# Kapitel ADS:I

## I. Einführung

- Begriffserklärung
- Beispiele für Probleme und algorithmische Lösungen
- Algorithm Engineering


© POTTHAST 2018

# Begriffserklärung

**Definition 1 (Problem, Probleminstanz)**

Ein Problem (aus dem Griechischen für „das, was [zur Lösung] vorgelegt wurde“), nennt man eine Aufgabe, deren Lösung mit Schwierigkeiten verbunden ist.

In der Informatik wird ein Problem, das computergestützt gelöst werden soll, formal definiert als Probleminstanz, für die eine bestimmte Lösung zu berechnen ist.

Wir betrachten das Problem als Menge aller erlaubten Instanzen gemäß seiner formalen Definition.


© POTTHAST 2018

# Begriffserklärung

**Definition 1 (Problem, Probleminstanz)**

Ein Problem (aus dem Griechischen für „das, was [zur Lösung] vorgelegt wurde“), nennt man eine Aufgabe, deren Lösung mit Schwierigkeiten verbunden ist.

In der Informatik wird ein Problem, das computergestützt gelöst werden soll, formal definiert als Probleminstanz, für die eine bestimmte Lösung zu berechnen ist.

Wir betrachten das Problem als Menge aller erlaubten Instanzen gemäß seiner formalen Definition.


© POTTHAST 2018

# Begriffserklärung

## Definition 1 (Problem, Probleminstanz)

Ein Problem (aus dem Griechischen für „das, was [zur Lösung] vorgelegt wurde“), nennt man eine Aufgabe, deren Lösung mit Schwierigkeiten verbunden ist.

In der Informatik wird ein Problem, das computergestützt gelöst werden soll, formal definiert als Probleminstanz, für die eine bestimmte Lösung zu berechnen ist.

Wir betrachten das Problem als Menge aller erlaubten Instanzen gemäß seiner formalen Definition.

**Beispiel:**

Problem: Sortieren

Instanz: $A$. Folge von $n$ Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von $A$, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Verfahren das für jede Instanz $A$ eine Lösung $A'$ berechnet.
$(A = (31, 41, 59, 26, 41, 58) \;\Rightarrow\; A' = (26, 31, 41, 41, 58, 59))$


© POTTHAST 2018

Bemerkungen:

- Die Schwierigkeit, ein Problem computergestützt zu lösen, besteht insbesondere darin, dass die Menge aller erlaubten Instanzen unendlich groß ist. Aus theoretischer Sicht lohnt sich nur unter dieser Voraussetzung die Entwicklung eines Algorithmus zur Lösung des Problems. Andernfalls genügt es, die Lösungen aller Probleminstanzen aufzuzählen und auf Anfrage nachzuschlagen und auszugeben.


© POTTHAST 2018

# Begriffserklärung

**Definition 2 (Algorithmus)**

Ein Algorithmus ist eine Folge von Anweisungen zur Lösung eines Problems. Für eine Probleminstanz als Eingabe, berechnet er deren Lösung als Ausgabe.

Statische Eigenschaften:

- Definiertheit
  Jede Anweisung ist präzise und eindeutig spezifiziert.
- Effektivität
  Jede Anweisung ist “leicht” ausführbar oder ein Algorithmus.
- Statische Finitheit
  Die Folge von Anweisungen ist endlich.
- Dynamische Finitheit
  Der Platzbedarf jeder Anweisung ist endlich.


© POTTHAST 2018

# Begriffserklärung

**Definition 2 (Algorithmus)**

Ein Algorithmus ist eine Folge von Anweisungen zur Lösung eines Problems. Für eine Probleminstanz als Eingabe, berechnet er deren Lösung als Ausgabe.

Dynamische Eigenschaften: (optional)

- Terminiertheit
  Die Ausgabe wird nach Ausführung endlich vieler Anweisungen ausgegeben.
- Determiniertheit
  Für eine bestimmte Eingabe wird immer dieselbe Ausgabe ausgegeben.
- Determinismus
  Für eine bestimmte Eingabe wird immer dieselbe Anweisungsfolge befolgt.


© POTTHAST 2018

# Begriffserklärung

**Definition 2 (Algorithmus)**

Ein Algorithmus ist eine Folge von Anweisungen zur Lösung eines Problems. Für eine Probleminstanz als Eingabe, berechnet er deren Lösung als Ausgabe.

Qualitative Eigenschaften: (optional)

- Korrektheit
  Für jede Probleminstanz als Eingabe löst die Ausgabe das Problem.
- Optimalität
  Für jede Probleminstanz als Eingabe löst die Ausgabe das Problem optimal.


© POTTHAST 2018

# Begriffserklärung

**Definition 2 (Algorithmus)**

Ein Algorithmus ist eine Folge von Anweisungen zur Lösung eines Problems. Für eine Probleminstanz als Eingabe, berechnet er deren Lösung als Ausgabe.

Quantitative Eigenschaften:

- Komplexität (Zeit)
  Obere Schranke für die benötigte Zeit zur Berechnung der Ausgabe.
- Komplexität (Platz)
  Obere Schranke für den benötigten Platz zur Berechnung der Ausgabe.
- Komplexität (Problem)
  Theoretische untere/obere Schranken für Zeit/Platz zur Problemlösung.


© POTTHAST 2018

Bemerkungen:

- Das Wort “Algorithmus” stammt aus dem 9. Jahrhundert und leitet sich vom Namen des choresmischen Mathematiker *Al-Chwarizmi* ab, der auf der Arbeit des aus dem 7. Jahrhundert stammenden indischen Mathematikers Brahmagupta aufbaute. In seiner ursprünglichen Bedeutung bezeichnete ein Algorithmus nur das Einhalten der arithmetischen Regeln unter Verwendung der indisch-arabischen Ziffern. Die ursprüngliche Definition entwickelte sich mit Übersetzung ins Lateinische weiter. [Wikipedia]
- Bis heute gibt es keine allgemein akzeptierte Definition des Konzeptes „Algorithmus“; zahlreiche Autoren haben versucht, es zu umschreiben; das obige ist der Versuch eines Konsenses.
- Umgangssprachlich kann man „Algorithmus“ in etwa mit „Rezept“ gleichsetzen.
- Eine Anweisung ist zum Beispiel dann leicht ausführbar, wenn man sie per Hand mit Stift und Papier in vernünftiger Zeit erledigen kann. [Knuth 1968-heute]
- Eingabe und Ausgabe sind Daten, kodiert als Folge von Bits.
- Die Berechnung der Ausgabe für eine Eingabe bezeichnen wir als Datenverarbeitung.
- Auch inkorrekte Algorithmen können nützlich sein, wenn ihre Fehlerrate kontrollierbar ist.


© POTTHAST 2018

# Begriffserklärung

## Definition 3 (Datenstruktur)

Eine Datenstruktur organisiert Daten gemäß einer Spezifikation im Speicher eines Computers, so dass sie der effektiven Verarbeitung durch einen Algorithmus zugänglich werden. Hierfür werden Algorithmen zur Manipulation der enthaltenen Daten bereitgestellt.

Datenstrukturen sind

- dynamisch
  Enthaltene Daten können manipuliert, also gelesen, hinzugefügt, entfernt, oder verändert werden.
- konsistent
  Die Algorithmen zur Manipulation enthaltener Daten sind korrekt: sie stellen die spezifizierte Organisationsstruktur der Daten sicher.
- zweckorientiert
  Sie dienen der Realisierung von Anweisungen im Computer im Sinne ihrer leichten Ausführbarkeit.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Problem: Sortieren

Instanz: $A$. Folge von $n$ Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von $A$, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Hintergrund:

- Die Problemdefinition ist reduziert auf das Wesentliche: Zahlen sortieren.
- In der Praxis wollen wir aber kompliziertere Datentypen verarbeiten.
- Der erste Schritt dafür ist, eine interessante Eigenschaft der Daten zu betrachten: z.B. Größe, Wert, Wichtigkeit, Erstelldatum, etc.
- Diese Eigenschaft wird quantifiziert und die Daten danach sortiert.
- Die interessierende Eigenschaft wird als Sortierschlüssel bezeichnet.
- Zu analysierende Daten werden mit Sortierschlüsseln verknüpft gespeichert.
- Der Einfachheit halber blenden wir die verknüpften Daten aus.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

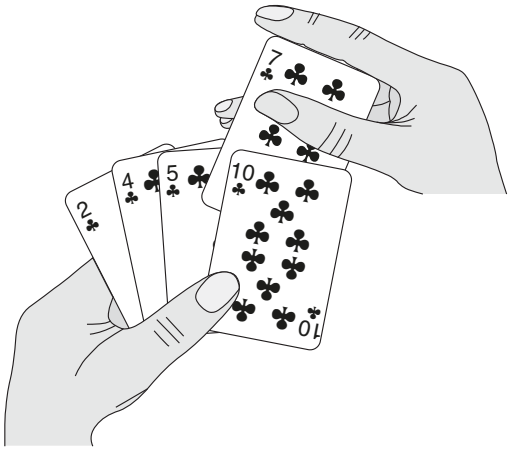
Wunsch: Ein Algorithmus, der für jede Instanz $A$ eine Lösung $A'$ berechnet.

Idee: Sortiere analog zum Aufnehmen und Einsortieren von Handkarten beim Karten spielen.

Anweisungen zum Karten sortieren:

- ❑ Eine Karte vom Stapel aufnehmen.
- ❑ Kartenwert merken.
- ❑ Angefangen bei der Karte rechts auf der Hand:
- ❑ Prüfe, ob sie einen höheren Wert hat.
- ❑ Wenn ja, schiebe sie nach rechts auf der Hand.
- ❑ Dann gehe zur links benachbarten Karte.
- ❑ Fahre fort, solange höherwertige Karten da sind.
- ❑ Füge die Karte an der nun freien Stelle ein.
- ❑ Fahre fort, solange Karten auf dem Stapel sind.

➜ Die Handkarten sind sortiert.





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j - 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i - 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| *A* | 5 | 2 | 4 | 6 | 1 | 3 |



© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

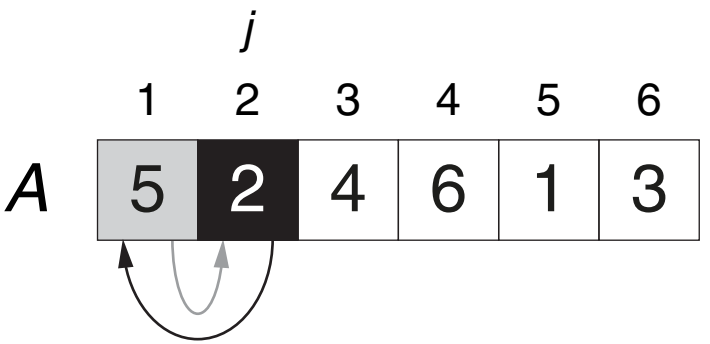
*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j - 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i - 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

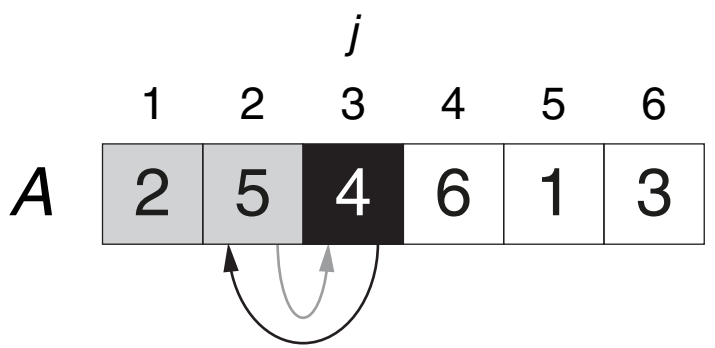
*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j − 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i − 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

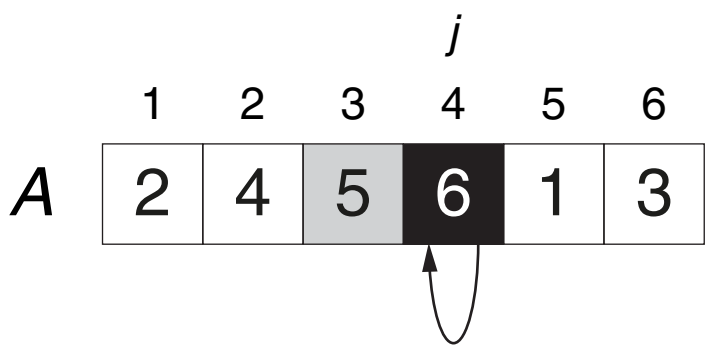
*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j - 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i - 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

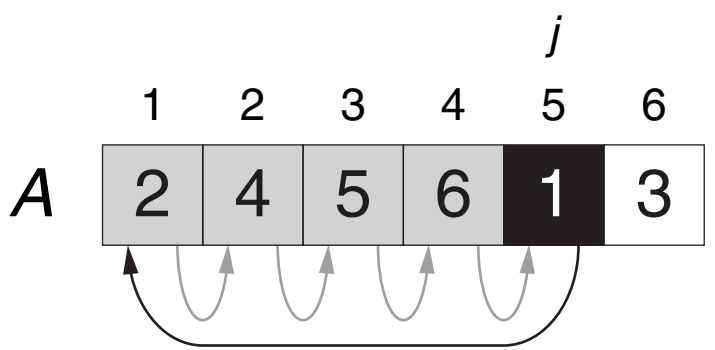
*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j - 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i - 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:






© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

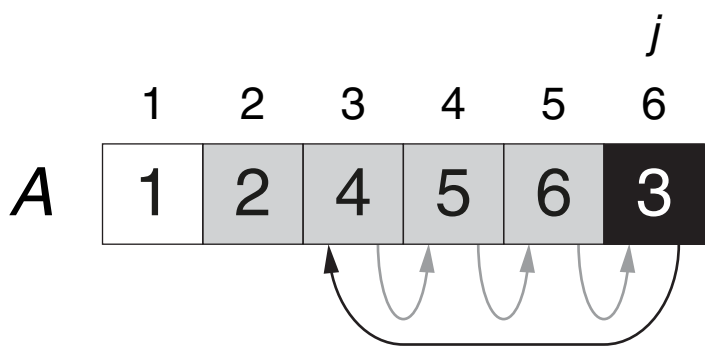
*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j − 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i − 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Sortierproblem

Algorithmus: Insertion Sort.

Eingabe: $A$. Array von $n$ Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von $A$.

*InsertionSort*$(A)$

```
1.  FOR j = 2 TO n DO
2.     a_j = A[j]
3.     i = j - 1
4.     WHILE i > 0 AND A[i] > a_j DO
5.        A[i + 1] = A[i]
6.        i = i - 1
7.     ENDDO
8.     A[i + 1] = a_j
9.  ENDDO
```

Datenstruktur Array:

- $n$ gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes $A[i]$

Beispiel:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| *A* | 1 | 2 | 3 | 4 | 5 | 6 |



© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

Suchproblem

| | |
|---|---|
| Problem: | Suchen |
| Instanz: | $A$. Folge von $n$ Zahlen $A = (a_1, a_2, \ldots, a_n)$.<br>$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt. |
| Lösung: | Der Index $i$ für ein $a_i$ in $A$, so dass $a_i = v$, sonst *NIL*. |


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

Suchproblem

| | |
|---|---|
| Problem: | Suchen |
| Instanz: | $A$. Folge von $n$ Zahlen $A = (a_1, a_2, \ldots, a_n)$.<br>$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt. |
| Lösung: | Der Index $i$ für ein $a_i$ in $A$, so dass $a_i = v$, sonst *NIL*. |
| Wunsch: | Ein Algorithmus, um $v$ in $A$ zu lokalisieren, falls vorhanden. |
| Idee: | Eine Zahl aus $A$ nach der anderen betrachten und vergleichen. |


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

| | |
|---|---|
| Problem: | Suchen |
| Instanz: | $A$. Folge von $n$ Zahlen $A = (a_1, a_2, \ldots, a_n)$. |
| | $v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt. |
| Lösung: | Der Index $i$ für ein $a_i$ in $A$, so dass $a_i = v$, sonst *NIL*. |
| Wunsch: | Ein Algorithmus, um $v$ in $A$ zu lokalisieren, falls vorhanden. |
| Idee: | Eine Zahl aus $A$ nach der anderen betrachten und vergleichen. |
| Algorithmus: | Sequential Search |
| Eingabe: | $A$. Array von $n$ Zahlen. |
| | $v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt. |
| Ausgabe: | Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$. |

*SequentialSearch*$(A, v)$

```
1.  FOR i = 1 TO n DO
      IF v == A[i] THEN return(i)
    ENDDO
2.  return(NIL)
```

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| *A* | 5 | 2 | 4 | 6 | 1 | 3 |


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Was, wenn $A$ sortiert wäre?

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 5 | 2 | 4 | 6 | 1 | 3 |

vs.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 1 | 2 | 3 | 4 | 5 | 6 |


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Was, wenn $A$ sortiert wäre?

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 5 | 2 | 4 | 6 | 1 | 3 |

vs.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 1 | 2 | 3 | 4 | 5 | 6 |

Wunsch: Ein Algorithmus, um $v$ in $A$ zu lokalisieren, falls vorhanden, wenn $A$ sortiert ist.

Idee: Mittig beginnen, dann mittig in der Hälfte weitersuchen, die $v$ enthalten müsste, usw.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Was, wenn $A$ sortiert wäre?

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 5 | 2 | 4 | 6 | 1 | 3 |

vs.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | 1 | 2 | 3 | 4 | 5 | 6 |

Wunsch: Ein Algorithmus, um $v$ in $A$ zu lokalisieren, falls vorhanden, wenn $A$ sortiert ist.

Idee: Mittig beginnen, dann mittig in der Hälfte weitersuchen, die $v$ enthalten müsste, usw.

Alltagsbeispiele: (größtenteils aus der entfernten Vergangenheit)

- Sammlungen von Büchern, Musik, Filmen, Spielen (inkl. TCGs), Dokumenten
- Suche in Bibliothekskatalogen, Lexika, Enzyklopädien, Telefonbüchern


© POTTHAST 2018

Bemerkungen:

- Das meiste aus den Alltagsbeispielen wurde zwischenzeitlich ersetzt durch Dateien auf Festplatten oder in der Cloud sowie Webdienste. Interessanterweise schert sich kaum jemand um eine akkurate Organisation seines digitalen Eigentums. Vielleicht, weil Hausgäste das Durcheinander nie zu sehen bekommen? Wer weiß... [xkcd 1077] [xkcd 1360]
- Bei Suchmaschinen entspricht die Einschränkung der Suche bezüglich verschiedener sogenannter Suchfacetten wie Preis, Größe, Gewicht, etc. einer einschränkenden Vorauswahl, die entfernt verwandt zur binären Suche ist.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

1. Initialisiere Variablen *low* und *high*, die das Intervall $A[\mathit{low}, \mathit{high}]$ des Arrays $A$ bezeichnen, das gerade betrachtet wird.
2. Solange $A[\mathit{low}, \mathit{high}]$ ein gültiges Intervall ist:
3. Berechne den ganzzahligen Index *mid* des mittleren Eintrags.
4. Wenn $A[\mathit{mid}]$ die gesuchte Zahl $v$ enthält, gebe *mid* aus.
5. Wenn $v$ größer ist, schränke das Intervall $A[\mathit{low}, \mathit{high}]$ so ein, dass die Hälfte des Intervalls ab *mid* (exklusive) übrig bleibt, ansonsten die andere Hälfte.
6. Fahre fort, bis das Intervall ungültig wird ($\mathit{low} > \mathit{high}$).
7. Wenn dieser Fall eintritt, ist $v$ nicht in $A$; gebe *NIL* aus.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *A* | 1 | 3 | 5 | 6 | 8 | 10 | 15 | 16 | 19 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

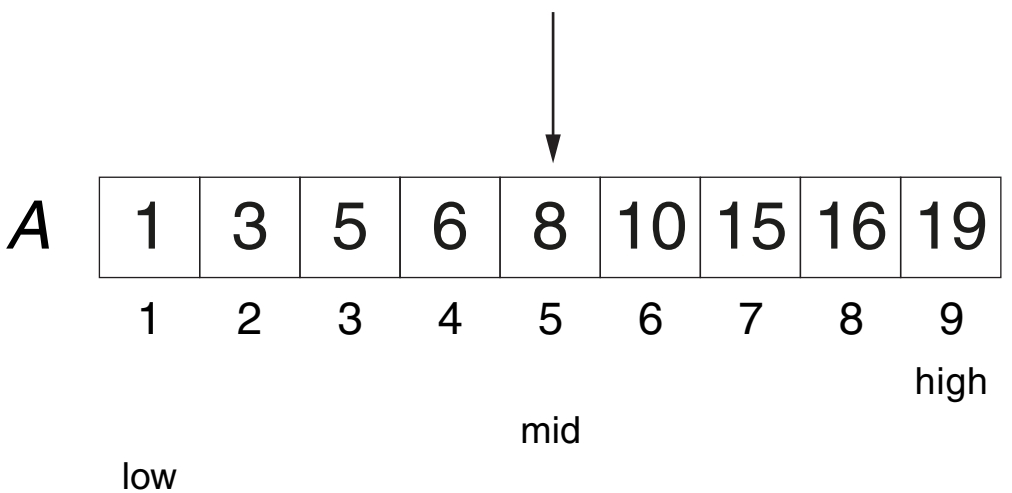
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

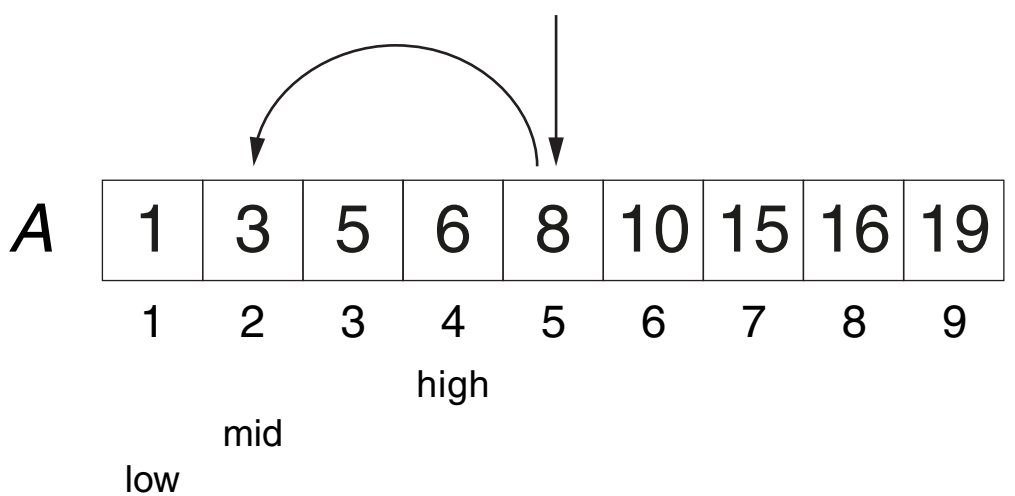
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

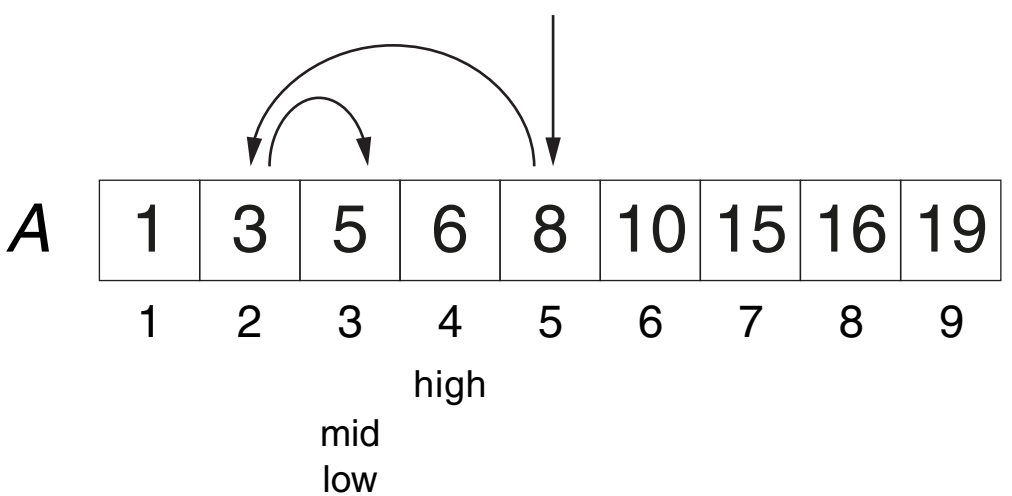
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

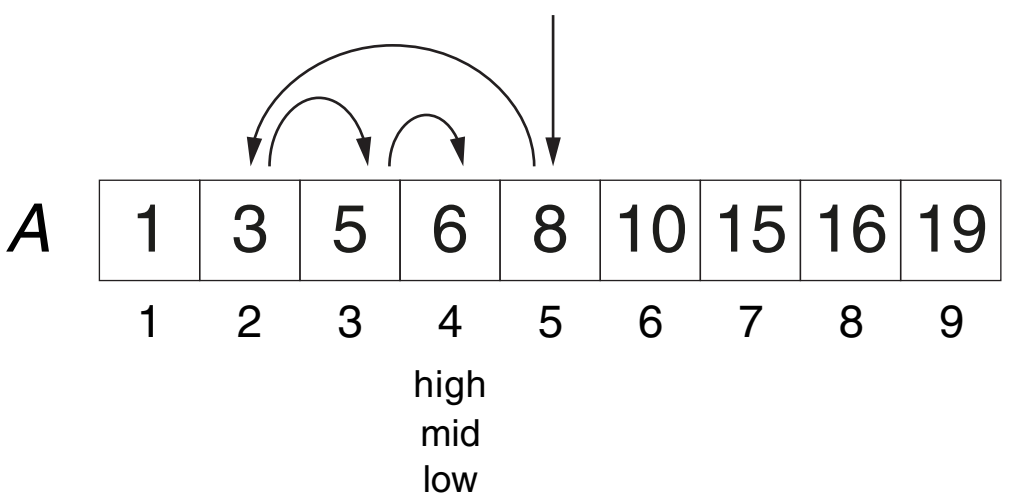
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

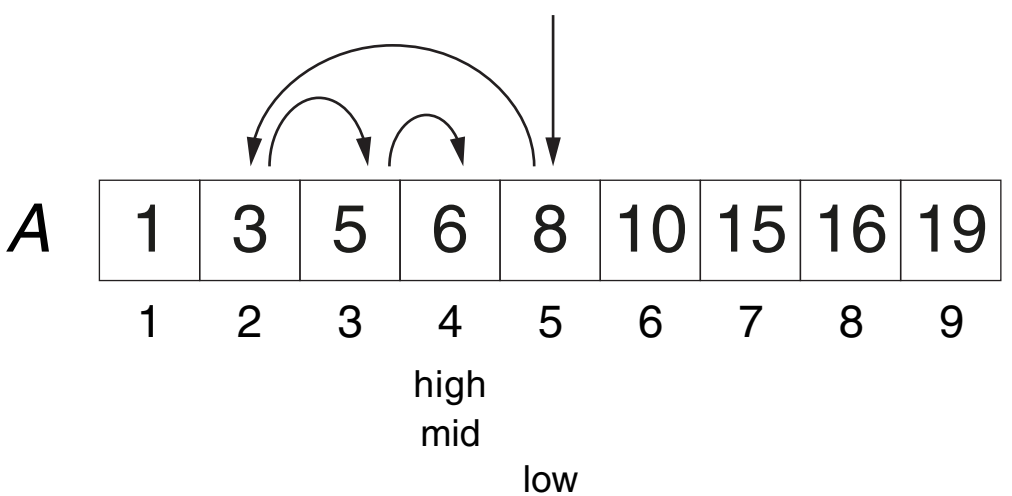
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel $v = 7$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

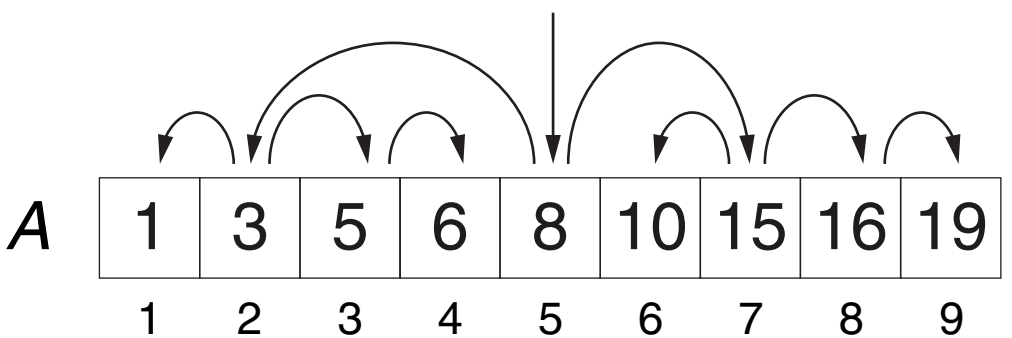
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Beispiel für allgemeines $v$:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

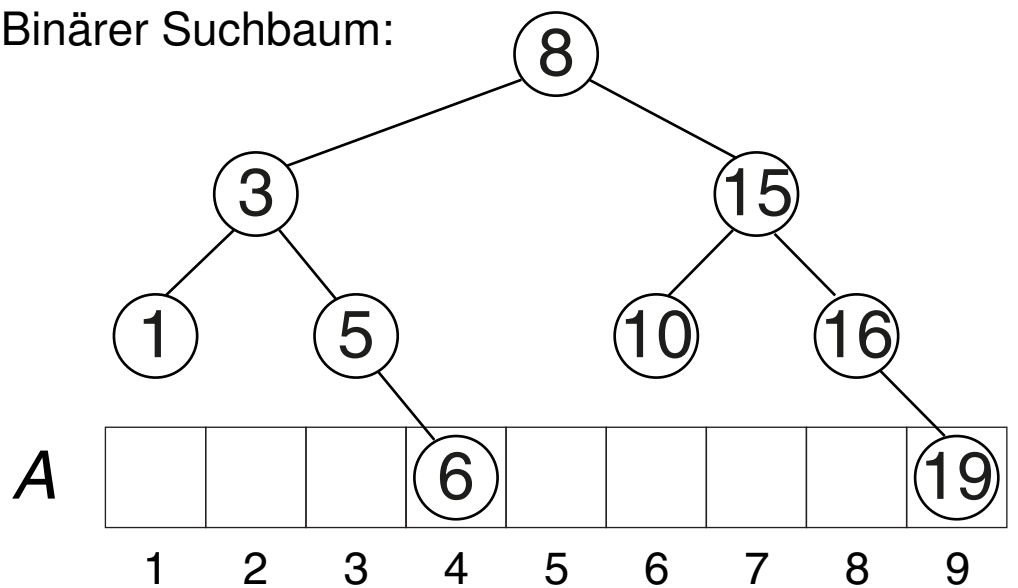
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

1. $low = 1$; $high = n$
2. **WHILE** $low \leq high$ **DO**
3. $mid = \lfloor (low + high)/2 \rfloor$
4. **IF** $v == A[mid]$
   **THEN** $return(mid)$
5. **IF** $v > A[mid]$
   **THEN** $low = mid + 1$
   **ELSE** $high = mid - 1$
6. **ENDDO**
7. $return(NIL)$

Binärer Suchbaum:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Suchproblem

Algorithmus: Binary Search.

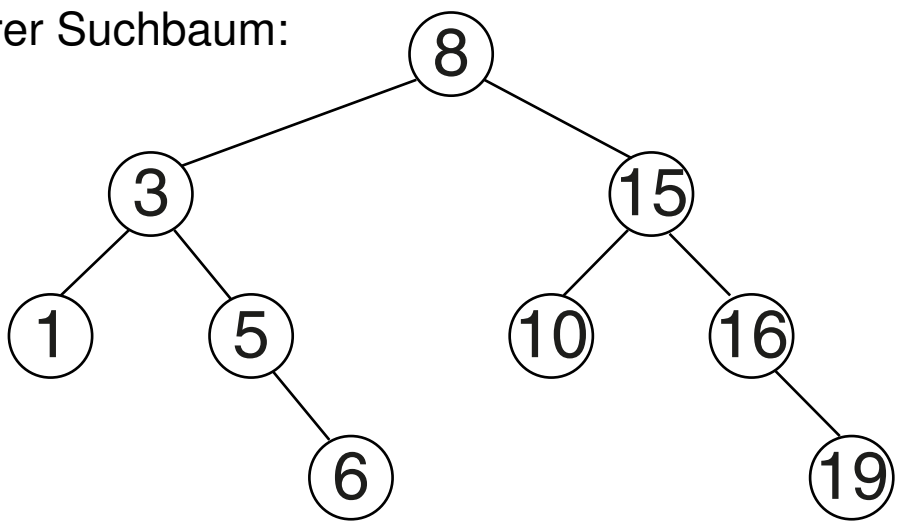
Eingabe: $A$. Array von $n$ sortierten Zahlen.
$v$. Zahl, für die geklärt werden soll, ob und wo in $A$ sie vorkommt.

Ausgabe: Die Position $i$ von $v$ in $A$ oder *NIL*, falls $v \notin A$.

*BinarySearch*$(A, v)$

```
1.  low = 1;  high = n
2.  WHILE low ≤ high DO
3.     mid = ⌊(low + high)/2⌋
4.     IF v == A[mid]
       THEN return(mid)
5.     IF v > A[mid]
       THEN low = mid + 1
       ELSE high = mid − 1
6.  ENDDO
7.  return(NIL)
```

Binärer Suchbaum:





© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:

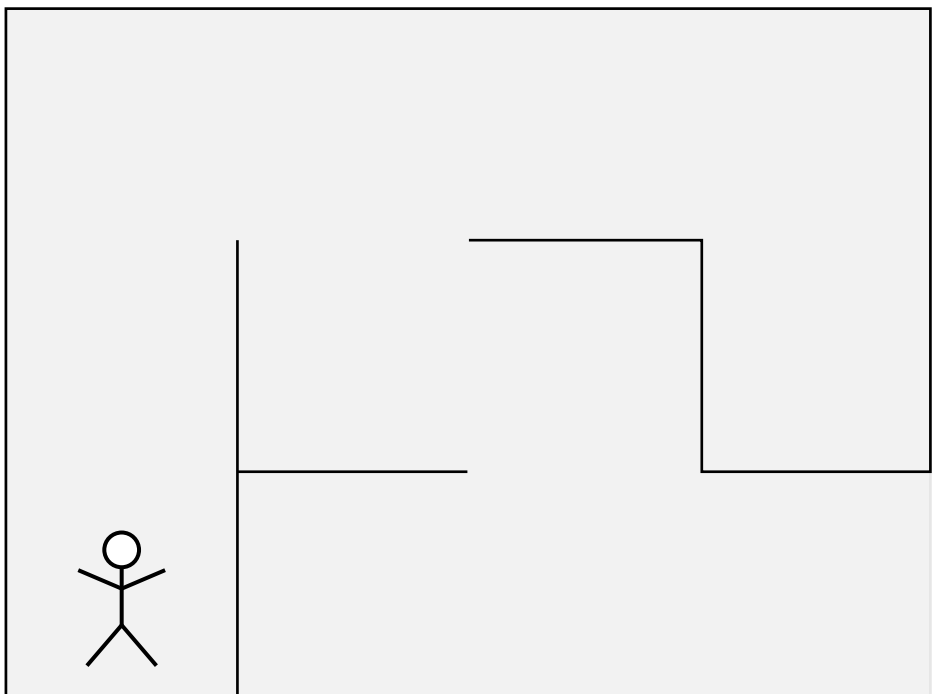
- Es ist dunkel.
- Bewegung ist in alle Himmelsrichtungen möglich.
- Wände begrenzen die Bewegungsfreiheit.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Wunsch: Ein Algorithmus zum Verlassen des Labyrinths.

Idee: An der Wand entlang tasten.

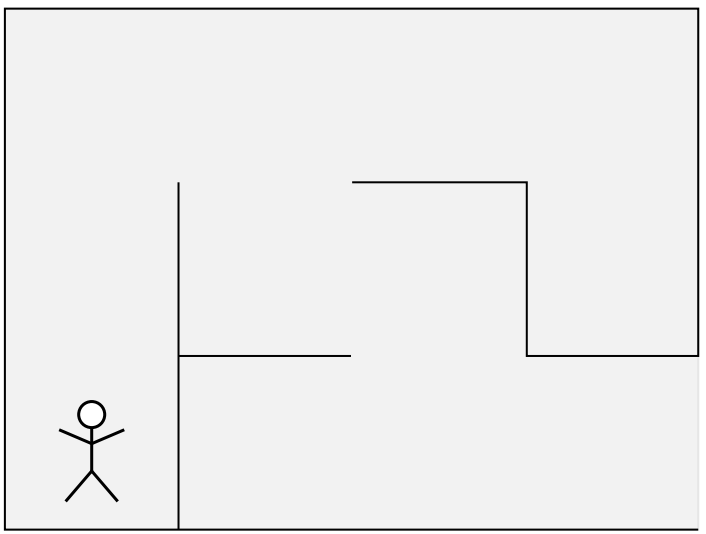


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

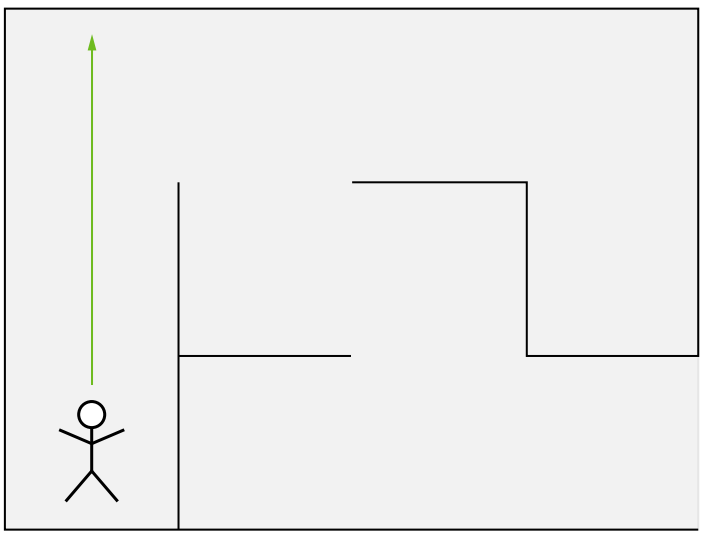
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

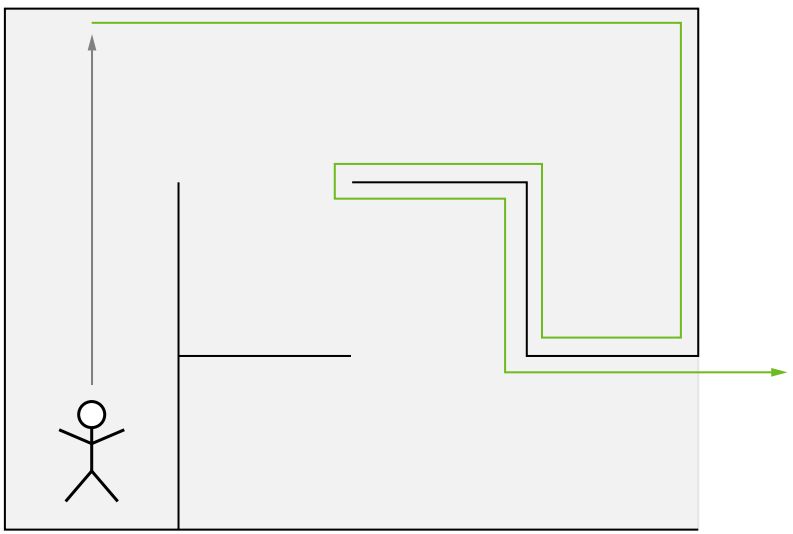
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

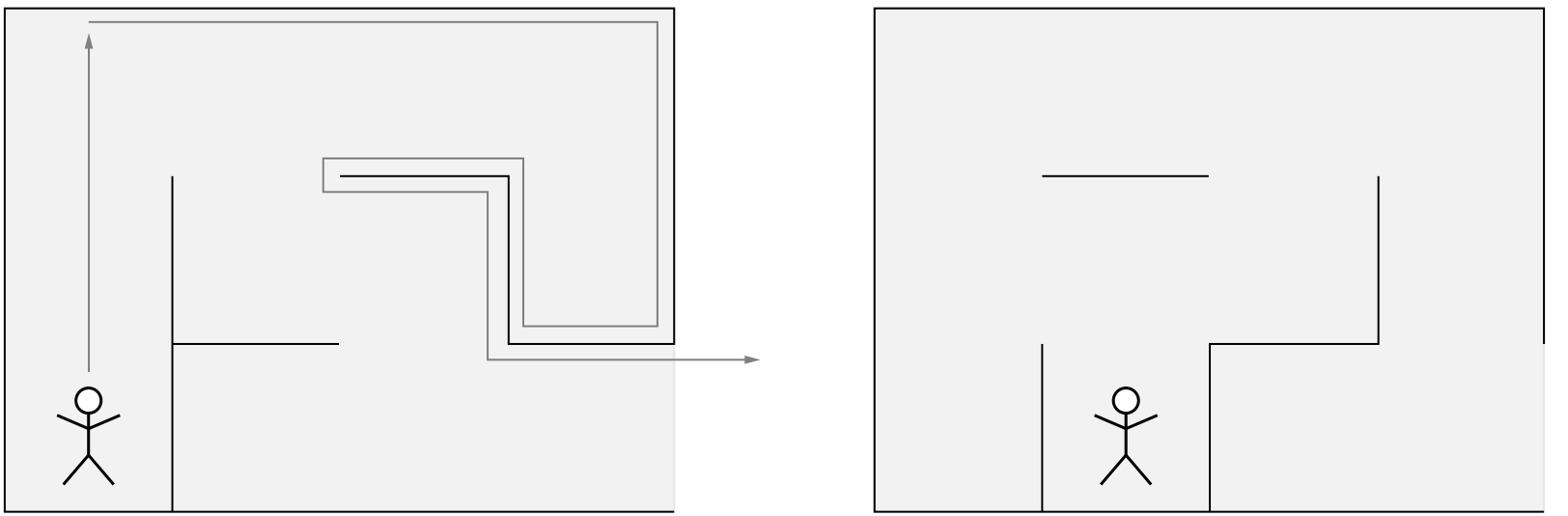
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

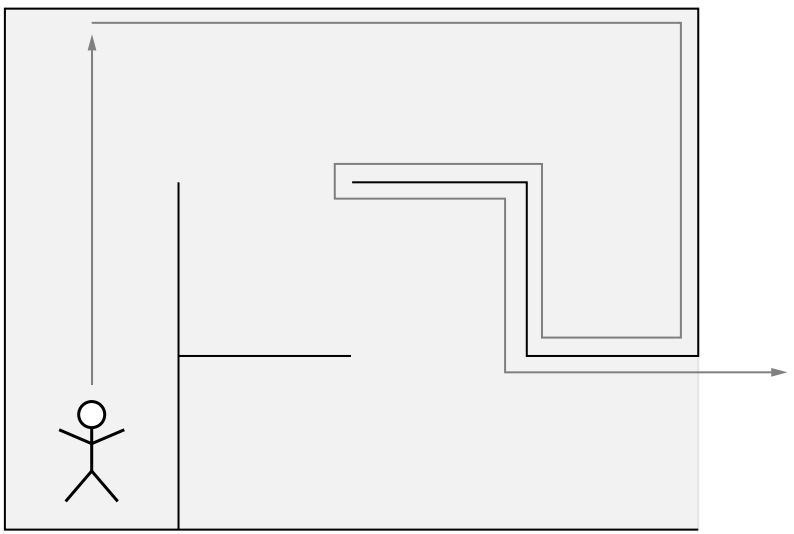
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.

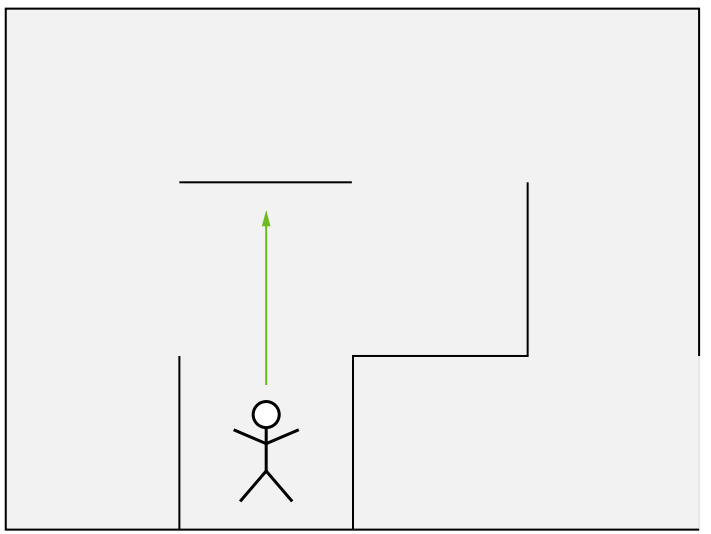

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

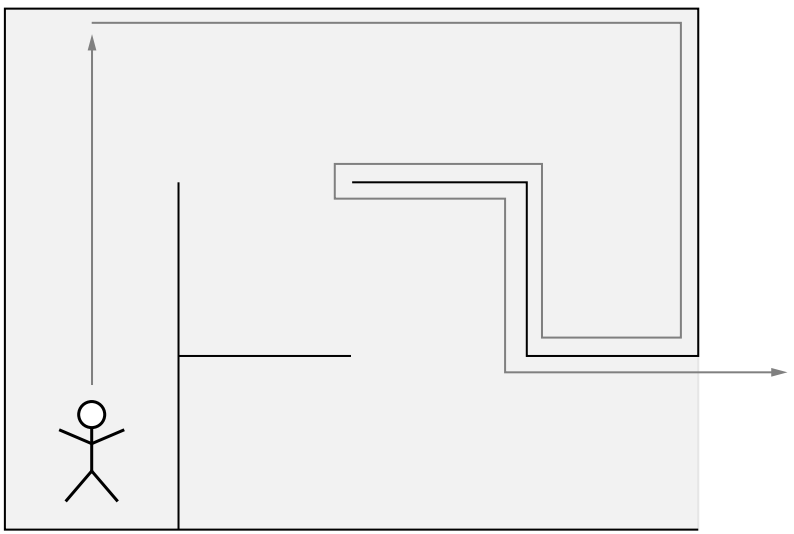
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.

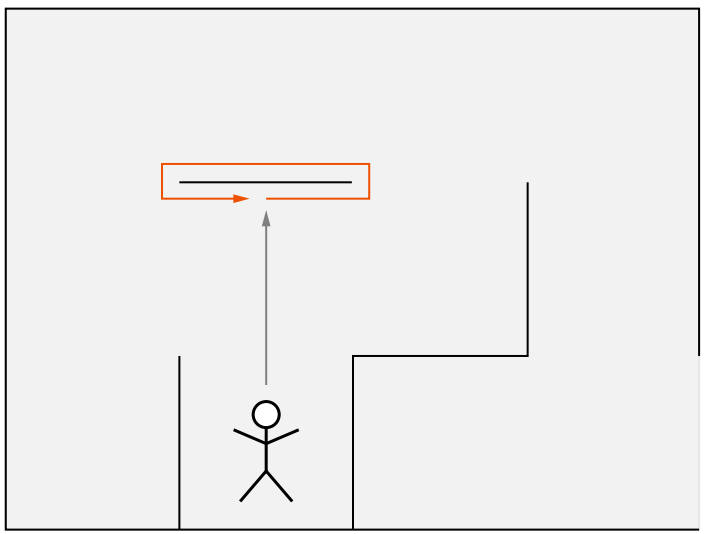

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*WallFollower*$(G, s)$

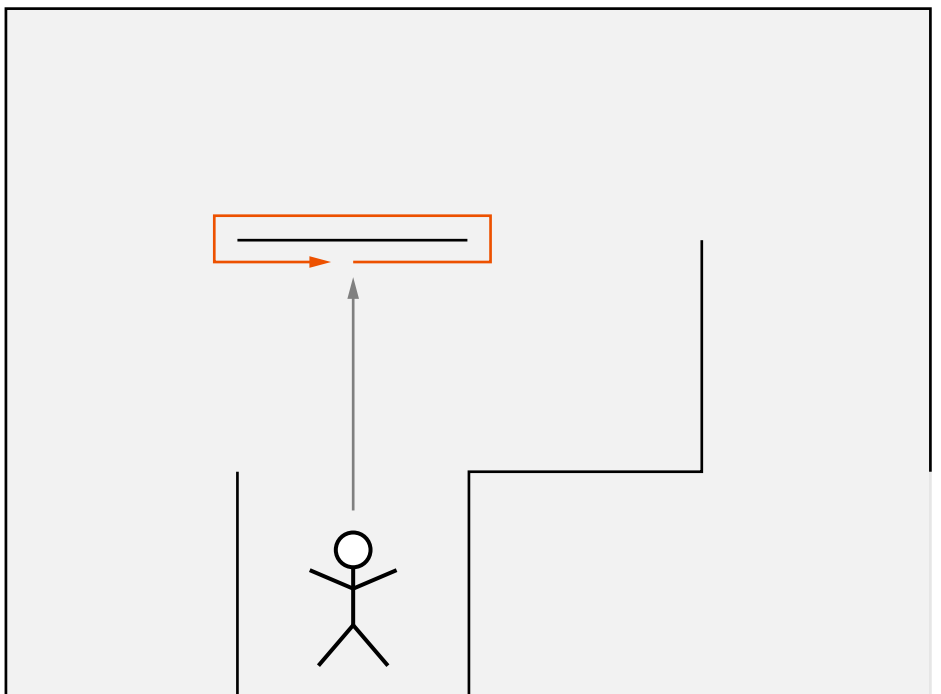
1. Gehe von $s$ in eine zufällige Richtung bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand bis ein Ausgang von $G$ erreicht ist.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Wunsch: Ein Algorithmus zum Verlassen des Labyrinths.

Idee: Wenn möglich, nach Norden gehen, sonst an der Wand entlang tasten.

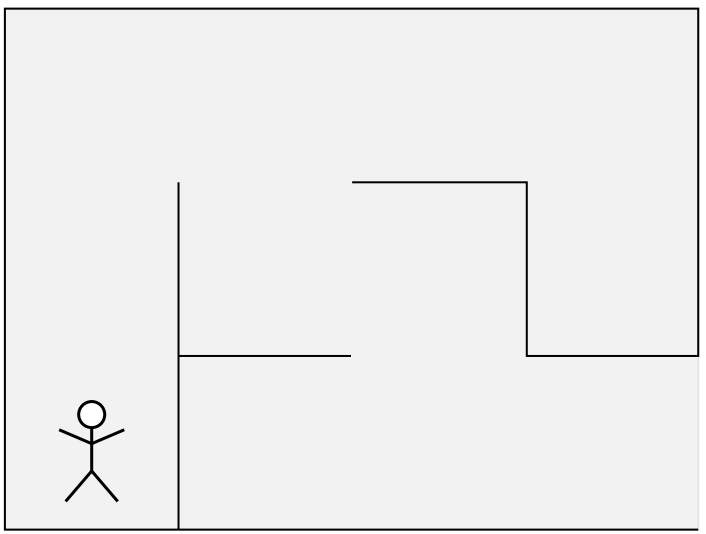


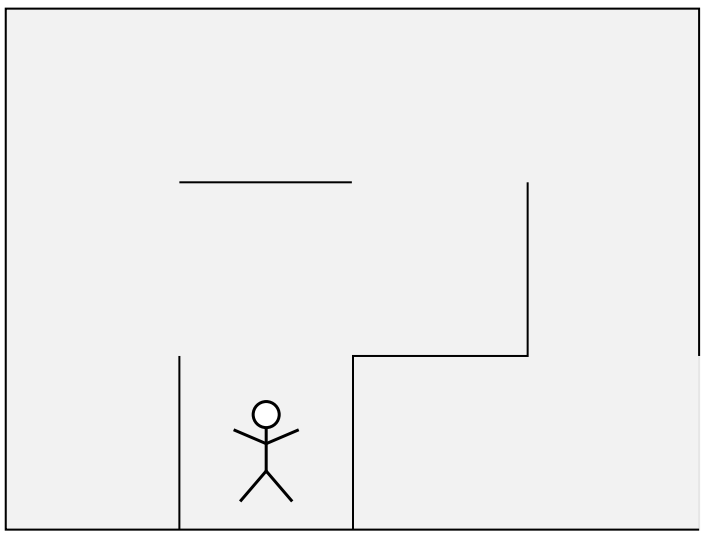
© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Oriented Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*OrientedWallFollower*$(G, s)$

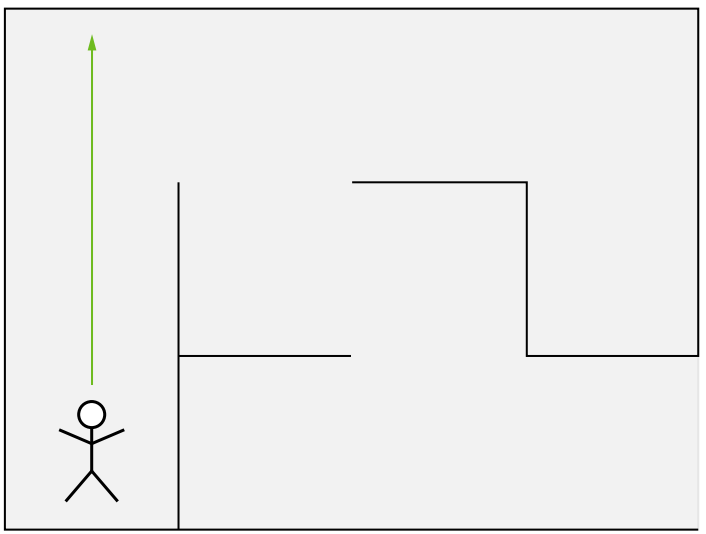
1. Gehe von $s$ nach Norden bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand zu einem Abzweig nach Norden. Beginne von vorn.

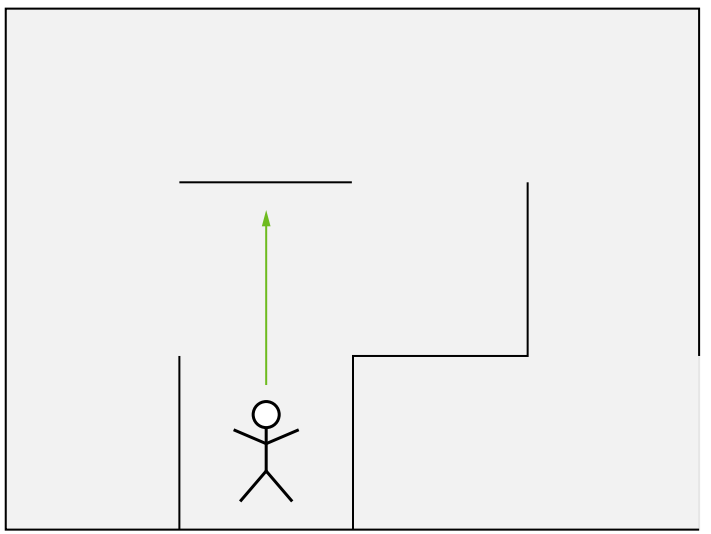

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Oriented Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*OrientedWallFollower*$(G, s)$

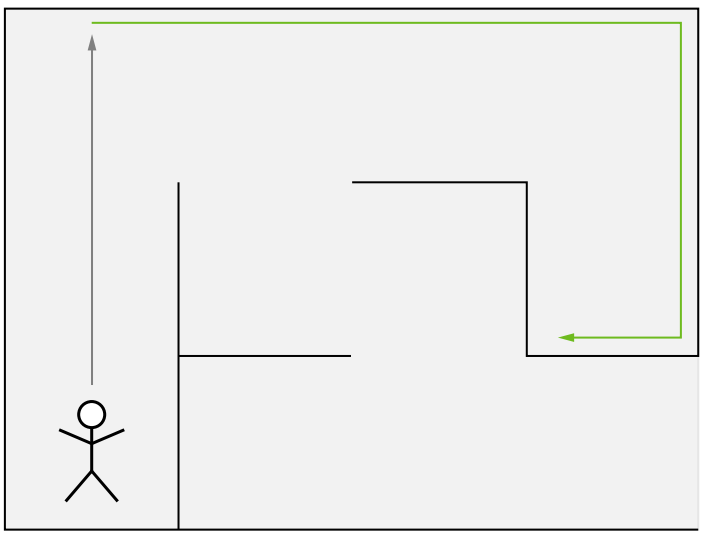
1. Gehe von $s$ nach Norden bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand zu einem Abzweig nach Norden. Beginne von vorn.

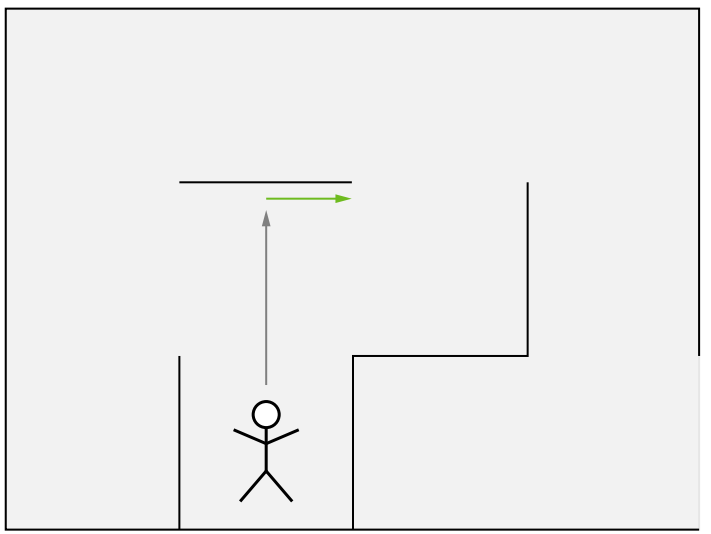

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Oriented Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*OrientedWallFollower*$(G, s)$

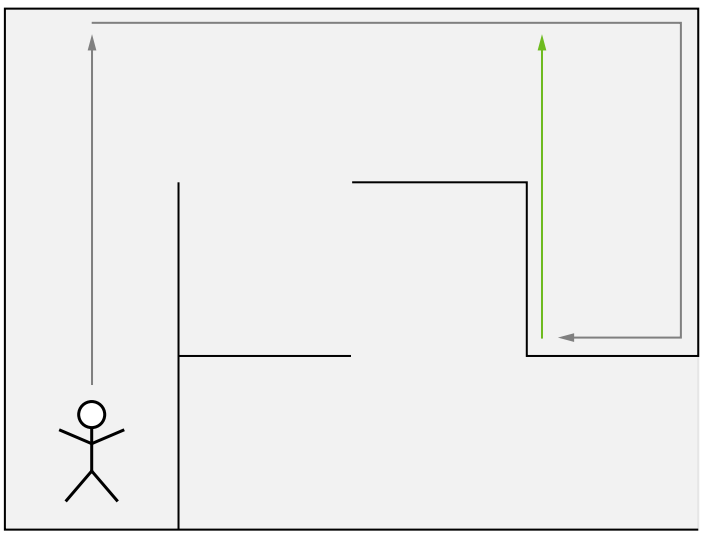
1. Gehe von $s$ nach Norden bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand zu einem Abzweig nach Norden. Beginne von vorn.

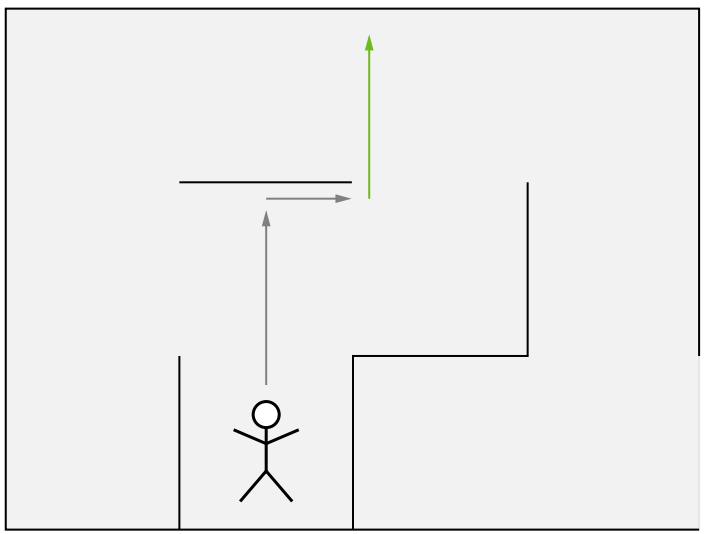

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Oriented Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*OrientedWallFollower*$(G, s)$

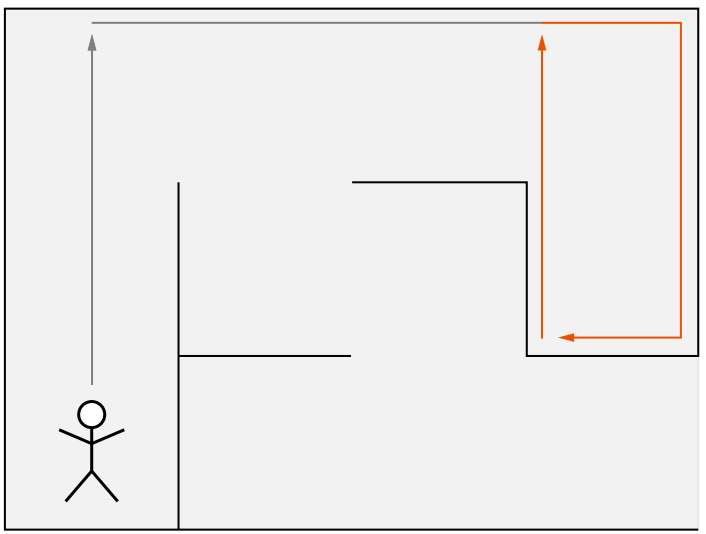
1. Gehe von $s$ nach Norden bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand zu einem Abzweig nach Norden. Beginne von vorn.

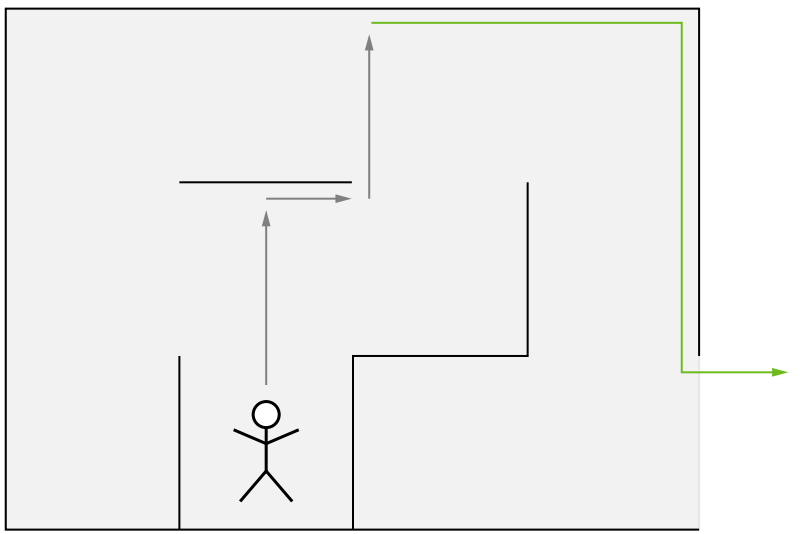

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:





Algorithmus: Oriented Wall Follower. (informell)

Eingabe: $G$. Labyrinth.
$s$. Startposition im Labyrinth.

Ausgabe: Ein Pfad zum Ausgang.

*OrientedWallFollower*$(G, s)$

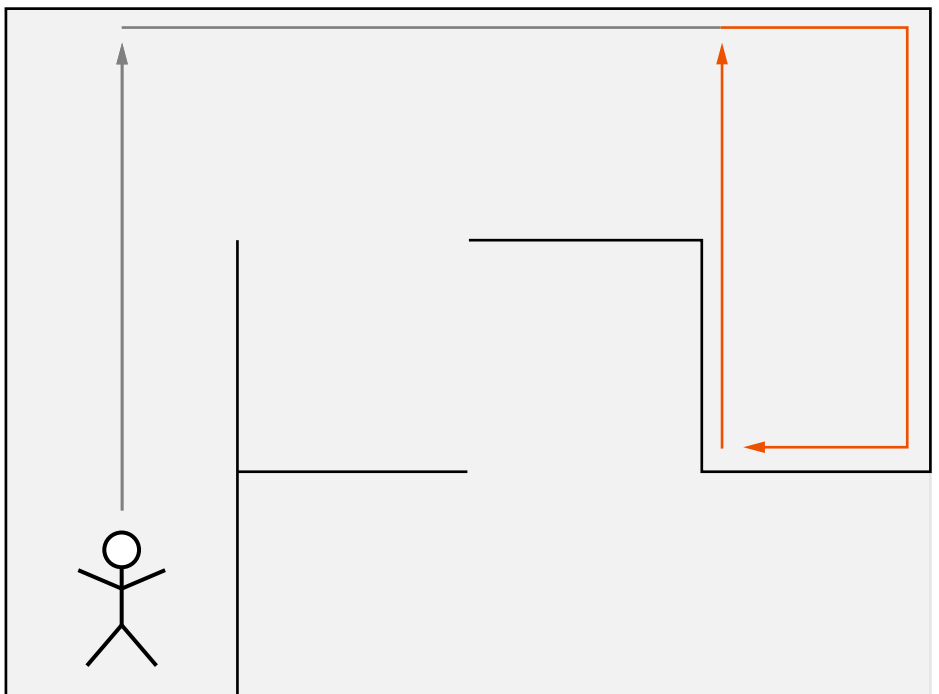
1. Gehe von $s$ nach Norden bis eine Wand oder ein Ausgang von $G$ erreicht ist.
2. Folge der Wand mit der linken Hand zu einem Abzweig nach Norden. Beginne von vorn.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

## Labyrinthproblem

Suche nach einem Ausweg:



Wunsch: Ein Algorithmus zum Verlassen des Labyrinths.

Idee: Nach Norden gehen, wenn man sich nicht um die eigene Achse gedreht hat, sonst an der Wand entlang tasten.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.


© POTTHAST 2018

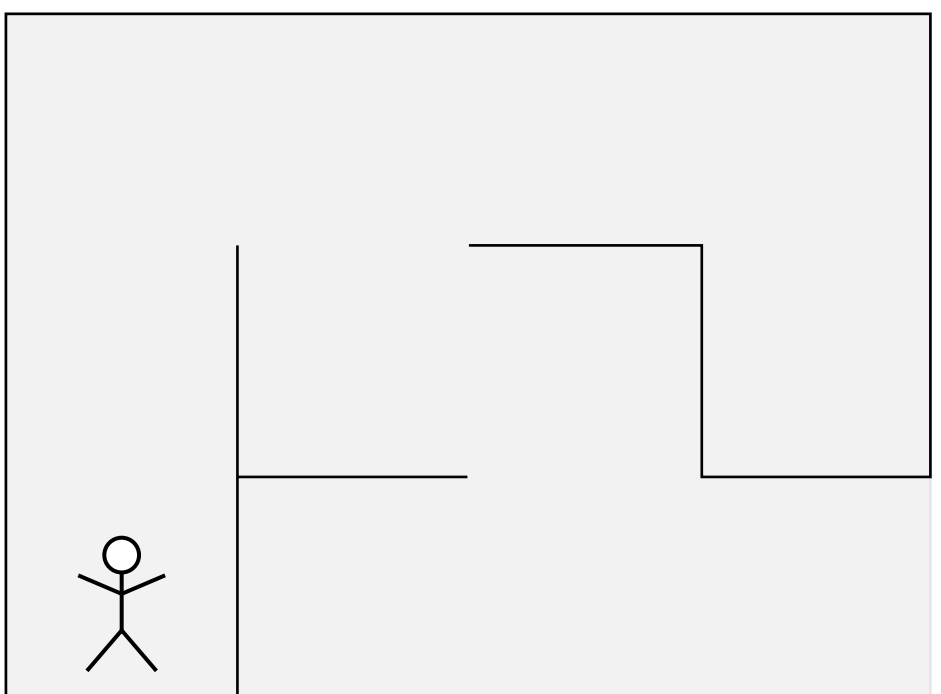
# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.

Modellierung mittels Datenstruktur Graph $G = (V, E)$:




© POTTHAST 2018

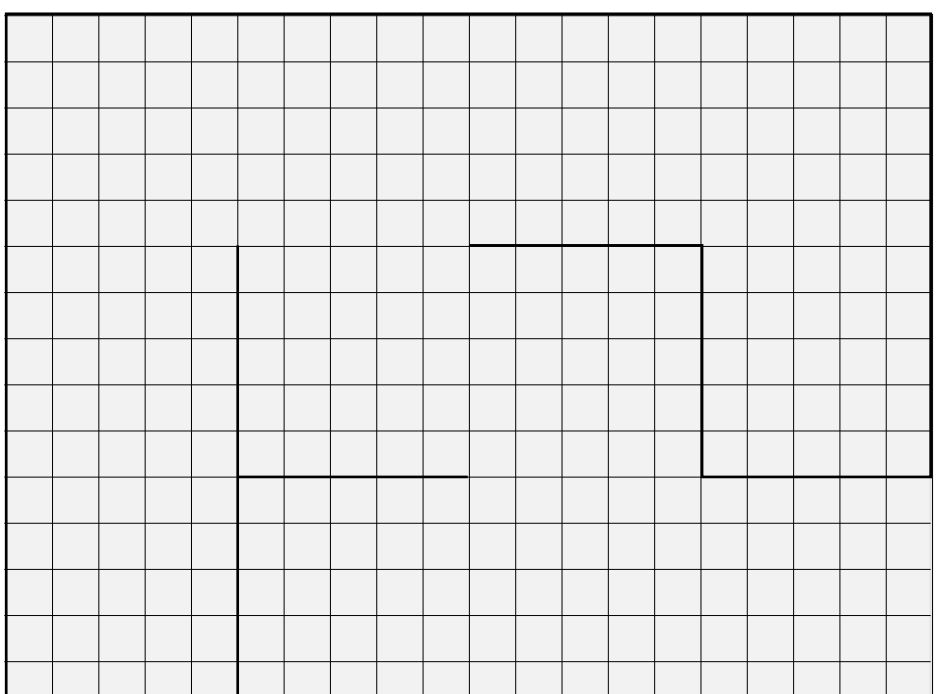
# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.

Modellierung mittels Datenstruktur Graph $G = (V, E)$:




© POTTHAST 2018

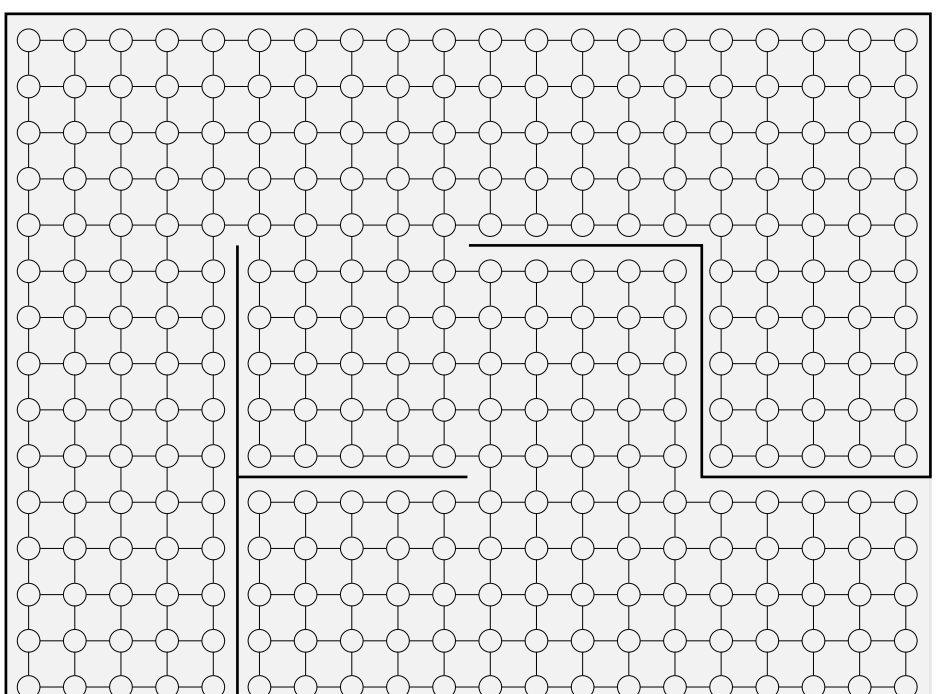
# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.

Modellierung mittels Datenstruktur Graph $G = (V, E)$:




© POTTHAST 2018

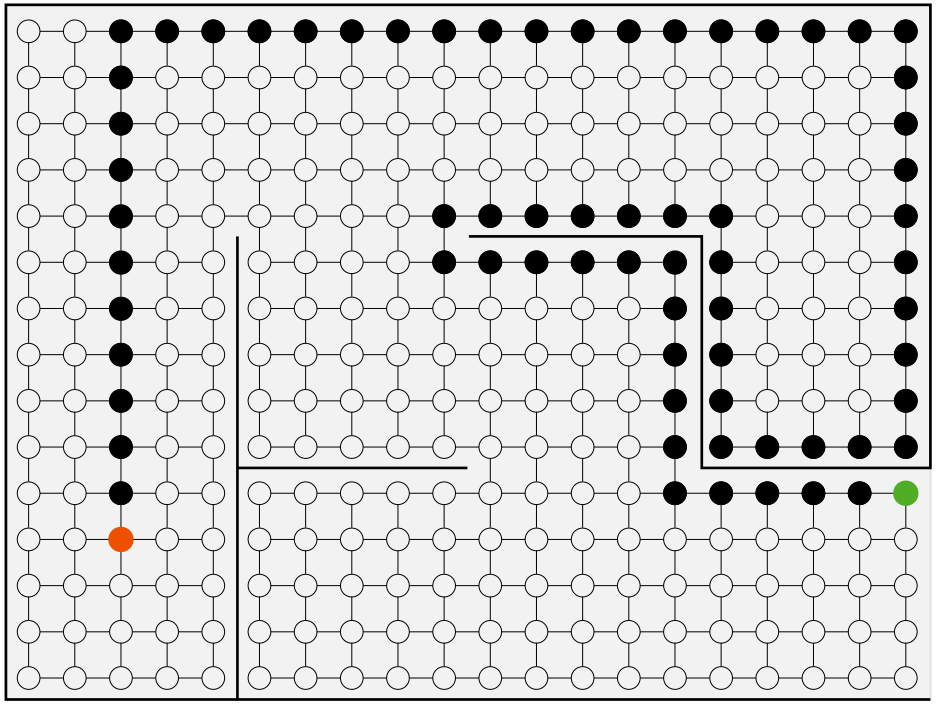
# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.

Modellierung mittels Datenstruktur Graph $G = (V, E)$:




© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

Algorithmus: Pledge Algorithm.

Eingabe: $G$. Labyrinth als Graph $G = (V, E)$.
$s$. Startposition im Labyrinth als Knoten $s \in V$.
$\star(n_0, \ldots, n_k)$. Prädikat; *True* falls $(n_0, \ldots, n_k)$ ein Lösungspfad ist.

Ausgabe: Ein Pfad von $s$ zu einem Ausgangsknoten in $G$.

PledgeAlgorithm$(G, s, \star)$

1. Initialisiere Lösungspfad $b$, aktuelle Position $n$, und Drehungszähler $t$.
2. Wiederhole Schritte 3-5 bis der Ausgang erreicht ist.
3. Wenn der aktuelle Lösungspfad $b$ bereits zum Ausgang führt, gebe $b$ aus.
4. Gehe nach Norden ($t = 0$) solange möglich und man nicht am Ausgang ist. Drehe nach rechts ($t = t + 1$).
5. Solange der Umdrehungszähler $t \neq 0$ und man nicht am Ausgang ist:
   - Wenn links eine Wand und der Weg voraus frei ist, mache einen Schritt vorwärts.
   - Wenn links eine Wand und der Weg voraus versperrt ist, drehe nach rechts ($t = t + 1$).
   - Wenn links keine Wand ist, drehe nach links ($t = t - 1$) und mache einen Schritt vorwärts.


© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

*PledgeAlgorithm*$(G, s, \star)$

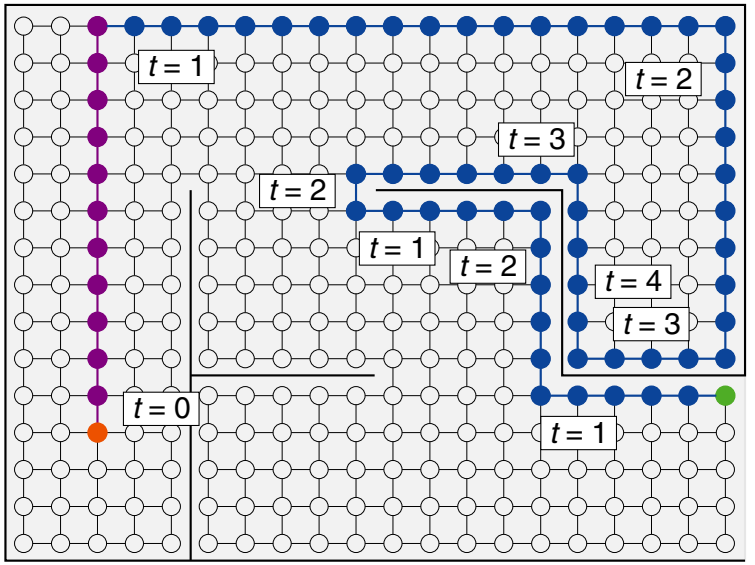
```
1.  b = (s);  n = s;  t = 0
2.  LOOP
3.    IF ⋆(b) THEN return(b) ENDIF
4.    WHILE hasEdge(n,t) AND NOT isExit(n) DO
        n = followEdge(n,t)
        b = append(b,n)
      ENDDO
      t = t + 1
5.    WHILE t ≠ 0 AND NOT isExit(n) DO
        IF hasWallLeft(n,t) THEN
          IF hasEdge(n,t)
          THEN n = followEdge(n,t)
               b = append(b,n)
          ELSE t = t + 1 ENDIF
        ELSE
          t = t − 1
          n = followEdge(n,t)
          b = append(b,n)
        ENDIF
      ENDDO
6.  ENDLOOP
```

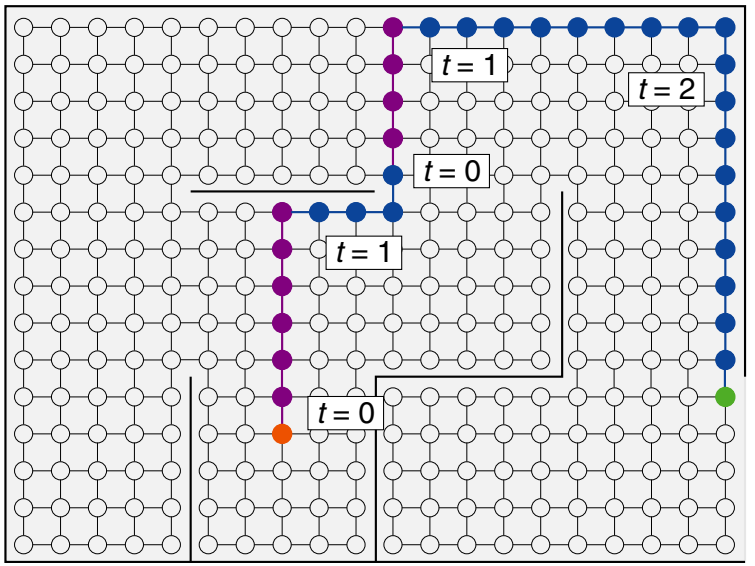

© POTTHAST 2018

# Beispiele für Probleme und algorithmische Lösungen

*PledgeAlgorithm*$(G, s, \star)$

```
1.  b = (s);  n = s;  t = 0
2.  LOOP
3.    IF ⋆(b) THEN return(b) ENDIF
4.    WHILE hasEdge(n,t) AND NOT isExit(n) DO
        n = followEdge(n,t)
        b = append(b,n)
      ENDDO
      t = t + 1
5.    WHILE t ≠ 0 AND NOT isExit(n) DO
        IF hasWallLeft(n,t) THEN
          IF hasEdge(n,t)
          THEN n = followEdge(n,t)
               b = append(b,n)
          ELSE t = t + 1 ENDIF
        ELSE
          t = t − 1
          n = followEdge(n,t)
          b = append(b,n)
        ENDIF
      ENDDO
6.  ENDLOOP
```





© POTTHAST 2018

Bemerkungen:

- Wir modellieren das Labyrinth als Graph $G = (V, E)$: Der Bewegungsraum wird durch Einteilung in Quadrate diskretisiert. Man kann sich von jedem Quadrat in eins der 4 angrenzenden Quadrate bewegen, es sei denn das Quadrat grenzt an eine Wand. Diagonale Bewegung ist nicht erlaubt. Die Quadrate bilden die Menge der Knoten $V$, und die erlaubten Bewegungen zwischen benachbarten Quadraten bilden die Menge der Kanten $E$.

- Der Turn-Counter $t$ dient der Zählung von Links- und Rechtsdrehungen um 90°, wobei $t > 0$ mehr Rechts- als Linksdrehungen, $t < 0$ mehr Links- als Rechtsdrehungen und $t = 0$ eine ausgeglichene Zahl von Links- und Rechtsdrehungen anzeigt.

  Gleichzeitig dient der Turn-Counter $t$ als Hilfsvariable, um in den Funktionen *hasEdge*$(n, t)$, *followEdge*$(n, t)$ und *hasWallLeft*$(n, t)$ die aktuelle Ausrichtung zu ermitteln. Wir vereinbaren, dass $t = 0$ für Norden, $t = 1$ für Osten, $t = 2$ für Süden und $t = 3$ für Westen steht. Für $t < 0$ gilt analog, dass $t = -1$ für Westen, $t = -2$ für Süden und $t = -3$ für Osten steht. Für $t < -3$ und $t > 3$ wird $t \bmod 4$ verwendet.

- Funktion *hasEdge*$(n, t)$ prüft für Knoten $n$, ob eine Kante gemäß der aktuellen Ausrichtung $t$ existiert. Wenn nicht, ist eine Wand voraus.

- Funktion *followEdge*$(n, t)$ liefert den zu Knoten $n$ adjazenten Knoten $n'$ gemäß der aktuellen Ausrichtung $t$. Das Ergebnis ist undefiniert, wenn kein solcher Knoten existiert.

- Funktion *hasWallLeft*$(n, t)$ prüft für einen Knoten $n$, ob links zur Ausrichtung $t$ eine Wand ist. Sie dient als Umsetzung für die Idee, die Hand an die Wand zu legen, um sie zu verfolgen.

- Funktion *append*$(b, n)$ fügt Knoten $n$ ans Ende der Teillösung $b$ an und gibt die neue Teillösung zurück.


© POTTHAST 2018

# Algorithm Engineering

## Fragen

- Terminiert der Algorithmus?
- Ist der Algorithmus korrekt?
- Gibt es einen besseren Algorithmus?
- Wie nahe am Optimum ist der Algorithmus?


© POTTHAST 2018

# Algorithm Engineering

## Fragen

- Terminiert der Algorithmus?
- Ist der Algorithmus korrekt?
- Gibt es einen besseren Algorithmus?
- Wie nahe am Optimum ist der Algorithmus?

## Zeitfragen [Medien] [algorithmwatch.org]

- Kann ich dem „Algorithmus“ vertrauen?
- Entscheidet der „Algorithmus“ in meinem Interesse?
- Entscheidet der „Algorithmus“ in jedermanns Interesse?
- Bevorteilt der „Algorithmus“ manche mehr als andere?


© POTTHAST 2018