Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

Motivation und Überblick

1. Vorlesung

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

- Kochrezept,

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

- Kochrezept,
- Wegsuche im Navigationsgerät,

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

- Kochrezept,
- Wegsuche im Navigationsgerät,
- Anfragen an Datenbanken und Suchmaschinen,

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

- Kochrezept,
- Wegsuche im Navigationsgerät,
- Anfragen an Datenbanken und Suchmaschinen,
- Raumplanung für Vorlesungen

Motivation

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

al-Chwarizmi
(ca. 780– 835)
persischer
Universalgelehrter
(Mathe, Astro, Geo)

Algorithmen sind allgegenwärtig:

- Kochrezept,
- Wegsuche im Navigationsgerät,
- Anfragen an Datenbanken und Suchmaschinen,
- Raumplanung für Vorlesungen
- ...

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- **Was wird gelöst:** Spezifikation des Problems.

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- **Was wird gelöst:** Spezifikation des Problems.
- **Wie wird es gelöst:** Beschreibung des Algorithmus zur Lösung des Problems.

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- **Was wird gelöst:** Spezifikation des Problems.
- **Wie wird es gelöst:** Beschreibung des Algorithmus zur Lösung des Problems.
- **Warum funktioniert das:** Korrektheitsbeweis (inkl. Terminierung).

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- **Was wird gelöst:** Spezifikation des Problems.
- **Wie wird es gelöst:** Beschreibung des Algorithmus zur Lösung des Problems.
- **Warum funktioniert das:** Korrektheitsbeweis (inkl. Terminierung).
- **Wie schnell:** Analyse der Laufzeit des Algorithmus.

Was ist Algorithmentheorie?

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems bestehend aus endlich vielen, wohldefinierten Einzelschritten.

Zu Entwurf und Analyse von Algorithmen gehören:

- **Was wird gelöst:** Spezifikation des Problems.
- **Wie wird es gelöst:** Beschreibung des Algorithmus zur Lösung des Problems.
- **Warum funktioniert das:** Korrektheitsbeweis (inkl. Terminierung).
- **Wie schnell:** Analyse der Laufzeit des Algorithmus.

Weiteres: Speicherplatz, Eleganz, etc.

Ziele der Algorithmentheorie

- Möglichst effiziente Algorithmen (und Datenstrukturen) für die verschiedensten Probleme entwerfen.

Ziele der Algorithmentheorie

- Möglichst effiziente Algorithmen (und Datenstrukturen) für die verschiedensten Probleme entwerfen.
- Techniken und Paradigmen zum Algorithmenentwurf bereitstellen.

Ziele der Algorithmentheorie

- Möglichst effiziente Algorithmen (und Datenstrukturen) für die verschiedensten Probleme entwerfen.
- Techniken und Paradigmen zum Algorithmenentwurf bereitstellen.
- Korrektheit der Algorithmen beweisen und Laufzeit analysieren.

Ziele der Algorithmentheorie

- Möglichst effiziente Algorithmen (und Datenstrukturen) für die verschiedensten Probleme entwerfen.
- Techniken und Paradigmen zum Algorithmenentwurf bereitstellen.
- Korrektheit der Algorithmen beweisen und Laufzeit analysieren.
- Implementierungen und Evaluationen für Standardalgorithmen bereitstellen.

Ziele der Vorlesung

- Breite Auswahl an wichtigen Problemen und Algorithmen zur Lösung vorstellen.

Ziele der Vorlesung

- Breite Auswahl an wichtigen Problemen und Algorithmen zur Lösung vorstellen.
- Wichtigste Paradigmen zum Algorithmenentwurf vermitteln.

Ziele der Vorlesung

- Breite Auswahl an wichtigen Problemen und Algorithmen zur Lösung vorstellen.
- Wichtigste Paradigmen zum Algorithmenentwurf vermitteln.
  $\rightarrow$ Die Vorlesung ist methodenbasiert aufgebaut.

Ziele der Vorlesung

- Breite Auswahl an wichtigen Problemen und Algorithmen zur Lösung vorstellen.
- Wichtigste Paradigmen zum Algorithmenentwurf vermitteln.
  $\rightarrow$ Die Vorlesung ist methodenbasiert aufgebaut.
- Soll euch in die Lage versetzen, Praxisprobleme zu formalisieren und effiziente Lösungen zu finden.

Ziele der Vorlesung

- Breite Auswahl an wichtigen Problemen und Algorithmen zur Lösung vorstellen.
- Wichtigste Paradigmen zum Algorithmenentwurf vermitteln.
  $\rightarrow$ Die Vorlesung ist methodenbasiert aufgebaut.
- Soll euch in die Lage versetzen, Praxisprobleme zu formalisieren und effiziente Lösungen zu finden.
- Ihr sollt in der Lage sein, die Korrektheit von Algorithmen zu beweisen und ihre Laufzeit zu analysieren.

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit
- Rekursion, Divide & Conquer, Rekursionsgleichungen

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit
- Rekursion, Divide & Conquer, Rekursionsgleichungen
- Greedy Algorithmen

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit
- Rekursion, Divide & Conquer, Rekursionsgleichungen
- Greedy Algorithmen
- Dynamische Programmierung

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit
- Rekursion, Divide & Conquer, Rekursionsgleichungen
- Greedy Algorithmen
- Dynamische Programmierung
- Sortieren, Packungsprobleme, Scheduling

Themenüberblick

- Algorithmenbegriff und Schreibweise (Pseudocode)
- Algorithmische Komplexität und asymptotische Laufzeit
- Rekursion, Divide & Conquer, Rekursionsgleichungen
- Greedy Algorithmen
- Dynamische Programmierung
- Sortieren, Packungsprobleme, Scheduling
- Graphenalgorithmen: kürzeste Wege, minimale Spannbäume, maximale Netzwerkflüsse und maximale Matchings

Fragen?

Algorithmenbegriff und Sortieren

Algorithmenbegriff

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems, besteht aus endlich vielen, wohldefinierten Einzelschritten.

Algorithmenbegriff

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems, besteht aus endlich vielen, wohldefinierten Einzelschritten.

Beispiel: Sortierproblem

Gegeben sei ein Array A mit n natürlichen Zahlen. Sortiere alle Elemente in A so um, dass für jedes $0 \leq i \leq n-2$ gilt $A[i] \leq A[i+1]$.

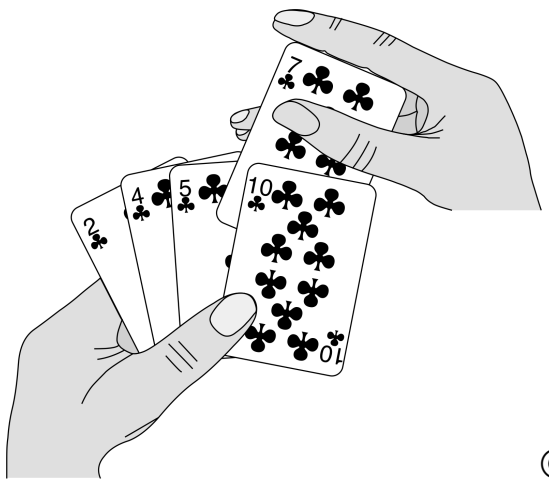
Algorithmenbegriff

Algorithmus

Eine eindeutige Handlungsvorschrift zur Lösung eines Problems, besteht aus endlich vielen, wohldefinierten Einzelschritten.

Beispiel: Sortierproblem

Gegeben sei ein Array A mit n natürlichen Zahlen. Sortiere alle Elemente in A so um, dass für jedes $0 \leq i \leq n-2$ gilt $A[i] \leq A[i+1]$.

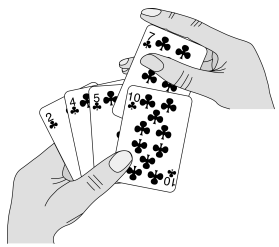


© Corman et al.

Insertion Sort

`Insertion Sort` (informell)

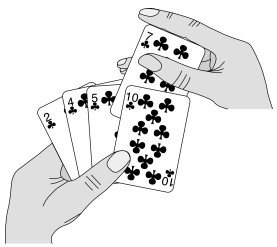
- Iteriere durch das Eingabearray A von links nach rechts.



Insertion Sort

`Insertion Sort` (informell)

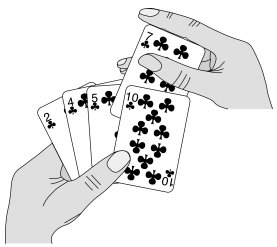
- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.



Insertion Sort

`Insertion Sort` (informell)

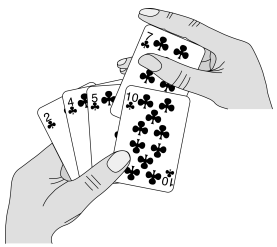
- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.
- Füge es an der richtigen Stelle im sortierten Teil von A ein.



Insertion Sort

`Insertion Sort` (informell)

- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.
- Füge es an der richtigen Stelle im sortierten Teil von A ein.

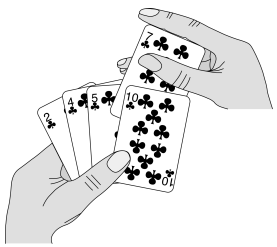


Liegt hier ein Algorithmus vor?

Insertion Sort

`Insertion Sort` (informell)

- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.
- Füge es an der richtigen Stelle im sortierten Teil von A ein.



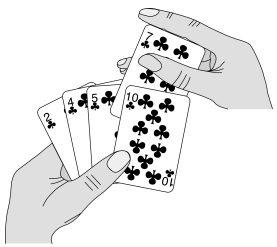
Liegt hier ein Algorithmus vor?

- Eindeutige Handlungsvorschrift?

Insertion Sort

`Insertion Sort` (informell)

- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.
- Füge es an der richtigen Stelle im sortierten Teil von A ein.



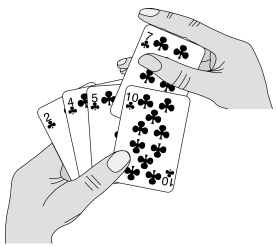
Liegt hier ein Algorithmus vor?

- Eindeutige Handlungsvorschrift?
- Was bedeutet z.B. "füge ein Element an der richtigen Stelle ein"?

Insertion Sort

`Insertion Sort` (informell)

- Iteriere durch das Eingabearray A von links nach rechts.
- Gehe zum ersten noch nicht behandelten Element in A.
- Füge es an der richtigen Stelle im sortierten Teil von A ein.



Liegt hier ein Algorithmus vor?

- Eindeutige Handlungsvorschrift?
- Was bedeutet z.B. "füge ein Element an der richtigen Stelle ein"?

Wir brauchen eine Formalisierung des Algorithmenbegriffs!

Problembegriff

- Menge von Eingaben A, Menge von Ausgaben B.

Problembegriff

- Menge von Eingaben A, Menge von Ausgaben B.
- Problem: (Partielle) Zuordnung der Eingaben zu den Ausgaben, d.h. eine (partielle) Funktion $P\colon A \to B$.

Problembegriff

- Menge von Eingaben A, Menge von Ausgaben B.
- Problem: (Partielle) Zuordnung der Eingaben zu den Ausgaben, d.h. eine (partielle) Funktion $P\colon A \to B$.
- Algorithmus soll zu einer gegebenen Eingabe $v \in A$ eine Ausgabe $w \in B$ berechnen.

Problembegriff

- Menge von Eingaben A, Menge von Ausgaben B.
- Problem: (Partielle) Zuordnung der Eingaben zu den Ausgaben, d.h. eine (partielle) Funktion $P\colon A \to B$.
- Algorithmus soll zu einer gegebenen Eingabe $v \in A$ eine Ausgabe $w \in B$ berechnen.
- Der Algorithmus löst Problem P falls er zu jeder Eingabe $v \in A$, auf der P definiert ist, die Ausgabe $w = P(v)$ berechnet.

Problembegriff

- Problem: (Partielle) Funktion $P : A \to B$.

Problembegriff

- Problem: (Partielle) Funktion $P : A \to B$.
- Unsere Probleme: Partielle Funktionen $P : \mathbb{Z}^* \to \mathbb{Z}^*$.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.
  - Input und Output sind also Strings aus ganzen Zahlen.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.
  - Input und Output sind also Strings aus ganzen Zahlen.
- Bitstrings, d.h. Wörter über dem Alphabet $\{0, 1\}$, würden ausreichen, da alle sinnvollen Eingaben als Bitstrings kodiert werden können.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.
  - Input und Output sind also Strings aus ganzen Zahlen.
- Bitstrings, d.h. Wörter über dem Alphabet $\{0, 1\}$, würden ausreichen, da alle sinnvollen Eingaben als Bitstrings kodiert werden können.
  - Kodierung von Zahlen in konkreten Implementierungen notwendig, aber zur Beschreibung von Algorithmen oft unbequem.

Problembegriff

- Problem: (Partielle) Funktion $P\colon A \to B$.
- Unsere Probleme: Partielle Funktionen $P\colon \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.
  - Input und Output sind also Strings aus ganzen Zahlen.
- Bitstrings, d.h. Wörter über dem Alphabet $\{0, 1\}$, würden ausreichen, da alle sinnvollen Eingaben als Bitstrings kodiert werden können.
  - Kodierung von Zahlen in konkreten Implementierungen notwendig, aber zur Beschreibung von Algorithmen oft unbequem.
  - Standardoperationen wie Addition, Multiplikation, ... können auf kleinen Zahlen (32/64-bit) in konstanter Zeit ausgeführt werden.

Problembegriff

- Problem: (Partielle) Funktion $P : A \to B$.
- Unsere Probleme: Partielle Funktionen $P : \mathbb{Z}^* \to \mathbb{Z}^*$.
  - $\mathbb{Z}$: die ganzen Zahlen.
  - Σ^*: die Menge alle endlichen Wörter über einem Alphabet Σ.
  - Input und Output sind also Strings aus ganzen Zahlen.
- Bitstrings, d.h. Wörter über dem Alphabet $\{0, 1\}$, würden ausreichen, da alle sinnvollen Eingaben als Bitstrings kodiert werden können.
  - Kodierung von Zahlen in konkreten Implementierungen notwendig, aber zur Beschreibung von Algorithmen oft unbequem.
  - Standardoperationen wie Addition, Multiplikation, ... können auf kleinen Zahlen (32/64-bit) in konstanter Zeit ausgeführt werden.
  - Auf großen Zahlen (1000+ bit z.B. in kryptographische Anwendungen) müssen wir zur Binärkodierung übergehen um Laufzeiten richtig abzuschätzen.

Problembegriff

Beispiel: Das Sortierproblem ist die Abbildung `sort`, die ein Wort $a_1 \dots a_n$ bestehend aus n Zahlen auf das Wort $a_{i_1} \dots a_{i_n}$ abbildet, so dass $a_{i_j} \leq a_{i_k}$ für alle $1 \leq j \leq k \leq n$.

Problembegriff

Beispiel: Das Sortierproblem ist die Abbildung `sort`, die ein Wort $a_1 \dots a_n$ bestehend aus n Zahlen auf das Wort $a_{i_1} \dots a_{i_n}$ abbildet, so dass $a_{i_j} \leq a_{i_k}$ für alle $1 \leq j \leq k \leq n$.

- Wir betrachten oft mathematische Objekte oder kompliziertere Datenstrukturen als Eingaben und beschreiben nicht formal wie diese kodiert sind.

Problembegriff

Beispiel: Das Sortierproblem ist die Abbildung `sort`, die ein Wort $a_1 \dots a_n$ bestehend aus n Zahlen auf das Wort $a_{i_1} \dots a_{i_n}$ abbildet, so dass $a_{i_j} \leq a_{i_k}$ für alle $1 \leq j \leq k \leq n$.

- Wir betrachten oft mathematische Objekte oder kompliziertere Datenstrukturen als Eingaben und beschreiben nicht formal wie diese kodiert sind.
- Die Kodierung ist immer möglich und wir sollten uns klarmachen, wie eine solche Kodierung aussieht, da sie die Laufzeit der Algorithmen maßgeblich bestimmen kann.

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)
- ...

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)
- ...

Berechnung/Algorithmus

- Eine Eingabe liegt in einem Speicher;

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)
- ...

Berechnung/Algorithmus

- Eine Eingabe liegt in einem Speicher;
- es wird eine feste endliche Folge von Befehlen aus einem festen endlichen Befehlssatz ausgeführt;

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)
- ...

Berechnung/Algorithmus

- Eine Eingabe liegt in einem Speicher;
- es wird eine feste endliche Folge von Befehlen aus einem festen endlichen Befehlssatz ausgeführt;
- jeder Befehl manipuliert den Speicher nach festen Regeln;

Maschinenmodelle

Viele verschiedene Maschinenmodelle zur Formalisierung des Begriffs der Berechnung bzw. des Algorithmus.

- Turingmaschine
- Random Access Machine (RAM)
- ...

Berechnung/Algorithmus

- Eine Eingabe liegt in einem Speicher;
- es wird eine feste endliche Folge von Befehlen aus einem festen endlichen Befehlssatz ausgeführt;
- jeder Befehl manipuliert den Speicher nach festen Regeln;
- das Ergebnis der Berechnung wird aus dem Speicher abgelesen.

Wie beschreibt man einen Algorithmus?

Pseudocode:

- semi-formelle Beschreibung des Algorithmus
- nicht vom Rechner ausführbar
- nicht standardisiert

Wie beschreibt man einen Algorithmus?

Pseudocode:

- semi-formelle Beschreibung des Algorithmus
- nicht vom Rechner ausführbar
- nicht standardisiert

Beispiel: Berechnung des Maximum zweier Zahlen

Input: a, b
Output: max(a,b)

1. $x = a$
2. **If** $b > a$ **then** $x = b$
3. **Return** x

Do-While, Repeat, For-Schleifen, rekursive Aufrufe, etc.

Pseudocode

- Soll der Veranschaulichung eines Algorithmus dienen.

Pseudocode

- Soll der Veranschaulichung eines Algorithmus dienen.
- Kann weitere Befehle als Abkürzungen enthalten, sogar natürliche Sprache und mathematische Notation, um von technischen Details zu abstrahieren.

Pseudocode

- Soll der Veranschaulichung eines Algorithmus dienen.
- Kann weitere Befehle als Abkürzungen enthalten, sogar natürliche Sprache und mathematische Notation, um von technischen Details zu abstrahieren.
- Müssen immer sicherstellen, dass es sich um eindeutige, wohldefinierte Anweisungen handelt.
- Für Aufwandsabschätzungen: bewusst machen, was die Kosten für jede benutzte Anweisung sind.

Pseudocode

- Soll der Veranschaulichung eines Algorithmus dienen.
- Kann weitere Befehle als Abkürzungen enthalten, sogar natürliche Sprache und mathematische Notation, um von technischen Details zu abstrahieren.
- Müssen immer sicherstellen, dass es sich um eindeutige, wohldefinierte Anweisungen handelt.
- Für Aufwandsabschätzungen: bewusst machen, was die Kosten für jede benutzte Anweisung sind.

Input: Array `A` der Länge `n`
Output: min(`A`)

1. sortiere `A`
2. **Return** `A[0]`

Wie lange braucht Schritt 1?
Wie sortieren wir ein Array?

Pseudocode Insertion Sort

Data: Array $\mathtt{A}[0, \ldots, \mathtt{n}-1]$ mit Zahlen aus $\mathbb{N}$

Pseudocode Insertion Sort

Data: Array $\mathtt{A}[\mathtt{0}, \ldots, \mathtt{n}-\mathtt{1}]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
  |
  |
  |
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
  |   |
  |
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
  |
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

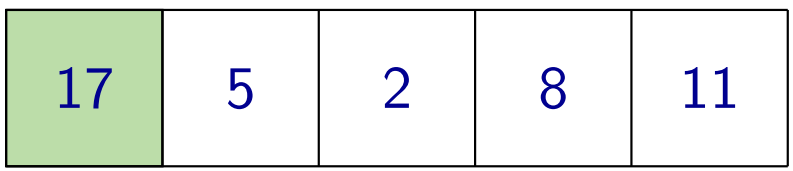
```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 1$

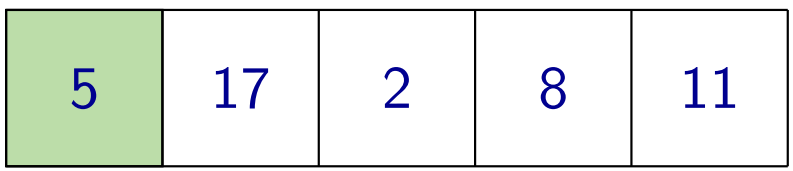
$j = 1$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 1$

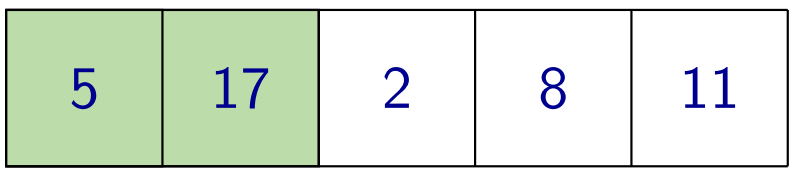
$j = 0$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 2$

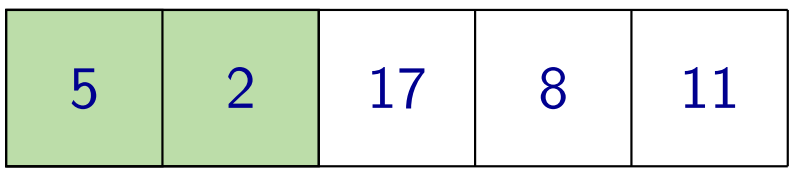
$j = 2$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 2$

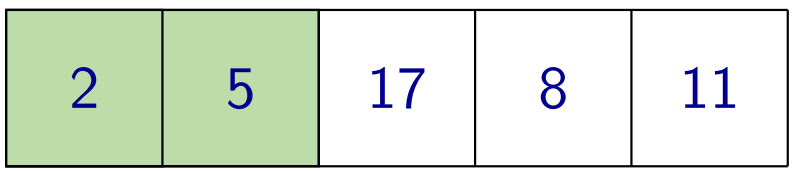
$j = 1$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



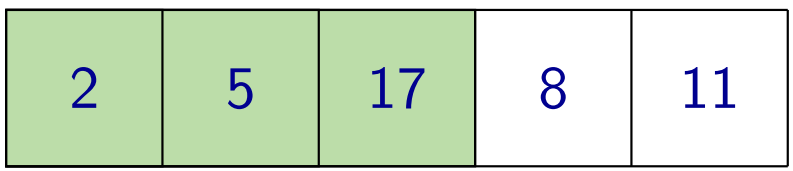
$i = 2$
$j = 0$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 3$

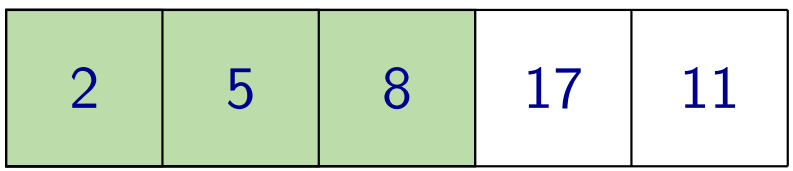
$j = 3$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 3$

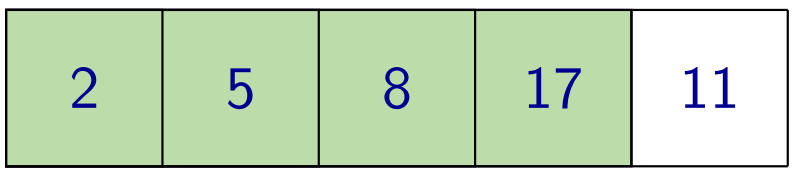
$j = 2$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 4$

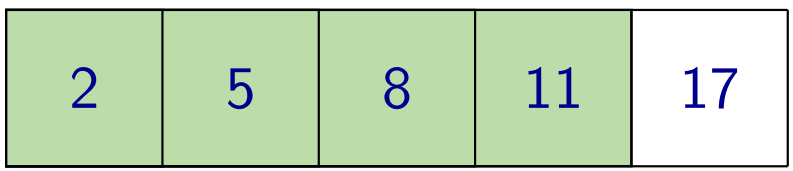
$j = 4$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:



$i = 4$

$j = 3$

Pseudocode Insertion Sort

Data: Array $A[0, \ldots, n-1]$ mit Zahlen aus $\mathbb{N}$
Result: Array in aufsteigender Reihenfolge sortiert

```
1 i := 1
2 while i < length(A) do
3 |   j := i
4 |   while j > 0 and A[j − 1] > A[j] do
5 |   |   swap A[j] and A[j − 1]
6 |   |   j := j − 1
7 |   i := i + 1
```

Bsp:

2	5	8	11	17

$i = 5$

$j = 3$

Analyse Insertion Sort

- Der Algorithmus arbeitet in-place, d.h. auf dem Eingabearray A und benötigt keinen zusätzlichen Speicherplatz.

Analyse `Insertion Sort`

- Der Algorithmus arbeitet in-place, d.h. auf dem Eingabearray A und benötigt keinen zusätzlichen Speicherplatz.
- In der i-ten Iteration der äußeren **while**-Schleife wird das Element `A[i]` mit Hilfe der inneren **while**-Schleife an die richtige Position gebracht.

Analyse `Insertion Sort`

- Der Algorithmus arbeitet in-place, d.h. auf dem Eingabearray A und benötigt keinen zusätzlichen Speicherplatz.
- In der i-ten Iteration der äußeren **while**-Schleife wird das Element $\texttt{A[i]}$ mit Hilfe der inneren **while**-Schleife an die richtige Position gebracht.
- Dazu wird $\texttt{A[i]}$ iterativ mit den Werten $\texttt{A[i} - 1\texttt{]}, \texttt{A[i} - 2\texttt{]}, \ldots$ verglichen und getauscht, bis die richtige Position gefunden ist.

Analyse `Insertion Sort`

Satz: Korrektheit `Insertion Sort`

Der Algorithmus terminiert und die Ausgabe von `Insertion Sort` ist stets eine aufsteigend sortierte Permutation der Eingabe.

Analyse `Insertion Sort`

Satz: Korrektheit `Insertion Sort`

Der Algorithmus terminiert und die Ausgabe von `Insertion Sort` ist stets eine aufsteigend sortierte Permutation der Eingabe.

Beweis. Per Induktion mit dem Konzept der Schleifeninvarianten.

Analyse `Insertion Sort`

Satz: Korrektheit `Insertion Sort`

Der Algorithmus terminiert und die Ausgabe von `Insertion Sort` ist stets eine aufsteigend sortierte Permutation der Eingabe.

Beweis. Per Induktion mit dem Konzept der Schleifeninvarianten.

- Es sei $\mathtt{A} = \mathtt{A}[0, \ldots, \mathtt{n}-1] = [\mathtt{a}_0, \ldots, \mathtt{a}_{\mathtt{n}-1}]$ ein Array mit n Zahlen.

Analyse Insertion Sort

Satz: Korrektheit Insertion Sort

Der Algorithmus terminiert und die Ausgabe von `Insertion Sort` ist stets eine aufsteigend sortierte Permutation der Eingabe.

Beweis. Per Induktion mit dem Konzept der Schleifeninvarianten.

- Es sei $\mathtt{A} = \mathtt{A}[0, \ldots, \mathtt{n}-1] = [\mathtt{a}_0, \ldots, \mathtt{a}_{\mathtt{n}-1}]$ ein Array mit n Zahlen.
- Schleifeninvariante: Bei Überprüfung der Abbruchbedingung in Zeile 2 ($\mathtt{i} < \mathtt{length(A)}$) ist
  - $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{\mathtt{i}-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ und es gilt $a_{j_k} \leq a_{j_\ell}$ für $j_k \leq j_\ell$ (die ersten i Zahlen aufsteigend sortiert)

Analyse Insertion Sort

Satz: Korrektheit Insertion Sort

Der Algorithmus terminiert und die Ausgabe von Insertion Sort ist stets eine aufsteigend sortierte Permutation der Eingabe.

Beweis. Per Induktion mit dem Konzept der Schleifeninvarianten.

- Es sei $\mathtt{A} = \mathtt{A}[0, \ldots, \mathtt{n}-1] = [\mathtt{a}_0, \ldots, \mathtt{a}_{\mathtt{n}-1}]$ ein Array mit n Zahlen.
- Schleifeninvariante: Bei Überprüfung der Abbruchbedingung in Zeile 2 ($\mathtt{i} < \mathtt{length(A)}$) ist
  - $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{\mathtt{i}-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ und es gilt $a_{j_k} \leq a_{j_\ell}$ für $j_k \leq j_\ell$ (die ersten i Zahlen aufsteigend sortiert)
  - und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).

Analyse Insertion Sort

Satz: Korrektheit Insertion Sort

Der Algorithmus terminiert und die Ausgabe von Insertion Sort ist stets eine aufsteigend sortierte Permutation der Eingabe.

Beweis. Per Induktion mit dem Konzept der Schleifeninvarianten.

- Es sei $\mathtt{A} = \mathtt{A}[0, \ldots, \mathtt{n}-1] = [\mathtt{a}_0, \ldots, \mathtt{a}_{\mathtt{n}-1}]$ ein Array mit n Zahlen.
- Schleifeninvariante: Bei Überprüfung der Abbruchbedingung in Zeile 2 ($\mathtt{i} < \mathtt{length(A)}$) ist
  - $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{\mathtt{i}-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ und es gilt $a_{j_k} \leq a_{j_\ell}$ für $j_k \leq j_\ell$ (die ersten i Zahlen aufsteigend sortiert)
  - und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).

Analyse `Insertion Sort`

Induktionsanfang:

- Im ersten Schleifendurchlauf gilt $i = 1$ und die Schleife wurde noch nicht ausgeführt.

Analyse Insertion Sort

Induktionsanfang:

- Im ersten Schleifendurchlauf gilt $i = 1$ und die Schleife wurde noch nicht ausgeführt.
- Also ist $\mathtt{A}[0, \ldots, 0] = [\mathtt{a}_0]$ aufsteigend sortiert und enthält genau die erste Zahl, wie behauptet.

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$.
Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$.
Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$.
Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).
- In den Zeilen 3 bis 7 werden nun alle Einträge aus $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$, die größer als das einzusortierende Element $\mathtt{a}_\mathtt{i}$ sind, iterativ um eine Position nach rechts geschoben.

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$. Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).
- In den Zeilen 3 bis 7 werden nun alle Einträge aus $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$, die größer als das einzusortierende Element $\mathtt{a}_\mathtt{i}$ sind, iterativ um eine Position nach rechts geschoben.
- Sei k minimal, so dass $j_k > 0$ und $a_{j_k} > a_i$.

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$.
Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).
- In den Zeilen 3 bis 7 werden nun alle Einträge aus $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$, die größer als das einzusortierende Element $\mathtt{a}_\mathtt{i}$ sind, iterativ um eine Position nach rechts geschoben.
- Sei k minimal, so dass $j_k > 0$ und $a_{j_k} > a_i$.
- Am Ende des Schleifendurchlaufs gilt dann
  - $\mathtt{A}[0, \ldots, \mathtt{i}] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{k-1}}, \mathtt{a}_\mathtt{i}, \mathtt{a}_{\mathtt{j}_k}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$. Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).
- In den Zeilen 3 bis 7 werden nun alle Einträge aus $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$, die größer als das einzusortierende Element $\mathtt{a}_\mathtt{i}$ sind, iterativ um eine Position nach rechts geschoben.
- Sei k minimal, so dass $j_k > 0$ und $a_{j_k} > a_i$.
- Am Ende des Schleifendurchlaufs gilt dann
  - $\mathtt{A}[0, \ldots, \mathtt{i}] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{k-1}}, \mathtt{a}_\mathtt{i}, \mathtt{a}_{\mathtt{j}_k}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$
  - und genau a_i wurde einsortiert, also $\{j_0, \ldots, j_i\} = \{0, \ldots, i\}$.

Analyse Insertion Sort

Induktionsschritt: Wir nehmen an, dass die Invariante gilt für ein $i \geq 1$. Also gilt zu Beginn der Iteration in der `while`-Schleife

- $\mathtt{A}[0, \ldots, \mathtt{i}-1] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$ für $0 \leq j_0, \ldots, j_{i-1} \leq i-1$ aufsteigend sortiert
- und es gilt $\{j_0, \ldots, j_{i-1}\} = \{0, \ldots, i-1\}$ (d.h. das Array enthält genau die i ersten Zahlen).
- In den Zeilen 3 bis 7 werden nun alle Einträge aus $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$, die größer als das einzusortierende Element $\mathtt{a}_\mathtt{i}$ sind, iterativ um eine Position nach rechts geschoben.
- Sei k minimal, so dass $j_k > 0$ und $a_{j_k} > a_i$.
- Am Ende des Schleifendurchlaufs gilt dann
  - $\mathtt{A}[0, \ldots, \mathtt{i}] = [\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{k-1}}, \mathtt{a}_\mathtt{i}, \mathtt{a}_{\mathtt{j}_k}, \ldots, \mathtt{a}_{\mathtt{j}_{i-1}}]$
  - und genau a_i wurde einsortiert, also $\{j_0, \ldots, j_i\} = \{0, \ldots, i\}$.

Analyse Insertion Sort

- Also gilt die Invariante auch zu Beginn der nächsten Iteration.

Analyse Insertion Sort

- Also gilt die Invariante auch zu Beginn der nächsten Iteration.
- Es folgt für $i = n$, dass $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{n-1}}]$ alle Zahlen des Eingabearrays enthält und aufsteigend sortiert ist.

Analyse Insertion Sort

- Also gilt die Invariante auch zu Beginn der nächsten Iteration.
- Es folgt für $i = n$, dass $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{n-1}}]$ alle Zahlen des Eingabearrays enthält und aufsteigend sortiert ist.
- Im dritten Schritt der Argumentation hätten wir auch formal beweisen müssen, dass $\mathtt{a}_\mathtt{i}$ an die richtige Stelle geschoben wird.

Analyse Insertion Sort

- Also gilt die Invariante auch zu Beginn der nächsten Iteration.
- Es folgt für $i = n$, dass $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{n-1}}]$ alle Zahlen des Eingabearrays enthält und aufsteigend sortiert ist.
- Im dritten Schritt der Argumentation hätten wir auch formal beweisen müssen, dass $\mathtt{a}_\mathtt{i}$ an die richtige Stelle geschoben wird.
  - Wieder per Induktion mit einer geeigneten Invariante.

Analyse Insertion Sort

- Also gilt die Invariante auch zu Beginn der nächsten Iteration.
- Es folgt für $i = n$, dass $[\mathtt{a}_{\mathtt{j}_0}, \ldots, \mathtt{a}_{\mathtt{j}_{n-1}}]$ alle Zahlen des Eingabearrays enthält und aufsteigend sortiert ist.
- Im dritten Schritt der Argumentation hätten wir auch formal beweisen müssen, dass $\mathtt{a}_\mathtt{i}$ an die richtige Stelle geschoben wird.
  - Wieder per Induktion mit einer geeigneten Invariante.

Laufzeitanalyse `Insertion Sort`

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1
2 while i < length(A) do
3     j := i
4     while j > 0 and A[j − 1] > A[j] do
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                                    1
2 while i < length(A) do
3     j := i
4     while j > 0 and A[j − 1] > A[j] do
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                                    1
2 while i < length(A) do                                    n
3     j := i
4     while j > 0 and A[j − 1] > A[j] do
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                                    1
2 while i < length(A) do                                    n
3     j := i                                                n − 1
4     while j > 0 and A[j − 1] > A[j] do
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                              1
2 while i < length(A) do                              n
3     j := i                                          n − 1
4     while j > 0 and A[j − 1] > A[j] do              ∑_{i=1}^{n−1} t_i
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                              1
2 while i < length(A) do                              n
3     j := i                                          n − 1
4     while j > 0 and A[j − 1] > A[j] do              ∑_{i=1}^{n−1} t_i
5         swap A[j] and A[j − 1]                      ∑_{i=1}^{n−1} t_i − 1
6         j := j − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                              1
2 while i < length(A) do                              n
3     j := i                                          n − 1
4     while j > 0 and A[j − 1] > A[j] do              ∑_{i=1}^{n−1} t_i
5         swap A[j] and A[j − 1]                      ∑_{i=1}^{n−1} t_i − 1
6         j := j − 1                                  ∑_{i=1}^{n−1} t_i − 1
7     i := i + 1
```

Laufzeitanalyse Insertion Sort

- Zählen zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Hängt von der Zahl n ab!
- Für $i \in \{1, \ldots, n-1\}$ bezeichne mit t_i, wie oft Zeile 4 in Iteration i der **while**-Schleife ausgeführt wird.
  (Die Abfrage in einer **while**-Schleife wird einmal mehr ausgeführt als die Schleife selbst)

```
1 i := 1                                              1
2 while i < length(A) do                              n
3     j := i                                          n − 1
4     while j > 0 and A[j − 1] > A[j] do              ∑_{i=1}^{n−1} t_i
5         swap A[j] and A[j − 1]                      ∑_{i=1}^{n−1} t_i − 1
6         j := j − 1                                  ∑_{i=1}^{n−1} t_i − 1
7     i := i + 1                                      n − 1
```

Laufzeitanalyse `Insertion Sort`

- Bezeichne mit c_i die Zahl der Prozessorbefehle, die benötigt wird um Zeile i einmal auszuführen.

Laufzeitanalyse `Insertion Sort`

- Bezeichne mit c_i die Zahl der Prozessorbefehle, die benötigt wird um Zeile i einmal auszuführen.
- Wir betrachten nur kleine Zahlen, darum nehmen wir jedes c_i als konstant an.

Laufzeitanalyse `Insertion Sort`

- Bezeichne mit c_i die Zahl der Prozessorbefehle, die benötigt wird um Zeile i einmal auszuführen.
- Wir betrachten nur kleine Zahlen, darum nehmen wir jedes c_i als konstant an.
- Gesamtlaufzeit beschränkt durch:

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6) \sum_{i=1}^{n-1} t_i.$$

Laufzeitanalyse Insertion Sort

- Bezeichne mit c_i die Zahl der Prozessorbefehle, die benötigt wird um Zeile i einmal auszuführen.
- Wir betrachten nur kleine Zahlen, darum nehmen wir jedes c_i als konstant an.
- Gesamtlaufzeit beschränkt durch:

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6) \sum_{i=1}^{n-1} t_i.$$

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)(n-1).$$

Laufzeitanalyse Insertion Sort

- Bezeichne mit c_i die Zahl der Prozessorbefehle, die benötigt wird um Zeile i einmal auszuführen.
- Wir betrachten nur kleine Zahlen, darum nehmen wir jedes c_i als konstant an.
- Gesamtlaufzeit beschränkt durch:

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\sum_{i=1}^{n-1} t_i.$$

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)(n-1).$$

- Worst Case: $t_i = i$ (a_i wird mit allen Vorgängern getauscht).

$$c_1 + (c_2 + c_3 + c_7)n + (c_4 + c_5 + c_6)\frac{n(n-1)}{2}.$$

Laufzeitanalyse `Insertion Sort`

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).

Laufzeitanalyse Insertion Sort

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).
  Für geeignete Konstanten $a, b \in \mathbb{N}$:

$$an + b \qquad \text{linear in } n.$$

Laufzeitanalyse `Insertion Sort`

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).
  Für geeignete Konstanten $a, b \in \mathbb{N}$:

$$an + b \qquad \text{linear in } n.$$

- Worst Case: $t_i = i$ (a_i wird mit allen Vorgängern getauscht).

Laufzeitanalyse Insertion Sort

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).
  Für geeignete Konstanten $a, b \in \mathbb{N}$:

$$an + b \qquad \text{linear in } n.$$

- Worst Case: $t_i = i$ (a_i wird mit allen Vorgängern getauscht).
  Für geeignete Konstanten $a, b, c \in \mathbb{N}$:

$$an^2 + bn + c \qquad \text{quadratisch in } n.$$

Laufzeitanalyse `Insertion Sort`

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).
  Für geeignete Konstanten $a, b \in \mathbb{N}$:

$$an + b \qquad \text{linear in } n.$$

- Worst Case: $t_i = i$ (a_i wird mit allen Vorgängern getauscht).
  Für geeignete Konstanten $a, b, c \in \mathbb{N}$:

$$an^2 + bn + c \qquad \text{quadratisch in } n.$$

- Die genauen Werte der Konstanten hängen vom Prozessor, vom Compiler, usw. ab. Dies können (und wollen) wir in der theoretischen Analyse nicht berücksichtigen.

Laufzeitanalyse Insertion Sort

- Best Case: $t_i = 1$ (a_i steht bereits an der richtigen Position).
  Für geeignete Konstanten $a, b \in \mathbb{N}$:

$$an + b \qquad \text{linear in } n.$$

- Worst Case: $t_i = i$ (a_i wird mit allen Vorgängern getauscht).
  Für geeignete Konstanten $a, b, c \in \mathbb{N}$:

$$an^2 + bn + c \qquad \text{quadratisch in } n.$$

- Die genauen Werte der Konstanten hängen vom Prozessor, vom Compiler, usw. ab. Dies können (und wollen) wir in der theoretischen Analyse nicht berücksichtigen.
- Uns interessiert nur die Größenordnung des Wachstums!

$\rightarrow$ **Formalisierung in nächster Vorlesung!**

Zusammenfassung

- Algorithmenbegriff
- Wichtig:
  - Problemspezifikation
  - Algorithmenbeschreibung (Pseudocode)
  - Korrektheitsbeweis
  - Laufzeit