Algorithmen und Datenstrukturen

D1. Sortieren von Strings

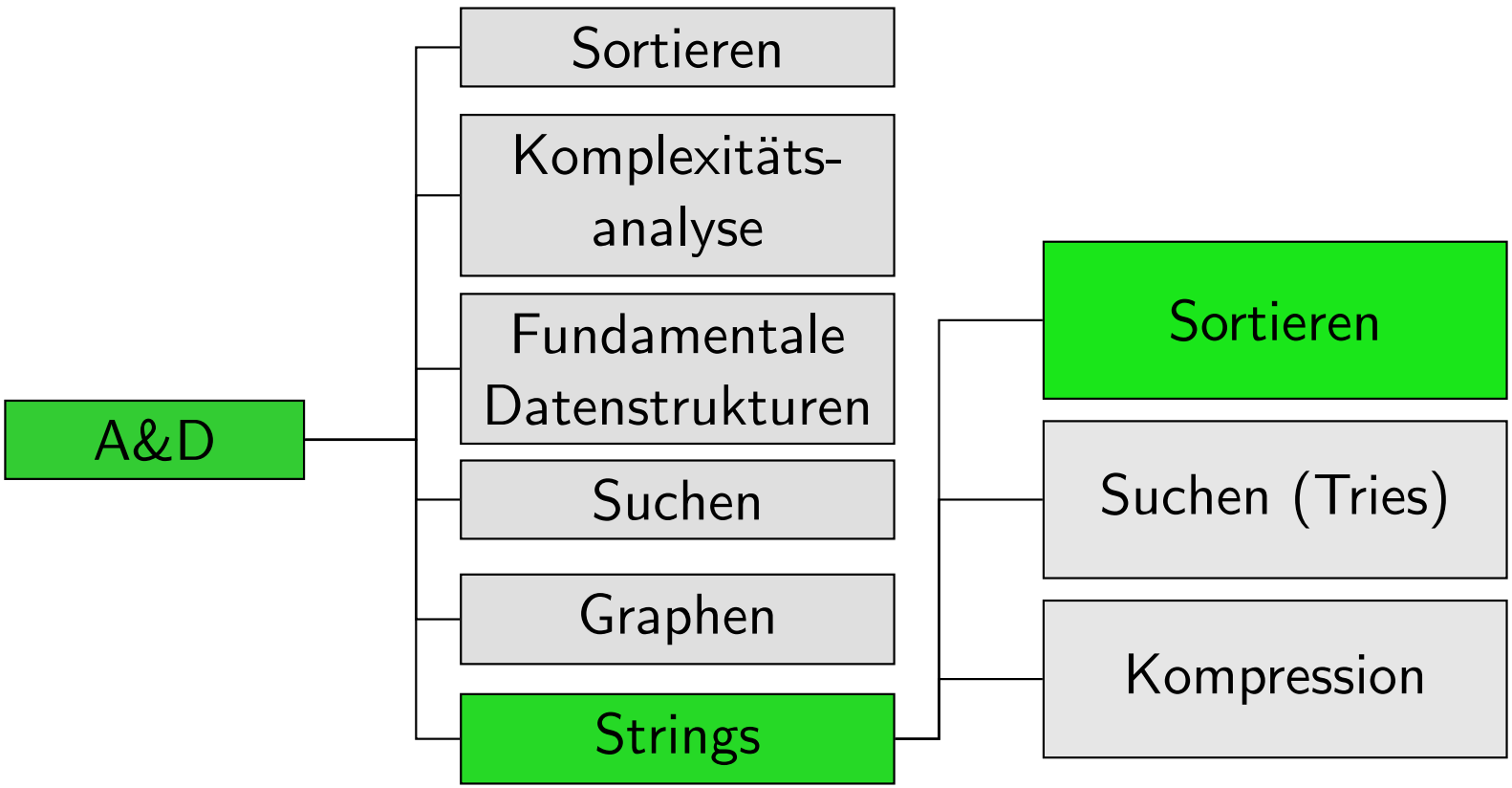
Gabi Röger und Marcel Lüthi

Universität Basel

16. Mai 2019

Algorithmen und Datenstrukturen
16. Mai 2019 — D1. Sortieren von Strings

Übersicht


A&D
Sortieren
Komplexitäts-
analyse
Fundamentale
Datenstrukturen
Suchen
Graphen
Strings
Sortieren
Suchen (Tries)
Kompression

String algorithmen oder generische Algorithmen?

- Alle Algorithmen zum Sortieren / Suchen wurden über beliebige Schlüssel definiert.
  - Können direkt auf Strings angewendet werden.
- Preis der Abstraktion / Allgemeinheit: Vorhandene Struktur der Schlüssel wird nicht ausgenutzt.

Frage

Können wir Eigenschaften von Strings ausnutzen um noch effizientere Algorithmen zu entwickeln?

Heutiges Programm

- Motivation
- Abstraktion: Alphabet
- LSD-Sort
- Quicksort für Strings

Repetition und Erweiterung bereits bekannter Konzepte

D1.1 Motivation

Strings als fundamentale Abstraktion

Strings / Text ist in vielen Bereichen grundlegende Repräsentation von Informationen

- Programmcode
- Datenrepräsentation im Web (HTML / Json / CSS)
- Kommunikation (E-Mail, Textmessages)
- Gensequenzen

Anwendung 1: Programmcode

Programme sind Strings

- Compiler / Interpreter interpretieren und transformieren Strings in ausführbare Programme
- IDEs bietet Funktionalität zur effizienten Suche und Manipulation von Code
  - Selektion von allen Wörtern, die Suchergebniss entsprechen
  - Suche nach regulärem Ausdrücken
  - Refactoring

Anwendung 2: Informations und Kommunikationssysteme

Text ist wichtigste Repräsentation für Information und Kommunikation im Internet

- E-Mail / SMS / ...: Text wird von einem an anderen Ort transferiert.
- Webbrowser: interpretiert CSS und HTML und stellt diesen formatiert dar.
- Suchmaschine: Muss grosse Mengen an Text effizient indizieren und durchsuchen.

Anwendung 3: Bioinformatik

> The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's, and C's. This string is the root data structure of an organism's biology.
>
> Maynard Olson - A time to sequence

- Analyse des Genoms eines Organismus
  - Beispiel: Genom Mensch besteht ist String aus ca. 3'000'000'000 Zeichen

Beispielprobleme

- Suchen von Sequenzen in grossen Datenbanken
- Vergleichen von (Sub)-Sequenzen von Strings
- Finden von häufig auftretenden Mustern
- ...

D1.2 Strings

Strings

String

Endliche Folge von Zeichen (Character)

- Strings sind unveränderlich (immutable). Einmal erzeugt können Strings nicht mehr verändert werden.
  - Ideale Schlüssel für Symboltabellen
- Intern häufig als Array von Zeichen implementiert.

0	1	2	3	4	5	6	7	8	9	10	11
A	T	T	A	C	K	A	T	D	A	W	N

Characters

Früher:

- 7 Bit Zeichensatz (ASCII)
- 8 Bit Zeichensatz (extended ASCII)

Heute:

- 8 oder 16 bit Unicode Zeichensatz (UTF-8, UTF-16)

Unterschied Java / Python

- Java Character entspricht 16 bit Unicode Zeichen (UTF-16)
- Python kennt keinen Charactertyp. Ausdruck s[i] ist (UTF-8) String der Länge 1.

Abstraktion: Alphabet

- Unicode umfasst 1'112'064 Zeichen.
- Kleineres Alphabet reicht für viele Anwendungen aus

Name	Radix (R)	Bits ($\log_2(R)$)	Zeichen
BINARY	2	1	0 1
DNA	4	2	A C G T
LOWERCASE	26	5	a - z
UPPERCASE	26	5	A-Z
ASCII	128	7	ASCII Characters
EXTENDED_ASCII	256	8	EXTENDED_ASCII
UNICODE	1'114'112	21	UNICODE

Alphabet

Abstraktion Alphabet erlaubt uns Code unabhängig vom benutzten Alphabet zu schreiben.

```
class Alphabet:
    def __init__(s : List[char])
    def toChar(index : Int) -> char
    def toIndex(c : Char) -> int
    def contains(c : Char) -> boolean
    def R() -> int   # Radix
```

D1.3 Sortieren von Strings

Sortieralgorithmen

Algorithmus	Laufzeit $O(\cdot)$	Speicherbedarf $O(\cdot)$	stabil
	best/avg./worst	best/avg./worst	
Selectionsort	n^2	1	nein
Insertionsort	$n/n^2/n^2$	1	ja
Mergesort	$n \log n$	n	ja
Quicksort	$n \log n/n \log n/n^2$	$\log n/\log n/n$	nein
Heapsort	$n \log n$	1	nein

$O(n \log n)$ ist beweisbar der lower bound für allgemeine, vergleichsbasierte, Sortierverfahren.

Idee 1

- Zeichen in Alphabet sind geordnet.
- Sortierung kann durch "Fachverteilen" hergestellt werden
  - Vergleiche: Radixsort

Erinnerung: Radixsort

- Zahlen: 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
- Wiederhole mit zweitletzter Stelle, etc.

- Grundlage LSD-Sortierverfahren

Idee 2

- Wie viele Character Vergleiche müssen durchgeführt werden um zwei Strings zu vergleichen?

0	1	2	3	4	5	6	7
p	r	e	f	e	t	c	h
p	r	e	f	i	x	e	s

- Worst case: Proportional zur Stringlänge
- Aber: Oft sublinear

Wir können Sortieralgorithmen so schreiben, dass sie Vergleiche auf einzelne Zeichen reduzieren.

- Grundlage von 3-Wege Quicksort für Strings

D1.4 LSD-Sortierverfahren

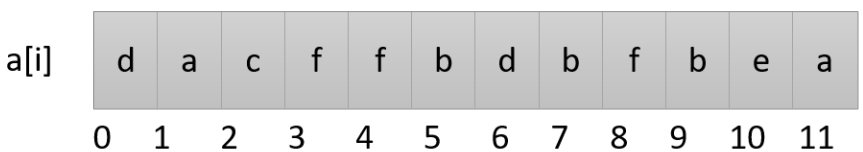
LSD-Sortierverfahren (1 Zeichen)

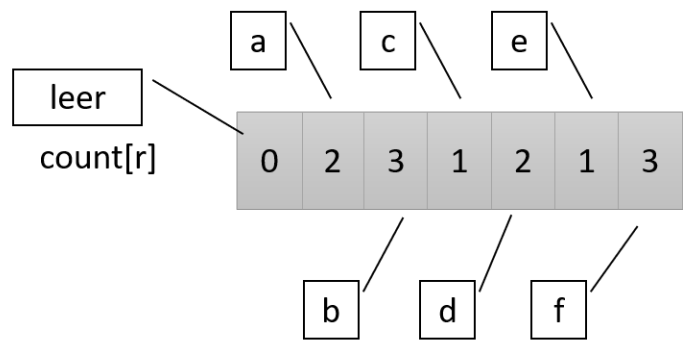
- Input: Array a, Output: Sortiertes array aux

```
N = len(a)  # Anzahl zu sortierender Zeichen
count = [0] * (alphabet.radix() + 1)
aux = [None] * N

# Zeichen zaehlen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    count[indexOfchar + 1] += 1

# Kummulative Summe
for r in range(0, alphabet.radix()):
    count[r+1] += count[r]

# Verteilen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    countForChar = count[indexOfchar]
    aux[countForChar] = a[i]
    count[indexOfchar] += 1
```

LSD-Sortierverfahren (1 Zeichen)

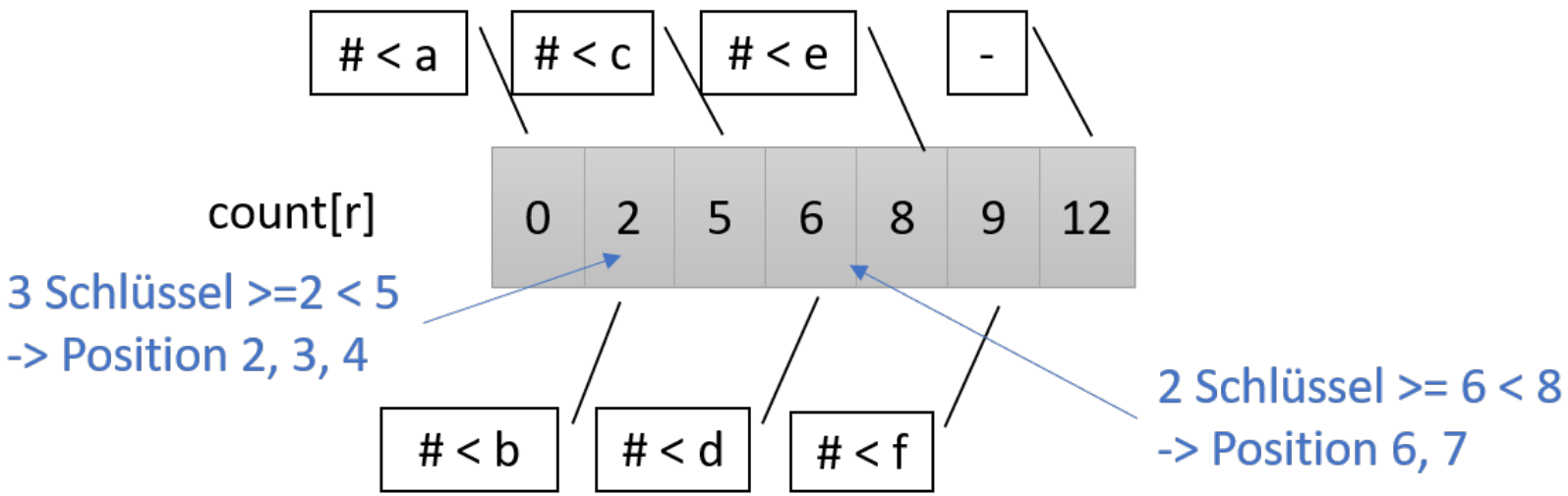
```
N = len(a)  # Anzahl zu sortierender Zeichen in array a
count = [0] * (alphabet.radix() + 1)

# Zeichen Zaehlen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    count[indexOfchar + 1] += 1
```





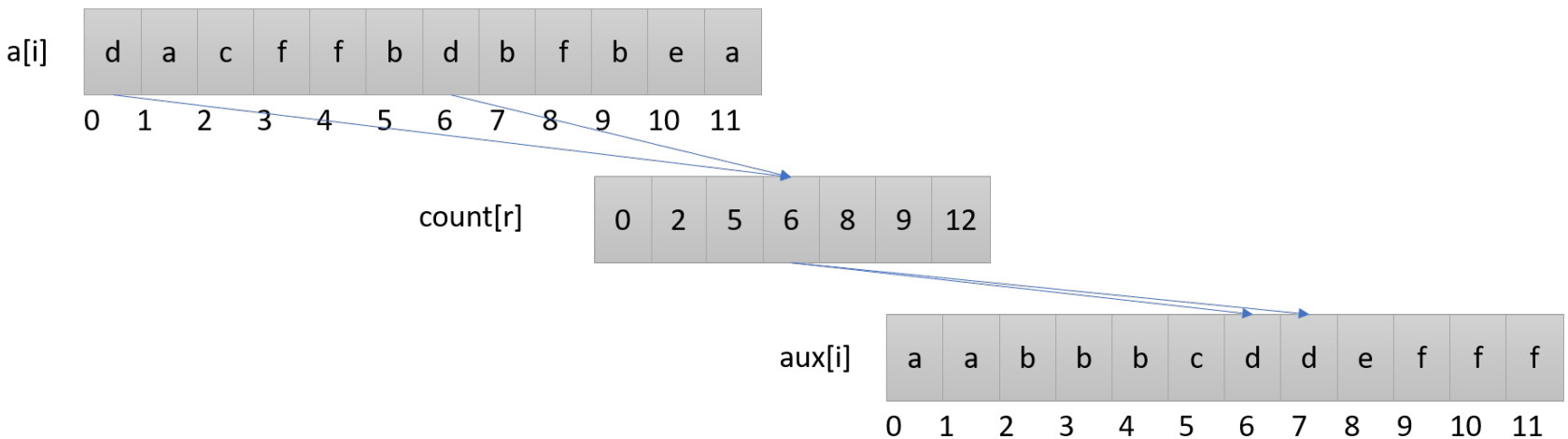
LSD-Sortierverfahren (1 Zeichen)

```
# Kummulative Summe
for r in range(0, alphabet.radix()):
    count[r+1] += count[r]
```



LSD-Sortierverfahren (1 Zeichen)

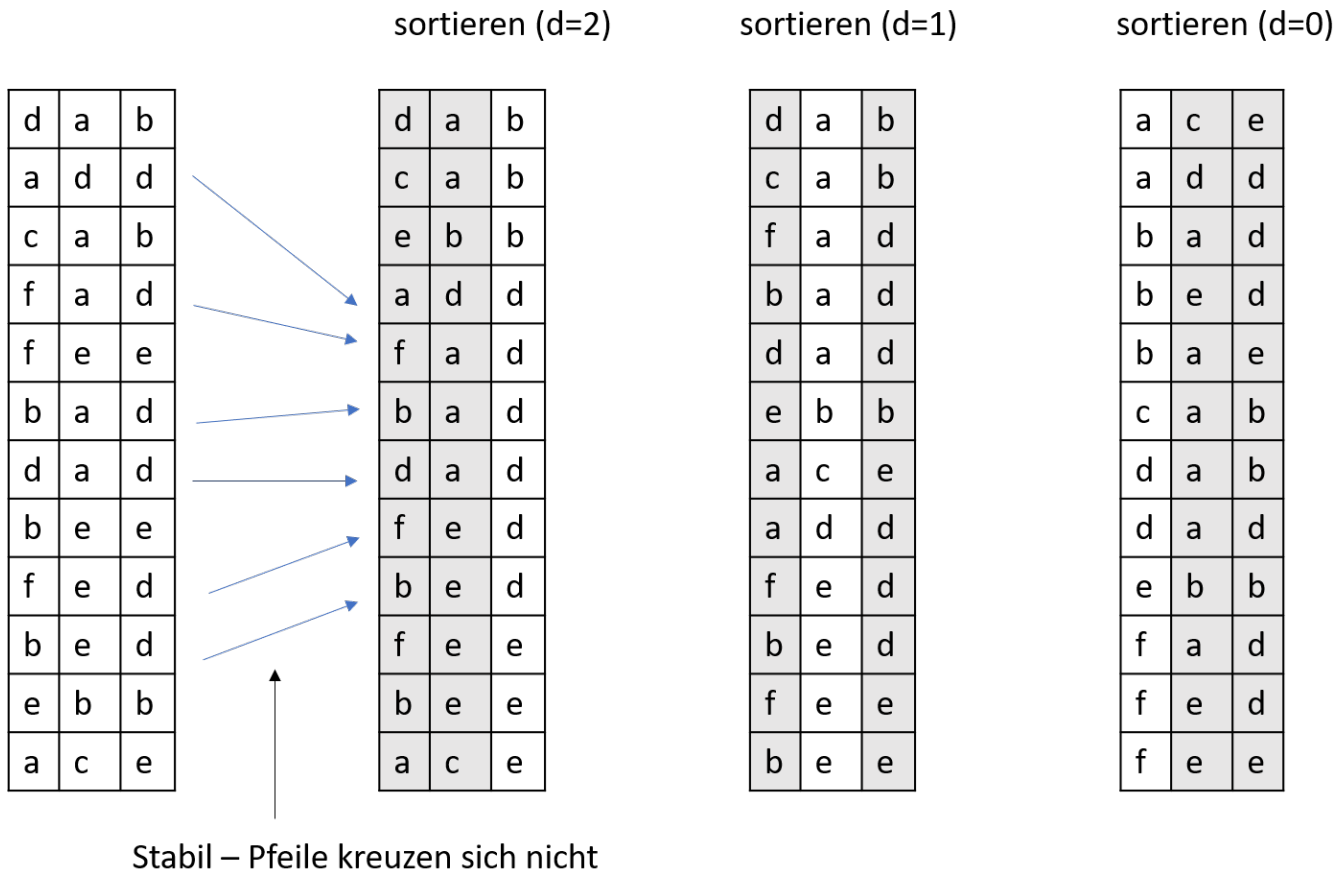
```
# Verteilen
for i in range(0, N):
    indexOfchar = alphabet.toIndex(a[i])
    countForChar = count[indexOfchar]
    aux[countForChar] = a[i]
    count[indexOfchar] += 1
```



LSD-Sortierverfahren (1 Zeichen)

- Verfahren ist stabil
- Zeitaufwand: Proportional zu $N + R$, wobei R Grösse des Alphabets ist
- Speicher: Proportional zu $N + R$ (aux-Array und count Array)

LSD-Sortierverfahren



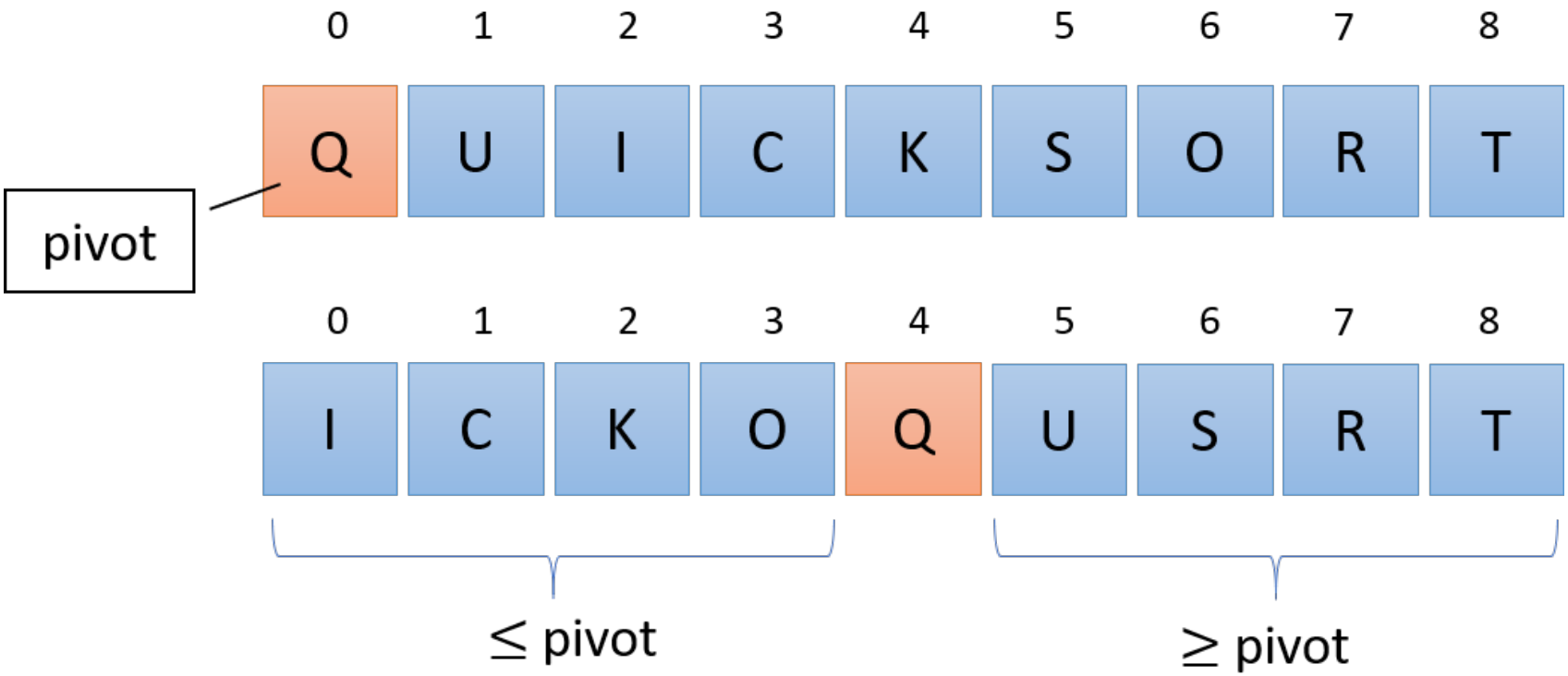
- Sortiere jedes Zeichen einzeln beginnend mit letztem (least significant digit)
- Funktioniert, da Sortierung stabil ist

LSD-Sortierverfahren

```
N = len(a);  aux = [None] * N ;   d = numDigits - 1
while d >= 0:
    count = [0] * (alphabet.radix() + 1)

    for i in range(0, N):
        indexOfcharAtPosdInA = alphabet.toIndex(a[i][d])
        count[indexOfcharAtPosdInA + 1] += 1

    for r in range(0, alphabet.radix()):
        count[r+1] += count[r]

    for i in range(0, N):
        indexOfCharAtPosdInA = alphabet.toIndex(a[i][d])
        countForChar = count[indexOfCharAtPosdInA]
        aux[countForChar] = a[i]
        count[indexOfCharAtPosdInA] += 1

    for i in range(0, N):
        a[i] = aux[i]
    d -= 1
```

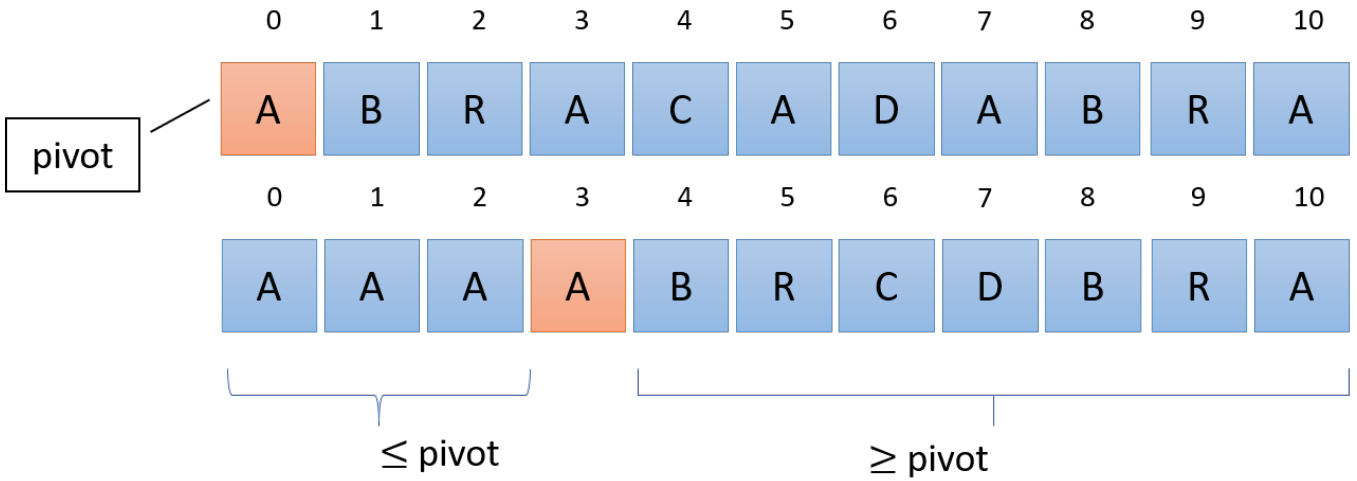
D1.5 Quicksort

Erinnerung: Quicksort



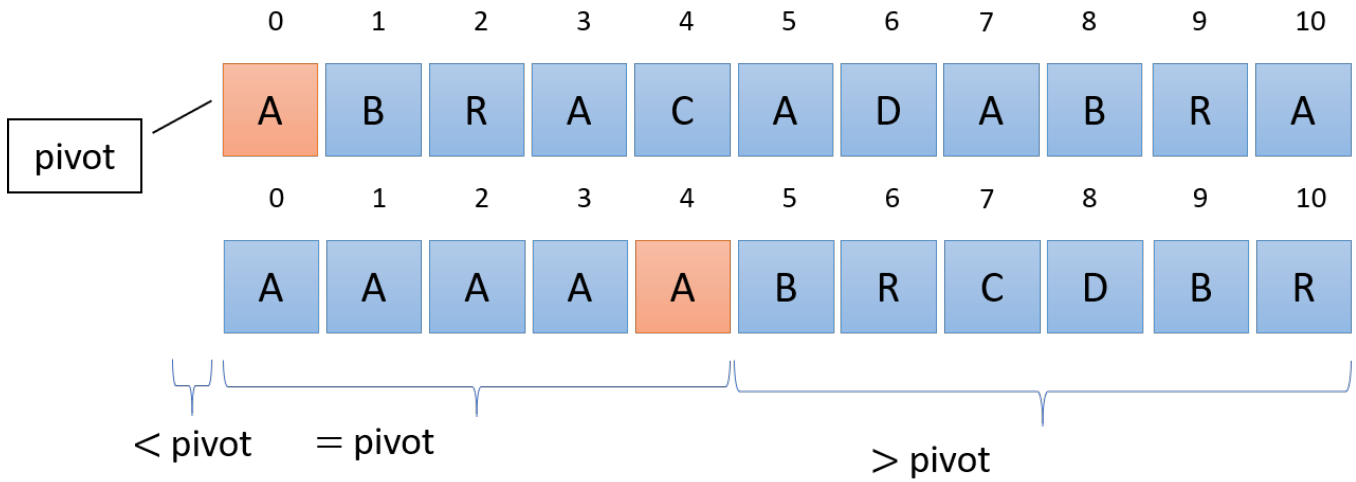
- Wähle Pivot Element
- Partitioniere Array
- Rekursion auf linkes und rechtes Teilarray

Quicksort: Gleiche Schlüssel



- Was passiert bei vielen gleichen Schlüsseln?
- Unnötige Partitionierung von gleichen Schlüsseln.

3-Wege Quicksort

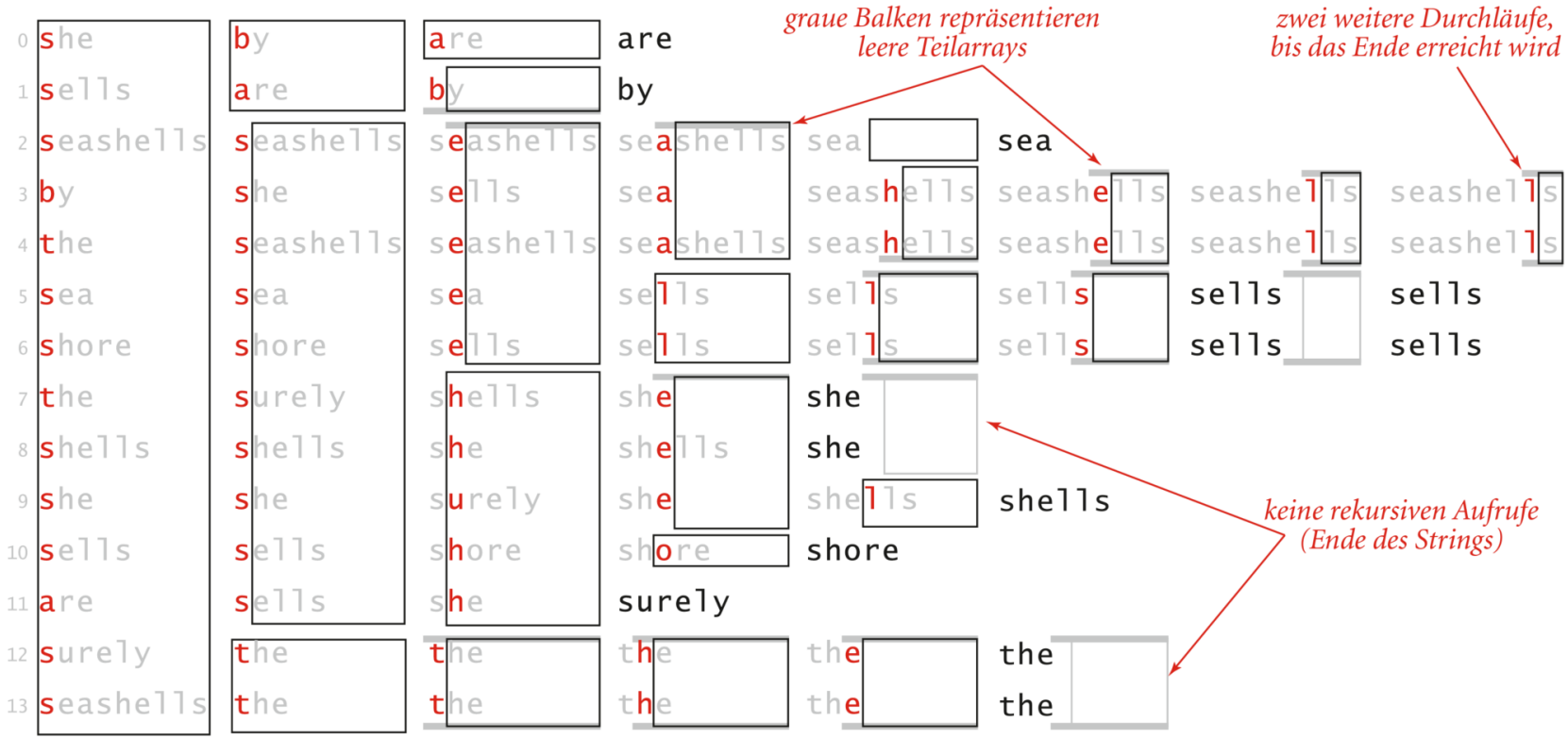


- Gleiche Schlüssel sind bereits sortiert.
  - Kein rekursiver Aufruf mehr nötig.

Quicksort für Strings

- 3-Wege Quicksort per Buchstabe
- Bei gleichen Anfangsbuchstaben, vergleiche nächsten Buchstaben.

Quicksort für Strings



Quelle: Sedgewick & Wayne, Algorithmen, Abbildung 5.18

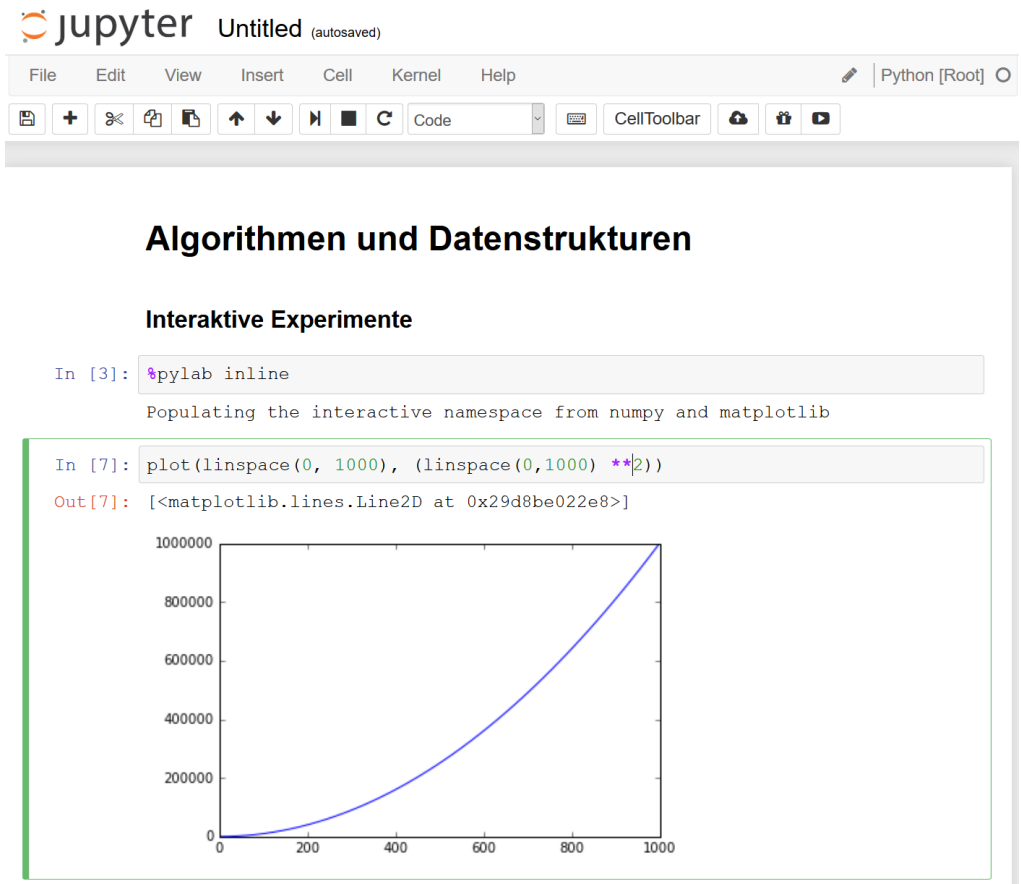
Laufzeit

Theorem

Um ein Array von N zufälligen Strings zu sortieren, benötigt der 3-Weg-Quicksort für Strings im Durchschnitt $\sim 2N\ln N$ Zeichenvergleiche.

- Gleiche Anzahl Vergleiche wie standard (3-Wege) Quicksort
- Aber: Wir haben Zeichenvergleiche und nicht Schlüsselvergleiche

Implementation



Jupyter Notebooks: Stringsort.ipynb