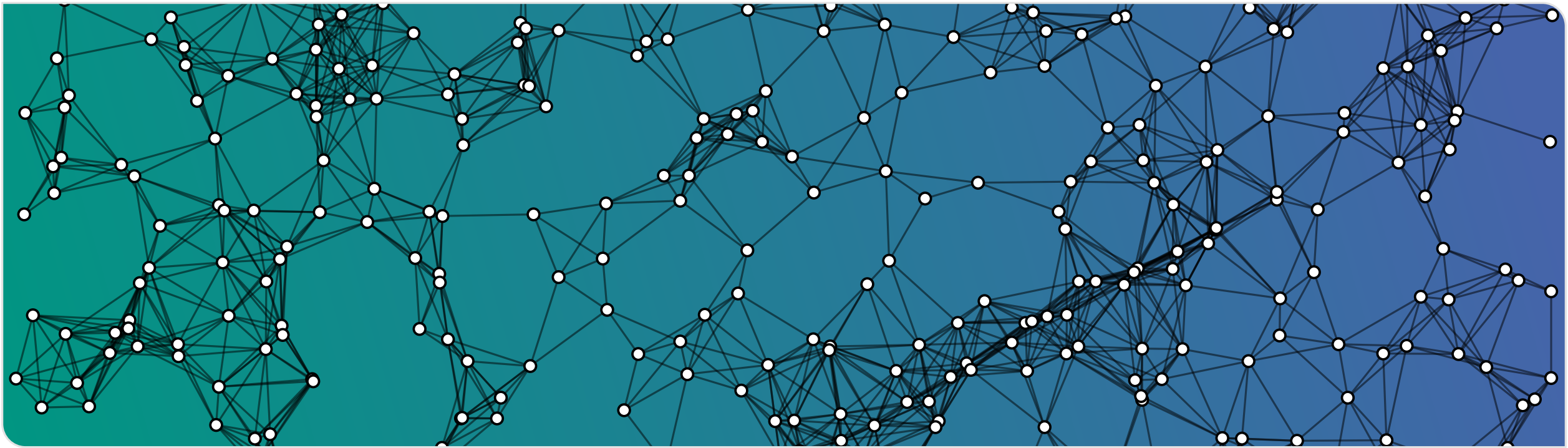
Karlsruher Institut für Technologie

Algorithmen 1

Sortieren: Bucketsort, Radixsort, Word-RAM



Überblick

Letztes Mal gesehen

- Sortieralgorithmen mit Laufzeit $O(n \log n)$ (Mergesort, Quicksort)
- wir müssen Elemente nur vergleichen können und sonst nichts über sie wissen
- vergleichsbasiert $\rightarrow$ die Algorithmen sind universell einsetzbar
- untere Schranke: jeder vergleichsbasierte Sortieralgo benötigt $\Omega(n \log n)$ Vergleiche

Überblick

Letztes Mal gesehen

- Sortieralgorithmen mit Laufzeit $O(n \log n)$ (Mergesort, Quicksort)
- wir müssen Elemente nur vergleichen können und sonst nichts über sie wissen
- vergleichsbasiert $\rightarrow$ die Algorithmen sind universell einsetzbar
- untere Schranke: jeder vergleichsbasierte Sortieralgo benötigt $\Omega(n \log n)$ Vergleiche

Plan für heute

- sortieren von Zahlen in $O(n)$: Bucketsort, Radixsort
- unter gewissen Annahmen an die Größe der Zahlen

Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen (wenn die Zahlen nicht zu groß sind)

Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen (wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$

$\langle 3, 7, 9, 3, 5, 1, 0, 1 \rangle$

Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

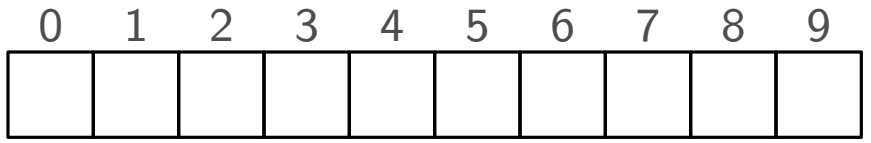
- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m

$\langle 3, 7, 9, 3, 5, 1, 0, 1 \rangle$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

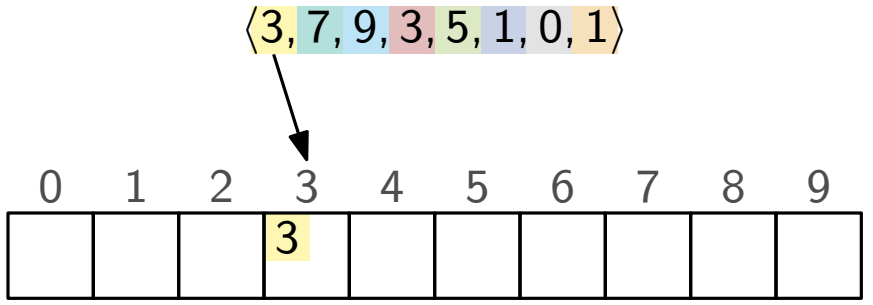
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

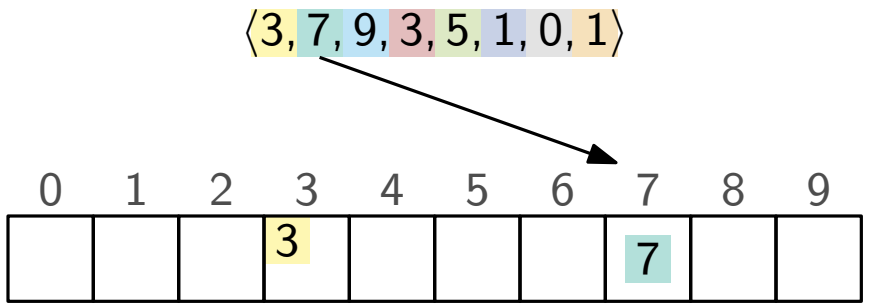
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

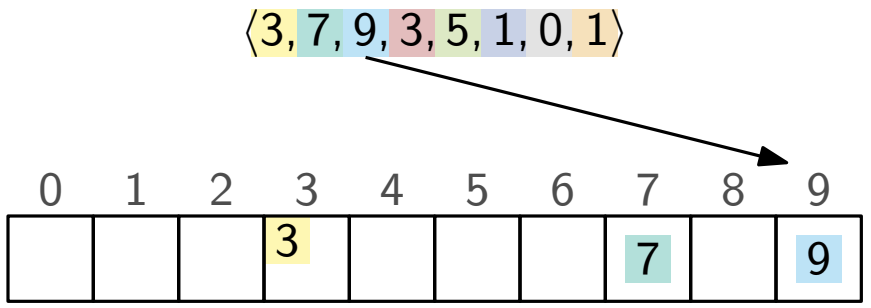
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

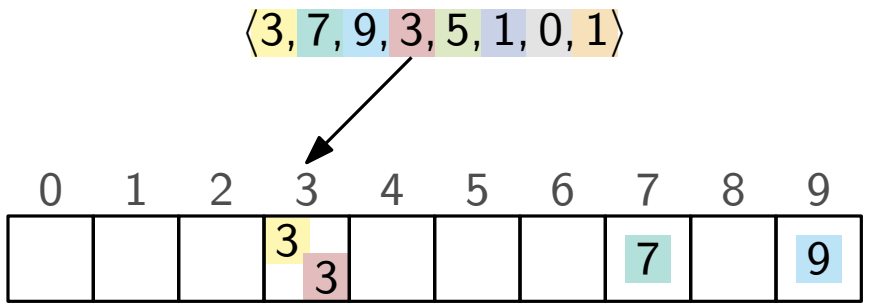
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

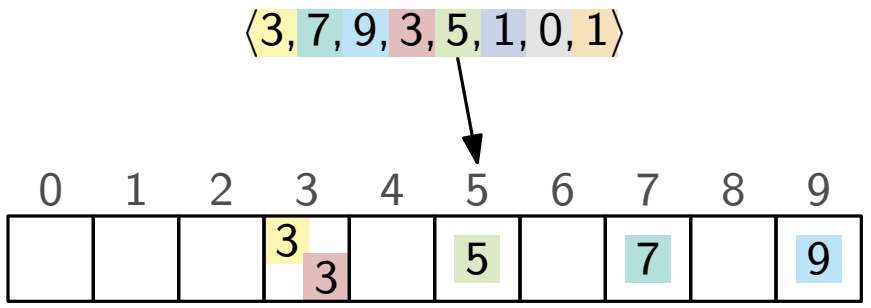
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

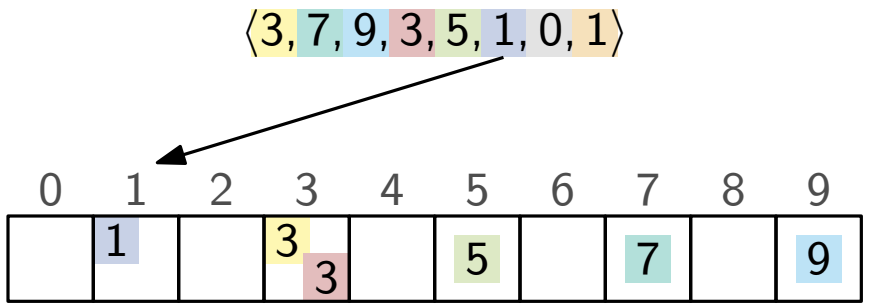
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

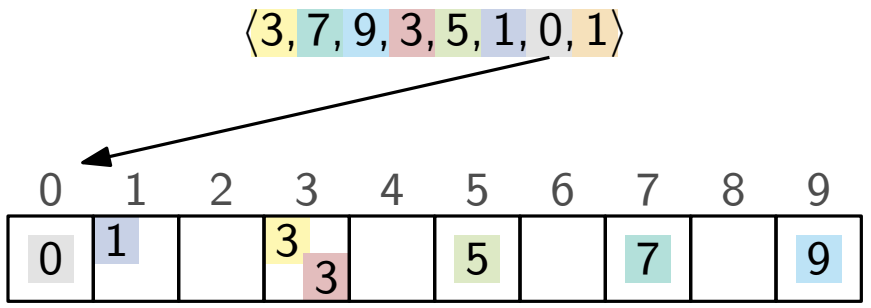
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

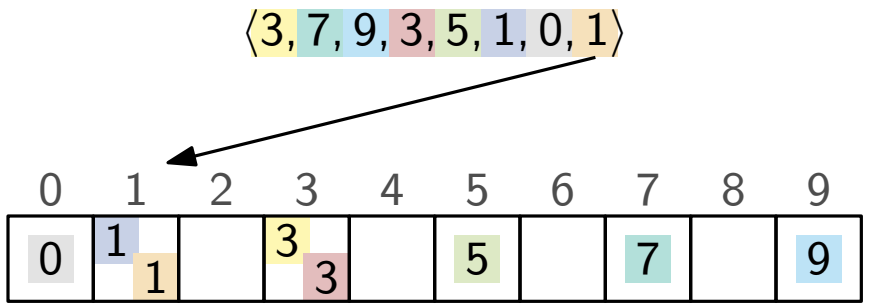
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

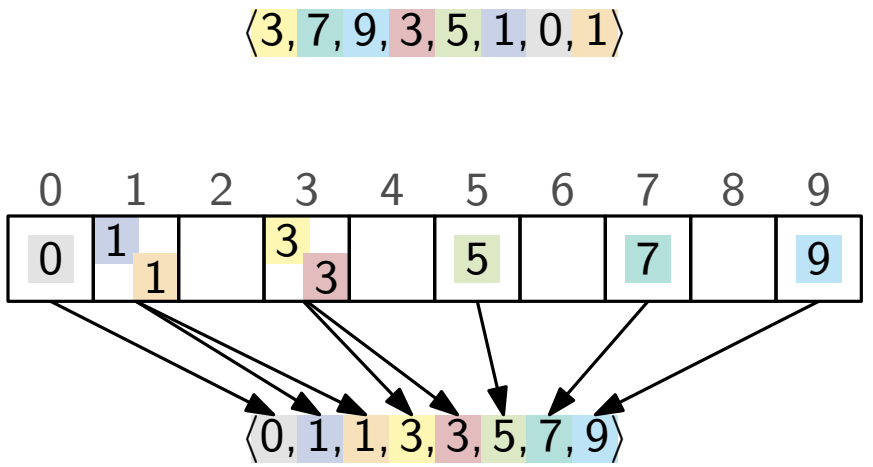
Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$
- lies sortierte Folge aus B ab



app.klicker.uzh.ch/join/algo1

Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

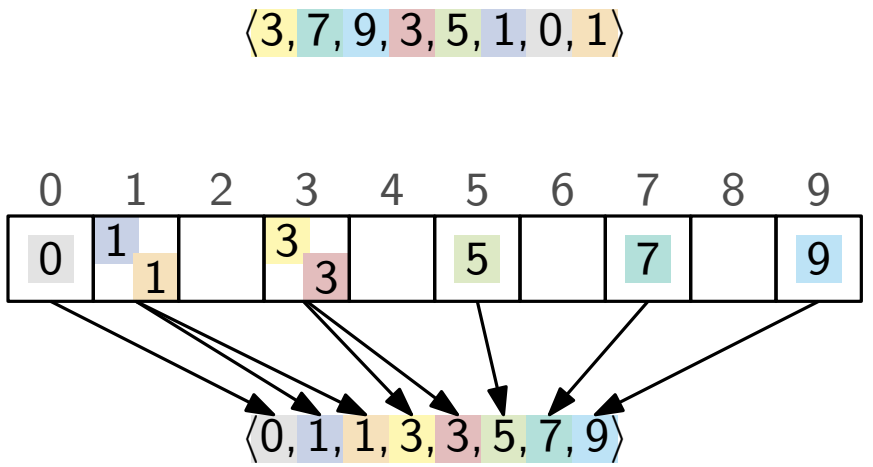
- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$
- lies sortierte Folge aus B ab

Was ist die Laufzeit?



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

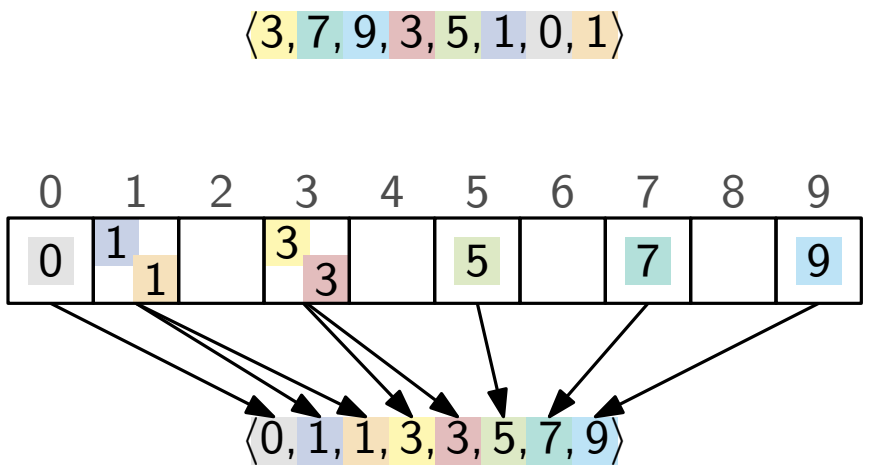
- zu sortieren: n Zahlen aus $[0, m)$
- erstelle Buckets: Array B der Größe m
- für jede Zahl x: speichere x in $B[x]$
- lies sortierte Folge aus B ab

Laufzeit

$\Theta(m)$

$\Theta(n)$

$\Theta(n+m)$



Bucketsort

Theorem
Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Was können wir mit Zahlen tun?

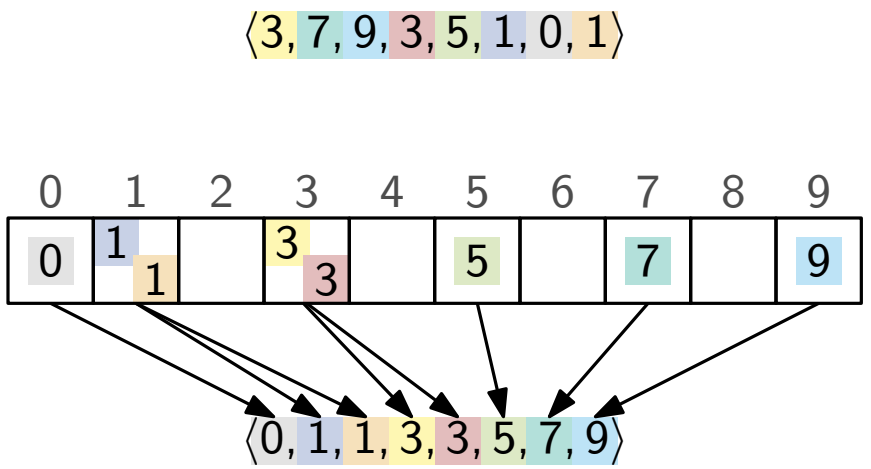
- vergleichen $\rightarrow$ hilft hier nicht
- Array-Zellen adressieren $\rightarrow$ das können wir nutzen

(wenn die Zahlen nicht zu groß sind)

Bucketsort

	Laufzeit
zu sortieren: n Zahlen aus $[0, m)$	
erstelle Buckets: Array B der Größe m	$\Theta(m)$
für jede Zahl x: speichere x in $B[x]$	$\Theta(n)$
lies sortierte Folge aus B ab	$\Theta(n+m)$

Laufzeit $\Theta(n+m) = \Theta(n)$, wenn $m \in O(n)$



Zahlen als Schlüssel

Mehr als nur Zahlen

- oft will man nicht tatsächlich Zahlen sortieren
- sondern komplexere Objekte bezüglich einer Kennzahl
- diese Kennzahl nennt man auch **Schlüssel (Key)**

Zahlen als Schlüssel

Mehr als nur Zahlen

- oft will man nicht tatsächlich Zahlen sortieren
- sondern komplexere Objekte bezüglich einer Kennzahl
- diese Kennzahl nennt man auch **Schlüssel (Key)**

Beispiel 1: Personen nach Alter in Jahren

- Eingabe: Tabelle mit Personendaten (n Zeilen)
- maximales Alter ist typischerweise klein (in $O(n)$)
- Bucketsort hat Laufzeit $\Theta(n)$

Name	Alter
Peter Arbeitsloser	26
Martyn Vorstand	38
Henryk Ingenieur	45
Kiki Unbekannt	28
Der Alte	92

Personen aus: Qualityland, Marc-Uwe Kling

Zahlen als Schlüssel

Mehr als nur Zahlen

- oft will man nicht tatsächlich Zahlen sortieren
- sondern komplexere Objekte bezüglich einer Kennzahl
- diese Kennzahl nennt man auch **Schlüssel (Key)**

Beispiel 1: Personen nach Alter in Jahren

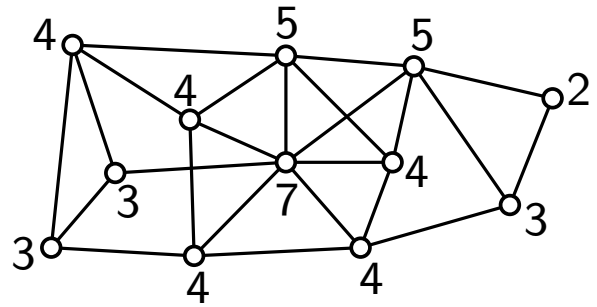
- Eingabe: Tabelle mit Personendaten (n Zeilen)
- maximales Alter ist typischerweise klein (in $O(n)$)
- Bucketsort hat Laufzeit $\Theta(n)$

Name	Alter
Peter Arbeitsloser	26
Martyn Vorstand	38
Henryk Ingenieur	45
Kiki Unbekannt	28
Der Alte	92

Personen aus: Qualityland, Marc-Uwe Kling

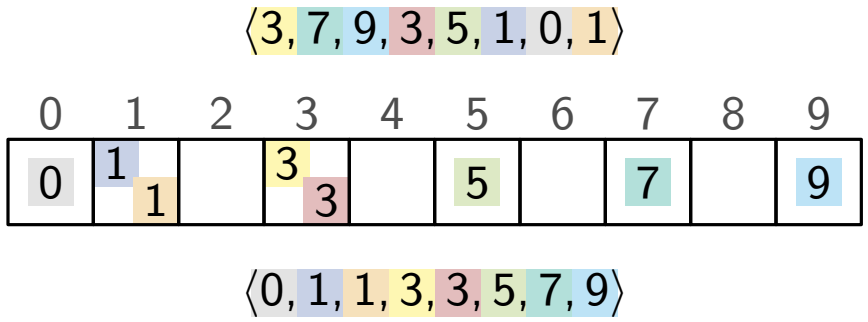
Beispiel 2: Knoten eines Graphen nach Grad (#Nachbarn)

- Eingabe: Graph mit n Knoten
- Grad jedes Knotens ist kleiner n
- Bucketsort hat Laufzeit $\Theta(n)$



Bucketsort: Implementierung

```
Bucketsort(input)
    m := largest key in the input + 1
    Array⟨Array⟩ buckets := array of size m of empty arrays
    for obj in input do
        buckets[obj.key].pushBack(obj)
    Array A := empty array
    for bucket in buckets do
        for obj in bucket do
            A.pushBack(obj)
    return A
```

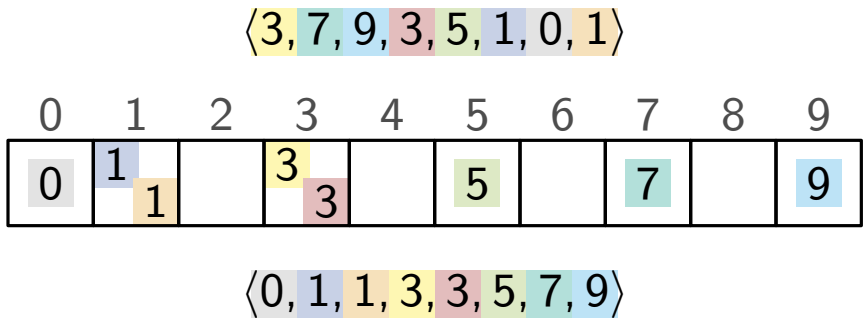


Bucketsort: Implementierung

```
Bucketsort(input)
    m := largest key in the input + 1
    Array⟨Array⟩ buckets := array of size m of empty arrays
    for obj in input do
        buckets[obj.key].pushBack(obj)
    Array A := empty array
    for bucket in buckets do                    // m buckets
        for obj in bucket do                    // n objects
            A.pushBack(obj)
    return A
```

Anmerkung

- verschachtelte Schleifen
  $\rightarrow$ Laufzeit $\Theta(m \cdot n)$?

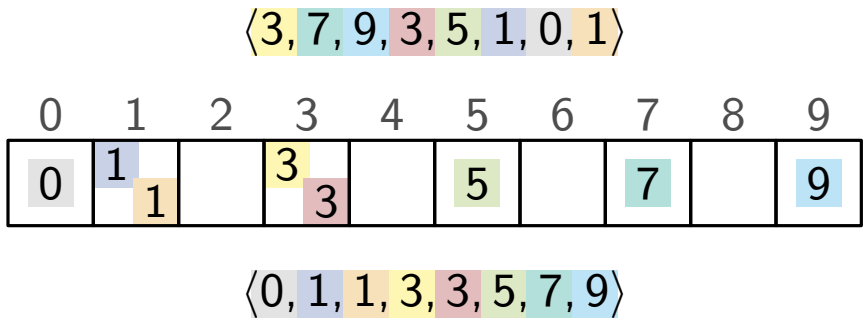


Bucketsort: Implementierung

```
Bucketsort(input)
    m := largest key in the input + 1
    Array⟨Array⟩ buckets := array of size m of empty arrays
    for obj in input do
        buckets[obj.key].pushBack(obj)
    Array A := empty array
    for bucket in buckets do                    // m buckets
        for obj in bucket do                    // n objects
            A.pushBack(obj)
    return A
```

Anmerkung

- verschachtelte Schleifen
  $\rightarrow$ Laufzeit $\Theta(m \cdot n)$?
- jedes Objekt liegt in nur einem Bucket
- daher: $\Theta(m + n)$

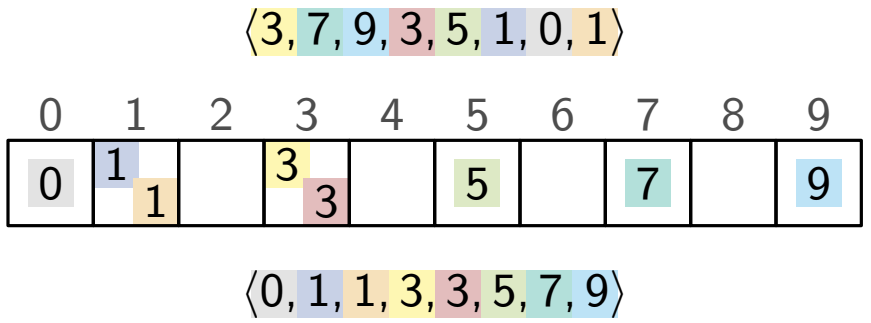


Bucketsort: Implementierung

```
Bucketsort(input)
    m := largest key in the input + 1
    Array⟨Array⟩ buckets := array of size m of empty arrays
    for obj in input do
        buckets[obj.key].pushBack(obj)
    Array A := empty array
    for bucket in buckets do                    // m buckets
        for obj in bucket do                    // n objects
            A.pushBack(obj)
    return A
```

Anmerkung

- verschachtelte Schleifen
  $\rightarrow$ Laufzeit $\Theta(m \cdot n)$?
- jedes Objekt liegt in nur einem Bucket
- daher: $\Theta(m + n)$



Theorem
Mit Bucketsort könne wir n natürliche Zahlen in $\Theta(n)$ Zeit sortieren, wenn die Größe der Zahlen in $O(n)$ liegt.

Stabiles Sortieren und lexikographische Ordnungen

Beobachtung: Bucketsort ist stabil
Wenn Objekt X in der Eingabe vor Objekt Y steht und $X.\mathbf{key} = Y.\mathbf{key}$, dann steht X in der Ausgabe vor Y.

```
Bucketsort(input)
    m := largest key in the input + 1
    Array⟨Array⟩ buckets := array of size m of empty arrays
    for obj in input do
        buckets[obj.key].pushBack(obj)
    Array A := empty array
    for bucket in buckets do
        for obj in bucket do
            A.pushBack(Obj)
    return A
```

Stabiles Sortieren und lexikographische Ordnungen

Beobachtung: Bucketsort ist stabil
Wenn Objekt X in der Eingabe vor Objekt Y steht und $X.\mathbf{key} = Y.\mathbf{key}$, dann steht X in der Ausgabe vor Y.

Lexikographische Ordnung auf den Nationen

- ordne nach Anzahl Goldmedaillen
- bei gleicher Anzahl Gold: nach Silbermedaillen
- bei gleicher Anzahl Gold und Silber: nach Bronze

Medaillenspiegel Olympia 2018

Nation	Gold	Silber	Bronze
NOR	14	14	11
GER	14	10	7
CAN	11	8	10
USA	9	8	6
SUI	5	6	4
FRA	5	4	6
GBR	1	0	4
POL	1	0	1
UKR	1	0	0

app.klicker.uzh.ch/join/algo1

Stabiles Sortieren und lexikographische Ordnungen

Beobachtung: Bucketsort ist stabil
Wenn Objekt X in der Eingabe vor Objekt Y steht und $X.\mathbf{key} = Y.\mathbf{key}$, dann steht X in der Ausgabe vor Y.

Lexikographische Ordnung auf den Nationen

- ordne nach Anzahl Goldmedaillen
- bei gleicher Anzahl Gold: nach Silbermedaillen
- bei gleicher Anzahl Gold und Silber: nach Bronze

Wie können wir mit Bucketsort lexikographisch sortieren?

Medaillenspiegel Olympia 2018

Nation	Gold	Silber	Bronze
NOR	14	14	11
GER	14	10	7
CAN	11	8	10
USA	9	8	6
SUI	5	6	4
FRA	5	4	6
GBR	1	0	4
POL	1	0	1
UKR	1	0	0

Stabiles Sortieren und lexikographische Ordnungen

Beobachtung: Bucketsort ist stabil
Wenn Objekt X in der Eingabe vor Objekt Y steht und $X.\mathbf{key} = Y.\mathbf{key}$, dann steht X in der Ausgabe vor Y.

Lexikographische Ordnung auf den Nationen

- ordne nach Anzahl Goldmedaillen
- bei gleicher Anzahl Gold: nach Silbermedaillen
- bei gleicher Anzahl Gold und Silber: nach Bronze

Sortieralgorithmus

- sortiere drei Mal, jeweils mit stabilem Sortieralgorithmus: erst nach Bronze, dann nach Silber, dann nach Gold
- letzte Sortierung nach Gold: mehr Gold $\Rightarrow$ weiter oben
- bei gleicher Anzahl Gold: vorherige Sortierung nach (Silber, Bronze) bleibt erhalten

Medaillenspiegel Olympia 2018

Nation	Gold	Silber	Bronze
NOR	14	14	11
GER	14	10	7
CAN	11	8	10
USA	9	8	6
SUI	5	6	4
FRA	5	4	6
GBR	1	0	4
POL	1	0	1
UKR	1	0	0

Und was ist mit größeren Zahlen?

Problem

- die Zahlen sind möglicherweise signifikant größer als die Anzahl zu sortierender Objekte
- zu viele Buckets $\rightarrow$ kostet Speicher und Laufzeit

Name	Einkommen
Peter Arbeitsloser	35 211
Martyn Vorstand	783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	46 823
Der Alte	46 129

Personen aus: Qualityland, Marc-Uwe Kling

Und was ist mit größeren Zahlen?

Problem

- die Zahlen sind möglicherweise signifikant größer als die Anzahl zu sortierender Objekte
- zu viele Buckets $\rightarrow$ kostet Speicher und Laufzeit

Idee für das Beispiel: dreistufiges Verfahren

- ordne lexikographisch nach (Ziffern 1–3, Ziffern 4–6, Ziffern 7–9)
- vorher: Bucketsort mit 10^9 Buckets
- jetzt: 3 Mal Bucketsort mit 10^3 Buckets

Name	Einkommen
Peter Arbeitsloser	35 211
Martyn Vorstand	783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	46 823
Der Alte	46 129

Personen aus: Qualityland, Marc-Uwe Kling

Name	Einkommen
Peter Arbeitsloser	000 035 211
Martyn Vorstand	000 783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	000 046 823
Der Alte	000 046 129

Personen aus: Qualityland, Marc-Uwe Kling

Und was ist mit größeren Zahlen?

Problem

- die Zahlen sind möglicherweise signifikant größer als die Anzahl zu sortierender Objekte
- zu viele Buckets $\rightarrow$ kostet Speicher und Laufzeit

Idee für das Beispiel: dreistufiges Verfahren

- ordne lexikographisch nach (Ziffern 1–3, Ziffern 4–6, Ziffern 7–9)
- vorher: Bucketsort mit 10^9 Buckets
- jetzt: 3 Mal Bucketsort mit 10^3 Buckets

Allgemein: asymptotische Verbesserung?

- Eingabe: n Zahlen aus $[0, n^c)$
- Anzahl Ziffern: $\log(n^c) = c \cdot \log n$

Name	Einkommen
Peter Arbeitsloser	35 211
Martyn Vorstand	783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	46 823
Der Alte	46 129

Personen aus: Qualityland, Marc-Uwe Kling

Name	Einkommen
Peter Arbeitsloser	000 035 211
Martyn Vorstand	000 783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	000 046 823
Der Alte	000 046 129

Personen aus: Qualityland, Marc-Uwe Kling

app.klicker.uzh.ch/join/algo1

Und was ist mit größeren Zahlen?

Problem

- die Zahlen sind möglicherweise signifikant größer als die Anzahl zu sortierender Objekte
- zu viele Buckets $\rightarrow$ kostet Speicher und Laufzeit

Name	Einkommen
Peter Arbeitsloser	35 211
Martyn Vorstand	783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	46 823
Der Alte	46 129

Personen aus: Qualityland, Marc-Uwe Kling

Idee für das Beispiel: dreistufiges Verfahren

- ordne lexikographisch nach (Ziffern 1–3, Ziffern 4–6, Ziffern 7–9)
- vorher: Bucketsort mit 10^9 Buckets
- jetzt: 3 Mal Bucketsort mit 10^3 Buckets

Name	Einkommen
Peter Arbeitsloser	000 035 211
Martyn Vorstand	000 783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	000 046 823
Der Alte	000 046 129

Personen aus: Qualityland, Marc-Uwe Kling

Allgemein: asymptotische Verbesserung?

- Eingabe: n Zahlen aus $[0, n^c)$
- Anzahl Ziffern: $\log(n^c) = c \cdot \log n$

Bei Aufspaltung in c Blöcke mit je $\log n$ Ziffern:
Wie groß werden die Zahlen pro Block? Welche Laufzeit erhalten wir?

Und was ist mit größeren Zahlen?

Problem

- die Zahlen sind möglicherweise signifikant größer als die Anzahl zu sortierender Objekte
- zu viele Buckets $\rightarrow$ kostet Speicher und Laufzeit

Name	Einkommen
Peter Arbeitsloser	35 211
Martyn Vorstand	783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	46 823
Der Alte	46 129

Personen aus: Qualityland, Marc-Uwe Kling

Idee für das Beispiel: dreistufiges Verfahren

- ordne lexikographisch nach (Ziffern 1–3, Ziffern 4–6, Ziffern 7–9)
- vorher: Bucketsort mit 10^9 Buckets
- jetzt: 3 Mal Bucketsort mit 10^3 Buckets

Name	Einkommen
Peter Arbeitsloser	000 035 211
Martyn Vorstand	000 783 491
Henryk Ingenieur	123 456 789
Kiki Unbekannt	000 046 823
Der Alte	000 046 129

Personen aus: Qualityland, Marc-Uwe Kling

Allgemein: asymptotische Verbesserung?

- Eingabe: n Zahlen aus $[0, n^c)$
- Anzahl Ziffern: $\log(n^c) = c \cdot \log n$
- aufspalten in c Blöcke mit je $\log n$ Ziffern $\rightarrow$ Zahlen im Intervall $[0, n)$
- also: c mal Bucketsort mit Laufzeit je $\Theta(n) \Rightarrow$ Gesamtlaufzeit $\Theta(c \cdot n)$

Sortieren polynomiell großer Zahlen

Gerade gesehen

- n Zahlen aus dem Intervall $[0, n^c) \to c$ mal Bucketsort mit Zahlen aus dem Intervall $[0, n)$
- Laufzeit $\Theta(cn) \to \Theta(n)$, wenn c konstant
- den Algorithmus nennt man auch **Radixsort** (bzw. LSD Radixsort)

Theorem
Mit Radixsort können wir n natürliche Zahlen in $\Theta(n)$ Zeit sortieren, wenn die Größe der Zahlen in $n^{O(1)}$ liegt, also polynomiell in n ist.

Sortieren polynomiell großer Zahlen

Gerade gesehen

- n Zahlen aus dem Intervall $[0, n^c) \rightarrow c$ mal Bucketsort mit Zahlen aus dem Intervall $[0, n)$
- Laufzeit $\Theta(cn) \rightarrow \Theta(n)$, wenn c konstant
- den Algorithmus nennt man auch **Radixsort** (bzw. LSD Radixsort)

Theorem
Mit Radixsort können wir n natürliche Zahlen in $\Theta(n)$ Zeit sortieren, wenn die Größe der Zahlen in $n^{O(1)}$ liegt, also polynomiell in n ist.

Was jetzt?

- explizitere Formulierung des Algorihtmus $\rightarrow$ erster Schritt Richtung Implementierung
- Korrektheit und Laufzeit nochmal klar machen
- Erklärung des Namens

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$
- sortiere die Zahlen erst bzgl. x_0
- sortiere dann stabil bzgl. x_1
- dann bzgl. x_2 und so weiter

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$
- sortiere die Zahlen erst bzgl. x_0
- sortiere dann stabil bzgl. x_1
- dann bzgl. x_2 und so weiter

Laufzeit

- $\Theta(k \cdot (n + b))$

k mal sortieren von n Zahlen der Größe b

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$
- sortiere die Zahlen erst bzgl. x_0
- sortiere dann stabil bzgl. x_1
- dann bzgl. x_2 und so weiter

Laufzeit

- $\Theta(k \cdot (n + b))$

k mal sortieren von n Zahlen der Größe b

- für $b = n$ gilt $k = \lceil c \rceil \Rightarrow \Theta(n)$

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$
- sortiere die Zahlen erst bzgl. x_0
- sortiere dann stabil bzgl. x_1
- dann bzgl. x_2 und so weiter

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Laufzeit

- $\Theta(k \cdot (n + b))$

k mal sortieren von n Zahlen der Größe b

- für $b = n$ gilt $k = \lceil c \rceil \Rightarrow \Theta(n)$

Korrektheit

- nach Sortierung bzgl. x_i gilt: Zahlen sind sortiert bzgl. $x_i \cdot b^i + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$
- Beweis: Induktion über i

Radixsort

Algorithmus

- Eingabe: n ganze Zahlen aus $[0, n^c)$
- Darstellung jeder Zahl x zu einer Basis b:
  $x_k \cdot b^k + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$ $\quad (k = \lceil \log_b n^c \rceil)$
- sortiere die Zahlen erst bzgl. x_0
- sortiere dann stabil bzgl. x_1
- dann bzgl. x_2 und so weiter

Beispiel von vorhin: $b = 1000$

$$000\,035\,211 = 000 \cdot 1000^2 + 035 \cdot 1000^1 + 211 \cdot 1000^0$$
$$000\,783\,491 = 000 \cdot 1000^2 + 783 \cdot 1000^1 + 491 \cdot 1000^0$$
$$123\,456\,789 = 123 \cdot 1000^2 + 456 \cdot 1000^1 + 789 \cdot 1000^0$$

Laufzeit

- $\Theta(k \cdot (n + b))$

k mal sortieren von n Zahlen der Größe b

- für $b = n$ gilt $k = \lceil c \rceil \Rightarrow \Theta(n)$

Nebenbemerkung

- die Basis in einem Zahlensystem nennt man auch **Radix** (binär $\rightarrow$ 2, dezimal $\rightarrow$ 10, etc.)
- daher der Name **Radixsort**

Korrektheit

- nach Sortierung bzgl. x_i gilt: Zahlen sind sortiert bzgl. $x_i \cdot b^i + \cdots + x_1 \cdot b^1 + x_0 \cdot b^0$
- Beweis: Induktion über i

Geht das alles wirklich so schnell?

Repräsentation der Zahlen im Computer

- Zahlen die bis zu n^c groß sind $\rightarrow \log(n^c) = c \log n$ Bits
- wir machen verschiedene Dinge mit den Zahlen:
  - Vergleichen
  - Speicherzellen adressieren
  - rechnen (z.B. $x_0 := x \bmod n$ um die letzten $\log n$ Bits von x zu erhalten)
- bisherige Annahme: das geht alles in konstanter Zeit

Geht das alles wirklich so schnell?

Repräsentation der Zahlen im Computer

- Zahlen die bis zu n^c groß sind $\rightarrow \log(n^c) = c \log n$ Bits
- wir machen verschiedene Dinge mit den Zahlen:
  - Vergleichen
  - Speicherzellen adressieren
  - rechnen (z.B. $x_0 := x \bmod n$ um die letzten $\log n$ Bits von x zu erhalten)
- bisherige Annahme: das geht alles in konstanter Zeit

Fragen über Fragen

- Sollte man nicht eher von $\Theta(\log n)$ Zeit pro Operation ausgehen?
- Hat Radixsort damit doch Laufzeit $\Theta(n \log n)$?
- Sorgen die $\Theta(n \log n)$ Vergleiche bei Merge- und Quicksort dann für Laufzeit $\Theta(n \log^2 n)$?

Hier übliche Notation

- $\log^2 n = \log(n) \cdot \log(n)$
- $\log \log n = \log(\log(n))$

Geht das alles wirklich so schnell?

Repräsentation der Zahlen im Computer

- Zahlen die bis zu n^c groß sind $\rightarrow \log(n^c) = c \log n$ Bits
- wir machen verschiedene Dinge mit den Zahlen:
  - Vergleichen
  - Speicherzellen adressieren
  - rechnen (z.B. $x_0 := x \bmod n$ um die letzten $\log n$ Bits von x zu erhalten)
- bisherige Annahme: das geht alles in konstanter Zeit

Fragen über Fragen

- Sollte man nicht eher von $\Theta(\log n)$ Zeit pro Operation ausgehen?
- Hat Radixsort damit doch Laufzeit $\Theta(n \log n)$?
- Sorgen die $\Theta(n \log n)$ Vergleiche bei Merge- und Quicksort dann für Laufzeit $\Theta(n \log^2 n)$?

Hier übliche Notation

- $\log^2 n = \log(n) \cdot \log(n)$
- $\log \log n = \log(\log(n))$

Antwort

- nein (zumindest nicht im Word-RAM Modell)

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$

Welche Annahme treffen wir für w?

- Möglichkeit 1: w konstant (typischer Computer: $w = 64$)
  - nur endlich viel Speicher adressierbar $\rightarrow$ theoretisch unschön (endlicher Automat)
  - arithmetische Operationen in $\Theta(\log n)$ $\rightarrow$ unrealistisch pessimistisch ($2^{64} > 1{,}8 \cdot 10^{19}$)

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$

Welche Annahme treffen wir für w?

- Möglichkeit 1: w konstant (typischer Computer: $w = 64$)
  - nur endlich viel Speicher adressierbar $\rightarrow$ theoretisch unschön (endlicher Automat)
  - arithmetische Operationen in $\Theta(\log n) \rightarrow$ unrealistisch pessimistisch ($2^{64} > 1{,}8 \cdot 10^{19}$)
- Möglichkeit 2: w unbeschränkt $\rightarrow$ zu optimistisch

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$

Welche Annahme treffen wir für w?

- Möglichkeit 1: w konstant (typischer Computer: $w = 64$)
  - nur endlich viel Speicher adressierbar $\rightarrow$ theoretisch unschön (endlicher Automat)
  - arithmetische Operationen in $\Theta(\log n) \rightarrow$ unrealistisch pessimistisch ($2^{64} > 1{,}8 \cdot 10^{19}$)
- Möglichkeit 2: w unbeschränkt $\rightarrow$ zu optimistisch
- Möglichkeit 3: $w \geq \log n$
  - genug Bits um alle benutzten Speicherzellen zu adressieren
  - nicht zu mächtig: $w \geq \log n$ ist realistisch für echte Computer

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$
  - $w \geq \log n$

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$
  - $w \geq \log n$

Implikationen fürs Sortieren polynomiell großer Zahlen

- Vergleiche und Operationen laufen in $O(1)$
- damit: Laufzeiten $\Theta(n)$ für Radixsort und $\Theta(n \log n)$ für Merge- oder Quicksort

Exkurs: Berechnungsmodell

Berechnungsmodell (informell)

- RAM (random-access machine): $O(1)$-Speicherzugriff mittels Adresse
- word RAM
  - jeder Speicherplatz hält ein **Wort** bestehend aus w Bits $\rightarrow$ ganze Zahlen in $[0, 2^w)$
  - arithmetische Operationen und Vergleiche auf Worten in $\Theta(1)$
  - $w \geq \log n$

Implikationen fürs Sortieren polynomiell großer Zahlen

- Vergleiche und Operationen laufen in $O(1)$
- damit: Laufzeiten $\Theta(n)$ für Radixsort und $\Theta(n \log n)$ für Merge- oder Quicksort

Sortieren noch größerer Zahlen

- Annahme: jede Zahl ist weiterhin durch ein Wort repräsentierbar, aber $w \in \omega(\log n)$
- weiterhin $O(1)$-Operationen, aber Radixsort braucht zu viele Sortierschritte
- geht in erwartet $O(n\sqrt{\log \log n})$ bzw. deterministisch in $O(n \log \log n)$

Zusammenfassung

Sortieren polynomiell großer Zahlen

- nutze aus, dass man mit Zahlen in $O(1)$ Zeit Speicherzellen adressieren kann (Arrays)
- $\Theta(n)$-Sortieren von linear großen Zahlen: Bucketsort
- $\Theta(n)$-Sortieren von polynomiell großen Zahlen: Aufspalten jeder Zahl in konstant viele linear große Zahlen $\rightarrow$ Radixsort
- in dem Kontext kennen gelernt: lexikographische Sortierung

Zusammenfassung

Sortieren polynomiell großer Zahlen

- nutze aus, dass man mit Zahlen in $O(1)$ Zeit Speicherzellen adressieren kann (Arrays)
- $\Theta(n)$-Sortieren von linear großen Zahlen: Bucketsort
- $\Theta(n)$-Sortieren von polynomiell großen Zahlen: Aufspalten jeder Zahl in konstant viele linear große Zahlen $\rightarrow$ Radixsort
- in dem Kontext kennen gelernt: lexikographische Sortierung

Exkurs: Berechnungsmodell word RAM

- verhält sich bei der Algorithmenanalyse meist wie gewünscht
- formalere Einführung von Berechnungsmodellen: TGI (nächstes Semester)