Algorithmen und Datenstrukturen
BAI3-AD – SoSe 2018

Prof. Dr. Marina Tropmann-Frick

BT7, Raum 10.86

marina.tropmann-frick@haw-hamburg.de

HAW
HAMBURG

Vorlesungsüberblick

1. Einführung

   Motivation, Übersicht, Begriffsklärung, Größenordnungen

2. **Lineare Datenstrukturen**

   Liste, Stack, Queue

3. Algorithmengrundlagen

   Die ersten Algorithmen mit Komplexitäts-(Aufwands-)analyse

4. Sortieren

   Sortieralgorithmen

5. Verzweigte Datenstrukturen

   Bäume, Operationen und Implementierungsmethoden

6. Grundlegende Graph-Algorithmen

   Graphen und Implementierungsmethoden, kürzeste Pfade, Breitensuche, Tiefensuche, Dijkstra's Algorithmus

7. Hash-Verfahren

   Hashfunktionen, Analyse

8. Optimierung

   Lineare Optimierung, Lösung Nichtlinearer Probleme

-- (Ausblick – weitere effiziente Algorithmen)*

2. Lineare Datenstrukturen

(1) Felder und Lineare Listen

(2) Stack

(3) Queue

(4) Spezielle Techniken

2. Lineare Datenstrukturen

Lineare Datenstrukturen

→ Daten sind linear angeordnet

Eine lineare Datenstruktur L ist eine Sequenz

$$L = (x_1, \dots, x_n).$$

Die lineare Datenstruktur ordnet Elemente (primitive Datentypen oder komplexere Datenstrukturen) in einer linearen Anordnung an.

→ „wie an einer Schnur angehängt“

Lineare Datenstrukturen

→ Prinzipiell zwei Ansätze:

1. Durchnummerieren → Zugriff schnell (in konstanter Zeit) über Index: Feld
2. Verknüpfen durch Verweise (Referenzen) auf nächstes und möglicherweise vorheriges Objekt: einfach und doppelt verkettete Strukturen (singly, doubly linked)

Feld (array) – fundamentale Datenstruktur

- In einem Feld werden einfache Daten oder Referenzen auf Objekte hintereinander angeordnet
- Die Anzahl der Einträge wird bei der Erzeugung des Feldes festgelegt
- Auf die Daten kann über ihren Index zugegriffen werden
- Der Zugriff auf die Daten erfolgt (ohne Einschränkung) in konstanter Zeit
- In meisten Programmiersprachen – zusammenhängender Speicherbereich, daher der schnelle Zugriff

Notation

`a[0..n-1]` für ein Feld der Länge n mit Namen `a`

Zugriff

auf das Element mit dem Index $i < \mathrm{n}$: `a[i]`

Zeitverbrauch und Platzbedarf:

- anlegen eines Feldes der Länge n einschließlich Initialisierung mit Standardwert $O(n)$, sogar $\Theta(n)$ → wird später eingeführt!
- Zugriff auf ein Feldelement (auslesen oder ändern) $\Theta(1)$ => konstant
- Platzbedarf $\Theta(n)$

Java:

für jeden Datentyp gibt es einen Feldtyp, der Daten dieses Typs als Elemente hat.

- Typbeschreibung: `Elementtyp[]`, z.B. `int[]`
- Anlegen: `new Elementtyp[n]`, z.B. `new int[5]`
- Anlegen mit Initialisierung: `new Elementtyp[n]{…, …}`
- Zugriff auf Feldelement: `Name[Ausdruck]`, z.B. `a[3]`

Anwendung → Fragestellung:

- Wir wollen ein Feld von Objekten `a[0..n-1]` sortieren
- Jedes Objekt besitzt einen Sortierschlüssel `key`, der aus einer reellen Zahl besteht
- Die Sortierung soll nach aufsteigenden Schlüsseln erfolgen

- Wir betrachten einen Spezialfall:
  - Die Schlüssel kommen aus einem kleinen Bereich, etwa aus $0, 1, \ldots N-1$ mit $N = 256$
  - Und das Feld ist groß, etwa $n \geq 1.000.000$

Beispielfeld: A

4	**6**	**5**	**2**	**3**	**5**	**2**	**5**

$\boldsymbol{n = 8}$

Ansatz (einfach) – **CountingSort**:

1. Wir durchlaufen das Feld einmal und zählen, wie häufig jeder Schlüssel vorkommt
2. Ergebnis wird in einem neuen Feld notiert
3. Danach legen wir ein neues Feld an
4. In das wir die Objekte in sortierter Reihenfolge schreiben
5. Dazu rechnen wir für jeden Schlüssel aus, wo das erste Objekt mit diesem Schlüssel stehen müsste
6. Dann durchlaufen wir das Feld von links nach rechts und schreiben die Objekte an die richtige Stelle in das Ergebnisfeld
7. Wobei die richtige Stelle durch die entsprechende Positionsvariable vorgegeben wird
8. Diese wird aktualisiert, nachdem ein Objekt übertragen wurde, und zwar durch erhöhen um 1

2. Lineare Datenstrukturen

(1) Felder

Beispielfeld: A

4	**6**	**5**	**2**	**3**	**5**	**2**	**5**

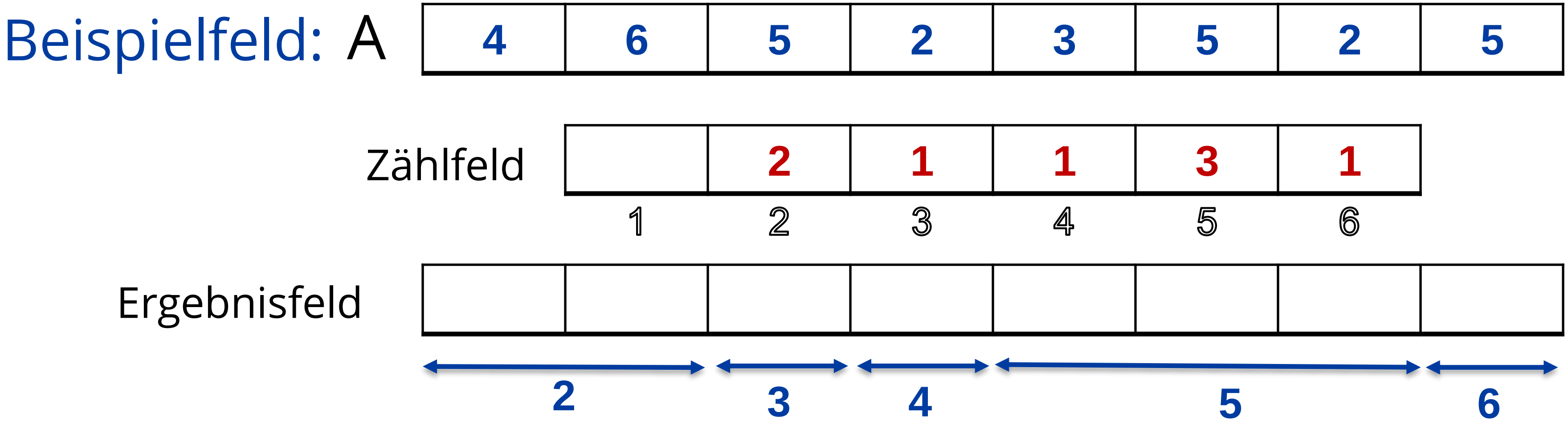
1. Wir durchlaufen das Feld einmal und zählen, wie häufig jeder Schlüssel vorkommt
2. Ergebnis wird in einem neuen Feld notiert

neues Feld → Zählfeld

	2	**1**	**1**	**3**	**1**
1	2	3	4	5	6

3. Danach legen wir ein neues Feld an
4. In das wir die Objekte in sortierter Reihenfolge schreiben

Ergebnisfeld



Beispielfeld: A

4	6	5	2	3	5	2	5

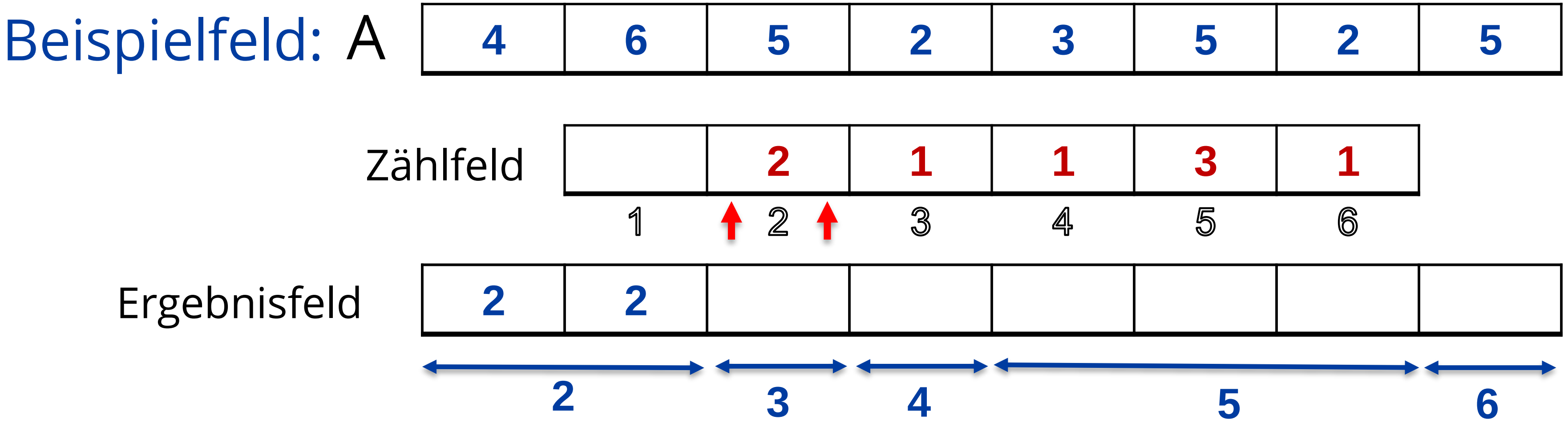
Zählfeld

1	2	3	4	5	6
	2	1	1	3	1

Ergebnisfeld

5. Dazu rechnen wir für jeden Schlüssel aus, wo das erste Objekt mit diesem Schlüssel stehen müsste

2. Lineare Datenstrukturen (1) Felder



Beispielfeld: A

4	6	5	2	3	5	2	5

Zählfeld

1	2	3	4	5	6
	2	1	1	3	1

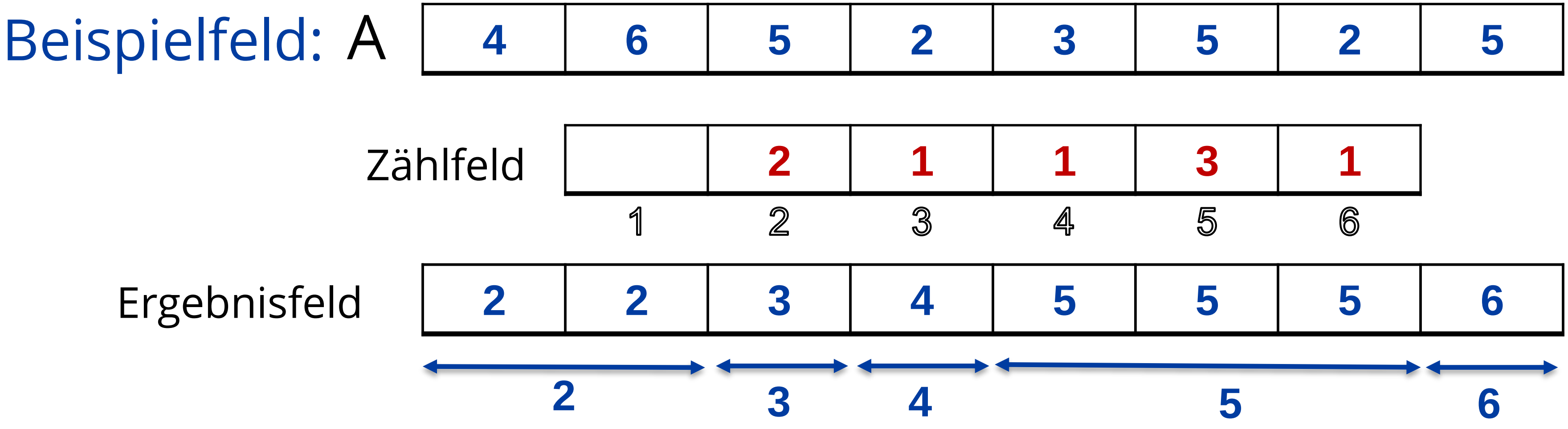
Ergebnisfeld

2	2						

6. Dann durchlaufen wir das Zählfeld von links nach rechts und schreiben die Objekte an die richtige Stelle in das Ergebnisfeld
7. Wobei die richtige Stelle durch die entsprechende Positionsvariable vorgegeben wird
8. Diese wird aktualisiert, nachdem ein Objekt übertragen wurde, und zwar durch erhöhen um 1

2. Lineare Datenstrukturen

(1) Felder



Beispielfeld: A

4	6	5	2	3	5	2	5

Zählfeld

1	2	3	4	5	6
	2	1	1	3	1

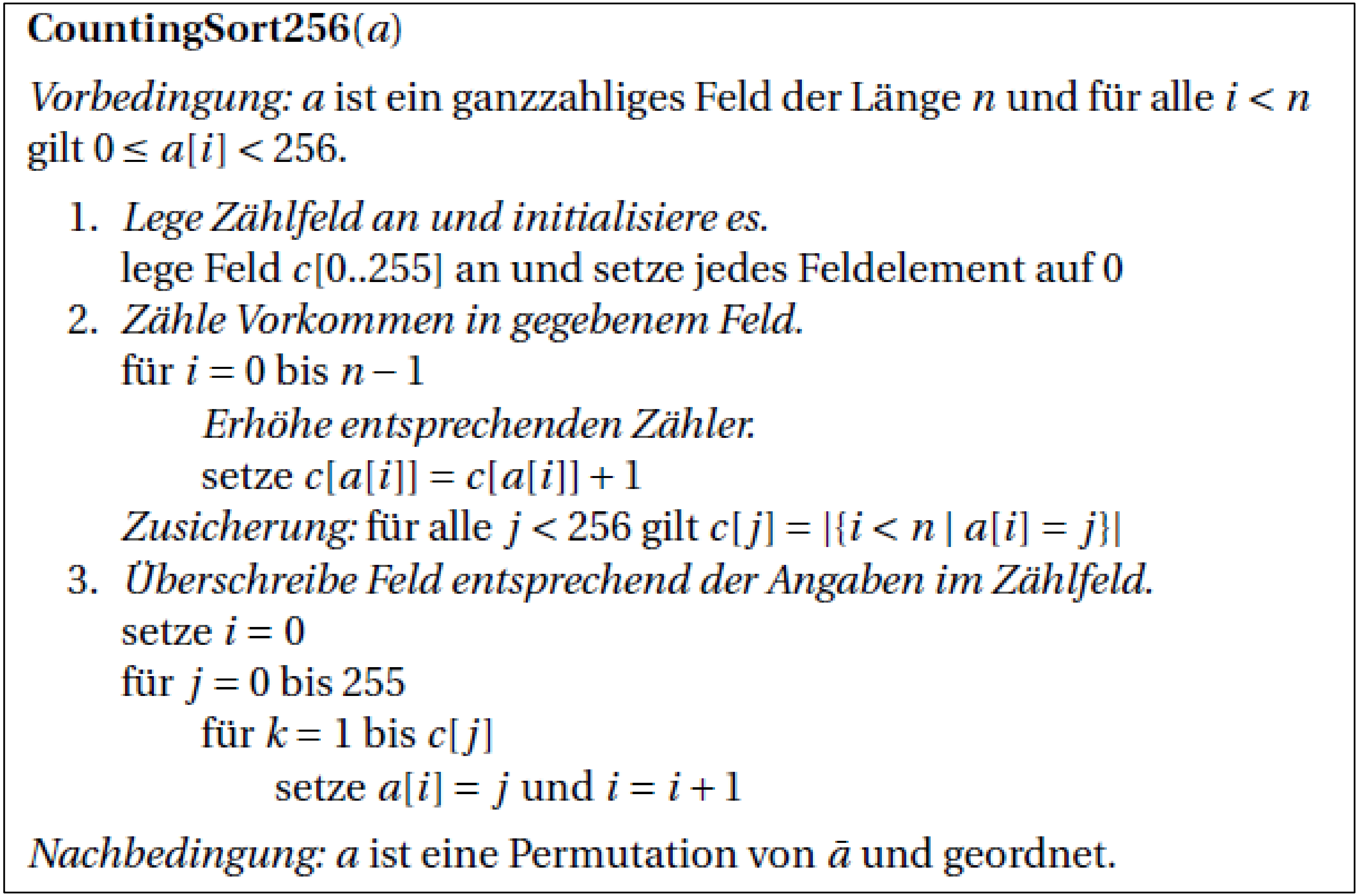
Ergebnisfeld

2	2	3	4	5	5	5	6

6. Dann durchlaufen wir das Zählfeld von links nach rechts und schreiben die Objekte an die richtige Stelle in das Ergebnisfeld
7. Wobei die richtige Stelle durch die entsprechende Positionsvariable vorgegeben wird
8. Diese wird aktualisiert, nachdem ein Objekt übertragen wurde, und zwar durch erhöhen um 1

2. Lineare Datenstrukturen

(1) Felder



```
CountingSort256($a$)

Vorbedingung: $a$ ist ein ganzzahliges Feld der Länge $n$ und für alle $i < n$
gilt $0 \leq a[i] < 256$.

   1. Lege Zählfeld an und initialisiere es.
      lege Feld $c[0..255]$ an und setze jedes Feldelement auf 0
   2. Zähle Vorkommen in gegebenem Feld.
      für $i = 0$ bis $n - 1$
          Erhöhe entsprechenden Zähler.
          setze $c[a[i]] = c[a[i]] + 1$
      Zusicherung: für alle $j < 256$ gilt $c[j] = |\{i < n \mid a[i] = j\}|$
   3. Überschreibe Feld entsprechend der Angaben im Zählfeld.
      setze $i = 0$
      für $j = 0$ bis $255$
          für $k = 1$ bis $c[j]$
              setze $a[i] = j$ und $i = i + 1$

Nachbedingung: $a$ ist eine Permutation von $\bar{a}$ und geordnet.
```

Zur Laufzeit für Feld der Länge n:

- Anlegen von c → 256 Zeiteinheiten, also konstant $\Theta(1)$
- Erste für-Schleife → $7n$ – pro Durchlauf einmal inkrementieren, einmal mit $n-1$ vergleichen, 4 Feldzugriffe, 1 inkrementieren, also $\Theta(n)$
- Initialisieren von i → eine Zeiteinheit, also $\Theta(1)$
- Zweite für-Schleife → 2x256 (j inkrementieren und vergleichen) $+ 5 * n$ (k inkrementieren und vergleichen, a[i] zuweisen, i inkrementieren und zuweisen), also $\Theta(n)$

Insgesamt ➡ die (pessimale) Laufzeit von CountingSort ist $\Theta(n)$

Speicherbedarf

- Asymptotische Analyse
- Objekte beanspruchen so viel Platz, wie bei einer Implementierung auf einem konkreten Rechner benötigt würde
- Die eigentlichen Referenzwerte benötigen konstanten Platz, genau wie die primitiven Werte

Zu beachten:

- Der Speicherplatz, der für die Eingabe benötigt wird, bleibt in der Regel unberücksichtigt
- In der Regel wird eine pessimale Analyse durchgeführt: maximale Speicherbelegung während der Ausführung aller Programmläufe zu einer festen Eingabegröße

Speicherbedarf

Implizite Annahme:

Die Freigabe nicht mehr benötigten Speichers erfolgt immer rechtzeitig

Jeder Aufruf einer Prozedur belegt Speicherplatz:

Verwaltungsinformation und ggf. lokale Variablen und Parameter
→ Insbesondere bei rekursiver Programmierung zu beachten

Speicherbedarf – CountingSort

c benötigt konstanten Platz: $\Theta(1)$

Platz für `a` bleibt unberücksichtigt

i, j, k belegen konstanten Platz: $\Theta(1)$

keine Rekursion

⇒ insgesamt: $\Theta(1)$

Einschub – Aufwand

Sprechweise:

Problem= algorithmische Fragestellung, gegeben durch mögliche Eingaben und zugehörige Ausgaben

Problemfall = konkrete Eingabe; auch Probleminstanz bezeichnet

Zur Beschreibung des Aufwands (Ressourcenverbrauchs) werden meistens Laufzeit (*running time*) und Speicherbedarf (*space*) herangezogen → Definitionen folgen.

Es gibt aber auch andere sinnvolle Aufwandskennzahlen:
z.B. Anzahl der Vergleiche, Anzahl der Zugriffe auf eine Datenstruktur, Anzahl der Schleifendurchläufe, . . .

Einschub – Aufwand

Aufwand zur Lösung eines Problemfalls, z.B. zu sortierendes Feld, wird in Abhängigkeit von einer charakteristischen Größe des Problemfalls betrachtet, z.B. Länge des zu sortierenden Feldes.
Wir sprechen von der **Eingabegröße** (*input size*).

Wir interessieren uns in der Regel für den schlechtesten Fall, um auf der sicheren Seite zu sein: Für eine Eingabegröße n betrachten wir das Maximum aller Laufzeiten für Problemfälle der Größe n.
Wir sprechen dann vom *pessimalen* (*worstcase*) Aufwand.

Alternative Betrachtungsweisen: *durchschnittlicher* Aufwand und *amortisierter* Aufwand.

Einschub – Aufwand → Laufzeit

Tatsächliche Laufzeiten hängen stark von der Hardware ab. Z.B. bewirkt ein schnellerer Prozessor eine Laufzeitverkürzung um einen (zumindest in etwa) konstanten Faktor.
Außerdem interessiert die Laufzeit für kleine Eingabegrößen (winzige Probleme) nicht sonderlich, da sie in der Regel klein ist. Daher beschränken wir uns auf asymptotische Betrachtungen.

Um Laufzeiten für Programme (und damit auch Algorithmen) bestimmen zu können, müssen wir den einzelnen Programmkonstrukten (*fiktive*) Laufzeiten zuordnen.
Dabei lassen wir uns von der Laufzeit auf einem generischen Modell einer üblichen Rechnerarchitektur leiten. Außerdem reicht es, dass wir dabei asymptotische Angaben machen, wenn wir insgesamt nur an einer asymptotischen Analyse interessiert sind.
Die Gesamtlaufzeit ergibt sich dann als Summe der Einzellaufzeiten.

Einschub – Aufwand → Laufzeit

Zu den Einzellaufzeiten:

Wenn keine komplizierten Daten verarbeitet werden, gehen wir von konstanter Laufzeit aus, d. h., wir setzen eine Zeiteinheit an.

Insbesondere setzen wir für den Zugriff auf ein Feldelement und die Zuweisung von Werten eine Zeiteinheit an.

Bsp. für nicht-konstante Laufzeit: Das Anlegen eines Feldes der Länge n (mit Initialisierung) hat Laufzeit $\Theta(n)$.Wir sprechen von **linearer Laufzeit** und setzen n als Laufzeit an.

2. Lineare Datenstrukturen

Fragen / Anregungen ?