ETH zürich

Departement Informatik

Programmieren und Problemlösen

Sortieren 2

Dennis Komm

Frühling 2021 – 15. April 2021

Stacks und Queues

Stacks und Queues

Bislang Zugriff auf beliebige Elemente in einer Liste mit eckigen Klammern

Stack

- Last-In First-Out
- Elemente können am Ende eingefügt werden
- Elemente können am selben Ende entnommen werden

Queue

- First-In First-Out
- Elemente können am Ende eingefügt werden
- Elemente können am Anfang entnommen werden

Queues

Queue – zwei Operationen

- `append(x)` fügt Element `x` an letzter Stelle ein
- `pop(0)` entfernt erstes Element und gibt es zurück
- In Python können Listen wie Queues genutzt werden

```
data = [1, 4, 5]
data.append(8)   <-- data = [1, 4, 5, 8]
data.pop(0)
data.pop(0)      <-- data = [5, 8]
```

Stacks

Stack – zwei Operationen

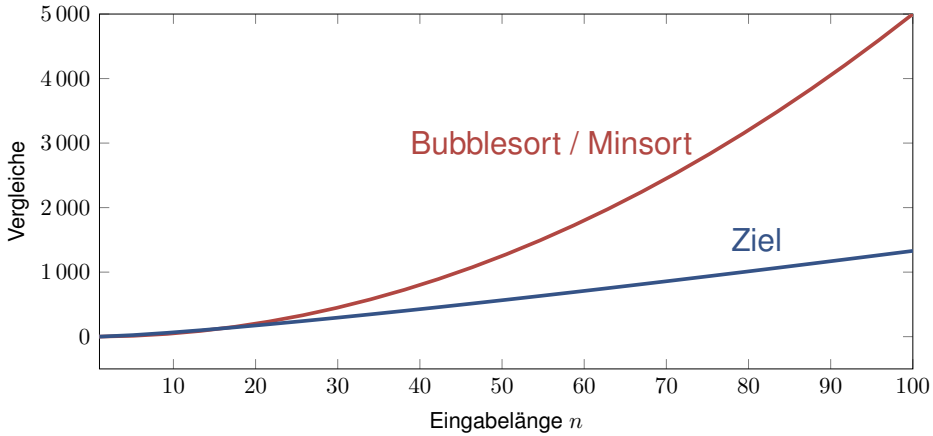
- `append(x)` fügt Element `x` an letzter Stelle ein
- `pop()` entfernt letztes Element und gibt es zurück
- In Python können Listen auch wie Stacks genutzt werden

```
data = [1, 4, 5]
data.append(8)   <-- data = [1, 4, 5, 8]
data.pop()
data.pop()       <-- data = [1, 4]
```


Sortieren 2

Mergesort

Zeitkomplexität von Bubblesort und Minsort



Wie kann man schneller sortieren?

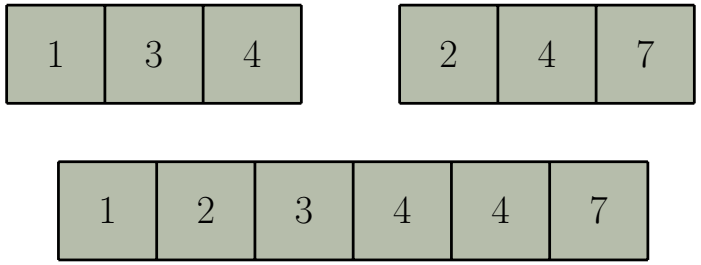
Idee

Zwei bereits sortierte Listen zusammenzufügen ist einfach

- Sortiere erst kleine Listen
- Füge diese zusammen
- Wiederhole

⇨ **Divide and Conquer**

Zusammenfügen sortierter Listen



Aufgabe – Zusammenfügen sortierter Listen

Schreiben Sie eine Funktion, die

- zwei sortierte Listen erhält
- eine sortierte Liste zurückgibt

Verwenden Sie zu diesem Zweck die Funktionen `pop(0)` und `append()`

Zusammenfügen sortierter Listen

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] > right[0]:
            result.append(right.pop(0))
        else:
            result.append(left.pop(0))
    return result + left + right
```

Solange nicht beide Listen leer sind

Füge kleineres der ersten Elemente hinten an

Eine der beiden gegebenen sortierten Listen kann noch Elemente enthalten

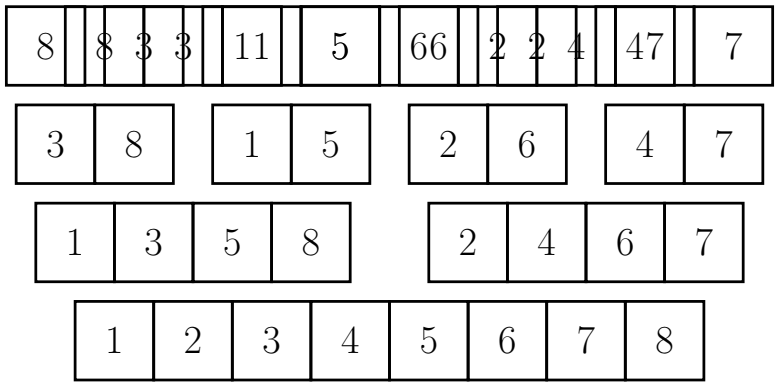
Mergesort

Divide and Conquer

Iterativ sortierte Listen zusammenfügen

- Zunächst „Listen“ der Länge 1 zu Listen der Länge 2 zusammenfügen
- Listen der Länge 2 zu Listen der Länge 4 zusammenfügen
- Listen der Länge 4 zu Listen der Länge 8 zusammenfügen
- Listen der Länge 8 zu Listen der Länge 16 zusammenfügen
- ...

Mergesort



Merge-Schritt

Einzelner Merge-Schritt

- Erhalte 2-dimensionale Liste, also Liste, die Listen erhält
- Mit Funktion `merge()` werden jeweils zwei aufeinanderfolgende Listen aus dieser Liste zusammenfügt
- Die letzte Liste wird nicht verändert, wenn die Anzahl Listen ungerade ist
- Das Resultat ist wieder eine 2-dimensionale Liste, die die zusammengefügten Listen enthält

Merge-Schritt

```
def mergestep(data):
    result = []
    while len(data) > 1:
        left = data.pop(0)
        right = data.pop(0)
        result.append(merge(left, right))
    return result + data
```

Solange mindestens zwei Listen existieren

Falls am Ende noch eine Liste übrig ist, füge sie an Fuge die ersten zwei Listen zusammen

Mergesort – Vollständiger Algorithmus

Vollständiger Algorithmus

- Eingabe als Liste `data` gegeben
- Mache aus jedem Element in `data` eine Liste mit einem Element
- Erhalte somit 2-dimensionale Liste
- Wende auf diese Liste immer wieder Funktion `mergestep()` an
- Am Ende hat Liste von Listen nur noch ein Element
- Dieses Element entspricht einer sortierten Liste

Mergesort – Vollständiger Algorithmus

```
def mergesort(data):
    result = []
    for item in data:
        result.append([item])
    while len(result) > 1:
        result = mergestep(result)
    return result[0]
```


Sortieren 2

Zeitkomplexität von Mergesort

Zeitkomplexität von Mergesort

Laufzeit von Mergesort ist proportional zu
Anzahl Merge-Schritte $\times$ Vergleiche pro Merge-Schritt

- Länge der sortierten Listen verdoppelt sich in jedem Merge-Schritt

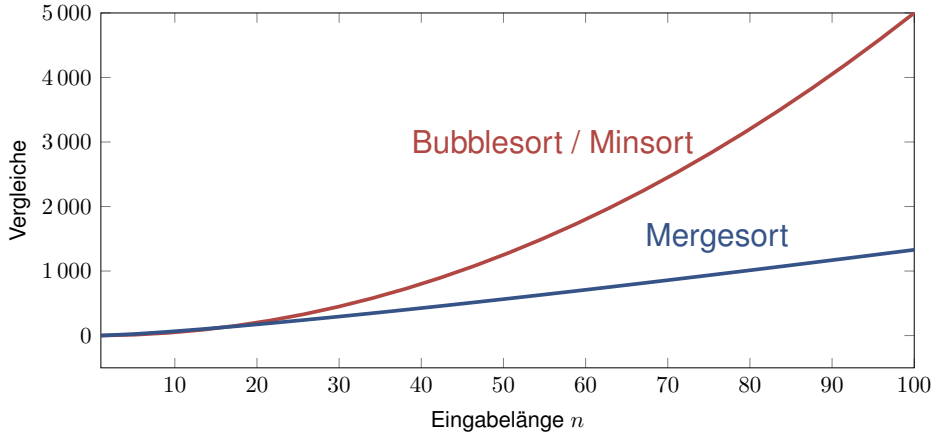
⇨ Ungefähr $\log_2 n$ Merge-Schritte für n Elemente

- In einem Merge-Schritt wird pro Vergleich ein Element in `result` geschrieben

⇨ Höchstens n Vergleiche pro Merge-Schritt

Die Laufzeit von Mergesort ist in $\mathcal{O}(n \log_2 n)$

Zeitkomplexität von Mergesort



Sortieren 2

Komplexität des Sortierens

Komplexität des Sortierens

Wie ändert sich die Laufzeit bei bestimmten Eingaben?

- Vorsortiert
- Absteigend sortiert
- Zufällig

Die Anzahl Vergleiche ist bei Mergesort
(und auch Bubble- und Minsort) immer gleich für ein festes n

- Das ist nicht immer so
- Unterschiedlicher Best-, Worst- und Average-Case
- **Timsort** nutzt z. B. vorsortierte Teillisten aus

Sortieren 2

Bucketsort

Sortieren weniger Elemente

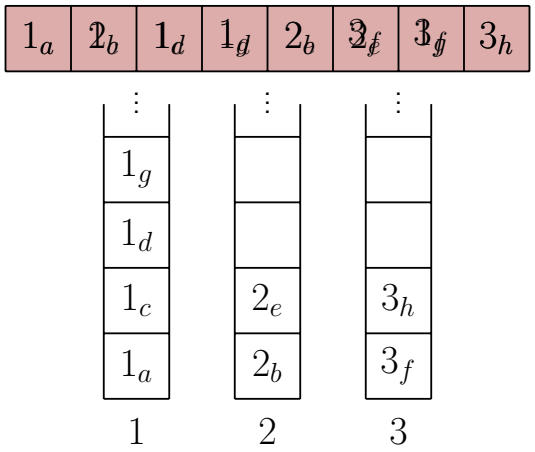
Sortieren komplexer Datensätze **nach einem Attribut**

Stabiles Sortieren: Daten mit gleichem Attribut haben gleiche Reihenfolge

Beispiel

Name	Vorname	Note
Adleman	Leonard	6
Caesar	Gaius Julius	3
de Vigenère	Blaise	5
Rivest	Ronald	6
Shamir	Adi	6

Bucketsort



Aufgabe – Bucketsort

Implementieren Sie Bucketsort

- als Python-Funktion
- mit drei **Stacks** `one`, `two` und `three` für die möglichen Werte 1, 2 und 3
- der die Stacks entsprechend der gegebenen Liste füllt
- und die Stacks am Ende zusammenfügt
  (mit Python-Listen ist das sehr einfach dank des +-Operators)

Bucketsort

```
def bucketsort(data):
    one = []
    two = []
    three = []
    for item in data:
        if item == 1:
            one.append(item)
        else:
            if item == 2:
                two.append(item)
            else:
                if item == 3:
                    three.append(item)
    return one + two + three
```

```
if item == 1:
    one.append(item)
elif item == 2:
    two.append(item)
elif item == 3:
    three.append(item)
```


Sortieren 2

Zeitkomplexität von Bucketsort

Zeitkomplexität von Bucketsort

- Sei n die Länge der Eingabe
- Sei k die Anzahl unterschiedlicher Werte
- Beim Füllen der Buckets max. $k - 1$ Vergleiche pro Element

⇨ Gesamtanzahl Vergleiche: ungefähr $k \cdot n$

Die Laufzeit von Bucketsort ist in $\mathcal{O}(n)$, wenn eine konstante Anzahl von Werten existiert

Zeitkomplexität von Bucketsort