Karlsruher Institut für Technologie

Tutorium Algorithmen 1

04 · Sortieren II · 13.5.2024
Peter Bohner   Tutorium 3

Was bisher geschah

- mergeSort: Teile Folge in zwei Teile, sortiere beide, füge sie zusammen ($O(n \log n)$)
  - Mergesort ist stabil (ändert die Reihenfolge gleicher Elemente nicht)
- quickSort: Teile Folge in „kleine“ und „große“ Elemente, verschieb sie in die jeweilige Hälfte, sortiere sie seperat ($O(n^2)$)

$\Rightarrow$ Bei zufälligen Pivotelement in erwartet $O(n \log n)$

- Alle Algorithmen die wir kennen benutzen die gleiche Idee (Einzelne Elemente vergleichen)
- VL: Sortieren, bei dem einzige Weise, Informationen über Elemente zu erhalten, diese paarweise zu vergleichen ist, geht nur in $\Omega(n \cdot \log(n))$

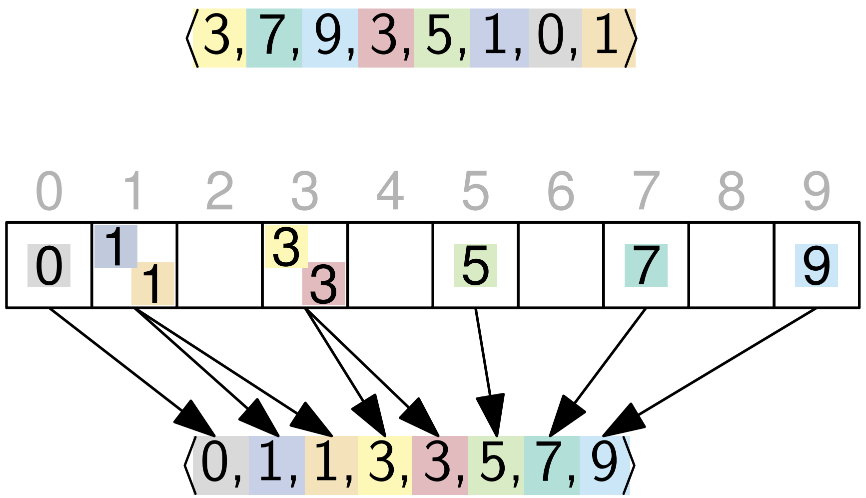
$\Rightarrow$ Schneller als Mergesort und Quicksort geht es nicht

> Wenn wir mehr Anforderungen an die Eingaben stellen bekommen wir evtl. schnellere Algorithmen

Peter Bohner - peter.bohner@student.kit.edu

Bucketsort

- Eingabe: n Zahlen aus $\mathbb{Z}_m$
- erstelle Array B mit m Buckets
- für jede Zahl x: speichere x in $B[x]$
- sammle Zahlen aus B auf



- stabil? Ja!
- Laufzeit?
  - $\Theta(n + m)$ (Array der Länge m und n Zahlen einfügen) bzw. $\Theta(n)$, wenn $m \in \mathcal{O}(n)$
- Warum kein Widerspruch zu unterer Schranke $\Theta(n \log(n))$?
  - durch Zahlen mehr Möglichkeiten als Vergleiche

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: Knoten sortieren

Gegeben ist ein einfacher ungerichteter Graph $G = (\mathbb{Z}_n, E)$.
Entwirf einen Algorithmus, der die Knotenmenge $V = \mathbb{Z}_n$ nach Knotengrad sortiert.
Welche Laufzeit hat dein Algorithmus?

Beobachtung: Die ein Knoten kann mindestens Grad 0 und maximal Grad $n - 1$ haben

- Lege ein Array A der Länge n mit 0 initialisiert an
- Iteriere über Knotenmenge und trage Knoten v_i in $A[deg(v_i)]$ ein
- Iteriere über A, um alle Knoten in einem Array ordentlich aufzureihen

Laufzeit (m $:= |E|$)

- Array anlegen: $\Theta(n)$
- Knotengrade berechnen (falls nicht gespeichert): $\Theta(n + m)$

$\Rightarrow$ Insgesamt: $\Theta(n + m)$ falls Knotegrade nicht bekannt ($\Theta(n)$ falls doch)

Peter Bohner - peter.bohner@student.kit.edu

LSD-Radixsort

Least Significant Digit Radixsort

- Wir wollen n Zahlen aus $\mathbb{Z}_m$ sortieren
- mit Bucketsort: $n = 9$, $m = 300 \Rightarrow$ langsam
- Beobachtung: Bucketsort ist <u>stabil</u>
- Idee: wir sortieren nach Ziffern, also $log_{10} m = 2, \cdots \approx 3$ mal
- LSD beginnt rechts (least significant bit)
- Auch andere Aufteilungen möglich (x Ziffern gruppiert)
  - Mit der Methode aus der Vorlesung lassen sich n Zahlen der Größe in $n^{\mathcal{O}(1)}$ in $\Theta(n)$ sortieren

Peter Bohner - peter.bohner@student.kit.edu

LSD-Radixsort

- Eingabe: n Zahlen aus $\mathbb{Z}_m$
- für jede Ziffer, beginnend mit der kleinsten:
  Bucketsort mit aktueller Ziffer als key

Sortiert die Folge $\langle 403, 6, 417, 18, 256, 350, 238, 999, 67 \rangle$ mit LSD-Radixsort.

LSD-Radixsort

403	006	417	018	256	350	238	999	067

0	1	2	3	4	5	6	7	8	9
350			403			006 256	417 067	018 238	999

350	403	006	256	417	067	018	238	999

0	1	2	3	4	5	6	7	8	9
403 006	417 018		238		350 256	067			999

403	006	417	018	238	350	256	067	999

0	1	2	3	4	5	6	7	8	9
006 018 067		238 256	350	403 417					999

006	018	067	238	256	350	403	417	999

MSD-Radixsort

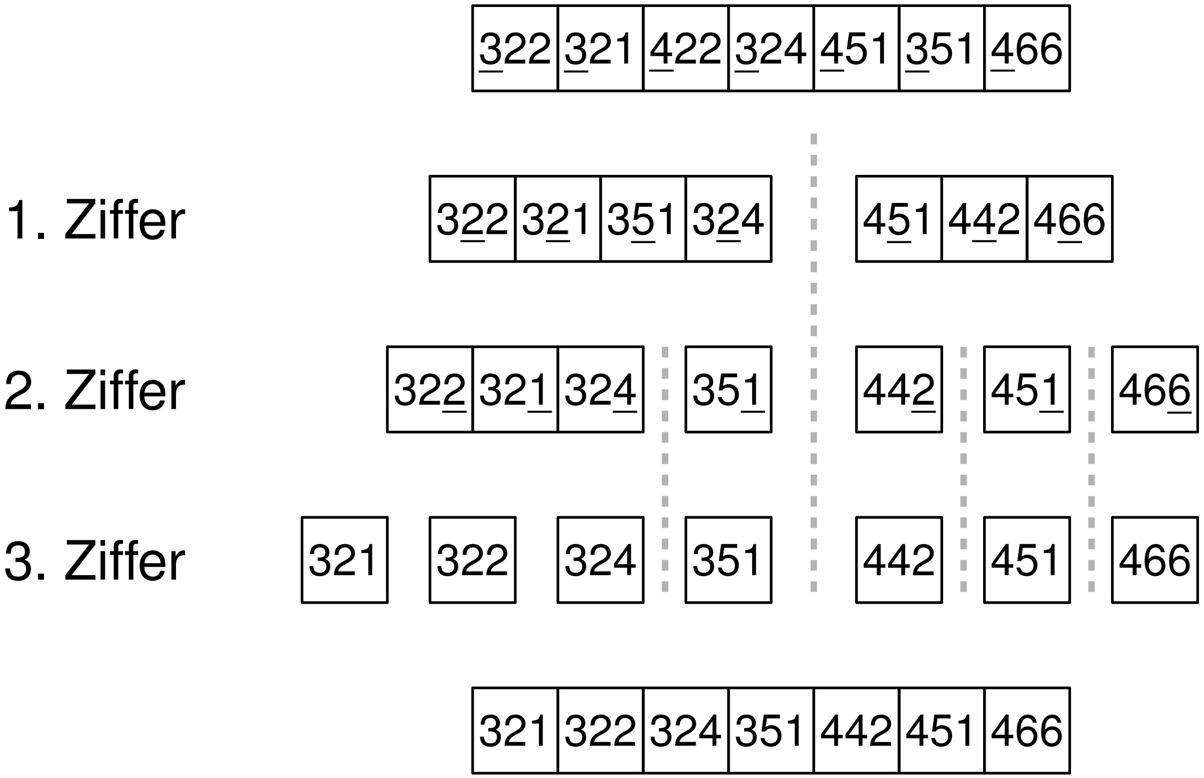
- Anstatt mit der niedrigsten können wir auch mit der höchsten Stelle anfangen
- Was geht schief wenn wir einfach wieder x mal Bucketsort anwenden?
  - Sortierung nach der 2ten Stelle macht die Sortierung nach der 1ten Stelle kaputt

Lösung:

- Sortiere nicht das ganze Array nochmal sondern nur im jeweiligen Bucket

Um Platz zu sparen, kann man trotzdem im gleichen Array bucketsorten, indem man sich für Zahlen merkt, aus welchem Bucket sie gekommen sind.

MSD-Radixsort Beispiel



MSD-Radixsort

Sortiert die Folge $\langle 403, 6, 417, 18, 256, 350, 238, 999, 67 \rangle$ mit MSD-Radixsort.

403	006	417	018	256	350	238	999	067

1. Ziffer

[006 | 018 | 067] [256 | 238] [350] [403 | 417] [999]

2. Ziffer

[006] [018] [067] [238] [256] [350] [403] [417] [999]

006	018	067	238	256	350	403	417	999

Peter Bohner - peter.bohner@student.kit.edu

Keys

Karlsruher Institut für Technologie

Was machen wir mit Objekten die keine Zahlen sind?

- Idee: Bilde Objekt auf eine Zahl ab (Schlüssel bzw **key**)
- Elemente werden anhand von $\textbf{key}(e)$ sortiert
- $\textbf{key}(e) \in \mathbb{N}$
- Vorteil: asymptotisch schneller
- Nachteil: nicht immer anwendbar

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: k-häufigste Buchstaben

Gegeben sei ein String s der Länge n, $s \in$ [a-zA-Z]$*$. Beschreibe einen Algorithmus, der in Linearzeit die k-häufigsten Buchstaben im String findet ($k \in \Theta(1)$).

- Definiere **key** Funktion, die alle Buchstaben auf die Zahlen 0 bis 51 mapped
- Stelle ein Array A der Länge 52 auf
- Zähle mit Bucketsort alle vorkommenden Buchstaben in ihren Buckets
- Erstelle Liste L mit fester Länge k
- Iteriere über A und füge das Tupel (Zeichen, i) in L ein, falls ein Eintrag mit Häufigkeit $j < i$ gefunden wird

Laufzeit: $\Theta(n) + \mathcal{O}(52 \cdot k) \Rightarrow \Theta(n)$

Peter Bohner - peter.bohner@student.kit.edu

Vergleichbasiertes vs. ganzzahliges Sortieren

- Vorteile ganzzahlig
  - Asymptotisch schneller
- Vorteile vergleichsbasiert
  - weniger Annahmen (z. B. wichtig für Algorithmenbibliotheken)
  - robust gegen beliebige Eingabeverteilungen
  - Cache-Effzienz weniger schwierig
  - bei langen Schlüsseln oft schneller

Peter Bohner - peter.bohner@student.kit.edu

Viele Sortieralgorithmen

Quicksort	erwartet $\mathcal{O}(n \log n)$
Mergesort	$\Theta(n \log n)$
Bucketsort	$\Theta(n + m) + \Theta(m)$ Space
LSD-Radixsort	$\Theta(c \cdot n)$
MSD-Radisort	$\Theta(c \cdot n)$

- m = größte vorkommende Zahl
- c = Anzahl Keys

Noch schneller sortieren als $\mathcal{O}(n)$? $\Rightarrow$ `https://en.wikipedia.org/wiki/Bitonic_sorter`

Aufgabe: Noch mal Sortieren

Gegeben ist eine Datenstruktur der Länge n, in der jede Zahl aus $\mathbb{Z}_n = \{0, \ldots, n-1\}$ genau einmal vorkommt. Die Datenstruktur unterstützt nur die Methode **swap**(i, j), welche in $\mathcal{O}(1)$ Zeit die Elemente an den Indizes i und j vertauscht.

Entwerft einen Algorithmus der diese Datenstruktur möglichst schnell, stabil und in-place sortiert.

Peter Bohner - peter.bohner@student.kit.edu

Lösung: Noch mal Sortieren

Algorithmus

- Wir nutzen aus, dass Element i in sortierter Folge an Position i steht
- Iteriere über die Datenstruktur
- Wen wir ein falsch eingeordnetes Element finden, dann swappe es mit **swap** and die richtige Stelle und das dortige hier her

Laufzeit

- Durch Datenstruktur iterieren: $\Theta(n)$
- Bei jedem **swap** platzieren wir ein Element korrekt $\Rightarrow$ höchstens n swaps

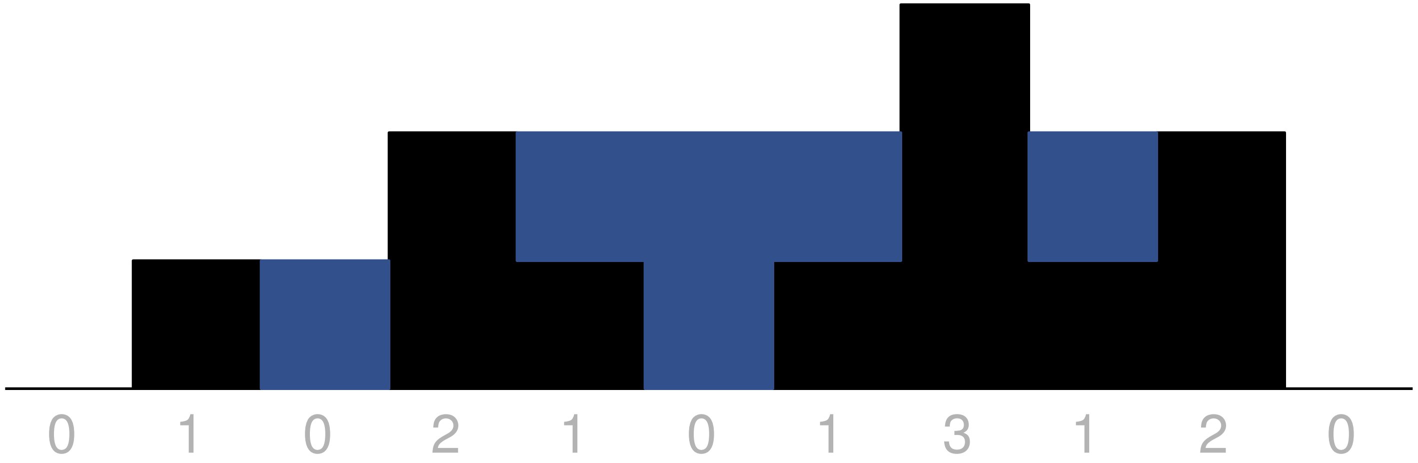
$\Rightarrow$Insgesamt: $\Theta(n)$

Peter Bohner - peter.bohner@student.kit.edu

Bonusaufgaben

Aufgabe: Regenwasser

Gegeben ist eine Heightmap in Form eines Arrays (Bsp. $\langle 0, 1, 0, 2, 1, 0, 1, 3, 1, 2, 0 \rangle$)
Die Zahlen des Arrays beschreiben die Höhe von Balken mit Breite 1. Zwischen den Balken sind *Löcher*, welche regnendes Wasser einfangen können.
Finde einen Algorithmus, der in Linearzeit die Menge an gesammeltem Regenwasser berechnet.



Lösung: Regenwasser

Algorithmus

Wir nutzen eine 2-Pointer Strategie

- Einen *left*-Pointer und einen *right*-Pointer, so wie Variablen $max_{x \in \{left,\, right\}}$ und *sum*
- Starte *left* auf Eintrag 0 mit max_{left} auf $A[0]$ und *right* auf Eintrag $n-1$ mit max_{right} auf $A[n-1]$
- Wenn $max_{left} \leq max_{right}$ bewege linken Pointer um eins nach rechts passe ggf. max_{left} an und addiere $max_{left} - A[left]$ auf *sum*
- Wenn $max_{left} > max_{right}$ bewege rechten Pointer nach links passe ggf. max_{right} an addiere $max_{right} - A[right]$ auf *sum*
- Sobald sich die Pointer in der Mitte treffen sind wir fertig

Peter Bohner - peter.bohner@student.kit.edu

Lösung: Regenwasser

Korrektheit

- Wenn $max_{left} \leq max_{right}$ gilt, so wissen wir, dass rechts von unserem aktuellen Pointer auf jeden Fall ein Balken mit ausreichender Höhe existiert, sodass Wasser eingefangen werden kann
- Gleiches gilt für die Annäherung von der rechten Seite
- Durch max_{left} wissen wir auch, wie hoch das Wasser links von unserem Pointer maximal stehen kann
- Das heißt in jedem Schritt berechnen wir den höchsten möglichen Wasserstand

Peter Bohner - peter.bohner@student.kit.edu

Aufgabe: Zaun Streichen

In dieser Aufgabe soll ein in Abschnitte unterteilter Zaun farbig gestrichen werden.

- Der Zaun ist in n Abschnitte $1, \ldots n$ unterteilt
- Jeder Abschnitt i hat eine Länge $L_i \in \mathbb{N}_+$
- Es stehen k Farben zur Verfügung
- Jeder Abschnitt soll in genau einer der Farben gestrichen werden
- Für jede Farbe soll gelten: Alle in dieser Farbe gestrichenen Abschnitte liegen nebeneinander
- Das Streichen von Abschnitt i braucht L_i Zeit
- Mit allen k Farben kann gleichzeitig an verschiedenen Stellen gestrichen werden
  (aber pro Farbe nur an einer Stelle gleichzeitig)

Finde einen möglichst effizienten Algorithmus, der die mindestens benötigte Zeit bestimmt, um den kompletten Zaun zu streichen. Bestimme dessen Laufzeit.

Hinweis: Binäre Suche

Peter Bohner - peter.bohner@student.kit.edu

Lösung: Zaun Streichen

Idee: Binäre Suche

- Intervall, in dem sich die Lösung (min. benötigte Zeit) befindet bekannt:
  - Können nicht schneller sein als der größte Abschnitt benötigt: mindestens $\max_{i \in \{1 \ldots n\}} L_i$
  - Brauchen höchstens so lange wie man mit nur einer Farbe brauchen würde: höchstens $\sum_{i=1}^{n} L_i$
- Außerdem: Längen aller Zaunabschnitte in $\mathbb{N} \implies$ Lösung in $\mathbb{N}$
- Binäre Suche über die möglichen Lösungen
- Leicht zu überprüfen ob bestimmte Zeit ausreicht:
  - Bestimme wie viele ganze Abschnitte in der 1. Farbe in dieser Zeit gestrichen werden können: Addiere Längen der Abschnitte solange dabei Zeit nicht überschritten
  - Wiederhole beginnend beim ersten noch ungestrichenen Abschnitt für nächste Farbe
  - Falls noch Abschnitt(e) aber keine Farbe mehr übrig: Zeit reicht nicht aus
  - Falls alle gestrichen werden können: Zeit reicht aus
- Zeit reicht nicht $\implies$ Lösung muss größer sein
- Zeit reicht $\implies$ Lösung ist höchstens so groß

Lösung: Zaun Streichen

Karlsruher Institut für Technologie

- mit $m = \sum_{i=1}^{n} L_i$ bezeichnen wir die Gesamtlänge des Zaunes
- Zu Beginn: $m - \max_{i \in \{1 \dots n\}} L_i + 1 \in \mathcal{O}(m)$ mögliche Lösungen
  $\Rightarrow$ Anzahl der benötigten Schritte in $\mathcal{O}(\log m)$
- In jedem Schritt: höchstens n-maliges Addieren und Vergleichen $\Rightarrow$ Laufzeit pro Schritt in $\mathcal{O}(n)$
- Insgesamt: Laufzeit in $\mathcal{O}(n \log m)$

Peter Bohner - peter.bohner@student.kit.edu

Zusammenfassung

Was haben wir gemacht?

- Bucketsort
- LSD- und MSD-Radixsort

Worauf könnt ihr euch nächstes Mal freuen?

- Hashing
- Die ersten Graphalgorithmen

Peter Bohner - peter.bohner@student.kit.edu

Fragen?

Fragen!

Ende

`https://xkcd.com/1323/`

Peter Bohner - peter.bohner@student.kit.edu