

Komplexität der Algorithmen

Laufzeit

Speicherplatz

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)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Idee InsertionSort

- *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)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

- Eingabegröße n
- $\text{length}(A) = n$
- verschiebe alle Elemente aus
- $A[1 \dots j-1]$, 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:
 - Insertionsort ist schneller auf aufsteigend sortierten Eingaben
 - Insertionsort 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) an die Laufzeit
 - **Obere Schranken** "f mit $f(n) \Rightarrow T(n)$ "
 - **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

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

QuadratischeMethode (int n)

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

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)

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 (RAM)

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 (RAM)

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)

1. sum = 0

2. **for** i \leftarrow 1 **to** n **do**

3. sum \leftarrow sum + 1

Zeit:

1

n+1

n

Nach der Ausführung hat sum den Wert n

Laufzeitanalyse – Beispiel

QuadratischeMethode (int n)

1. sum = 0

2. **for** i \leftarrow 1 **to** n **do**

3. **for** j \leftarrow 1 **to** n **do**

4. sum \leftarrow sum + 1

Zeit:

1

n+1

n * (n + 1)

n²

Nach der Ausführung hat sum den Wert n²

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

$n+1$

$n * (n+1)$

$n^2 * (n+1)$

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

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Was ist die Eingabegröße?

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Was ist die Eingabegröße?

Die Länge des Feldes A

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

$\sum t_j$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

$\sum t_j$

$\sum t_j$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

$\sum t_j$

$\sum t_j$

$n-1$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

$\sum t_j$

$\sum t_j$

$n-1$

$5n-4+3 \sum t_j$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

Worst-Case Analyse

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

Average-Case Analyse

- Für jedes n definiere Laufzeit
 $T(n) = \text{Durchschnitt}$ 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
 $T(n) = \text{Minimum}$ über alle Eingaben der Größe n
- „Nicht“ garantiert für jede Eingabe / „bester Fall“

Insertion Sort – Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1 + \sum t_j$

$\sum t_j$

$\sum t_j$

$n-1$

$\overline{5n-4+3 \sum t_j}$

t_j : Anzahl Wiederholungen der while-Schleife bei Laufindex j

Insertion Sort – Laufzeitanalyse

Worst-Case Analyse

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

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

Insertion Sort – Laufzeitanalyse

Worst-Case Analyse

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

$$\begin{aligned} T(n) &= 5n - 4 + 3 \cdot \sum_{j=2}^n (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^n j \\ &= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} \end{aligned}$$

Insertion Sort – Laufzeitanalyse

Worst-Case Analyse

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

$$\begin{aligned} T(n) &= 5n - 4 + 3 \cdot \sum_{j=2}^n (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^n j \\ &= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} \end{aligned}$$

- Abstraktion von multiplikativen Konstanten

→ **O-Notation (Groß-Oh-Notation)**

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 $n \rightarrow \infty$
- Klassifiziere Laufzeiten durch Angabe von „einfachen Vergleichsfunktionen“

O-Notation – Obere Schranke

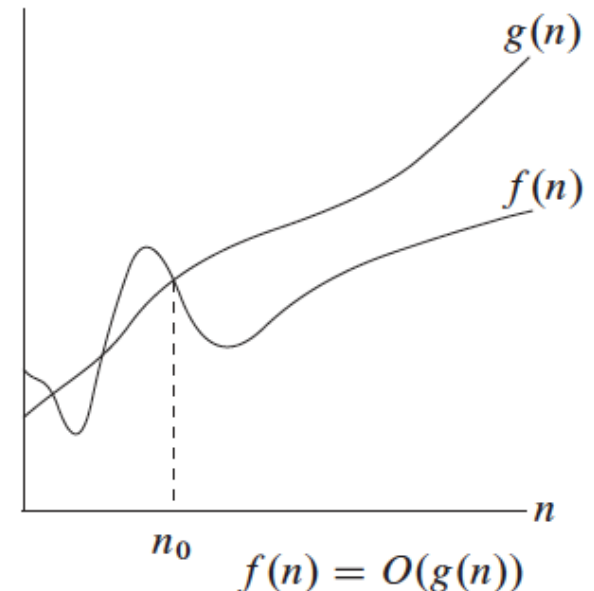
O-Notation

- $f(n) \in O(g(n)) = \{f(n) :$
- $(\text{wobei } f(n), g(n) > 0)$

$$f(n) \leq g(n)\}$$

Interpretation

- $f(n) \in O(g(n))$ bedeutet, dass $f(n)$ für $n \rightarrow \infty$ 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)



O-Notation – Obere Schranke: Beispiele

Beispiele

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

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

- $\Omega(f(n)) = \{g(n) : \exists c > 0, n_0 > 0, \text{ so dass f\u00fcr alle } n \geq n_0 \text{ gilt } g(n) \geq c \cdot f(n)\}$
- (wobei $f(n), g(n) > 0$)

Interpretation

- $g(n) \in \Omega(f(n))$ bedeutet, dass $g(n)$ f\u00fcr $n \rightarrow \infty$ **mindestens** so stark w\u00e4chst wie $f(n)$
- Beim Wachstum ignorieren wir Konstanten

Ω -Notation – Untere Schranke: Beispiele

Beispiele

- $10n \in \Omega(n)$
- $1000n \notin \Omega(n^2)$
- $n^2 \in \Omega(n)$
- $\Omega(1000n) = \Omega(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$

Θ -Notation – Obere und Untere Schranke

Θ -Notation

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

Beispiele

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

Echte obere und untere Schranken

o -Notation

- $o(f(n)) \in \{g(n): \forall c > 0 \exists n_0 > 0, \text{ so dass f\u00fcr alle } n \geq n_0 \text{ gilt } c \cdot g(n) < f(n)\}$
- $(f(n), g(n) > 0)$

ω -Notation

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

Beispiele

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

$f \in o(g)$	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 $f(n) \in O(g(n))$

Insertion Sort – Worst Case Laufzeitanalyse

Worst-Case Analyse (Insertion Sort)

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

$$\begin{aligned} T(n) &= 5n - 4 + 3 \cdot \sum_{j=2}^n (j-1) = 2n - 4 + 3 \cdot \sum_{j=1}^n j \\ &= 2n - 4 + 3 \cdot \frac{n(n+1)}{2} = \frac{3n^2 + 7n - 8}{2} = \Theta(n^2) \end{aligned}$$

- 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 – Best Case Laufzeitanalyse

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}(A)$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$

Zeit:

n

$n-1$

$n-1$

$n-1$

$n-1$

$5n-4$

$= O(n)$

Selection Sort – mit swap

SelectionSort(Array A)

1. **for** $j \leftarrow 1$ **to** $\text{length}(A) - 1$ **do**
2. $\text{min} \leftarrow j$
3. **for** $i \leftarrow j + 1$ **to** $\text{length}(A)$ **do**
4. **if** $A[i] < A[\text{min}]$ **then** $\text{min} \leftarrow i$
5. $\text{swap}(A, \text{min}, j)$

Idee SelectionSort

- *Die ersten $j-1$ Elemente sind sortiert (zu Beginn $j=1$)*
- *Innerhalb eines Schleifendurchlaufs wird das j -kleinste Element (entspricht dem kleinste 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** $j \leftarrow \text{length}(A) - 1$ **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 BubbleSort

- *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:

$$T(n) = O(n^2)$$

CountSort(Array A)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** $j \leftarrow 1$ **to** $\text{length}(A)$ **do**

3. $C[A[j]] \leftarrow C[A[j]] + 1$

4. $k \leftarrow 1$

5. **for** $j \leftarrow 1$ **to** $\text{length}(C)$ **do**

6. **for** $i \leftarrow 1$ **to** $C[j]$ **do**

7. $A[k] \leftarrow j$

8. $k \leftarrow k + 1$

➤ Annahmen:

➤ Eingabegröße n

➤ $\text{length}(A) = n$

➤ **Wertebereich von A: $1 - m$**

➤ **$\text{length}(C) = m$**

➤ 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)

1. C ist Hilfsarray mit 0 initialisiert

2. **for** $j \leftarrow 1$ **to** $\text{length}(A)$ **do**

3. $C[A[j]] \leftarrow C[A[j]] + 1$

4. $k \leftarrow 1$

5. **for** $j \leftarrow 1$ **to** $\text{length}(C)$ **do**

6. **for** $i \leftarrow 1$ **to** $C[j]$ **do**

7. $A[k] \leftarrow j$

8. $k \leftarrow k + 1$

➤ Annahmen:

➤ Eingabegröße n

➤ $\text{length}(A) = n$

➤ **Wertebereich von A: 1 – m**

➤ **$\text{length}(C) = m$**

➤ $O(n)$

➤ $O(n)$

➤ c

➤ $O(m)$

➤ $O(n)$

➤ $O(n)$

➤ $O(n)$

$O(n + m)$

Count Sort – Worst Case Laufzeit

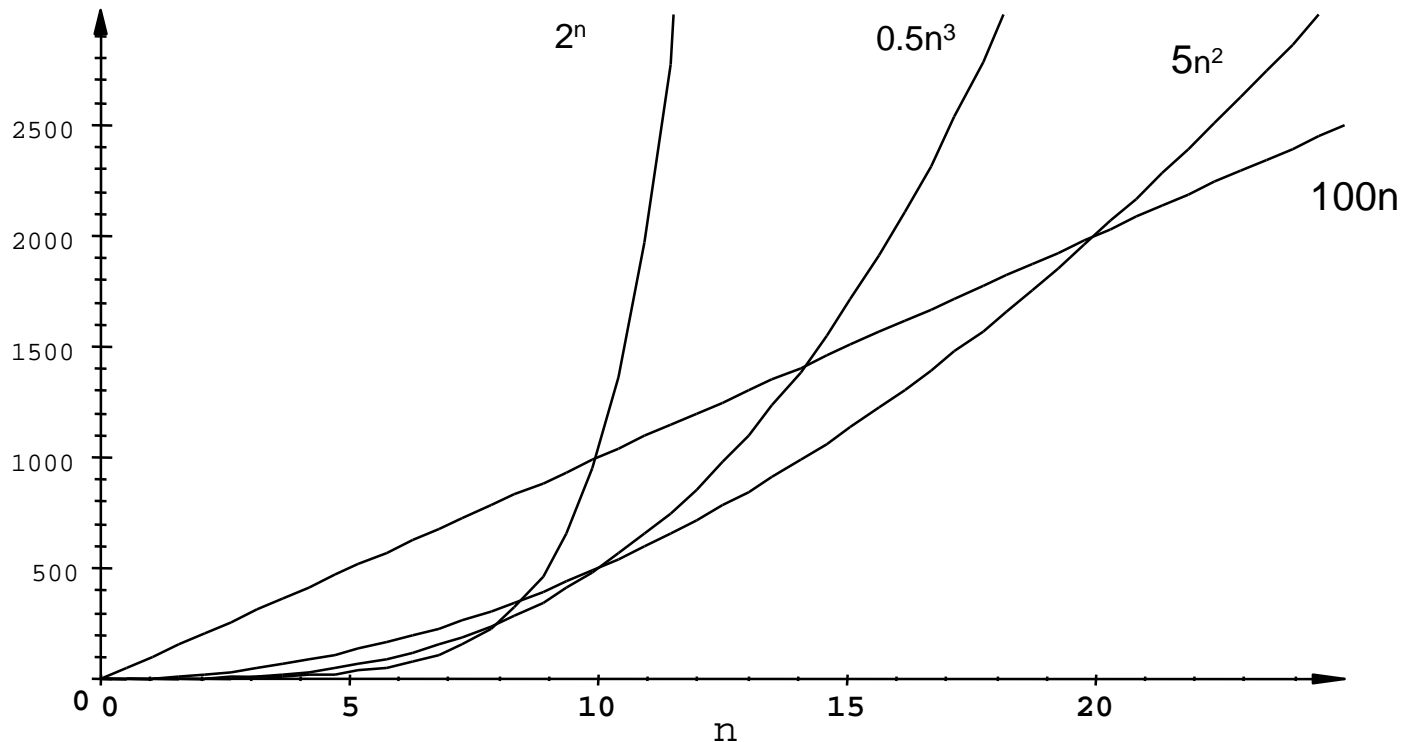
- Die Laufzeit hängt auch vom Wertebereich der Zahlen, d.h. von m , ab:
 $T(n, m)$
- Worst Case Laufzeit Insgesamt:

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

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

Laufzeiten – Diskussion

- Wir haben 4 Algorithmen mit den folgenden Laufzeiten, welchen wählen Sie?
- Es kann sein, dass 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 $n \rightarrow \infty$
- Ignorieren von Konstanten \rightarrow O-Notation

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.