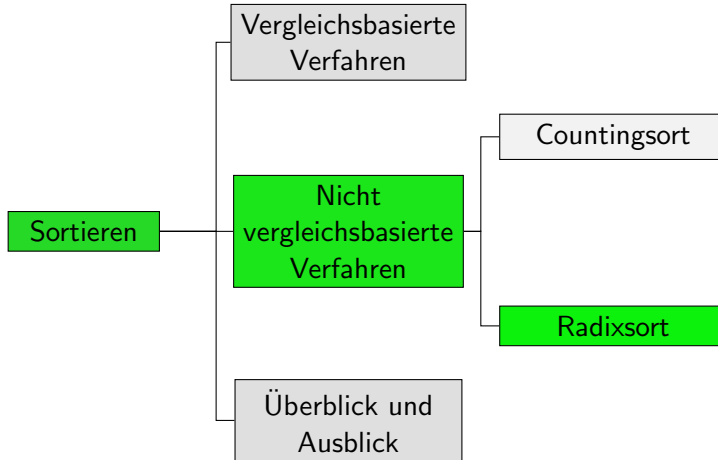
Countingsort: Algorithmus

```
1 def sort(array, k):
2     counts = [0] * k # list of k zeros
3     for elem in array:
4         counts[elem] += 1
5
6     pos = 0
7     for i in range(k):
8         occurrences_of_i = counts[i]
9         for j in range(occurrences_of_i):
10             array[pos + j] = i
11         pos += occurrences_of_i
```

Laufzeit: $O(n + k)$ (n Grösse der Eingabesequenz)

→ Für festes k linear

Sortierverfahren



Radixsort: Idee

„Sortieren durch Fachverteilen“

- Annahme: Schlüssel sind Zahlen im Dezimalsystem
z.B. 763, 983, 96, 286, 462

Radixsort: Idee

„Sortieren durch Fachverteilen“

- Annahme: Schlüssel sind Zahlen im Dezimalsystem

z.B. 763, 983, 96, 286, 462

- Teile Zahlen nach **letzter** Stelle auf:

0	1	2	3	4	5	6	7	8	9
		462	763			96			
			983			286			

Radixsort: Idee

„Sortieren durch Fachverteilen“

- Annahme: Schlüssel sind Zahlen im Dezimalsystem
z.B. 763, 983, 96, 286, 462

- Teile Zahlen nach **letzter** Stelle auf:

0	1	2	3	4	5	6	7	8	9
		462	763			96			
			983			286			

- Sammle Zahlen von vorne nach hinten/oben nach unten auf
462, 763, 983, 96, 286

Radixsort: Idee

„Sortieren durch Fachverteilen“

- Annahme: Schlüssel sind Zahlen im Dezimalsystem
z.B. 763, 983, 96, 286, 462

- Teile Zahlen nach **letzter** Stelle auf:

0	1	2	3	4	5	6	7	8	9
		462	763			96			
			983			286			

- Sammle Zahlen von vorne nach hinten/oben nach unten auf
462, 763, 983, 96, 286
- Teile Zahlen nach **vorletzter** Stelle auf, sammle sie auf.
- Teile Zahlen nach **drittletzter** Stelle auf, sammle sie auf.
- usw. bis alle Stellen betrachtet wurden.

Radixsort: Beispiel

- Eingabe: 263, 983, 96, 462, 286

Radixsort: Beispiel

- Eingabe: 263, 983, 96, 462, 286
- Aufteilung nach letzter Stelle:

0	1	2	3	4	5	6	7	8	9
		462	263			96			
			983			286			

Aufsammeln ergibt: 462, 263, 983, 96, 286

Radixsort: Beispiel

- Eingabe: 263, 983, 96, 462, 286

- Aufteilung nach letzter Stelle:

0	1	2	3	4	5	6	7	8	9
		462	263			96			
			983			286			

Aufsammeln ergibt: 462, 263, 983, 96, 286

- Aufteilung nach vorletzter Stelle:

0	1	2	3	4	5	6	7	8	9
						462		983	96
						263		286	

Aufsammeln ergibt: 462, 263, 983, 286, 96

Radixsort: Beispiel

- Eingabe: 263, 983, 96, 462, 286

- Aufteilung nach letzter Stelle:

0	1	2	3	4	5	6	7	8	9
		462	263			96			
			983			286			

Aufsammeln ergibt: 462, 263, 983, 96, 286

- Aufteilung nach vorletzter Stelle:

0	1	2	3	4	5	6	7	8	9
						462		983	96
						263		286	

Aufsammeln ergibt: 462, 263, 983, 286, 96

- Aufteilung nach drittletzter Stelle:

0	1	2	3	4	5	6	7	8	9
096		263		462					983
		286							

Aufsammeln ergibt: 96, 263, 286, 462, 983

Radixsort: Algorithmus (für beliebige Basis)

```
1 def sort(array, base=10):
2     if not array: # array is empty
3         return
4     iteration = 0
5     max_val = max(array) # identify largest element
6     while base ** iteration <= max_val:
7         buckets = [[] for num in range(base)]
8         for elem in array:
9             digit = (elem // (base ** iteration)) % base
10            buckets[digit].append(elem)
11        pos = 0
12        for bucket in buckets:
13            for elem in bucket:
14                array[pos] = elem
15                pos += 1
16        iteration += 1
```

Radixsort: Laufzeit

- m : Maximale Anzahl Stellen in Repräsentation mit gegebener Basis b .
- n : Länge der Eingabesequenz
- Laufzeit in $O(m \cdot (n + b))$

Radixsort: Laufzeit

- m : Maximale Anzahl Stellen in Repräsentation mit gegebener Basis b .
- n : Länge der Eingabesequenz
- Laufzeit in $O(m \cdot (n + b))$

Für festes m und b hat Radixsort lineare Laufzeit.

Zusammenfassung

Zusammenfassung

- Jedes **vergleichsbasierte Sortierverfahren** hat **mindestens leicht überlineare Laufzeit**.
- **Quicksort** ist ein **Divide-and-Conquer**-Verfahren, das die Elemente relativ zu einem **Pivotelement** aufteilt.
- **Countingsort** und **Radixsort** sind **nicht vergleichsbasiert** und erlauben (unter bestimmten Restriktionen) ein Sortieren in **linearer Zeit**.
- Sie machen jedoch zusätzliche Einschränkungen an die verwendeten Schlüssel.