14. Zeichen und Texte II

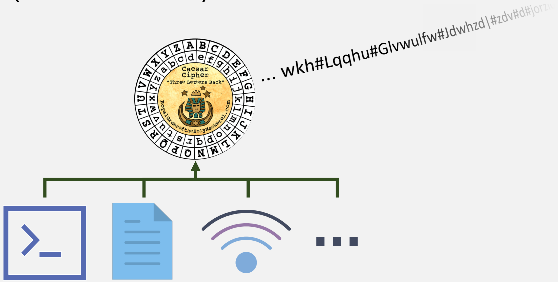
Caesar-Code mit Streams, Texte als Strings, String-Operationen

Caesar-Code: Generalisierung

```cpp
void caesar(int s) {
  std::cin >> std::noskipws;

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

- Momentan nur von `std::cin` nach `std::cout`

- Besser: von beliebiger Zeichenquelle (Konsole, Datei, ...) zu beliebiger Zeichensenke (Konsole, ...)



Icons: flaticon.com; authors Smashicons, Kirill Kazachek; CC 3.0 BY

Caesar-Code: Generalisierung

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

  in >> std::noskipws;

  char next;
  while (in >> next) {
    out << shift(next, s);
  }
}
```

- `std::istream`/`std::ostream` ist ein *generischer Eingabe-/Ausgabestrom* an `char`s

Caesar-Code: Generalisierung

```cpp
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

  in >> std::noskipws;

  char next;
  while (in >> next) {
    out << shift(next, s);
  }
}
```

- `std::istream`/`std::ostream` ist ein *generischer Eingabe-/Ausgabestrom* an `char`s
- Aufruf der Funktion erfolgt mit *spezifischen Strömen*, z.B.: Konsole (`std::cin`/`cout`), Dateien (`std::i`/`ofstream`), Strings (`std::i`/`ostringstream`)

Caesar-Code: Generalisierung, Beispiel 1

```
#include <iostream>
...

// in void main():
caesar(std::cin, std::cout, s);
```

Aufruf der generischen `caesar`-Funktion: Von `std::cin` nach `std::cout`

Caesar-Code: Generalisierung, Beispiel 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Aufruf der generischen `caesar`-Funktion: Von Datei zu Datei

Caesar-Code: Generalisierung, Beispiel 3

```
#include <iostream>
#include <sstream>
...

// in void main():
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

caesar(from, std::cout, s);
```

Aufruf der generischen `caesar`-Funktion: Von einem String nach `std::cout`

Texte

- Text „`Sein oder nicht sein`“ könnte als `vector<char>` repräsentiert werden

Texte

- Text „`Sein oder nicht sein`“ könnte als `vector<char>` repräsentiert werden
- Texte sind jedoch allgegenwärtig, daher existiert in der Standardbibliothek ein eigener Typ für sie: `std::string` (Zeichenkette)
- Benutzung benötigt `#include <string>`

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

- Mit variabler Länge initialisieren:

```
std::string text(n, 'a')
```

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

- Mit variabler Länge initialisieren:

```
std::string text(n, 'a')
```

- Texte vergleichen:

```
if (text1 == text2) ...
```

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Einzelne Zeichen lesen:

```
if (text[0] == 'a') ... // or text.at(0)
```

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Einzelne Zeichen lesen:

```
if (text[0] == 'a') ... // or text.at(0)
```

- Einzelne Zeichen schreiben:

```
text[0] = 'b'; // or text.at(0)
```

Benutzung von `std::string`

- Strings konkatenieren (zusammensetzen):

```
text = ":-";
text += ")";
assert(text == ":-)");
```

- Viele weitere Operationen, bei Interesse siehe
  `https://en.cppreference.com/w/cpp/string`

15. Vektoren II

Mehrdimensionale Vektoren/Vektoren von Vektoren, Kürzeste Wege, Vektoren als Funktionsargumente

Mehrdimensionale Vektoren

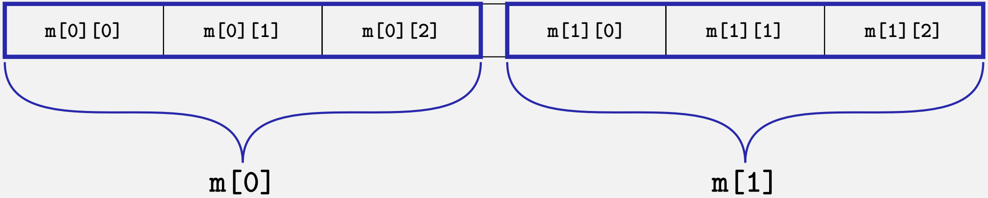
- Zum Speichern von mehrdimensionalen Strukturen wie Tabellen, Matrizen, ...

- ... können *Vektoren von Vektoren* verwendet werden:

```
std::vector<std::vector<int>> m; // An empty matrix
```

Mehrdimensionale Vektoren

Im Speicher: flach



Mehrdimensionale Vektoren

Im Speicher: flach

$$\underbrace{\boxed{\texttt{m[0][0]} \mid \texttt{m[0][1]} \mid \texttt{m[0][2]}}}_{\texttt{m[0]}}\;\underbrace{\boxed{\texttt{m[1][0]} \mid \texttt{m[1][1]} \mid \texttt{m[1][2]}}}_{\texttt{m[1]}}$$

Im Kopf: Matrix

Spalten →

Zeilen ↓

	0	1	2
0	`m[0][0]`	`m[0][1]`	`m[0][2]`
1	`m[1][0]`	`m[1][1]`	`m[1][2]`

Mehrdimensionale Vektoren: Initialisierungsbeispiele

Mittels Literalen:

```
// A 3-by-5 matrix
std::vector<std::vector<std::string>> m = {
  {"ZH", "BE", "LU", "BS", "GE"},
  {"FR", "VD", "VS", "NE", "JU"},
  {"AR", "AI", "OW", "IW", "ZG"}
};

assert(m[1][2] == "VS");
```

Mehrdimensionale Vektoren: Initialisierungsbeispiele

Auf bestimmte Grösse füllen:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
  m(a, std::vector<int>(b, 1));
```

Mehrdimensionale Vektoren: Initialisierungsbeispiele

Auf bestimmte Grösse füllen:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
  m(a, std::vector<int>(b, 1));
```

(Es gibt noch viele weitere Wege, Vektoren zu initialisieren)

Mehrdimensionale Vektoren und Typ-Aliasse

- Auch möglich: Vektoren von Vektoren von Vektoren von ...:
  `std::vector<std::vector<std::vector<...>>>`
- Typnamen können offensichtlich laaaaaaaang werden

Mehrdimensionale Vektoren und Typ-Aliasse

- Auch möglich: Vektoren von Vektoren von Vektoren von ...:
  `std::vector<std::vector<std::vector<...>>>`
- Typnamen können offensichtlich laaaaaaaang werden
- Dann hilft die Deklaration eines *Typ-Alias*:

using *Name* = *Typ*;

Name, unter dem der Typ neu auch angesprochen werden kann

bestehender Typ

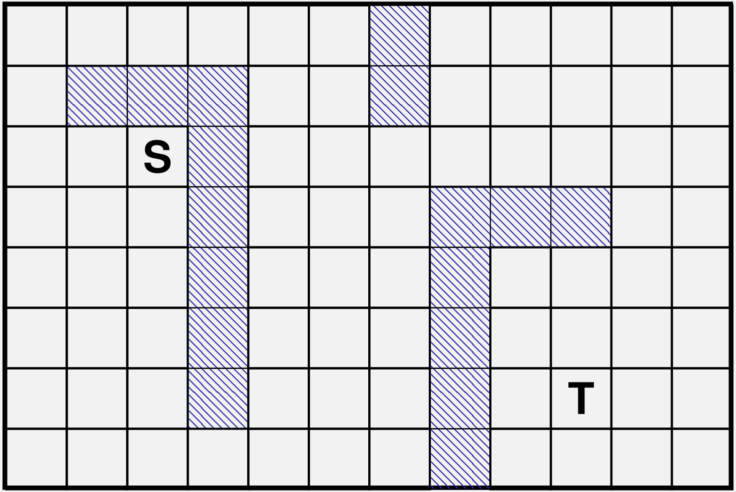
Typ-Aliasse: Beispiel

```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

// POST: Matrix 'm' was output to stream 'out'
void print(const imatrix& m, std::ostream& out);

int main() {
  imatrix m = ...;
  print(m, std::cout);
}
```

Hinweis: `const`-Referenz für Effizienz (keine Kopie) und Sicherheit (unveränderlich)

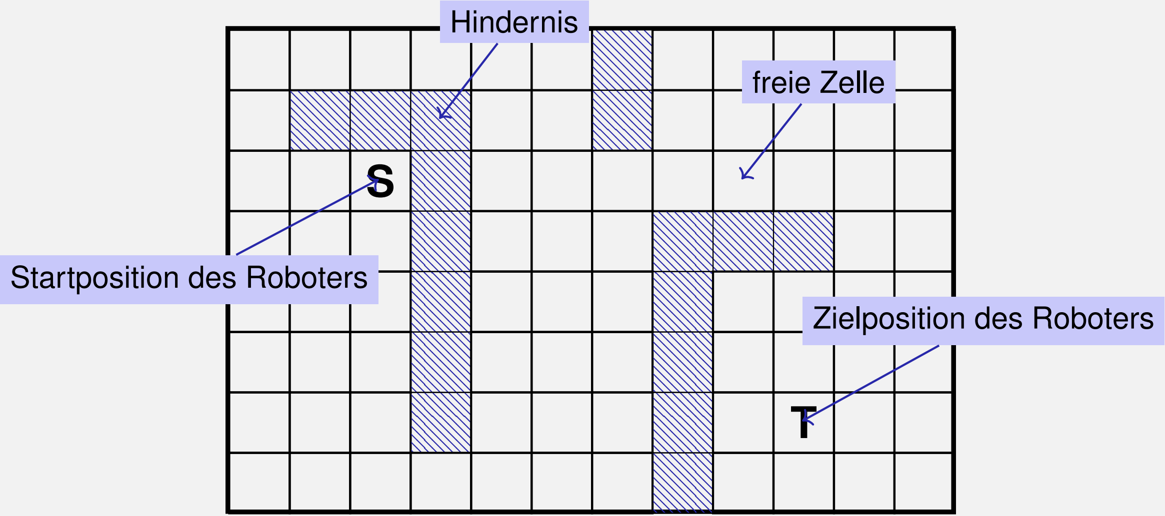
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



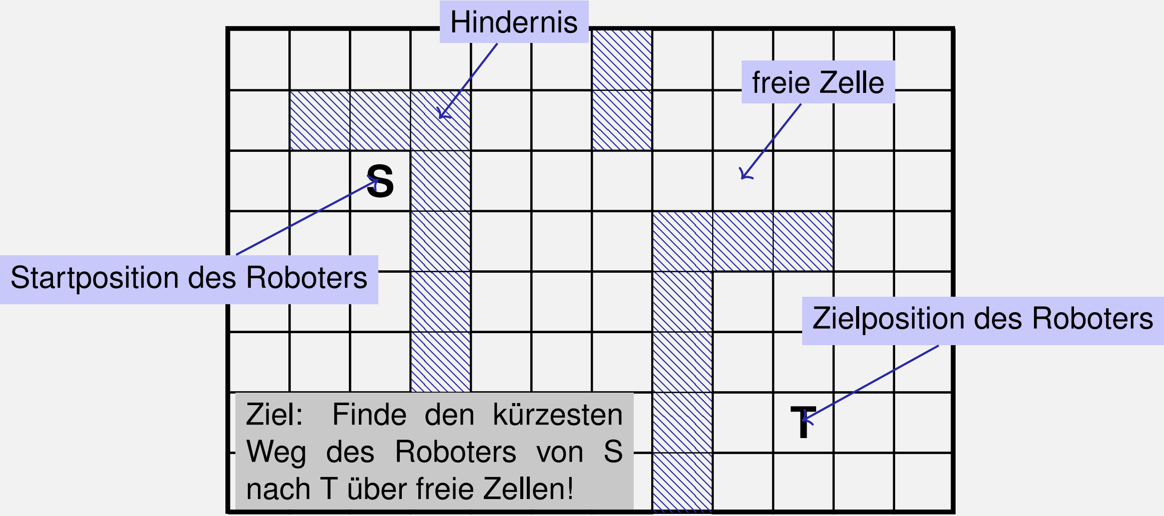
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



Ziel: Finde den kürzesten Weg des Roboters von S nach T über freie Zellen!

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
6	5	4		8	9	10		22	21	20	21
7	6	5	6	7	8	9		23	22	21	22

Ein (scheinbar) anderes Problem

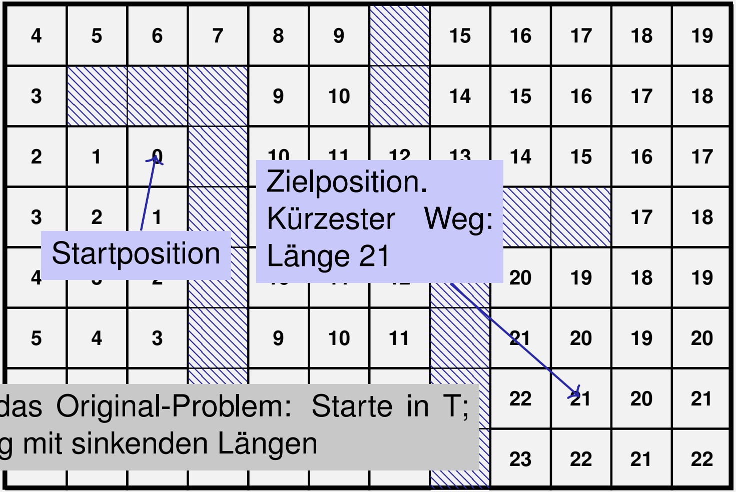
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
								22	21	20	21
								23	22	21	22

Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

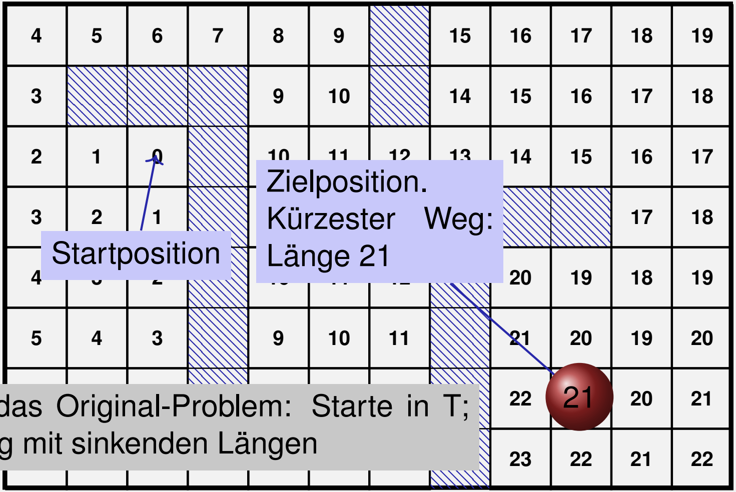
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

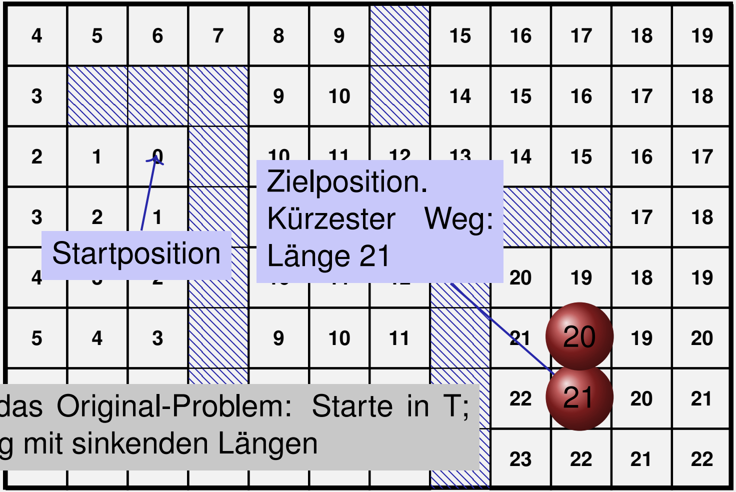
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

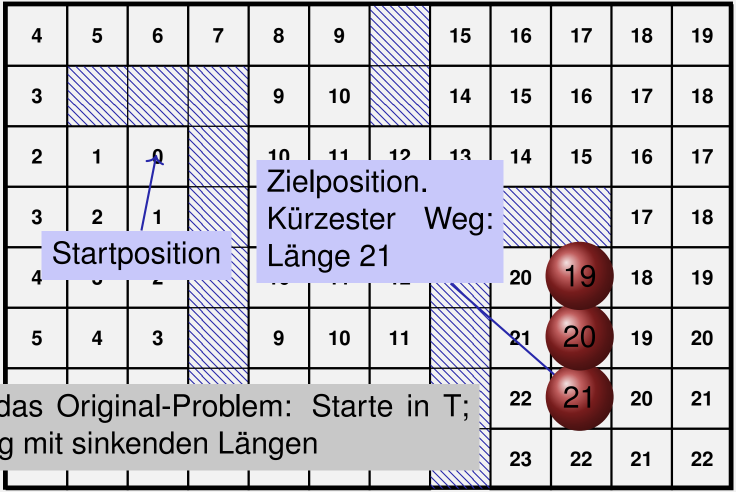
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

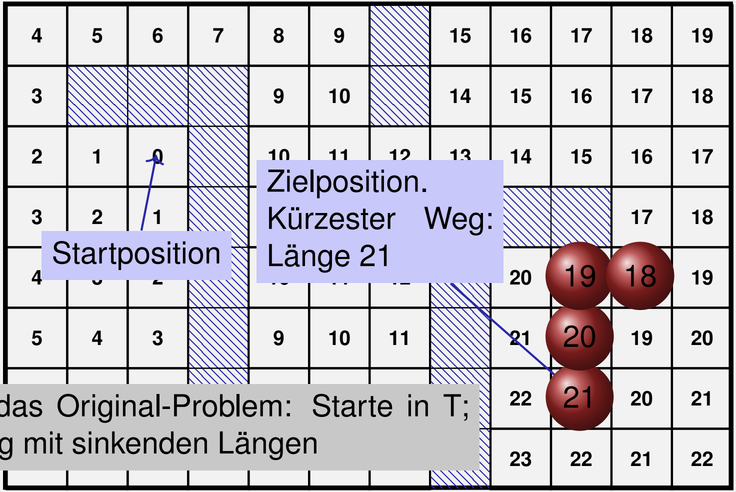
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

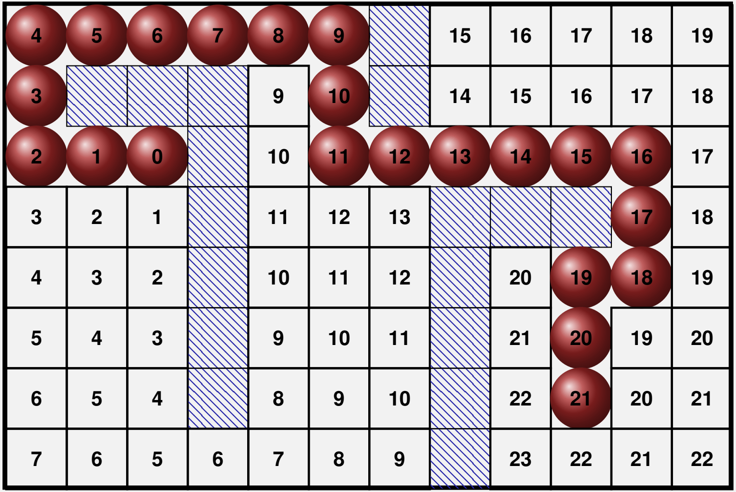
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



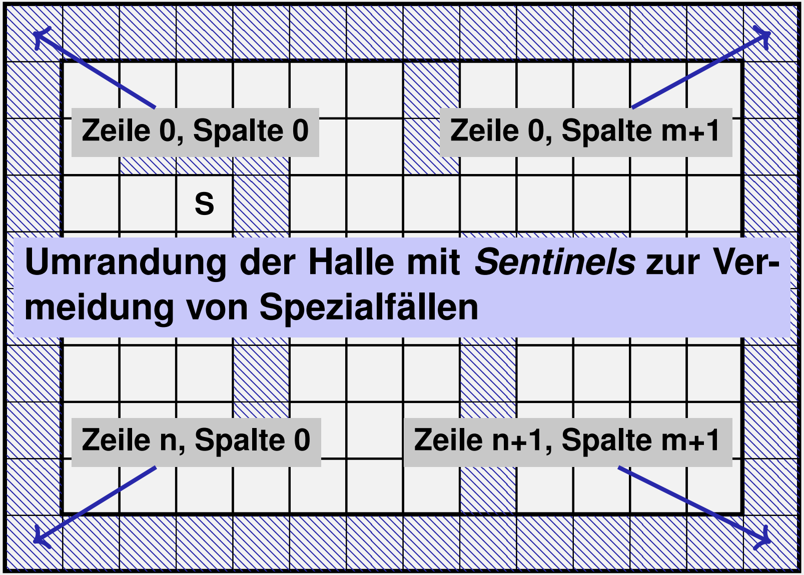
Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

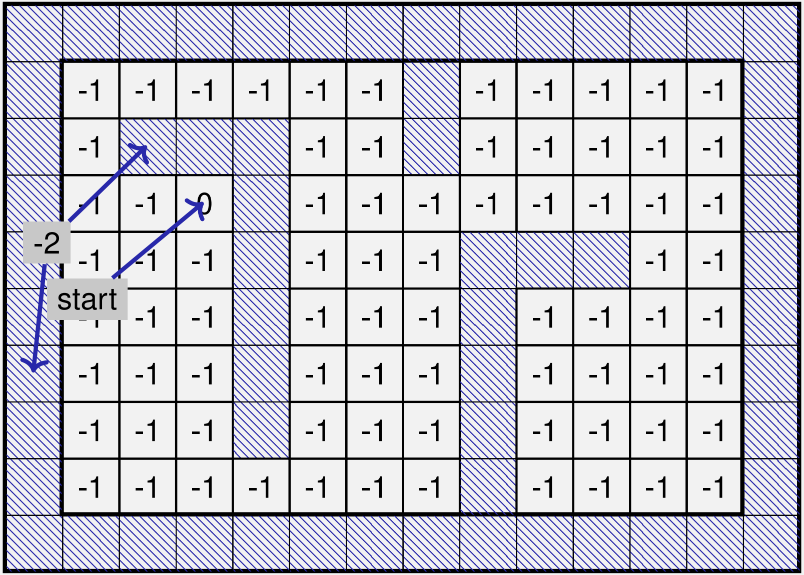


Vorbereitung: Wächter (*Sentinels*)



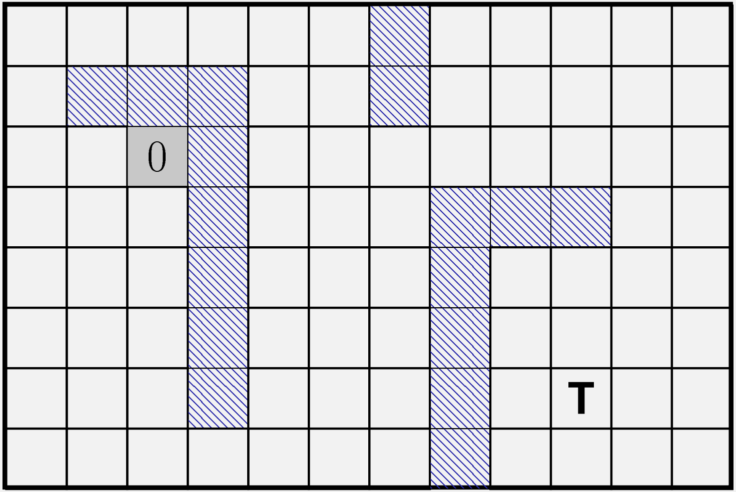
Umrandung der Halle mit *Sentinels* zur Vermeidung von Spezialfällen

Vorbereitung: Initiale Markierung



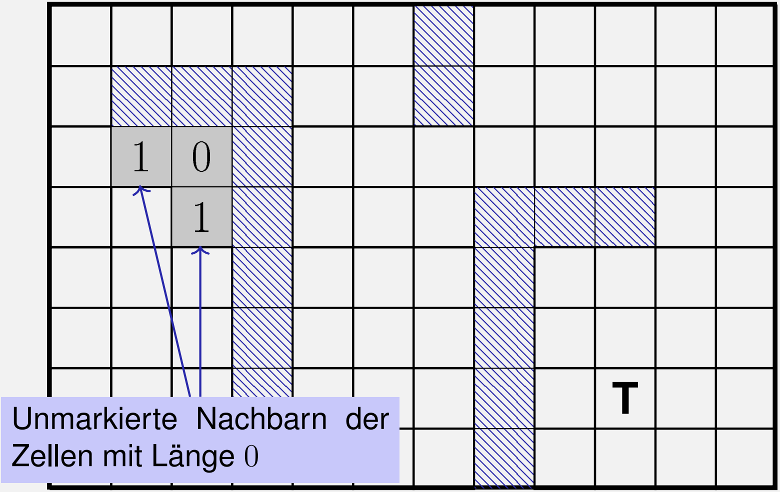
Markierung aller Zellen mit ihren Weglängen

Schritt 0: Alle Zellen mit Weglänge 0



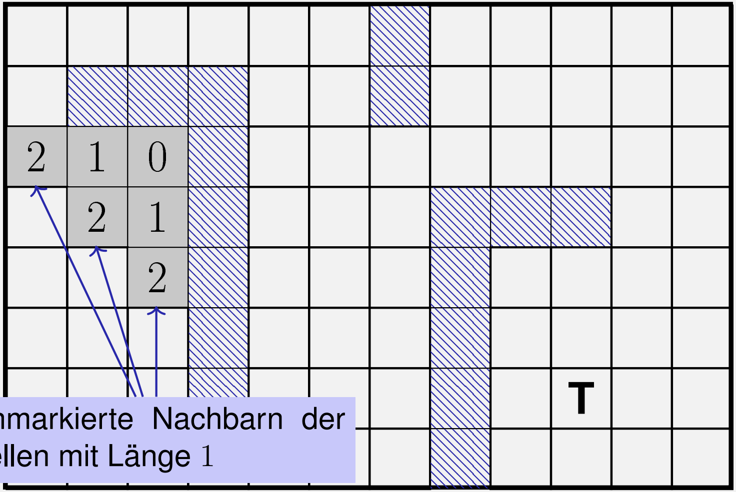
Markierung aller Zellen mit ihren Weglängen

Schritt 1: Alle Zellen mit Weglänge 1



Markierung aller Zellen mit ihren Weglängen

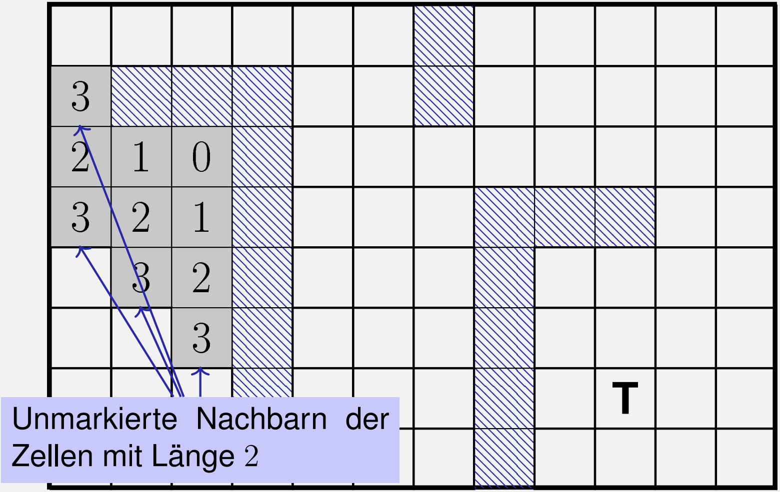
Schritt 2: Alle Zellen mit Weglänge 2



Unmarkierte Nachbarn der Zellen mit Länge 1

Markierung aller Zellen mit ihren Weglängen

Schritt 3: Alle Zellen mit Weglänge 3



Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != −1) continue;
      if (floor[r−1][c] == i−1 || floor[r+1][c] == i−1 ||
          floor[r][c−1] == i−1 || floor[r][c+1] == i−1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != −1) continue;
      if (floor[r−1][c] == i−1 || floor[r+1][c] == i−1 ||
          floor[r][c−1] == i−1 || floor[r][c+1] == i−1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

zeigt an, ob in einem Durchlauf durch alle Zellen Fortschritt gemacht wurde

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
     if (floor[r][c] != -1) continue;
     if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
         floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
       floor[r][c] = i; // label cell with i
       progress = true;
     }
    }
  if (!progress) break;
}
```

Gehe über alle Zellen

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

Zelle schon markiert oder Hindernis

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != −1) continue;
      if (floor[r−1][c] == i−1 || floor[r+1][c] == i−1 ||
          floor[r][c−1] == i−1 || floor[r][c+1] == i−1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != −1) continue;
      if (floor[r−1][c] == i−1 || floor[r+1][c] == i−1 ||
          floor[r][c−1] == i−1 || floor[r][c+1] == i−1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

Ein Nachbar hat Weglänge $i - 1$. Die Wächter garantieren immer 4 Nachbarn.

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != −1) continue;
      if (floor[r−1][c] == i−1 || floor[r+1][c] == i−1 ||
          floor[r][c−1] == i−1 || floor[r][c+1] == i−1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

Kein Fortschritt, alle erreichbaren Zellen markiert; fertig.

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. Tiefensuche wird typischerweise in Algorithmik-Vorlesungen diskutiert)

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. Tiefensuche wird typischerweise in Algorithmik-Vorlesungen diskutiert)
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. Tiefensuche wird typischerweise in Algorithmik-Vorlesungen diskutiert)
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden
- Verbesserung: Für Markierung i, durchlaufe nur die Nachbarn der Zellen mit Markierung $i-1$
- Verbesserung: Stoppe, sobald das Ziel erreicht wurde

16. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration, n-Damen Problem, Lindenmayer System

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich rekursiv definierbar.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich rekursiv definierbar.
- Das heisst, die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \le 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

Rekursion in $C{+}{+}$: Genauso!

$$n! = \begin{cases} 1, & \text{falls } n \le 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

```
// POST: return value is n!
unsigned int fac(unsigned int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife . . .

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife . . .
- . . . nur noch schlechter („verbrennt“ Zeit und Speicher)

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife . . .
- . . . nur noch schlechter („verbrennt“ Zeit und Speicher)

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife . . .
- . . . nur noch schlechter („verbrennt“ Zeit und Speicher)

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Ein Euro ist ein Euro.

Wim Duisenberg, erster Präsident der EZB

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir *garantierten Fortschritt Richtung einer Abbruchbedingung (≈ Basisfall)*

Beispiel `fac(n)`:

- Rekursion endet falls $n \leq 1$
- Rekursiver Aufruf mit neuem Argument $< n$
- Abbruchbedingung wird daher garantiert erreicht

```
unsigned int fac(
    unsigned int n) {

  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Aufruf von `fac(4)`

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Auswertung des Rückgabeausdrucks

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Rekursiver Aufruf mit Argument $n - 1 == 3$

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei n. Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac(unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es wird mit dem n des aktuellen Aufrufs gearbeitet: $n = 3$

Initialisierung des formalen Arguments

Der Aufrufstapel

```
std:cout << fac(4)
```

Der Aufrufstapel

Bei jedem Funktionsaufruf:

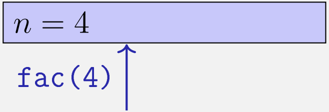
fac(4)

```
std:cout << fac(4)
```

Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel

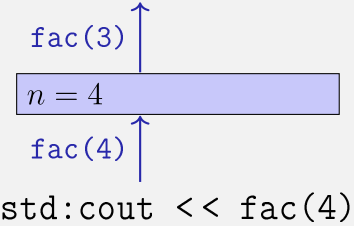



```
std:cout << fac(4)
```

Der Aufrufstapel

Bei jedem Funktionsaufruf:

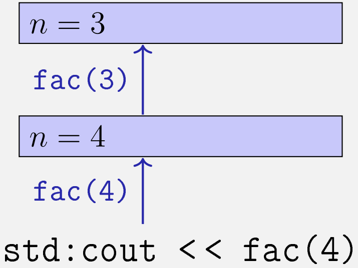
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

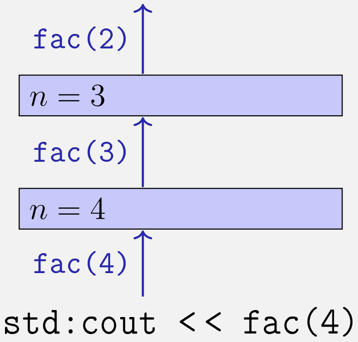
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

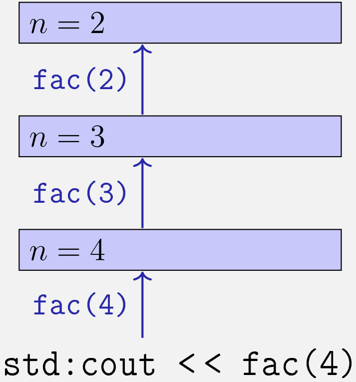
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

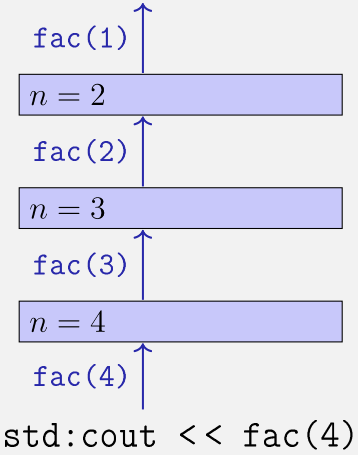
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

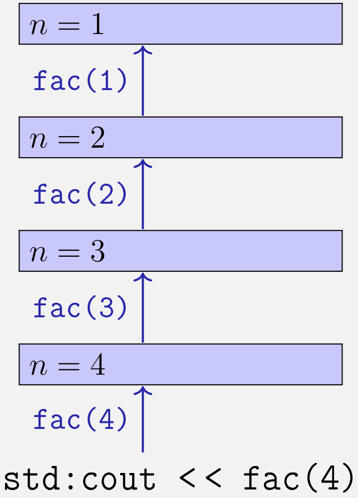
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

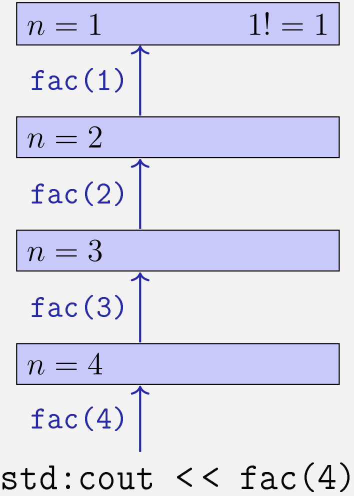
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

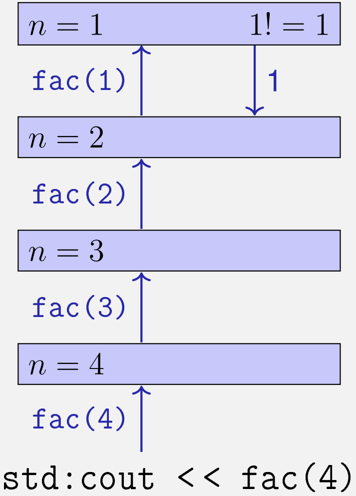
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Funktionsaufruf:

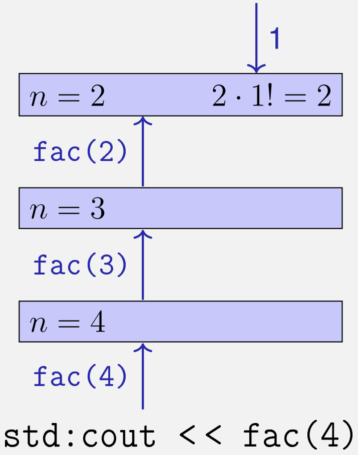
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

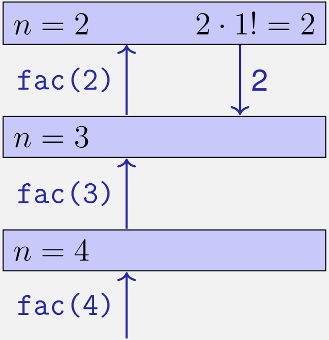
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

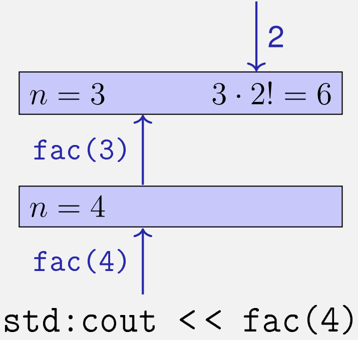


`std:cout << fac(4)`

Der Aufrufstapel

Bei jedem Funktionsaufruf:

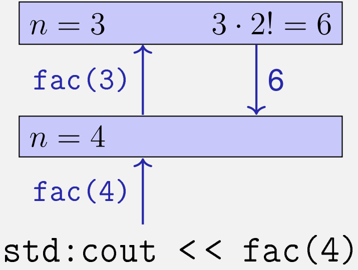
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

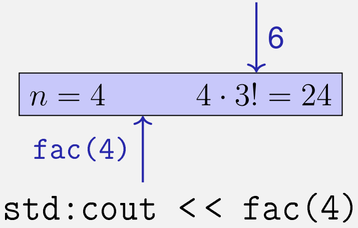
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

$\boxed{n = 4 \qquad 4 \cdot 3! = 24}$

`fac(4)` $\uparrow$ $\downarrow$ 24

`std:cout << fac(4)`

Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

24

```
std:cout << fac(4)
```

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\mathrm{gcd}(a, b)$ zweier natürlicher Zahlen a und b

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\mathrm{gcd}(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion (Beweis im Skript):

$$\mathrm{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \mathrm{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in C++

$$\mathrm{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \mathrm{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
unsigned int gcd(unsigned int a, unsigned int b) {
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

Euklidischer Algorithmus in C++

$$\operatorname{gcd}(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \operatorname{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
unsigned int gcd(unsigned int a, unsigned int b) {
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

Fibonacci-Zahlen in Zürich

Fibonacci-Zahlen in C++

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fibonacci-Zahlen in C++

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fibonacci-Zahlen in C++

Laufzeit

`fib(50)` dauert „ewig“, denn es berechnet
F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal,
F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen `a` und `b`)

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen `a` und `b`)
- Berechne die nächste Zahl als Summe von `a` und `b`

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen `a` und `b`)
- Berechne die nächste Zahl als Summe von `a` und `b`

Kann rekursiv und iterativ implementiert werden, letzteres ist einfacher/direkter

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  unsigned int a = 0;  // F_0
  unsigned int b = 1;  // F_1
  for (unsigned int i = 2; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a b

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  unsigned int a = 0;  // F_0
  unsigned int b = 1;  // F_1
  for (unsigned int i = 2; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a b

Schnelle Fibonacci-Zahlen in $C++$

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  unsigned int a = 0;  // F_0
  unsigned int b = 1;  // F_1
  for (unsigned int i = 2; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (\underbrace{F_{i-1}}_{a}, \underbrace{F_i}_{b})$$

Schnelle Fibonacci-Zahlen in $C{+}{+}$

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  unsigned int a = 0;  // F_0
  unsigned int b = 1;  // F_1
  for (unsigned int i = 2; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

sehr schnell auch bei `fib(50)`

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

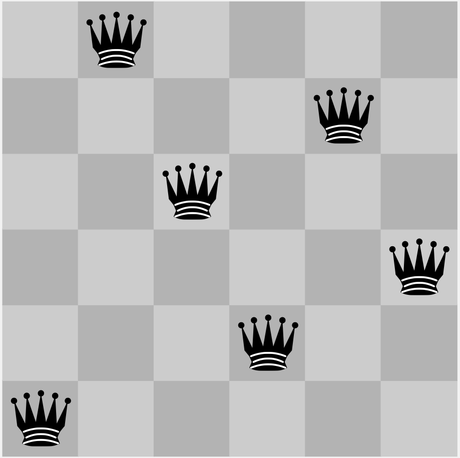
a

b

Die Macht der Rekursion

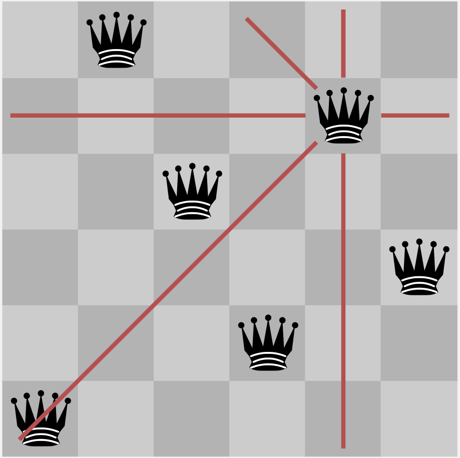
- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich deutlich einfacher lösbar.
- Beispiele: *das n-Damen-Problem*, Die Türme von Hanoi, Parsen von Ausdrücken, *Sudoku-Löser*, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren)

Das n-Damen Problem



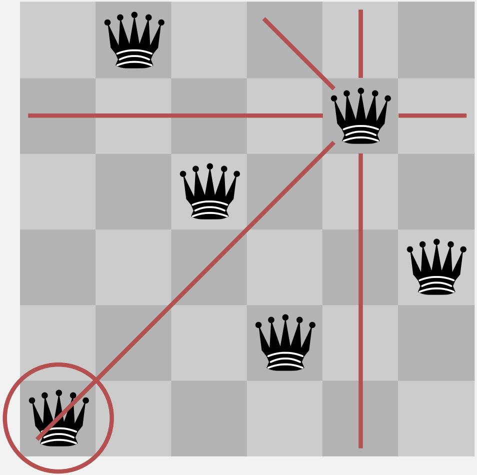
- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n-Damen Problem



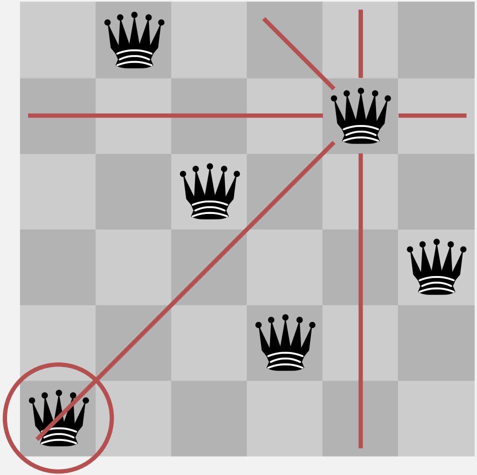
- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n-Damen Problem



- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n-Damen Problem



- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?
- Wenn ja, wie viele Lösungen gibt es?

Lösung?

- Durchprobieren aller Möglichkeiten?

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!

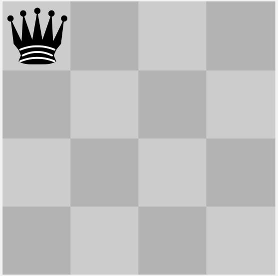
Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile: n^n Möglichkeiten. Besser – aber auch noch zu viele.

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile: n^n Möglichkeiten. Besser – aber auch noch zu viele.
- Idee: Unsinnige Versuche gar nicht erst weiterverfolgen, stattdessen falsche Züge zurücknehmen $\Rightarrow$ *Backtracking*

Lösung mit Backtracking

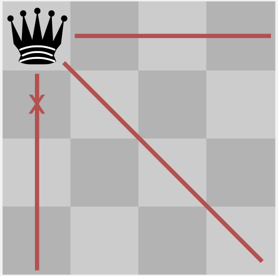


Erste Dame

queens

0
0
0
0

Lösung mit Backtracking

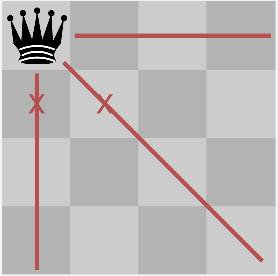


Verbotene Felder: hier dürfen keine anderen Damen stehen.

queens

0
0
0
0

Lösung mit Backtracking

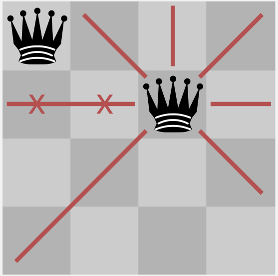


Verbotene Felder: hier dürfen keine anderen Damen stehen.

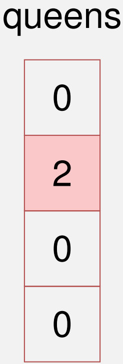
queens

0
1
0
0

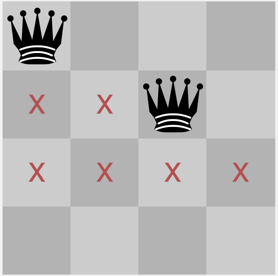
Lösung mit Backtracking



Nächste Dame in nächster Zeile (keine Kollision)



Lösung mit Backtracking

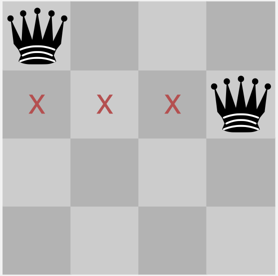


Alle Felder in nächster Zeile verboten. Zurück! (Backtracking!)

queens

0
2
4
0

Lösung mit Backtracking

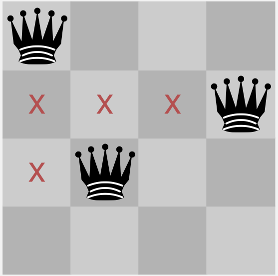


Dame eins weiter setzen und wieder versuchen

queens

0
3
0
0

Lösung mit Backtracking

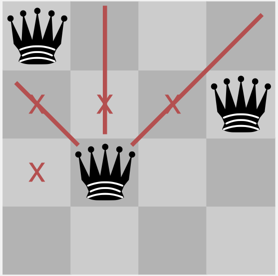


Nächste Zeile

queens

0
3
1
0

Lösung mit Backtracking

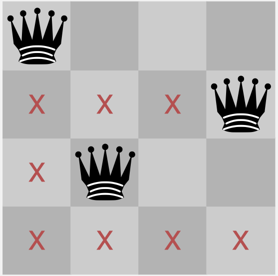


Ok (nur bereits gesetzte Damen müssen getestet werden)

queens

0
3
1
0

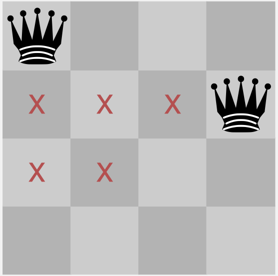
Lösung mit Backtracking



Alle Felder der nächsten Zeile verboten. Zurück.

queens
0
3
1
4

Lösung mit Backtracking

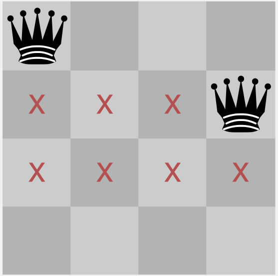


Weiter in der vorigen Zeile

queens

0
3
1
0

Lösung mit Backtracking

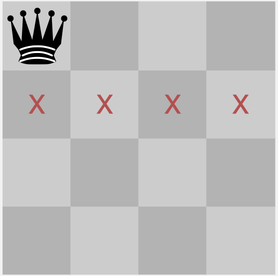


Alle restlichen Felder auch verboten. Weiter zurück (backtracking)

queens

0
3
4
0

Lösung mit Backtracking

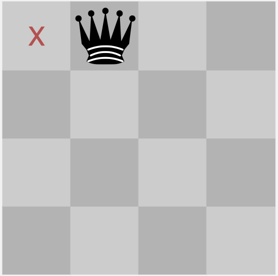


Alle Felder dieser Zeile führten zu keiner Lösung. Weiter zurück (back-tracking)

queens

0
4
0
0

Lösung mit Backtracking

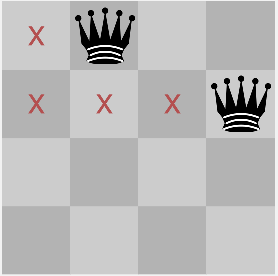


Setze Dame wieder eins weiter.

queens

1
0
0
0

Lösung mit Backtracking

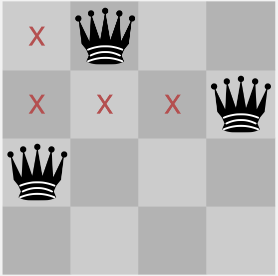


nächste Zeile

queens

1
3
0
0

Lösung mit Backtracking

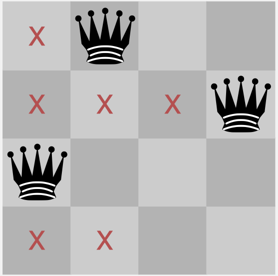


nächste Zeile

queens

1
3
0
0

Lösung mit Backtracking

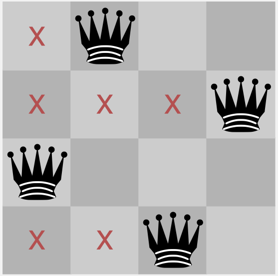


nächste Zeile

queens

1
3
0
1

Lösung mit Backtracking

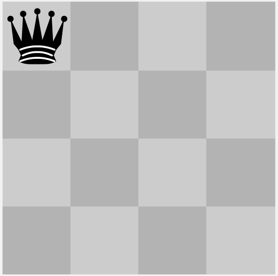


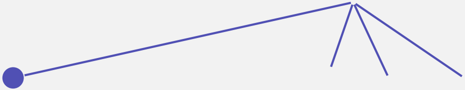
Lösung gefunden

queens

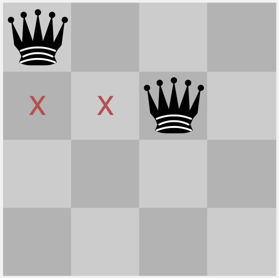
1
3
0
2

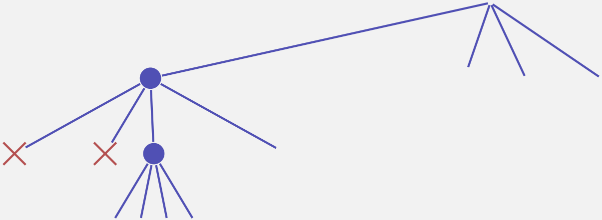
Suchstrategie als Baum visualisiert



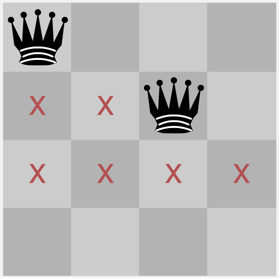


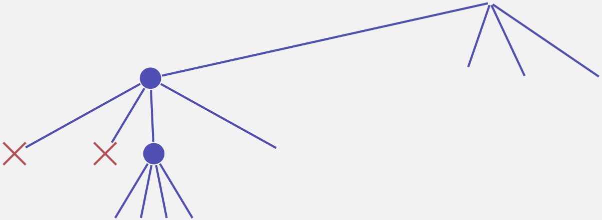
Suchstrategie als Baum visualisiert



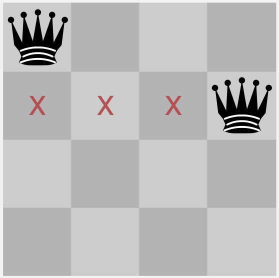


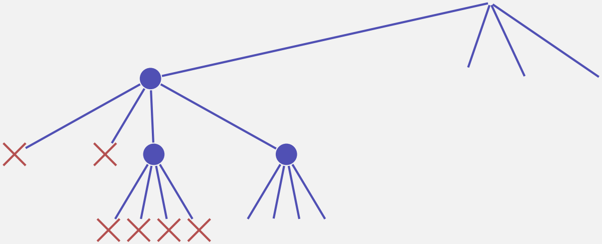
Suchstrategie als Baum visualisiert



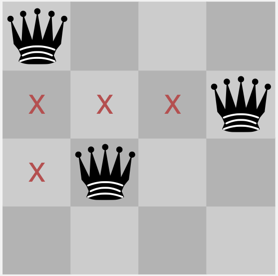


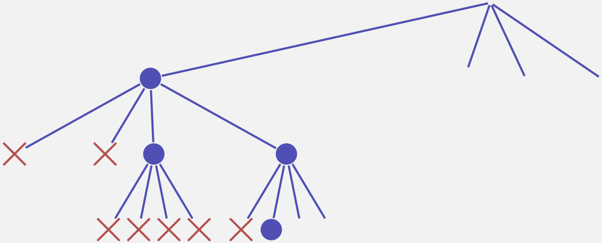
Suchstrategie als Baum visualisiert



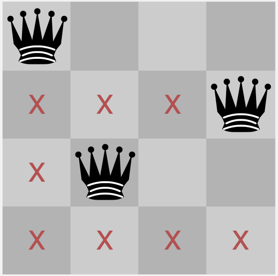


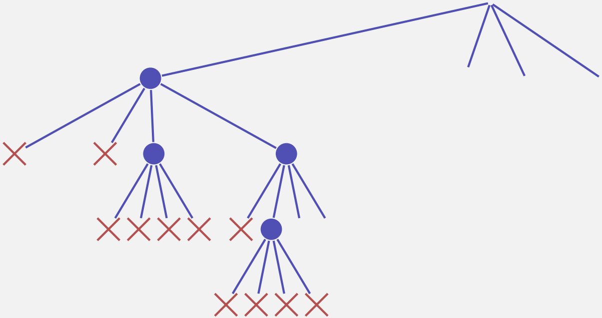
Suchstrategie als Baum visualisiert



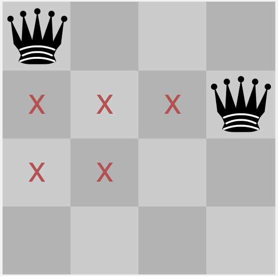


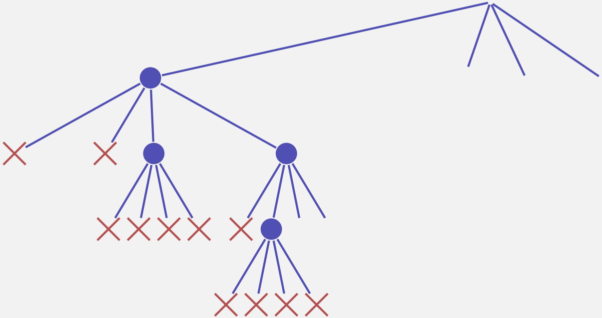
Suchstrategie als Baum visualisiert



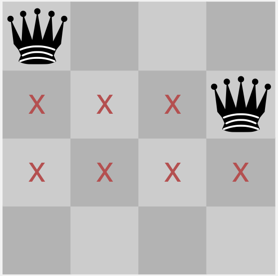


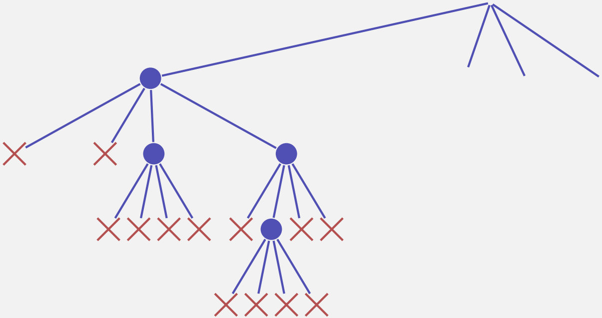
Suchstrategie als Baum visualisiert



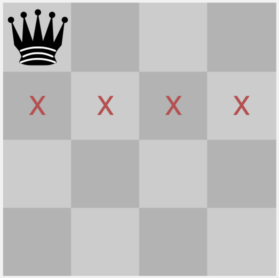


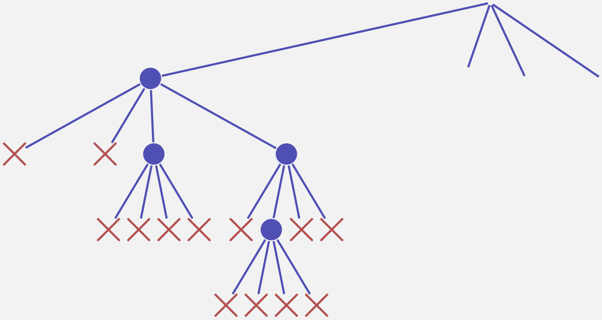
Suchstrategie als Baum visualisiert



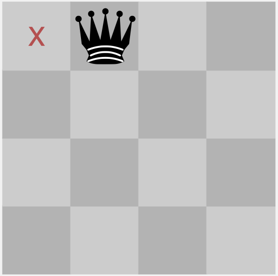


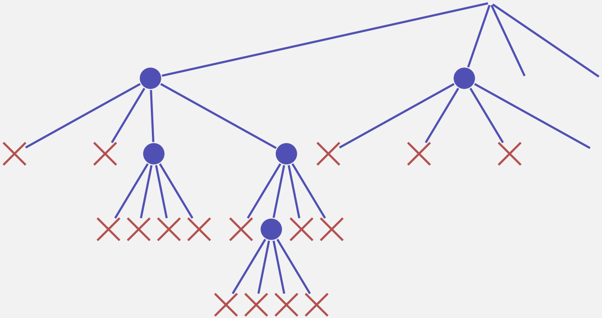
Suchstrategie als Baum visualisiert



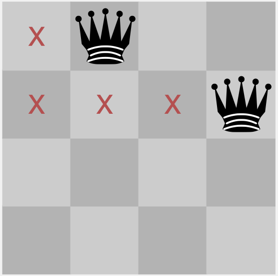


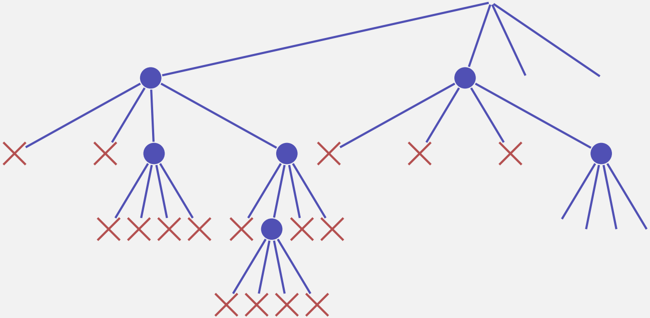
Suchstrategie als Baum visualisiert



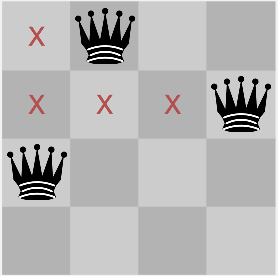


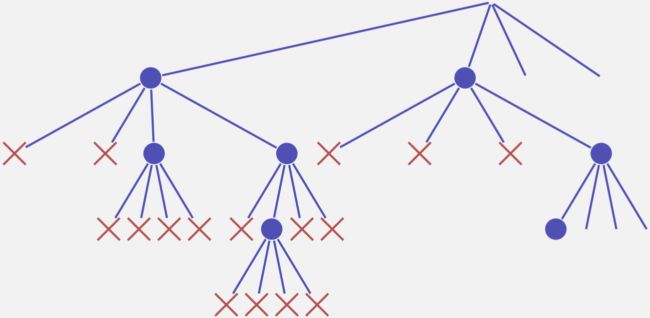
Suchstrategie als Baum visualisiert



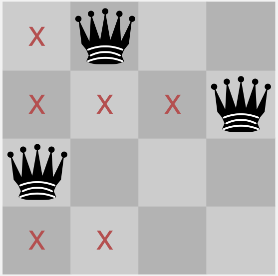


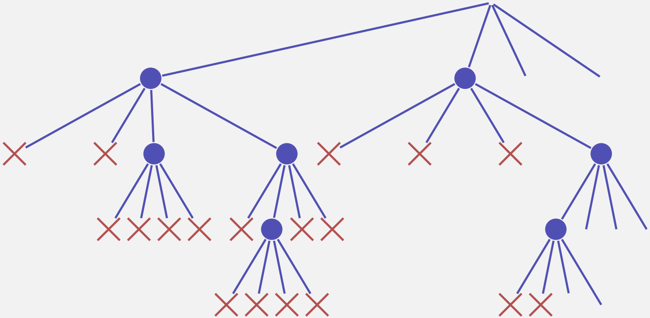
Suchstrategie als Baum visualisiert



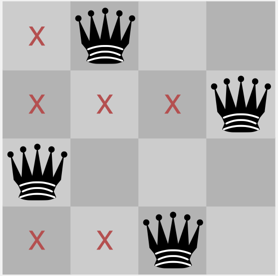


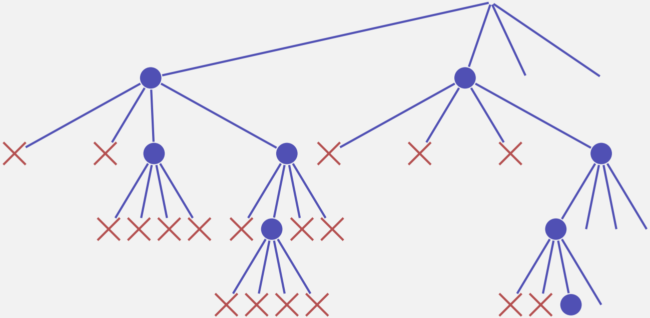
Suchstrategie als Baum visualisiert





Suchstrategie als Baum visualisiert





Prüfe Dame

```
using Queens = std::vector<unsigned int>;

// post: returns if queen in the given row is valid, i.e.
//       does not share a common row, column or diagonal
//       with any of the queens on rows 0 to row-1
bool valid(const Queens& queens, unsigned int row) {
  unsigned int col = queens[row];
  for (unsigned int r = 0; r != row; ++r) {
    unsigned int c = queens[r];
    if (col == c || col - row == c - r || col + row == c + r)
      return false; // same column or diagonal
  }
  return true; // no shared column or diagonal
}
```

Rekursion: Finde eine Lösung

```
// pre: all queens from row 0 to row−1 are valid,
//      i.e. do not share any common row, column or diagonal
// post: returns if there is a valid position for queens on
//      row .. queens.size(). if true is returned then the
//      queens vector contains a valid configuration.
bool solve(Queens& queens, unsigned int row) {
  if (row == queens.size())
    return true;
  for (unsigned int col = 0; col != queens.size(); ++col) {
    queens[row] = col;
    if (valid(queens, row) && solve(queens,row+1))
        return true; // (else check next position)
  }
  return false; // no valid configuration found
}
```

Rekursion: Zähle alle Lösungen

```
// pre: all queens from row 0 to row-1 are valid,
//   i.e. do not share any common row, column or diagonal
// post: returns the number of valid configurations of the
//   remaining queens on rows row ... queens.size()
int nSolutions(Queens& queens, unsigned int row) {
  if (row == queens.size())
    return 1;
  int count = 0;
  for (unsigned int col = 0; col != queens.size(); ++col) {
    queens[row] = col;
    if (valid(queens, row))
      count += nSolutions(queens,row+1);
  }
  return count;
}
```

Hauptprogramm

```
// pre: positions of the queens in vector queens
// post: output of the positions of the queens in a graphical way
void print(const Queens& queens);

int main() {
  int n;
  std::cin >> n;
  Queens queens(n);
  if (solve(queens,0)) {
    print(queens);
    std::cout << "# solutions:" << nSolutions(queens,0) << std::endl;
  } else
    std::cout << "no solution" << std::endl;
  return 0;
}
```