Komplexität von Algorithmen: Laufzeit, Speicherplatz

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 23/24

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort

VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort

VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren

VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue

VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen

VL 6 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C

VL 7 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort

VL 8 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise

VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen

VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort

VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung

VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem

VL 13 „Q & A“: Offene Vorlesung/Wiederholung

Grundlagen der Algorithmen-Analyse

Inhalt:

- Wie beschreibt man einen Algorithmus?
- Rechenmodell
- Laufzeitanalyse (Zeitkomplexität)
- Speicherplatzanalyse (Raumkomplexität)
- Wie beweist man die Korrektheit eines Algorithmus?

Insertion Sort

```
InsertionSort(Array A)
for j ← 2 to length(A) do
    key ← A[j]
    i ← j-1
    while i>0 and A[i]>key do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

Insertion Sort

```
InsertionSort(Array A)
for j ← 2 to length(A) do
    key ← A[j]
    i ← j-1
    while i>0 and A[i]>key do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

Idee Insertion Sort:

- Die ersten j-1 Elemente sind sortiert (zu Beginn j=2)
- Innerhalb eines Schleifendurchlaufs wird das j-te Element in die sortierte Folge eingefügt
- Am Ende ist die gesamte Folge sortiert

Insertion Sort – Laufzeit?

```
InsertionSort(Array A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j-1
        while i>0 and A[i]>key do
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

Insertion Sort – Laufzeit?

```
InsertionSort(Array A)
for j ← 2 to length(A) do
    key ← A[j]
    i ← j-1
    while i>0 and A[i]>key do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

- Eingabegröße n

Insertion Sort – Laufzeit?

```
InsertionSort(Array A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j-1
        while i>0 and A[i]>key do
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

- Eingabegröße n
- length(A) = n

Insertion Sort – Laufzeit?

InsertionSort(Array A)
1. **for** j ← 2 **to** length(A) **do**
2. key ← A[j]
3. i ← j-1
4. **while** i>0 and A[i]>key **do**
5. A[i+1] ← A[i]
6. i ← i-1
7. A[i+1] ← key

- Eingabegröße n
- length(A) = n

- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts

Insertion Sort – Laufzeit?

```
InsertionSort(Array A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j-1
        while i>0 and A[i]>key do
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

- Eingabegröße n
- length(A) = n

- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts

- Speichere key in Lücke

Insertion Sort – Laufzeit?

```
InsertionSort(Array A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j-1
        while i>0 and A[i]>key do
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

- Eingabegröße n
- length(A) = n

- verschiebe alle Elemente aus
- A[1…j-1], die größer als key
- sind eine Stelle nach rechts

- Speichere key in Lücke

Kernfrage:

Wie kann man die Laufzeit eines Algorithmus vorhersagen?

Insertion Sort – Laufzeit?

Laufzeit hängt ab von:

- Größe der Eingabe (Parameter n)
- Art der Eingabe
- Beobachtung:
  - Insertion Sort ist schneller auf aufsteigend sortierten Eingaben
  - Insertion Sort ist langsam auf absteigend sortierten Eingaben

Laufzeitanalyse – Beobachtungen

- Analyse
  - Laufzeit als Funktion der Eingabegröße
  - Wie?
    - Parametrisiert, d.h. in Abhängigkeit von der Eingabegröße
    - Eingabegröße: n
    - Laufzeit: T(n)
- Ziel
  - Finde Schranken (Garantien) für die Laufzeit
    - Obere Schranken “f mit $f(n) \geq T(n)$“
    - Untere Schranken “f mit $f(n) \leq T(n)$“
    - …

Laufzeitanalyse – Beispiel

LineareMethode(int n)

1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. sum $\leftarrow$ sum + 1

Laufzeitanalyse – Beispiel

LineareMethode(int n)

1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. sum $\leftarrow$ sum + 1

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)

1. sum = 0
2. **for** i ← 1 **to** n **do**
3. **for** j ← 1 **to** n **do**
4. sum ← sum + 1

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)

1. sum = 0
2. **for** i ← 1 **to** n **do**
3.   **for** j ← 1 **to** n **do**
4.   sum ← sum + 1

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

```
KubischeMethode(int n)
1.    sum = 0
2.    for i ← 1 to n do
3.        for j ← 1 to n do
4.            for k ← 1 to n do
5.                sum ← sum + 1
```

Laufzeitanalyse – Beispiel

KubischeMethode(int n)

1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. **for** j $\leftarrow$ 1 **to** n **do**
4. **for** k $\leftarrow$ 1 **to** n **do**
5. sum $\leftarrow$ sum + 1

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beobachtungen

- Tatsächliche Laufzeit hängt ab von vielen Faktoren
  - Hardware
    (Prozessor, Cache, Pipelining)
  - Software
    (Betriebssystem, Programmiersprache, Compiler)

Laufzeitanalyse – Beobachtungen

- Tatsächliche Laufzeit hängt ab von vielen Faktoren
  - Hardware
    (Prozessor, Cache, Pipelining)
  - Software
    (Betriebssystem, Programmiersprache, Compiler)

- Ziel
  - Laufzeitanalyse soll unabhängig von Hard- und Software gelten
  - D.h. wird in der Regel auf der Basis von Pseudocode gemacht
  - D.h. C-, Java-, oder Programmiersprachen-Implementierungsdetails sollen abstrahiert werden

Laufzeitanalyse – Beobachtungen

- Tatsächliche Laufzeit hängt ab von vielen Faktoren
  - Rechnerarchitektur
  - Übersetzer
  - Implementierung

- Laufzeitanalyse
  - Größenordnung der Laufzeit, also das asymptotische Verhalten der Laufzeit als Funktion der Eingabegröße n.
  - D.h., wir ignorieren Details, z.B. konstante Faktoren
  - Dadurch erhält man Laufzeit- / Wachstumsklassen (logarithmisch, linear, quadratisch, exponentiell, etc.), in die man Algorithmen einordnet.

Maschinenmodell
Formal: Random Access Machine

- Maschinenmodell - uniform
  - Eine Pseudocode-Instruktion braucht einen Zeitschritt
  - Wird eine Instruktion r-mal aufgerufen, werden r Zeitschritte benötigt
  - Die Zahlengröße spielt keine Rolle

Maschinenmodell
Formal: Random Access Machine

- Maschinenmodell – logarithmisch
  - Rechen- und Speicheroperationen werden per Bit berechnet d.h. die Größe der Zahlen ist wichtig und geht logarithmisch ein
  - Sonstige Pseudocode-Instruktionen brauchen einen Zeitschritt
  - Wird eine Instruktion r-mal auf k-Bit Zahlen (d.h. Zahlen bis zu 2^k) aufgerufen, werden r * k Zeitschritte benötigt

- Hinweis
  - Oft sind die Ergebnisse gleich, da der Zahlenbereich beschränkt ist auf int, long, long long, etc. Damit handelt es sich nur um Konstanten.
  - Das logarithmische Modell ist das übliche Modell für die Analyse

Laufzeitanalyse – Beispiel

```
LineareMethode(int n)
    sum = 0
    for i ← 1 to n do
        sum ← sum + 1
```

Zeit:

Laufzeitanalyse – Beispiel

LineareMethode(int n)

1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. sum $\leftarrow$ sum + 1

Zeit:

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

LineareMethode(int n)
1. sum = 0
2. **for** i ← 1 **to** n **do**
3. sum ← sum + 1

Zeit:
1

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

LineareMethode(int n)
1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. sum $\leftarrow$ sum + 1

Zeit:
1
n+1

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

LineareMethode(int n)	Zeit:
1. sum = 0	1
2. **for** i ← 1 **to** n **do**	n+1
3. sum ← sum + 1	n

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)

1. sum = 0
2. **for** i ← 1 **to** n **do**
3. **for** j ← 1 **to** n **do**
4. sum ← sum + 1

Zeit:

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)

Zeit:

1. sum = 0
2. **for** i $\leftarrow$ 1 **to** n **do**
3. **for** j $\leftarrow$ 1 **to** n **do**
4. sum $\leftarrow$ sum + 1

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)

1. sum = 0
2. **for** i ← 1 **to** n **do**
3. **for** j ← 1 **to** n **do**
4. sum ← sum + 1

Zeit:

1

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)	Zeit:
sum = 0	1
for i $\leftarrow$ 1 **to** n **do**	n+1
for j $\leftarrow$ 1 **to** n **do**	
sum $\leftarrow$ sum + 1	

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

QuadratischeMethode(int n)	Zeit:
1. sum = 0	1
2. **for** i ← 1 **to** n **do**	n+1
3. **for** j ← 1 **to** n **do**	n * (n + 1)
4. sum ← sum + 1	

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

	QuadratischeMethode(int n)	Zeit:
1.	sum = 0	1
2.	**for** i ← 1 **to** n **do**	n+1
3.	**for** j ← 1 **to** n **do**	n * (n + 1)
4.	sum ← sum + 1	n^2

Nach der Ausführung hat sum den Wert n^2

Laufzeitanalyse – Beispiel

```
KubischeMethode(int n)
    sum = 0
    for i ← 1 to n do
        for j ← 1 to n do
            for k ← 1 to n do
                sum ← sum + 1
```

Laufzeitanalyse – Beispiel

KubischeMethode(int n)

```
1.    sum = 0
2.    for i ← 1 to n do
3.        for j ← 1 to n do
4.            for k ← 1 to n do
5.                sum ← sum + 1
```

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beispiel

Zeit:

KubischeMethode(int n)
1. sum = 0
2. **for** i ← 1 **to** n **do**
3. **for** j ← 1 **to** n **do**
4. **for** k ← 1 **to** n **do**
5. sum ← sum + 1

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beispiel

```
KubischeMethode(int n)
    sum = 0
    for i ← 1 to n do
        for j ← 1 to n do
            for k ← 1 to n do
                sum ← sum + 1
```

Zeit:

1

n+1

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beispiel

```
KubischeMethode(int n)
    sum = 0
    for i ← 1 to n do
        for j ← 1 to n do
            for k ← 1 to n do
                sum ← sum + 1
```

Zeit:

1

n+1

n * (n + 1)

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beispiel

KubischeMethode(int n)
1. sum = 0
2. **for** i ← 1 **to** n **do**
3. **for** j ← 1 **to** n **do**
4. **for** k ← 1 **to** n **do**
5. sum ← sum + 1

Zeit:
1
n+1
$n * (n + 1)$
$n^2 * (n + 1)$

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Beispiel

	Zeit:
KubischeMethode(int n)	
sum = 0	1
for i ← 1 **to** n **do**	n+1
for j ← 1 **to** n **do**	$n * (n + 1)$
for k ← 1 **to** n **do**	$n^2 * (n + 1)$
sum ← sum + 1	n^3

Nach der Ausführung hat sum den Wert n^3

Laufzeitanalyse – Größenordnungen

n	linear	quadratisch	kubisch	exponentiell
1	1 μs	1 μs	1 μs	2 μs
10	10 μs	100 μs	1 ms	1 ms
20	20 μs	400 μs	8 ms	1 sec
30	30 μs	900 μs	27 ms	18 min
40	40 μs	2 ms	64 ms	13 Tage
50	50 μs	3 ms	125 ms	36 Jahre
60	60 μs	4 ms	216 ms	36 560 Jahre
100	100 μs	10 ms	1 sec	$4 \cdot 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	...

Insertion Sort – Laufzeitanalyse

Technische Universität Berlin

```
InsertionSort(Array A)
for j ← 2 to length(A) do
    key ← A[j]
    i ← j-1
    while i>0 and A[i]>key do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

Was ist die Eingabegröße?

Open Distributed Sytems
ODS

Insertion Sort – Laufzeitanalyse

```
InsertionSort(Array A)
    for j ← 2 to length(A) do
        key ← A[j]
        i ← j-1
        while i>0 and A[i]>key do
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

Was ist die Eingabegröße?

Die Länge des Feldes A

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** j ← 2 **to** length(A) **do**
2. key ← A[j]
3. i ← j-1
4. **while** i>0 and A[i]>key **do**
5. A[i+1] ← A[i]
6. i ← i-1
7. A[i+1] ← key

Zeit:
n

Was ist die Eingabegröße?
Die Länge des Feldes A

Insertion Sort – Laufzeitanalyse

Technische Universität Berlin

InsertionSort(Array A)	Zeit:
1. **for** j ← 2 **to** length(A) **do**	n
2. key ← A[j]	n-1
3. i ← j-1	
4. **while** i>0 and A[i]>key **do**	
5. A[i+1] ← A[i]	
6. i ← i-1	
7. A[i+1] ← key	

Open Distributed Sytems
ODS

Insertion Sort – Laufzeitanalyse

	InsertionSort(Array A)	Zeit:
1.	**for** j $\leftarrow$ 2 **to** length(A) **do**	n
2.	key $\leftarrow$ A[j]	n-1
3.	i $\leftarrow$ j-1	n-1
4.	**while** i>0 and A[i]>key **do**	
5.	A[i+1] $\leftarrow$ A[i]	
6.	i $\leftarrow$ i-1	
7.	A[i+1] $\leftarrow$ key	

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for j ← 2 **to** length(A) **do**	n
key ← A[j]	$n-1$
i ← j-1	$n-1$
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] ← A[i]	
i ← i-1	
A[i+1] ← key	

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for j ← 2 **to** length(A) **do**	n
key ← A[j]	$n-1$
i ← j-1	$n-1$
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] ← A[i]	$\sum t_j$
i ← i-1	
A[i+1] ← key	

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for j ← 2 **to** length(A) **do**	n
key ← A[j]	n-1
i ← j-1	n-1
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] ← A[i]	$\sum t_j$
i ← i-1	$\sum t_j$
A[i+1] ← key	

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for $j \leftarrow 2$ **to** length(A) **do**	n
key $\leftarrow$ A[j]	$n-1$
$i \leftarrow j-1$	$n-1$
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] $\leftarrow$ A[i]	$\sum t_j$
$i \leftarrow i-1$	$\sum t_j$
A[i+1] $\leftarrow$ key	$n-1$

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

Technische Universität Berlin

InsertionSort(Array A)	Zeit:
for j $\leftarrow$ 2 **to** length(A) **do**	n
key $\leftarrow$ A[j]	$n-1$
i $\leftarrow$ j-1	$n-1$
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] $\leftarrow$ A[i]	$\sum t_j$
i $\leftarrow$ i-1	$\sum t_j$
A[i+1] $\leftarrow$ key	$n-1$
	$5n-4+3\sum t_j$

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Open Distributed Sytems
ODS

Laufzeitanalyse

- Worst-Case Analyse
  - Für jedes n definiere Laufzeit
    $T(n)$ = Maximum über alle Eingaben der Größe n
  - Garantie für jede Eingabe / „schlechtester Fall“
  - Üblich für Laufzeitanalyse

Laufzeitanalyse

- Worst-Case Analyse
  - Für jedes n definiere Laufzeit
    T(n) = Maximum über alle Eingaben der Größe n
  - Garantie für jede Eingabe / „schlechtester Fall“
  - Üblich für Laufzeitanalyse
- Average-Case Analyse
  - Für jedes n definiere Laufzeit
    T(n) = Durchschnitt über alle Eingaben der Größe n
  - Hängt von Definition des Durchschnitts ab (wie sind die Eingaben verteilt)

Laufzeitanalyse

- Worst-Case Analyse
  - Für jedes n definiere Laufzeit
    T(n) = Maximum über alle Eingaben der Größe n
  - Garantie für jede Eingabe / „schlechtester Fall“
  - Üblich für Laufzeitanalyse
- Average-Case Analyse
  - Für jedes n definiere Laufzeit
    T(n) = Durchschnitt über alle Eingaben der Größe n
  - Hängt von Definition des Durchschnitts ab (wie sind die Eingaben verteilt)
- Best-Case Analyse
  - Für jedes n definiere Laufzeit
    T(n) = Minimum über alle Eingaben der Größe n
  - „Nicht“ garantiert für jede Eingabe / „bester Fall“

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for j $\leftarrow$ 2 **to** length(A) **do**	n
key $\leftarrow$ A[j]	$n-1$
i $\leftarrow$ j-1	$n-1$
while i>0 and A[i]>key **do**	$n-1 + \sum t_j$
A[i+1] $\leftarrow$ A[i]	$\sum t_j$
i $\leftarrow$ i-1	$\sum t_j$
A[i+1] $\leftarrow$ key	$n-1$
	$5n-4+3\sum t_j$

t_j: Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse
  - t = j-1 für absteigend sortierte Eingabe (schlechtester Fall)

$$T(n) = 5n - 4 + 3\sum_{j=2}^{n}(j-1)$$

$$= 2n + 3n - 4 + 3\sum_{j=1}^{n-1} j$$

$$= 2n - 4 + 3\left(n + \sum_{j=1}^{n-1} j\right)$$

Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse

$$T(n) = 2n - 4 + 3\sum_{j=1}^{n} j$$

$$= 2n - 4 + 3\,\frac{n(n+1)}{2}$$

$$= \frac{4n - 8 + 3n^2 + 3n}{2}$$

$$T(n) = \frac{3n^2 + 7n - 8}{2}$$

Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse
  - t = j-1 für absteigend sortierte Eingabe (schlechtester Fall)

$$T(n) = 5n - 4 + 3 \cdot \sum_{j=2}^{n} (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^{n} j$$

$$= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2}$$

  - Abstraktion von multiplikativen Konstanten

→ O-Notation (Groß-O-Notation)

Laufzeitanalyse – Beobachtungen

- Diskussion

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren
  - Je nach Rechnerarchitektur oder/und genutzten Befehlen könnte also z.B. 3n+4 langsamer sein als 5n+7
    - Fall 1: b = a; b += a; b += a; Fall 2: b = 3 * a;

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren
  - Je nach Rechnerarchitektur oder/und genutzten Befehlen könnte also z.B. $3n+4$ langsamer sein als $5n+7$
    - Fall 1: b = a; b += a; b += a;   Fall 2: b = 3 * a;
  - Betrachte nun Algorithmus A mit Laufzeit $100n$ und Algorithmus B mit Laufzeit $5n^2$

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren
  - Je nach Rechnerarchitektur oder/und genutzten Befehlen könnte also z.B. $3n+4$ langsamer sein als $5n+7$
    - Fall 1: b = a; b += a; b += a; Fall 2: b = 3 * a;
  - Betrachte nun Algorithmus A mit Laufzeit $100n$ und Algorithmus B mit Laufzeit $5n^2$
    - Ist n klein, so ist Algorithmus B schneller

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren
  - Je nach Rechnerarchitektur oder/und genutzten Befehlen könnte also z.B. 3n+4 langsamer sein als 5n+7
    - Fall 1: b = a; b += a; b += a;   Fall 2: b = 3 * a;
  - Betrachte nun Algorithmus A mit Laufzeit 100n und Algorithmus B mit Laufzeit $5n^2$
    - Ist n klein, so ist Algorithmus B schneller
    - Ist n groß, so wird das Verhältnis Laufzeit B / Laufzeit A beliebig groß

Laufzeitanalyse – Beobachtungen

- Diskussion
  - Die konstanten Faktoren sind wenig aussagekräftig, da wir bereits bei den einzelnen Befehlen konstante Faktoren ignorieren
  - Je nach Rechnerarchitektur oder/und genutzten Befehlen könnte also z.B. 3n+4 langsamer sein als 5n+7
    - Fall 1: b = a; b += a; b += a; Fall 2: b = 3 * a;
  - Betrachte nun Algorithmus A mit Laufzeit 100n und Algorithmus B mit Laufzeit $5n^2$
    - Ist n klein, so ist Algorithmus B schneller
    - Ist n groß, so wird das Verhältnis Laufzeit B / Laufzeit A beliebig groß
    - Algorithmus B braucht also einen beliebigen Faktor mehr Laufzeit als A (wenn die Eingabe groß genug ist)

Asymptotische Laufzeitanalyse

- Idee: asymptotische Laufzeitanalyse
  - Ignoriere konstante Faktoren
  - Betrachte das Verhältnis von Laufzeiten für $n \to \infty$
  - Klassifiziere Laufzeiten durch Angabe von „einfachen Vergleichsfunktionen“

O-Notation – Obere Schranke

- O-Notation
  - $f(n) \in O(g(n)) = \{f(n) : \exists\ c > 0, n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n)\}$

O-Notation – Obere Schranke

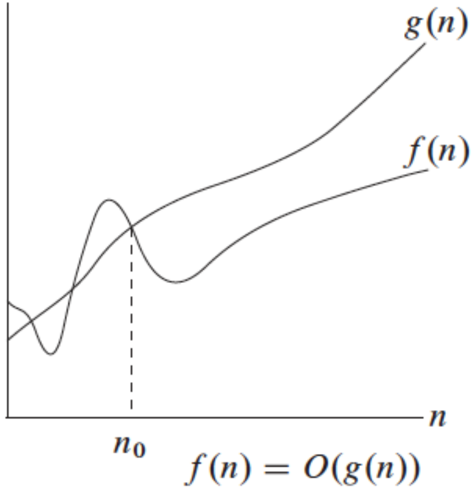
- O-Notation
  - $\boxed{f(n) \in O(g(n))} = \{f(n) : \exists\ c > 0,\ n_0 > 0,$ so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n)\}$

O-Notation – Obere Schranke

- O-Notation
  - $\boxed{f(n) \in O(g(n))} = \{f(n) :$ $f(n) \leq$ $g(n)\}$

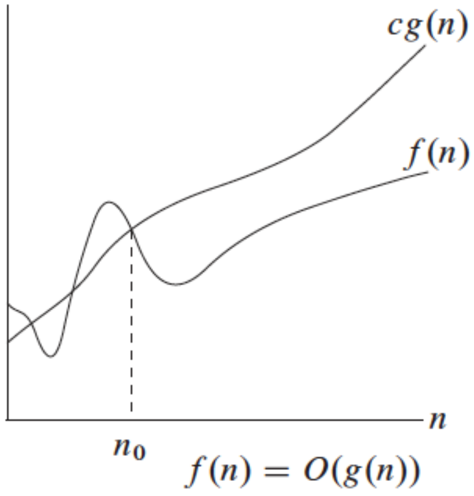
O-Notation – Obere Schranke

- O-Notation
  - $f(n) \in O(g(n)) = \{f(n) : \exists$ $n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) \leq$ $g(n)\}$



O-Notation – Obere Schranke

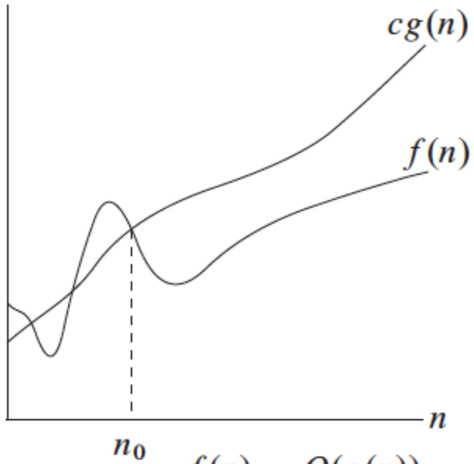
- O-Notation
  - $f(n) \in O(g(n)) = \{f(n) : \exists\ c > 0,\ n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n)\}$



O-Notation – Obere Schranke

- O-Notation
  - $f(n) \in O(g(n)) = \{f(n) : \exists\, c > 0, n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n)\}$
  - (wobei f(n), g(n) > 0)

- Interpretation
  - $f(n) \in O(g(n))$ bedeutet, dass f(n) für $n \to \infty$ höchstens genauso stark wächst wie g(n)
  - Beim Wachstum ignorieren wir Konstanten
  - Man sagt, f wird von g dominiert oder f wächst nicht stärker als g (Abschätzung nach oben)



$f(n) = O(g(n))$

O-Notation – Obere Schranke: Beispiele

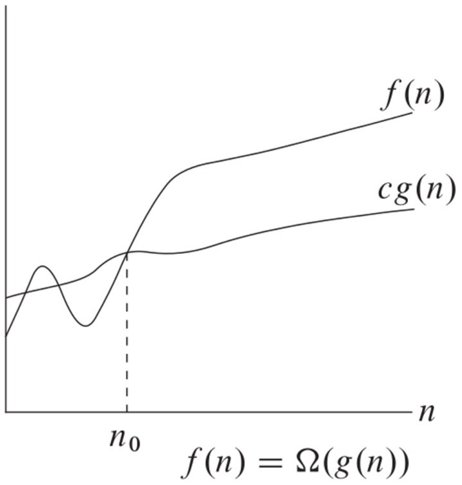
- Beispiele
  - $10\,n \in O(n)$
  - $10\,n \in O(n^2)$
  - $n^2 \notin O(1000\,n)$
  - $O(1000\,n) \in O(n)$

- Hierarchie
  - $O(\log n) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^c) \subseteq O(2^n)$ (für $c \geq 2$)

Ω-Notation – Untere Schranke

- Ω-Notation
  - $f(n) \in \Omega(g(n)) = \{g(n) : \exists\, c > 0, n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) \geq c \cdot g(n)\}$
  - (wobei f(n), g(n)>0)

- Interpretation
  - $f(n) \in \Omega(g(n))$ bedeutet, dass f(n) für $n \to \infty$ mindestens so stark wächst wie g(n)
  - Beim Wachstum ignorieren wir Konstanten



Ω-Notation – Untere Schranke: Beispiele

- Beispiele
  - $10\, n \in \Omega(n)$
  - $1000\, n \notin \Omega(n^2)$
  - $n^2 \in \Omega(n)$
  - $\Omega(1000\, n) \in \Omega(n)$

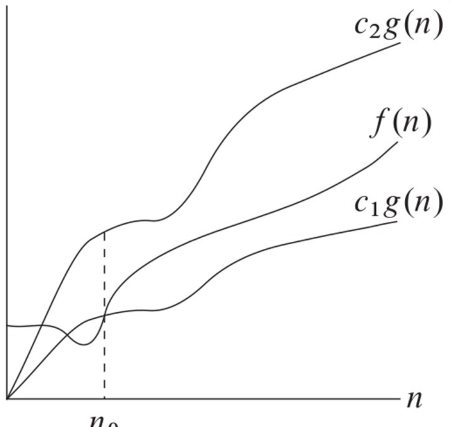
- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

Obere / untere Schranke: Beispiele

- Obere Schranke
  - $10\,n \in O(n)$
  - $10\,n \in O(n^2)$
  - $n^2 \notin O(1000\,n)$
  - $O(1000\,n) \in O(n)$
- Untere Schranke
  - $10\,n \in \Omega(n)$
  - $1000\,n \notin \Omega(n^2)$
  - $n^2 \in \Omega(n)$
  - $\Omega(1000\,n) \in \Omega(n)$

Θ-Notation – Obere und Untere Schranke

- Θ-Notation
  - $f(n) \in \Theta(g(n)) = \{f(n) : \exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, n_0 > 0, \forall\, n \geq n_0,$ sodass gilt $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$
  - $(f(n), g(n) > 0)$
  - Andere Schreibweise:
    $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ **und** $f(n) \in \Omega(g(n))$



$f(n) = \Theta(g(n))$

Θ-Notation – Obere und Untere Schranke

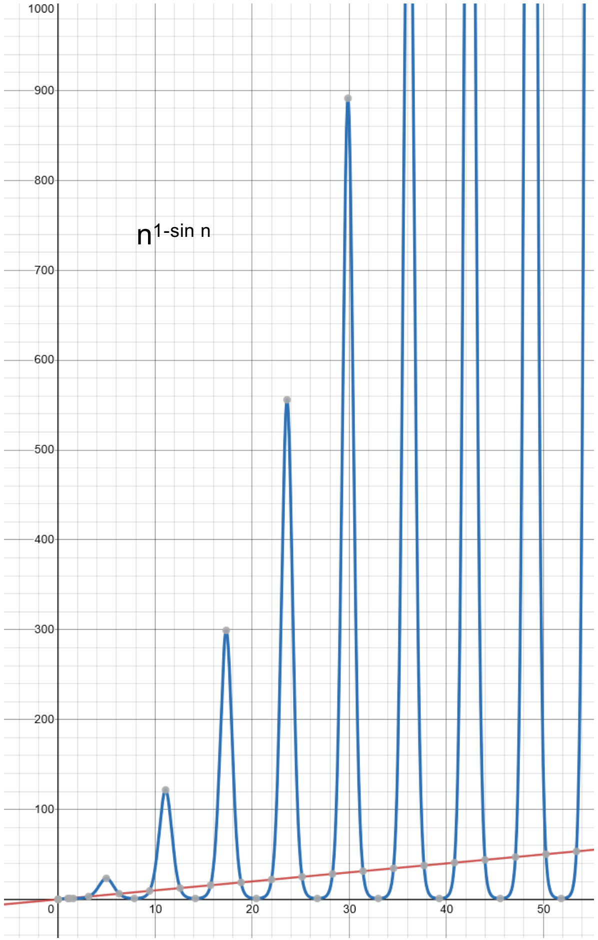
- Θ-Notation
  - $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ **und**
    $f(n) = \Omega(g(n))$

- Beispiele
  - $1000\, n \in \Theta(n)$
  - $10\, n^2 + 1000\, n \in \Theta(n^2)$
  - $n^{1-\sin n} \notin \Theta(n)$

Θ-Notation – Obere und Untere Schranke

- Θ-Notation
  - $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ **und** $f(n) = \Omega(g(n))$

- Beispiele
  - $1000\, n \in \Theta(n)$
  - $10\, n^2 + 1000\, n \in \Theta(n^2)$
  - $n^{1-\sin n} \notin \Theta(n)$



Open Distributed Sytems
ODS

Echte obere und untere Schranken

- o-Notation – echte obere Schranke
  - $o(f(n)) = \{g(n): \forall c > 0\ \exists n_0 > 0$, so dass für alle $n \geq n_0$ gilt $c \cdot g(n) < f(n)\}$
  - $(f(n), g(n) > 0)$

- ω-Notation – echte untere Schranke
  - $\omega(g(n)) = \{f(n): \forall c > 0, \exists n_0 > 0$, so dass für alle $n \geq n_0$ gilt $f(n) > c \cdot g(n)\}$
  - $(f(n), g(n) > 0)$

- Damit gilt $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

Laufzeitanalyse

- Beispiele
  - $n \in o(n^2)$
  - $n \notin o(n)$
  - $n^2 \in \omega(n)$
  - $n \notin \omega(n)$

- Eine weitere Interpretation
  - Grob gesprochen sind O, Ω, Θ, o, ω die „asymptotischen Versionen“ von $\leq$, $\geq$, $=$, $<$, $>$ (in dieser Reihenfolge)

- Schreibweise
  - Wir schreiben häufig $f(n) = O(g(n))$ anstelle von $f(n) \in O(g(n))$

Laufzeitanalyse

- Beispiele
  - $n \in o(n^2)$
  - $n \notin o(n)$
  - $n^2 \in \omega(n)$
  - $n \notin \omega(n)$

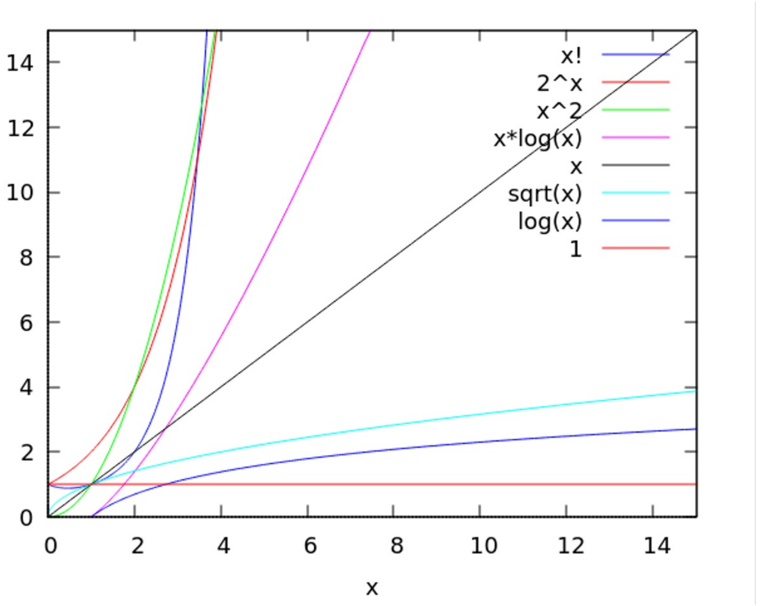
$f \in o(g)$	Wachstum von f	$<$	Wachstum von g
$f \in O(g)$	Wachstum von f	$\leq$	Wachstum von g
$f \in \Theta(g)$	Wachstum von f	$=$	Wachstum von g
$f \in \Omega(g)$	Wachstum von f	$\geq$	Wachstum von g
$f \in \omega(g)$	Wachstum von f	$>$	Wachstum von g

- Eine weitere Interpretation
  - Grob gesprochen sind O, Ω, Θ, o, ω die „asymptotischen Versionen" von $\leq$, $\geq$, $=$, $<$, $>$ (in dieser Reihenfolge)

- Schreibweise
  - Wir schreiben häufig $f(n) = O(g(n))$ anstelle von $f(n) \in O(g(n))$

Typische Laufzeitklassen

- n! – faktoriell
- $O(2^n)$ – exponentiell
- $O(n^2)$ – quadratisch (polynomiell)
- O(n log(n)) – super-linear
- O(n) – linear
- O(sqrt(x)) – Wurzelfunktion
- O(log(n)) – logarithmisch
- O(1) – beschränkt / konstant



Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse
  - $t_j = j-1$ für absteigend sortierte Eingabe (schlechtester Fall)

$$T(n) = 5n - 4 + 3 \cdot \sum_{j=2}^{n} (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^{n} j$$

$$= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} =$$

Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse
  - $t_j = j\text{-}1$ für absteigend sortierte Eingabe (schlechtester Fall)

$$T(n) = 5n - 4 + 3 \cdot \sum_{j=2}^{n} (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^{n} j$$

$$= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} = \boxed{\Theta(n^2)}$$

Insertion Sort – Laufzeitanalyse

- Worst-Case Analyse
  - t = j-1 für absteigend sortierte Eingabe (schlechtester Fall)

$$T(n) = 5n - 4 + 3 \cdot \sum_{j=2}^{n} (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^{n} j$$

$$= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} = \boxed{\Theta(n^2)}$$

  - D.h. Korrekt:
    - $O(n^2)$, $\Omega(n^2)$
    - $O(n^3)$, $\Omega(n)$

Falsch:

$o(n^2)$, $\Omega(n^3)$

$O(n)$

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)	Zeit:
for j ← 2 **to** length(A) **do**	n
key ← A[j]	n-1
i ← j-1	n-1
while i>0 and A[i]>key **do**	n-1
A[i+1] ← A[i]	
i ← i-1	
A[i+1] ← key	n-1
	5n-4

Best-Case Analyse
(aufsteigend sortiertes Array)

Insertion Sort – Laufzeitanalyse

Best-Case Analyse
(aufsteigend sortiertes Array)

InsertionSort(Array A)	Zeit:
for j ← 2 **to** length(A) **do**	n
key ← A[j]	n-1
i ← j-1	n-1
while i>0 and A[i]>key **do**	n-1
A[i+1] ← A[i]	
i ← i-1	
A[i+1] ← key	n-1
	5n-4 $= O(n)$

Selection Sort – mit swap

```
SelectionSort(Array A)
for j ←1 to length(A) - 1 do
      min ← j
      for i ← j + 1 to length(A) do
            if A[i] < A[min] then min ← i
      swap(A, min, j)
```

Idee Selection Sort

- Die ersten j-1 Elemente sind sortiert (zu Beginn j=1)
- Innerhalb eines Schleifendurchlaufs wird das j-kleineste Element (entspricht des kleinsten aus dem Rest) an die sortierte Folge „angehängt“
- Am Ende ist die gesamte Folge sortiert

Selection Sort – Worst Case Laufzeit

- Suchen des kleinsten verbleibenden Elementes:
  - Im ersten Durchlauf c“· n Operationen, dann c“·(n-1), dann c“·(n-2), usw.
  - Dann eine swap Operation
- Worst Case Laufzeit Insgesamt:

$$T(n) = c'n + c''\sum_{i=1}^{n} i = c'n + c''\frac{n(n+1)}{2} = O(n^2)$$

Bubble Sort

BubbleSort(Array A)
1. **for j** $\leftarrow$ length(A) -1 **downto** 1 **do**
2. **for** i $\leftarrow$1 **to** j **do**
3. **if** A[i] > A[i+1] **then** swap(A, i, i+1)

Idee Bubble Sort

- Die letzten Elemente von j bis n sind sortiert (zu Beginn j=n-1)
- Die größten Elemente steigen auf (bubblen), wie Luftblasen, die zu ihrer richtigen Position aufsteigen
- Am Ende ist die gesamte Folge sortiert

Bubble Sort

- Komplexität:

$$T(n) = O\left(n^2\right)$$

Count Sort

```
CountSort(Array A)

1.  C ist Hilfsarray mit 0 initialisiert

2.  for j ←1 to length(A)  do
3.          C[ A[j] ] ← C[ A[j] ]  + 1

4.  k  ← 1
5.  for j ←1 to length(C) do
6.        for i ←1  to C[j] do
7.              A[k] ← j
8.              k  ← k + 1
```

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
- Zähle, wie häufig jedes Element vorkommt

- Füge jedes Element der Reihe nach entsprechend seiner Häufigkeit in das Array hinein.

Count Sort – Laufzeit

CountSort(Array A)

```
C ist Hilfsarray mit 0 initialisiert

for j ←1 to length(A)  do
        C[ A[j] ] ← C[ A[j] ]  + 1

k  ← 1
for j ←1 to length(C) do
    for i ←1  to C[j] do
        A[k] ← j
        k  ← k + 1
```

- Annahmen:
  - Eingabegröße n
  - length(A) = n
  - **Wertebereich von A: 1 – m**
  - **length(C) = m**

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** j ← 1 **to** length(A) **do**
3. C[A[j]] ← C[A[j]] + 1

4. k ← 1
5. **for** j ← 1 **to** length(C) **do**
6. **for** i ← 1 **to** C[j] **do**
7. A[k] ← j
8. k ← k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
  - O(n)

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** j $\leftarrow$1 **to** length(A) **do**
3. C[A[j]] $\leftarrow$ C[A[j]] + 1

4. k $\leftarrow$ 1
5. **for** j $\leftarrow$1 **to** length(C) **do**
6. **for** i $\leftarrow$1 **to** C[j] **do**
7. A[k] $\leftarrow$ j
8. k $\leftarrow$ k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
  - O(n)
  - O(n)

Count Sort – Laufzeit

CountSort(Array A)

```
C ist Hilfsarray mit 0 initialisiert

for j ←1 to length(A)  do
        C[ A[j] ] ← C[ A[j] ]  + 1

k  ← 1
for j ←1 to length(C) do
    for i ←1  to C[j] do
        A[k] ← j
        k  ← k + 1
```

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
    - O(n)
    - O(n)
    - C

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** j $\leftarrow$1 **to** length(A) **do**
3. C[A[j]] $\leftarrow$ C[A[j]] + 1

4. k $\leftarrow$ 1
5. **for** j $\leftarrow$1 **to** length(C) **do**
6. **for** i $\leftarrow$1 **to** C[j] **do**
7. A[k] $\leftarrow$ j
8. k $\leftarrow$ k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
  - O(n)
  - O(n)
  - C
  - O(m)

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. for j ←1 **to** length(A) **do**
3. C[A[j]] ← C[A[j]] + 1

4. k ← 1
5. for j ←1 **to** length(C) **do**
6. **for** i ←1 **to** C[j] **do**
7. A[k] ← j
8. k ← k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
   - O(n)
   - O(n)
   - C
   - O(m)
   - O(n)

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** j $\leftarrow$1 **to** length(A) **do**
3. C[A[j]] $\leftarrow$ C[A[j]] + 1

4. k $\leftarrow$ 1
5. **for** j $\leftarrow$1 **to** length(C) **do**
6. **for** i $\leftarrow$1 **to** C[j] **do**
7. A[k] $\leftarrow$ j
8. k $\leftarrow$ k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
  - O(n)
  - O(n)
  - C
  - O(m)
  - O(n)
  - O(n)

Count Sort – Laufzeit

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. for j $\leftarrow$1 **to** length(A) **do**
3. C[A[j]] $\leftarrow$ C[A[j]] + 1

4. k $\leftarrow$ 1
5. for j $\leftarrow$1 **to** length(C) **do**
6. **for** i $\leftarrow$1 **to** C[j] **do**
7. A[k] $\leftarrow$ j
8. k $\leftarrow$ k + 1

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
- O(n)
- O(n)
- C
- O(m)
- O(n)
- O(n)
- O(n)

Count Sort – Laufzeit

Technische Universität Berlin

CountSort(Array A)

```
1.  C ist Hilfsarray mit 0 initialisiert

2.  for j ←1 to length(A)  do
3.          C[ A[j] ] ← C[ A[j] ]  + 1

4.  k ← 1
5.  for j ←1 to length(C) do
6.      for i ←1  to C[j] do
7.          A[k] ← j
8.          k ← k + 1
```

- Annahmen:
- Eingabegröße n
- length(A) = n
- **Wertebereich von A: 1 – m**
- **length(C) = m**
- O(n)
- O(n)
- C
- O(m)
- O(n)
- O(n)
- O(n)

→ O(n + m)

Open Distributed Sytems ODS

Count Sort – Worst Case Laufzeit

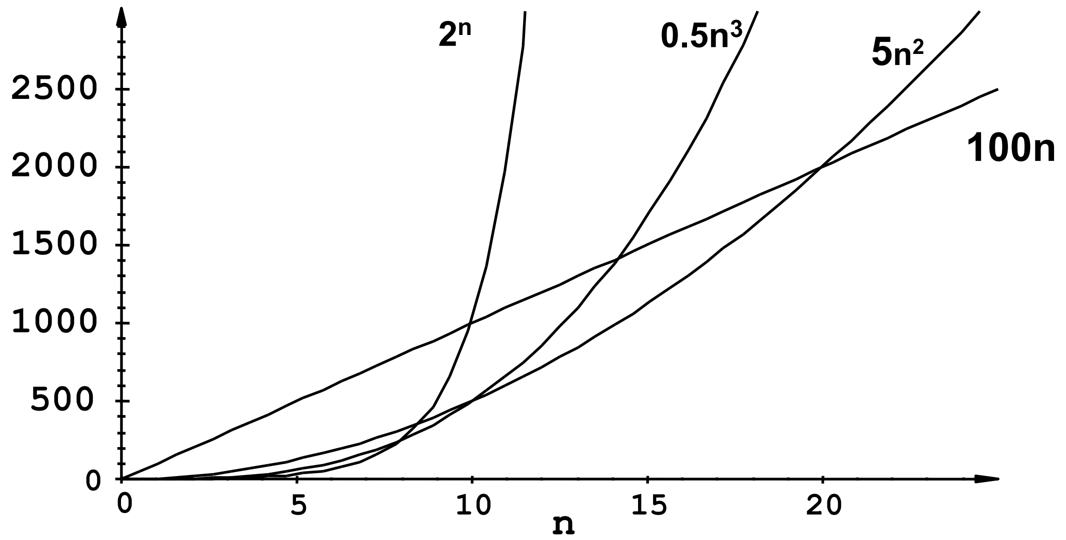
- Die Laufzeit hängt auch vom Wertebereich der Zahlen, d.h. von m ab: T(n, m)
- Worst Case Laufzeit insgesamt:

$$T(n,m) = O(n+m)$$

- Ist m = O(n), dann hat Count Sort eine lineare Laufzeit

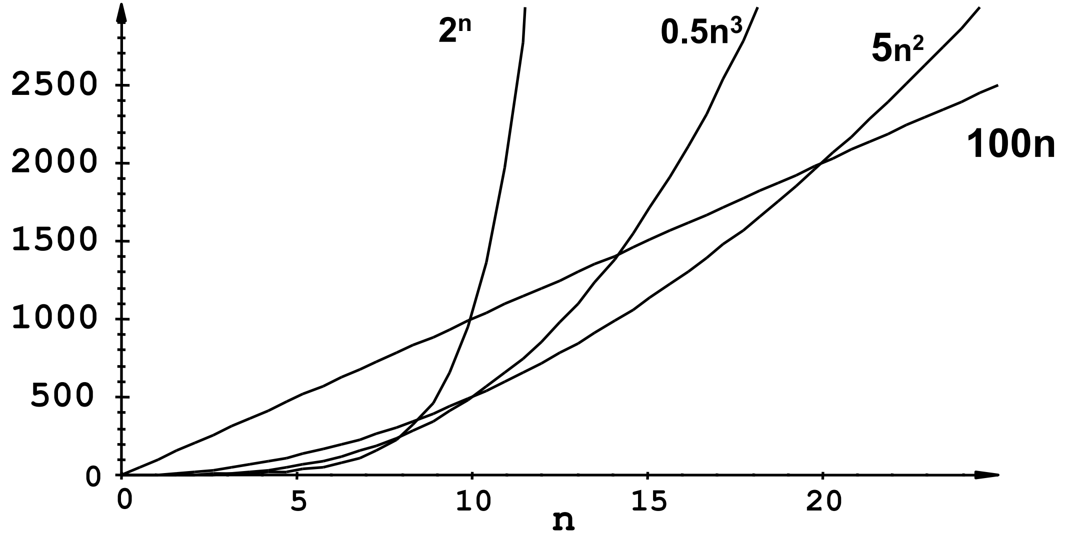
Laufzeiten – Diskussion

- Wir haben 4 Algorithmen mit den folgenden Laufzeiten, welchen wählen Sie?



Laufzeiten – Diskussion

- Wir haben 4 Algorithmen mit den folgenden Laufzeiten, welchen wählen Sie?
- Es kann sein, das für gewisse n (in diesem Fall n < 20) die Laufzeit des effizientesten Algorithmus (O(n)) nicht am besten ist!



Laufzeit – Zusammenfassung

- Rechenmodell
  - Abstrahiert von maschinennahen Einflüssen wie Cache, Pipelining, Prozessor, etc.
  - Jede Pseudocodeoperation braucht einen Zeitschritt

- Laufzeitanalyse
  - Normalerweise Worst-Case, manchmal Average-Case (sehr selten auch Best-Case)
  - Asymptotische Analyse für $n \to \infty$
  - Ignorieren von Konstanten $\to$ O-Notation

Raumkomplexität

- Speicherplatz
  - Speicherbedarf ist wichtig und ein interessantes Maß
  - Häufig jedoch kein sehr selektives Kriterium zur Unterscheidung von Algorithmen
  - Der Speicherbedarf unterschiedlicher Algorithmen für dasselbe Problem unterscheidet sich meist nur um einen (geringen) konstanten Faktor.
  - Allerdings kann zeitlicher Aufwand durch räumlichen Aufwand ersetzt werden und umgekehrt, z.B.:
    - Bei wiederholt auszuführenden identischen Berechnungen kann man das Ergebnis speichern und wiederverwenden
  - Der Speicherbedarf wächst häufig mit der Menge der Daten, mehr als quadratisches Wachstum ist selten.

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort

VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort

VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren

VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue

VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen

VL 6 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C

VL 7 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort

VL 8 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise

VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen

VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort

VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung

VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem

VL 13 „Q & A“: Offene Vorlesung/Wiederholung