13. Vectors and Strings II

Strings, Multidimensional Vector/Vectors of Vectors, Shortest Paths, Vectors as Function Arguments

Texts

- Text “`to be or not to be`” could be represented as `vector<char>`

Texts

- Text “`to be or not to be`” could be represented as `vector<char>`
- Texts are ubiquitous, however, and thus have their own typ in the standard library: `std::string`
- Requires `#include <string>`

Using `std::string`

- Declaration, and initialisation with a literal:

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

Using `std::string`

- Declaration, and initialisation with a literal:

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

- Initialise with variable length:

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

Using `std::string`

- Declaration, and initialisation with a literal:

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

- Initialise with variable length:

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

- Comparing texts:

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

Using `std::string`

- Querying size:

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

Using `std::string`

- Querying size:

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

- Reading single characters:

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

Using `std::string`

- Querying size:

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

- Reading single characters:

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

- Writing single characters:

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

Using `std::string`

- Concatenate strings:

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

- Many more operations; if interested, see `https://en.cppreference.com/w/cpp/string`

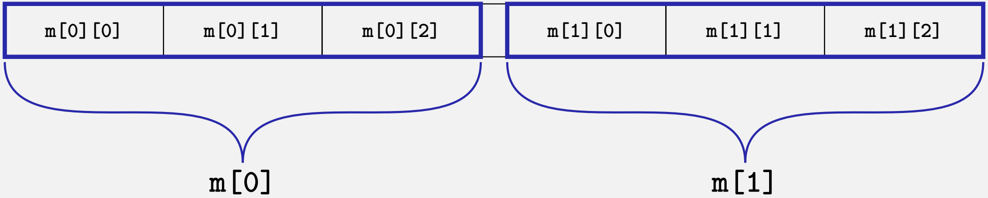
Multidimensional Vectors

- For storing multidimensional structures such as tables, matrices, ...
- ... *vectors of vectors* can be used:

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

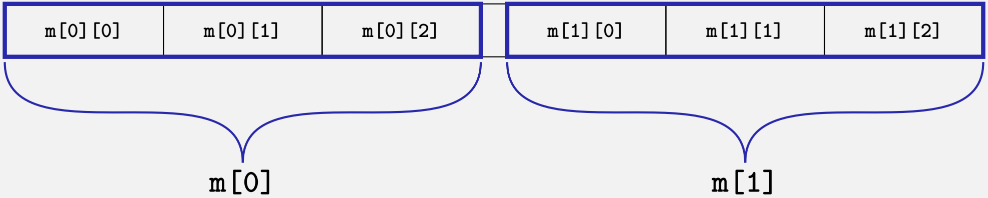
Multidimensional Vectors

In memory: flat



Multidimensional Vectors

In memory: flat



in our head: matrix

columns

rows

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

Multidimensional Vectors: Initialisation Examples

Using literals:

```
// 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");
```

Multidimensional Vectors: Initialisation Examples

Fill to specific size:

```
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));
```

Multidimensional Vectors: Initialisation Examples

Fill to specific size:

```
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));
```

(Many further ways of initialising a vector exist)

Multidimensional Vectors and Type Aliases

- Also possible: vectors of vectors of vectors of ...:
  `std::vector<std::vector<std::vector<...>>>`
- Type names can obviously become looooooong

Multidimensional Vectors and Type Aliases

- Also possible: vectors of vectors of vectors of ...:
  `std::vector<std::vector<std::vector<...>>>`
- Type names can obviously become looooooong
- The declaration of a *type alias* helps here:

`using` *Name* = *Typ*;

Name that can now be used to access the type

existing type

Type Aliases: Example

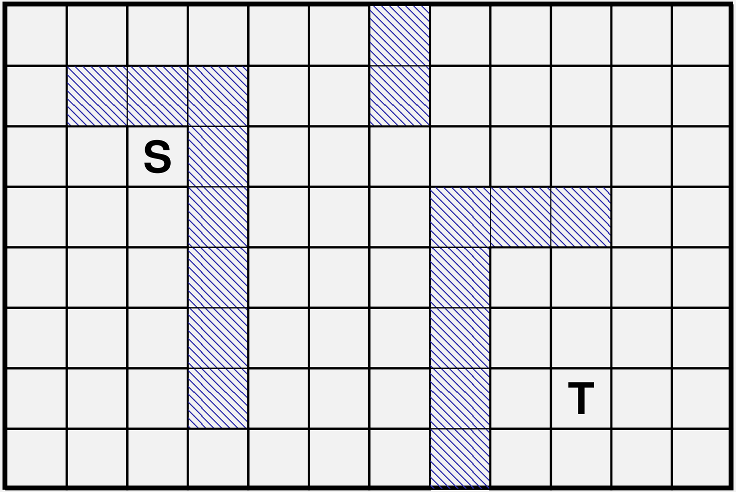
```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

// POST: Matrix 'm' was printed to stream 'to'
void print(imatrix m, std::ostream to);

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

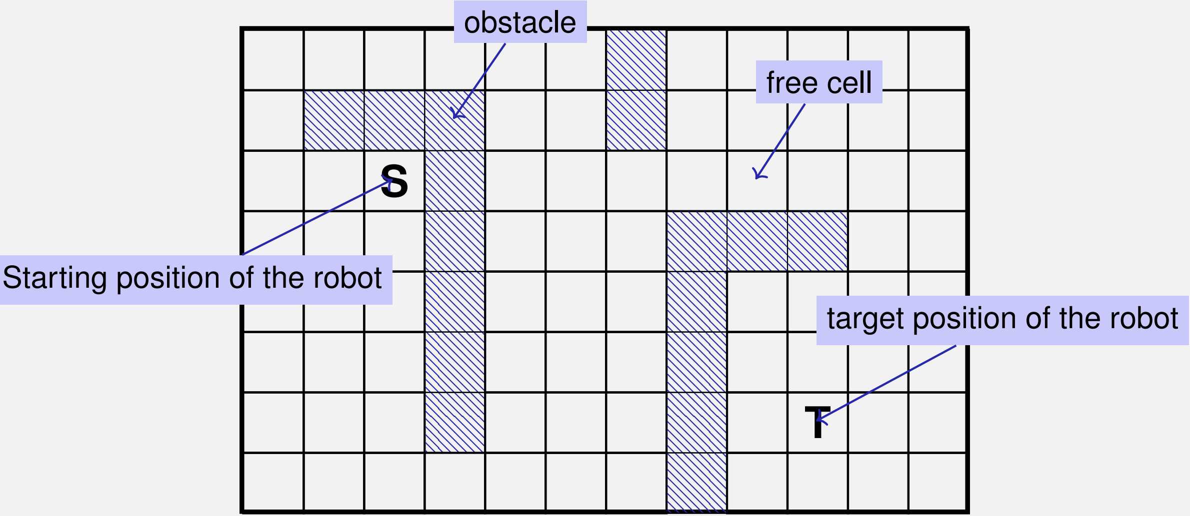
Application: Shortest Paths

Factory hall ($n \times m$ square cells)



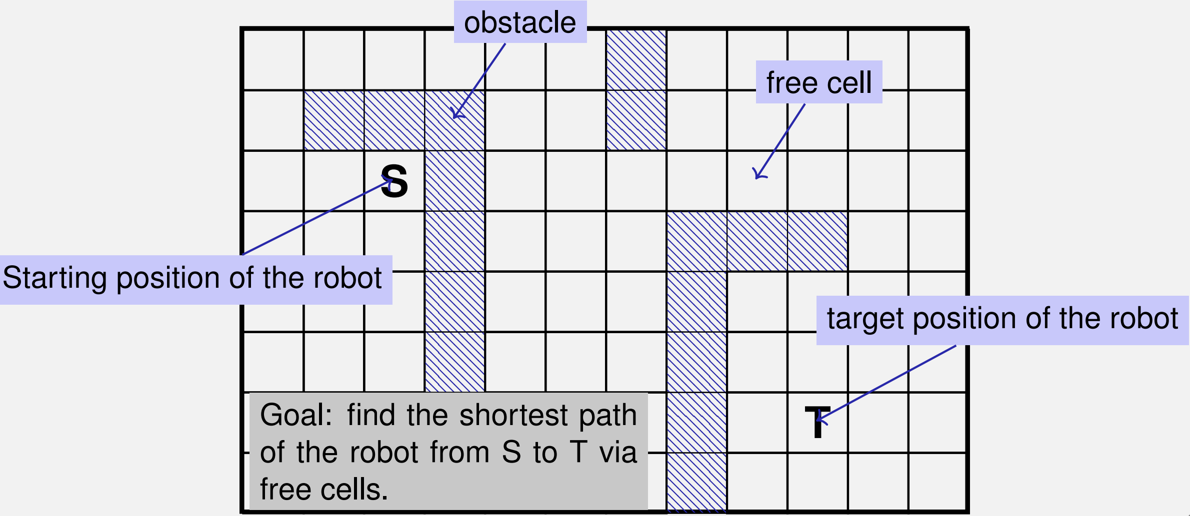
Application: Shortest Paths

Factory hall ($n \times m$ square cells)



Application: Shortest Paths

Factory hall ($n \times m$ square cells)



Goal: find the shortest path of the robot from S to T via free cells.

This problem appears to be different

Find the *lengths* of the shortest paths to *all* possible targets.

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

This problem appears to be different

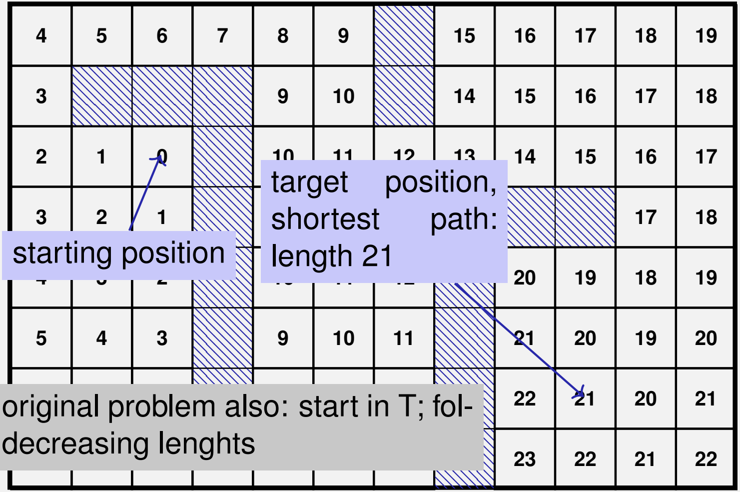
Find the *lengths* of the shortest paths to *all* possible targets.

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

This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

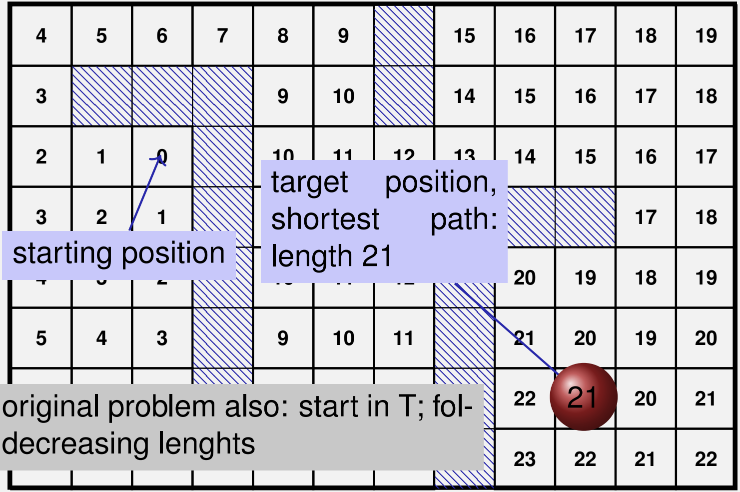
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

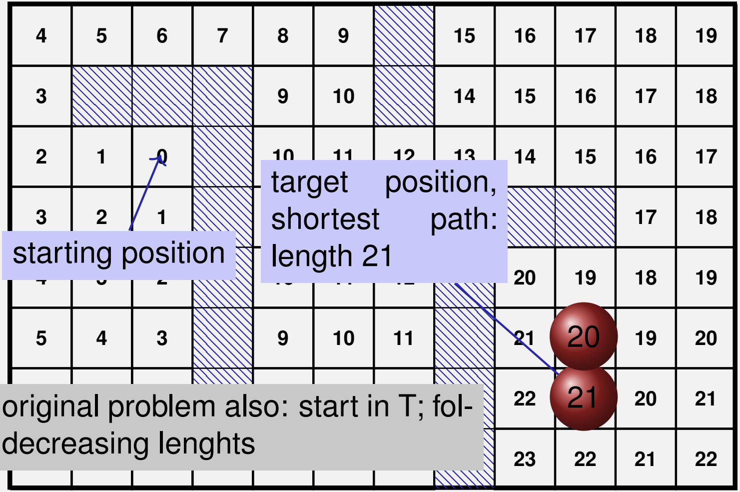
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

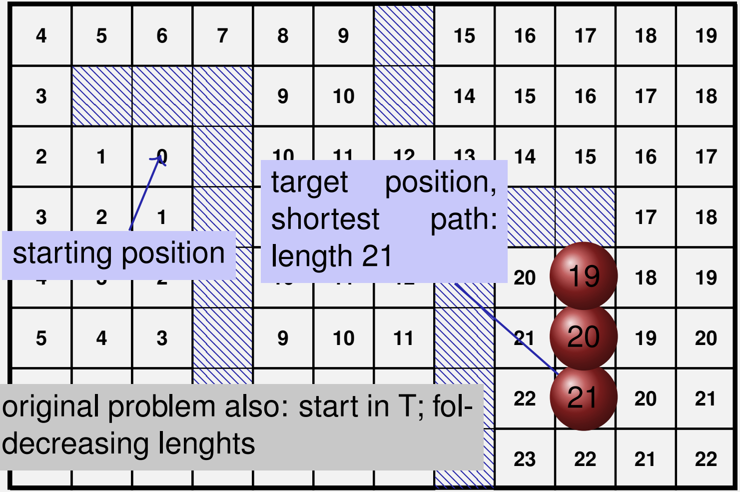
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

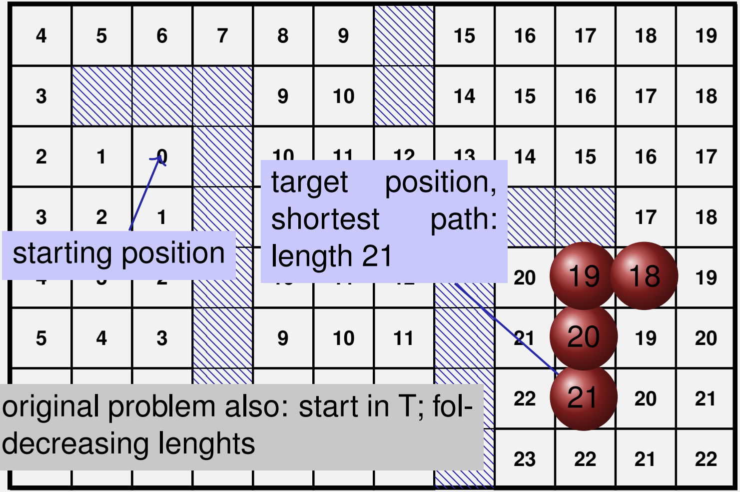
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

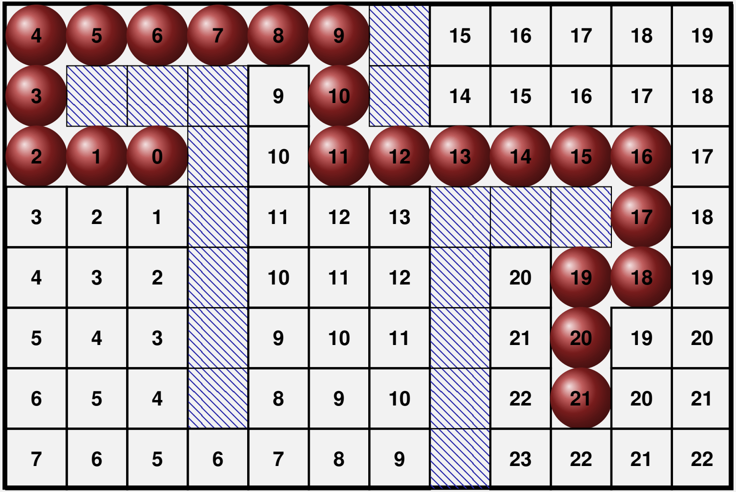
Find the *lengths* of the shortest paths to *all* possible targets.



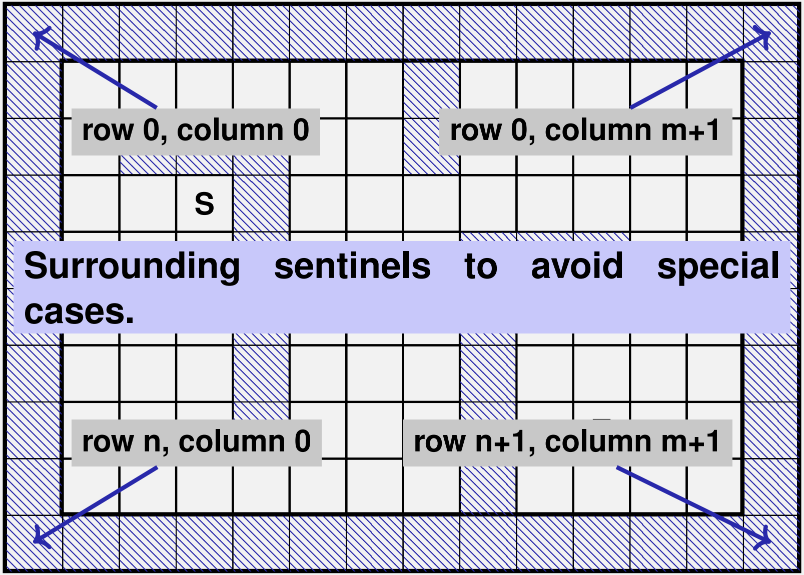
This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

Find the *lengths* of the shortest paths to *all* possible targets.

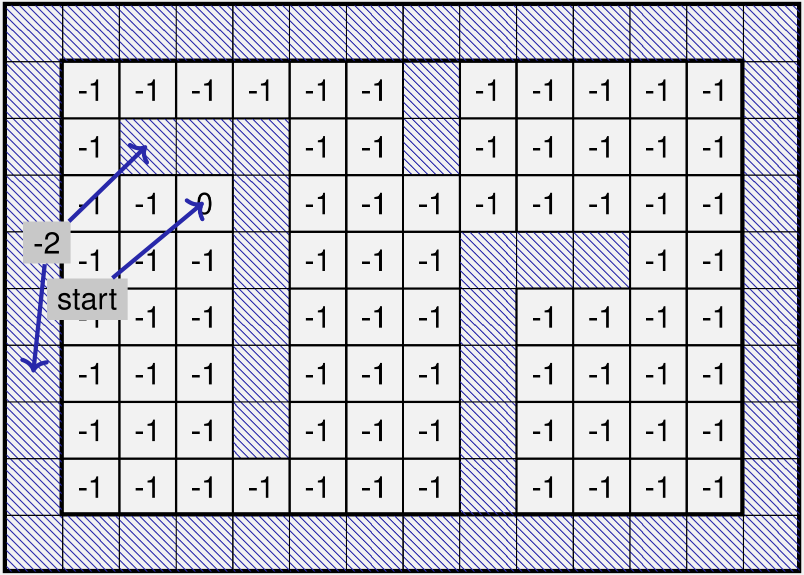


Preparation: Sentinels



Surrounding sentinels to avoid special cases.

Preparation: Initial Marking



The Shortest Path Program

```
// define a two-dimensional array of dimensions
// (n+2) x (m+2) to hold the floor
// plus extra walls around
std::vector<std::vector<int> >
    floor (n+2, std::vector<int>(m+2));


// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

The Shortest Path Program

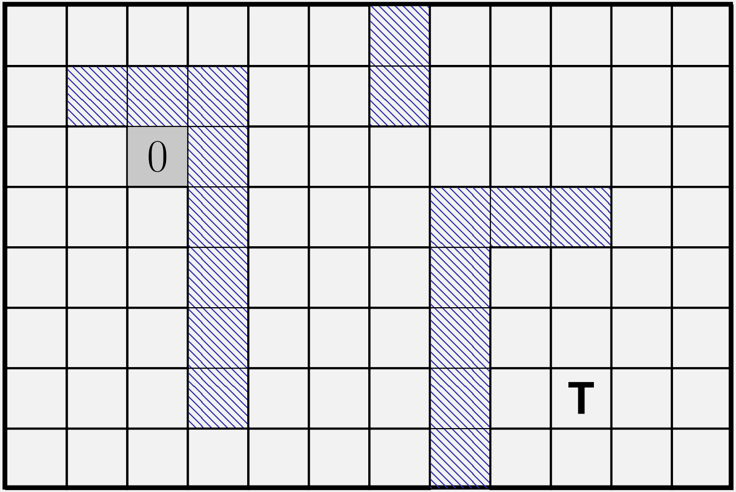
```
// define a two−dimensional array of dimensions
// (n+2) x (m+2) to hold the floor
// plus extra walls around
std::vector<std::vector<int> >
    floor (n+2, std::vector<int>(m+2));
```

Sentinel

```
// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

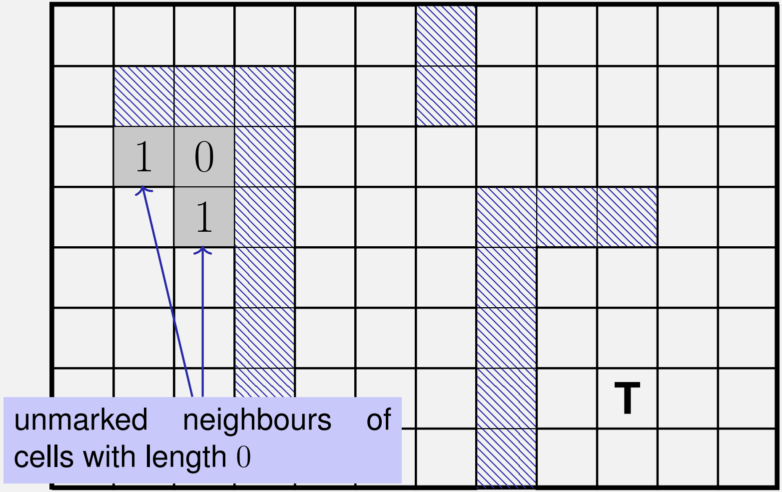
Mark all Cells with their Path Lengths

Step 0: all cells with path length 0



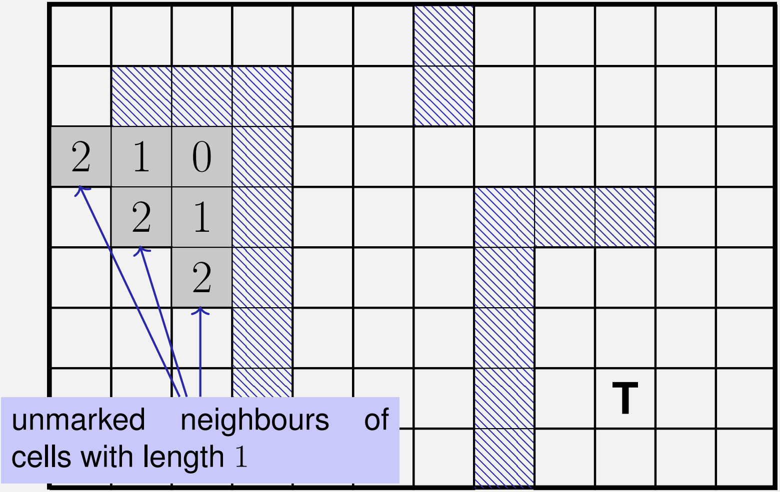
Mark all Cells with their Path Lengths

Step 1: all cells with path length 1



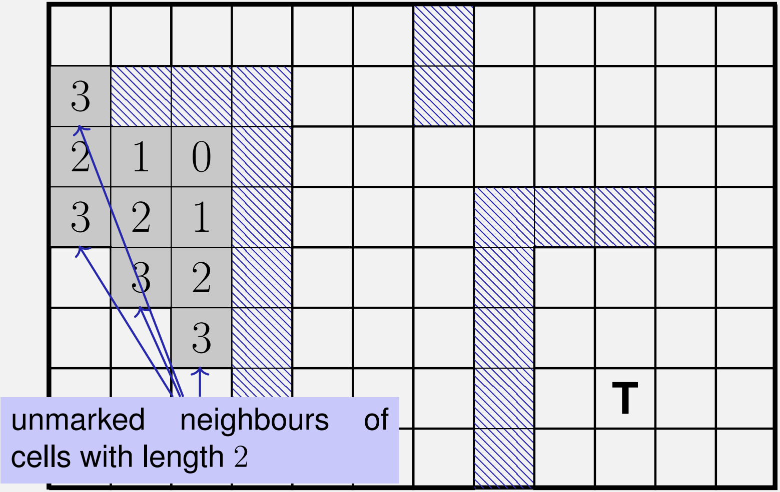
Mark all Cells with their Path Lengths

Step 2: all cells with path length 2



Mark all Cells with their Path Lengths

Step 3: all cells with path length 3



Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```

Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```

indicates if in sweep through all cells there was progress

Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```

sweep over all cells

Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```

cell already marked or obstacle

Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

a neighbour has path length $i-1$. The sentinels guarantee that there are always 4 neighbours

```
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;
}
```

Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```

no progress, all reachable cells marked; done.

The Shortest Paths Program

- Algorithm: *Breadth First Search*

The Shortest Paths Program

- Algorithm: *Breadth First Search*
- The program can become pretty slow because for each i all cells are traversed

The Shortest Paths Program

- Algorithm: *Breadth First Search*
- The program can become pretty slow because for each i all cells are traversed
- Improvement: for marking with i, traverse only the neighbours of the cells marked with $i-1$.
- Improvement: stop once the goal has been reached

Vectors as Function Arguments

- Recall the following:

```
#include <iostream>
#include <vector>

// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);

int main() {
  std::vector<std::vector<int>> m = ...;
  print(m);
}
```

Printing a Matrix: Version 1

- Recall the following:

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);
...
print(m);
```

Printing a Matrix: Version 1

- Recall the following:

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);
...
print(m);
```

- Disadvantage: When calling `print(m)` the (potentially large) matrix `m` will be copied (*call-by-value*) $\Rightarrow$ inefficient

Printing a Matrix: Version 2

- Better: Pass by reference (*call-by-reference*)

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>>& m);
...
print(m);
```

Printing a Matrix: Version 2

- Better: Pass by reference (*call-by-reference*)

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>>& m);
...
print(m);
```

- Disadvantage: `print(m)` could modify the matrix $\Rightarrow$ potentially error-prone

Printing a Matrix: Version 3

- Better: Pass by `const` reference

```
// POST: Matrix 'm' was printed to std::cout
void print(const std::vector<std::vector<int>>& m);
...
print(m);
```

Printing a Matrix: Version 3

- Better: Pass by `const` reference

```
// POST: Matrix 'm' was printed to std::cout
void print(const std::vector<std::vector<int>>& m);
...
print(m);
```

- Now: Efficient, but nevertheless not more error-prone

14. Recursion 1

Mathematical Recursion, Termination, Call Stack, Examples, Recursion vs. Iteration, n-Queen Problem, Lindenmayer Systems

Mathematical Recursion

- Many mathematical functions can be naturally defined recursively.

Mathematical Recursion

- Many mathematical functions can be naturally defined recursively.
- This means, the function appears in its own definition

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

Recursion in C++: In the same Way!

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

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

Infinite Recursion

- is as bad as an infinite loop. . .

Infinite Recursion

- is as bad as an infinite loop. . .
- . . . but even worse: it burns time and memory

Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time and memory

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

Infinite Recursion

- is as bad as an infinite loop. . .
- . . . but even worse: it burns time and memory

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

Ein Euro ist ein Euro.

Wim Duisenberg, erster Präsident der EZB

Recursive Functions: Termination

As with loops we need

- progress towards termination

Recursive Functions: Termination

As with loops we need

- progress towards termination

`fac(n):`
terminates immediately for $n \leq 1$, otherwise the function is called recusively with $<$ `n` .

Recursive Functions: Termination

As with loops we need

- progress towards termination

`fac(n)`:
terminates immediately for $n \leq 1$, otherwise the function is called recusively with $< n$.

"n is getting smaller for each call"

Recursive Functions: Evaluation

Example: `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
}
```

Call of `fac(4)`

Recursive Functions: Evaluation

Example: `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
}
```

Initialization of the formal argument

Recursive Functions: Evaluation

Example: `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
}
```

Evaluation of the return expression

Recursive Functions: Evaluation

Example: `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
}
```

recursive call with argument $n-1 == 3$

Recursive Functions: Evaluation

Example: `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
}
```

Initialization of the formal argument

Recursive Functions: Evaluation

Example: `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
}
```

Now there are two n. That of `fac(4)` and that of `fac(3)`

Initialization of the formal argument

Recursive Functions: Evaluation

Example: `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
}
```

The n of the current call is used: $n = 3$

Initialization of the formal argument

The Call Stack

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

The Call Stack

For each function call:

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

The Call Stack

For each function call:

- push value of the call argument onto the stack

$n = 4$

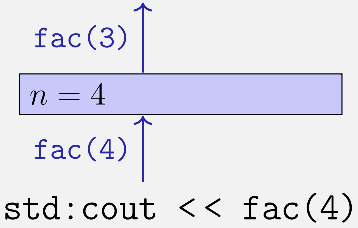
`fac(4)`

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

The Call Stack

For each function call:

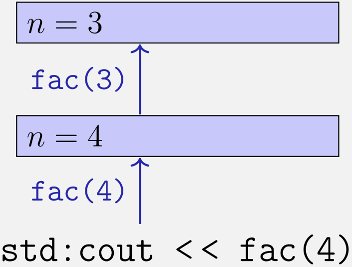
- push value of the call argument onto the stack



The Call Stack

For each function call:

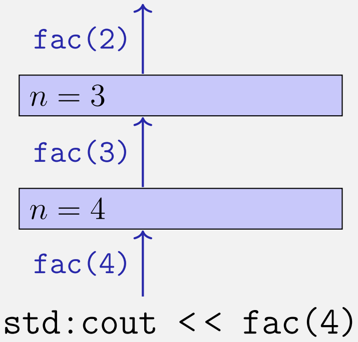
- push value of the call argument onto the stack



The Call Stack

For each function call:

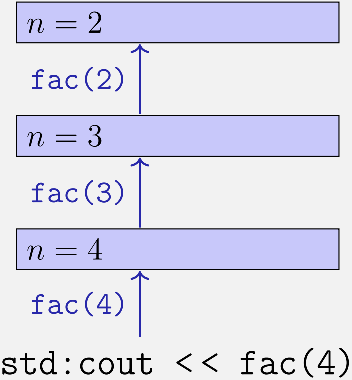
- push value of the call argument onto the stack



The Call Stack

For each function call:

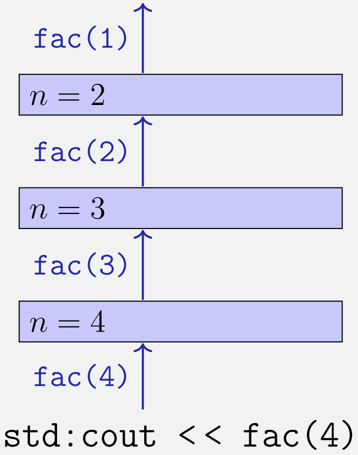
- push value of the call argument onto the stack



The Call Stack

For each function call:

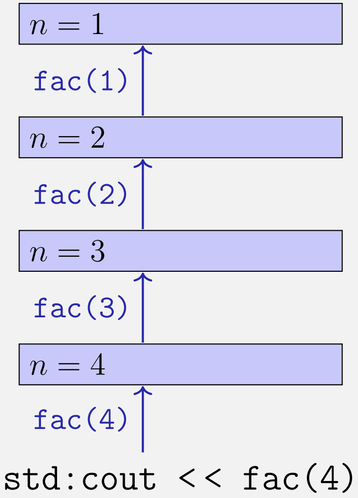
- push value of the call argument onto the stack



The Call Stack

For each function call:

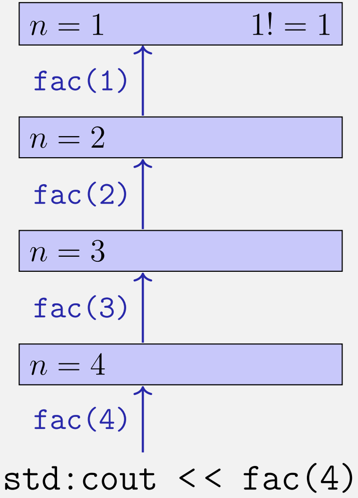
- push value of the call argument onto the stack



The Call Stack

For each function call:

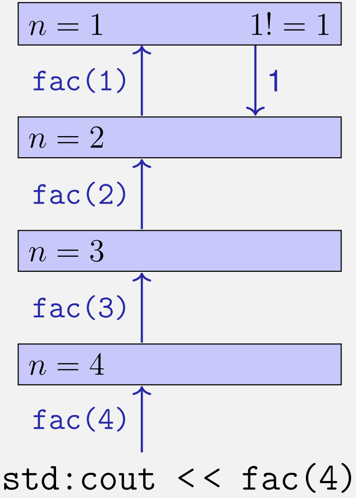
- push value of the call argument onto the stack
- always work with the top value



The Call Stack

For each function call:

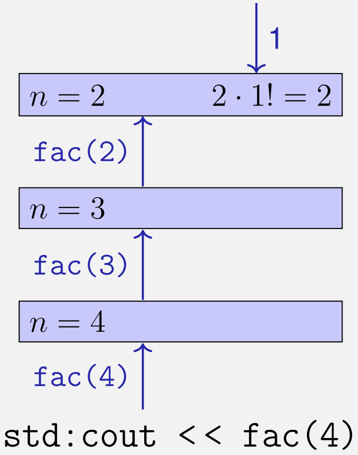
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

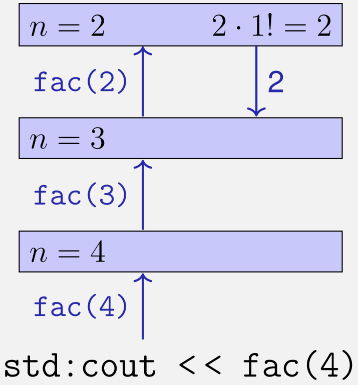
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

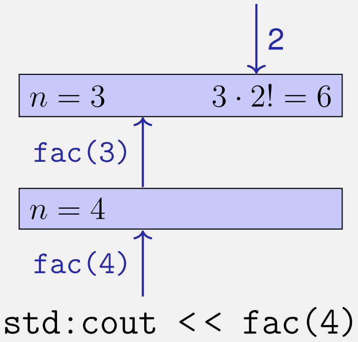
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

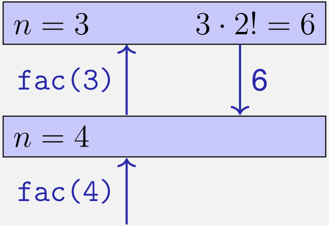
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack

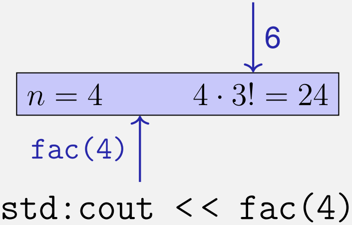


`std:cout << fac(4)`

The Call Stack

For each function call:

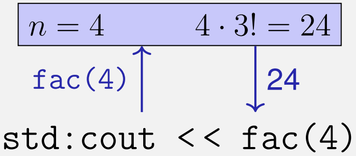
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack

$$\texttt{std:cout << } \overset{24}{\downarrow}\texttt{fac(4)}$$

Euclidean Algorithm

- finds the greatest common divisor $\gcd(a, b)$ of two natural numbers a and b

Euclidean Algorithm

- finds the greatest common divisor $\gcd(a, b)$ of two natural numbers a and b
- is based on the following mathematical recursion (proof in the lecture notes):

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Euclidean Algorithm in C++

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

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

Euclidean Algorithm in C++

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

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

Termination: $a \bmod b < b$, thus b gets smaller in each recursive call.

Fibonacci Numbers

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

Fibonacci Numbers

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

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

Fibonacci Numbers in Zurich

Fibonacci Numbers in C++

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

```
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 Numbers in $\mathrm{C{+}{+}}$

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

```
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
}
```

Correctness and termination are clear.

Fibonacci Numbers in $\mathrm{C{+}{+}}$

Laufzeit

`fib(50)` takes “forever” because it computes
F_{48} two times, F_{47} 3 times, F_{46} 5 times, F_{45} 8 times, F_{44} 13 times, F_{43} 21 times ... F_1 ca. 10^9 times (!)

```
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
}
```

Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!

Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!
- Memorize the most recent two numbers (variables `a` and `b`)!

Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!
- Memorize the most recent two numbers (variables `a` and `b`)!
- Compute the next number as a sum of `a` and `b`!

Fast Fibonacci Numbers in $\mathrm{C}{+}{+}$

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; 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

Fast Fibonacci Numbers in $\mathrm{C++}$

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; 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

Fast Fibonacci Numbers in C++

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; 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;
}
```

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

Fast Fibonacci Numbers in C++

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; 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;
}
```

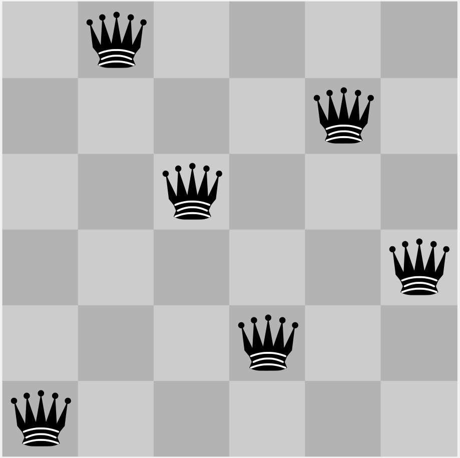
very fast, also for `fib(50)`

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

The Power of Recursion

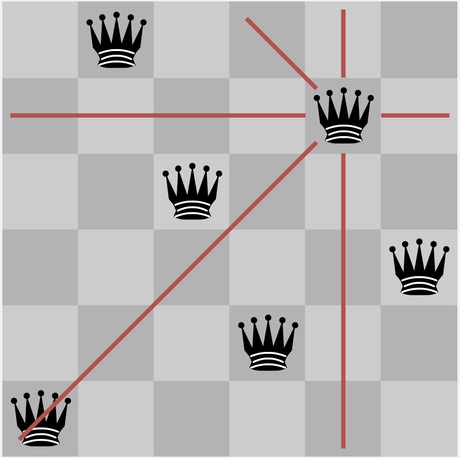
- Some problems appear to be hard to solve without recursion. With recursion they become significantly simpler.
- Examples: *The n-Queens-Problem*, The towers of Hanoi, *Sudoku-Solver*, Expression Parsers, Reversing In- or Output, Searching in Trees, Divide-And-Conquer (e.g. sorting)

The n-Queens Problem



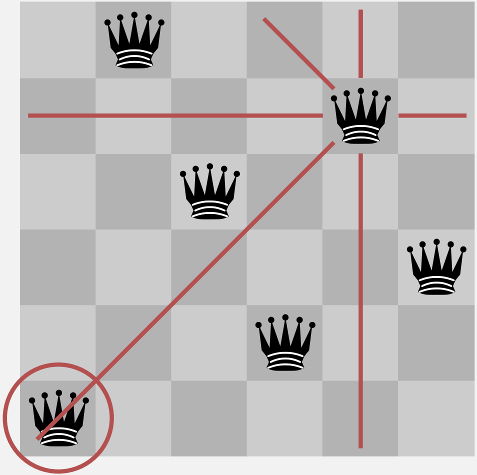
- Provided is a $n\ times n$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?

The n-Queens Problem



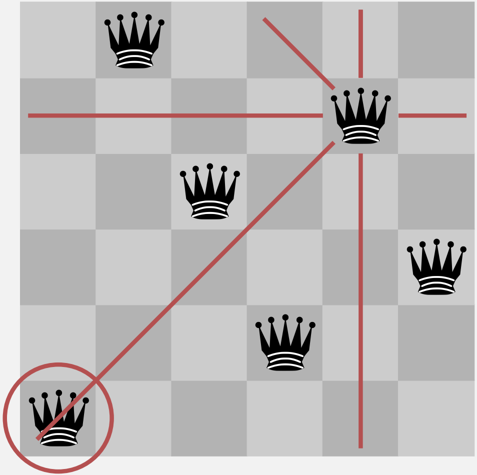
- Provided is a $n\ timesn$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?

The n-Queens Problem



- Provided is a $n \ timesn$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?

The n-Queens Problem



- Provided is a $n\ times n$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?
- If yes, how many solutions are there?

Solution?

- Try all possible placements?

Solution?

- Try all possible placements?
- $\binom{n^2}{n}$ possibilities. Too many!

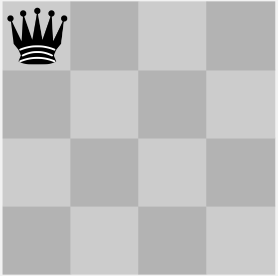
Solution?

- Try all possible placements?
- $\binom{n^2}{n}$ possibilities. Too many!
- n^n possibilities. Better – but still too many.

Solution?

- Try all possible placements?
- $\binom{n^2}{n}$ possibilities. Too many!
- n^n possibilities. Better – but still too many.
- Idea: Do not follow paths that obviously fail. (Backtracking)

Solution with Backtracking

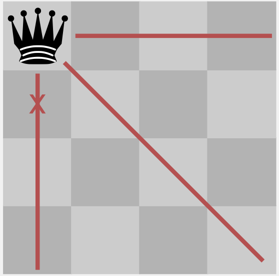


First Queen

queens

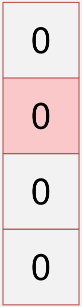
0
0
0
0

Solution with Backtracking

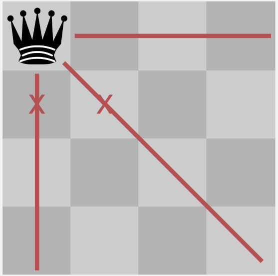


Forbidden Squares: no other queens may be here.

queens



Solution with Backtracking

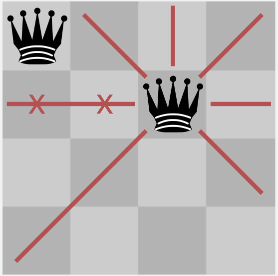


Forbidden Squares: no other queens may be here.

queens

0
1
0
0

Solution with Backtracking

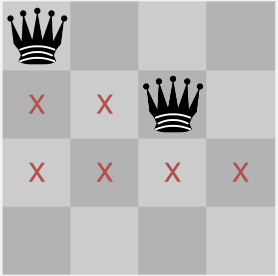


Second Queen in next row (no collision)

queens

0
2
0
0

Solution with Backtracking

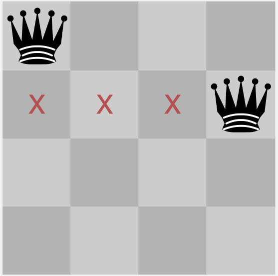


All squares in next row forbiden. Track back !

queens

0
2
4
0

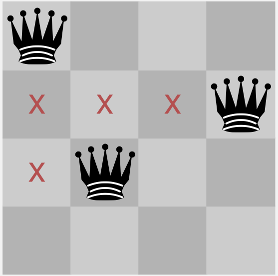
Solution with Backtracking



Move queen one step further and try again

queens
0
3
0
0

Solution with Backtracking

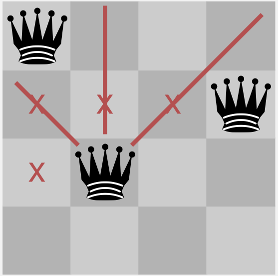


next row

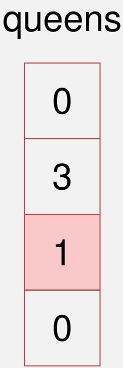
queens

0
3
1
0

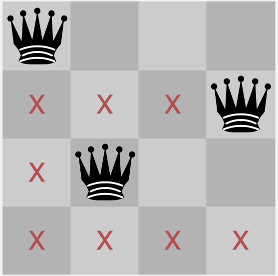
Solution with Backtracking



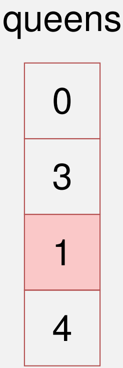
Ok (only previous queens have to be tested)



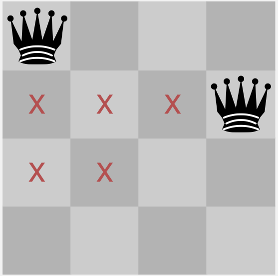
Solution with Backtracking



All squares of the next row forbidden. Track back.



Solution with Backtracking

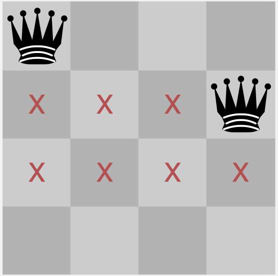


Continue in previous row.

queens

0
3
1
0

Solution with Backtracking

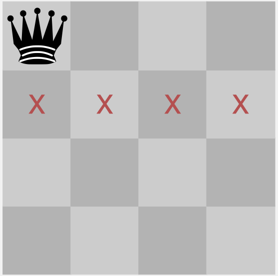


Remaining squares also forbidden. Track back!

queens

0
3
4
0

Solution with Backtracking

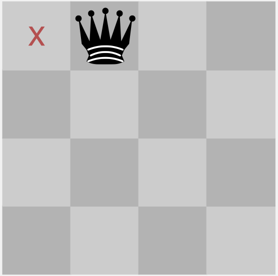


All squares of this row did not yield a solution. Track back!

queens

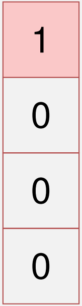
0
4
0
0

Solution with Backtracking

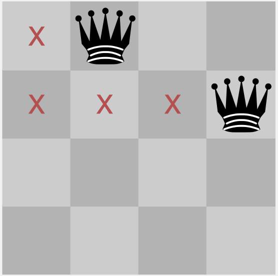


again advance queen by one square

queens



Solution with Backtracking

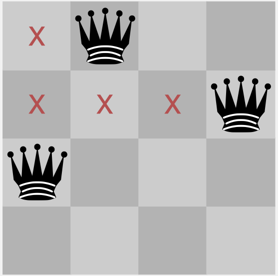


next row

queens

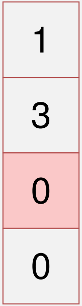
1
3
0
0

Solution with Backtracking

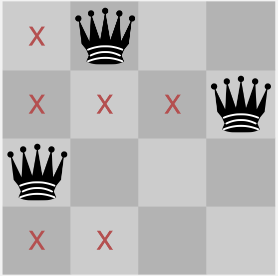


next row

queens

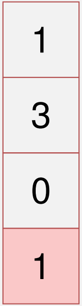


Solution with Backtracking

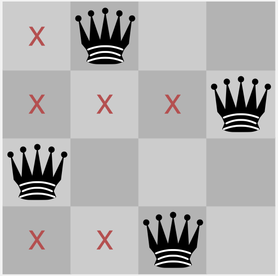


next row

queens



Solution with Backtracking

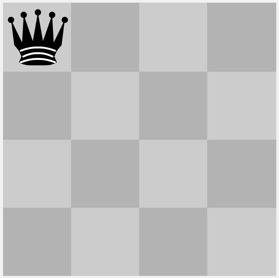


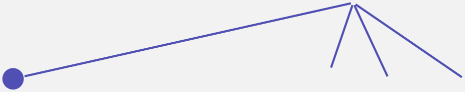
Found a solution

queens

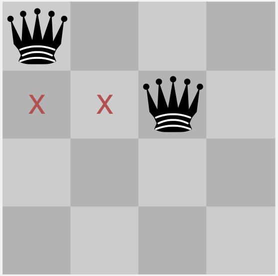
1
3
0
2

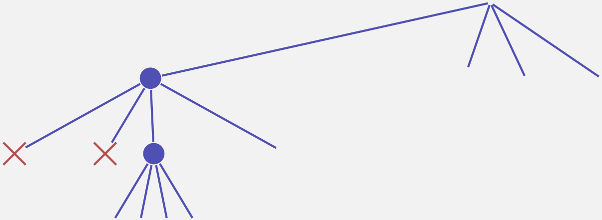
Search Strategy Visualized as a Tree



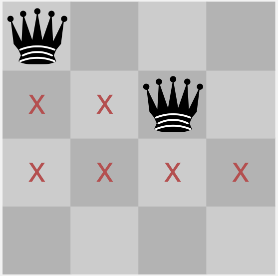


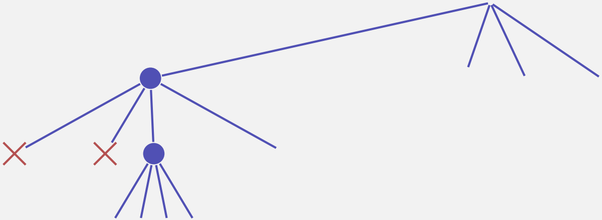
Search Strategy Visualized as a Tree



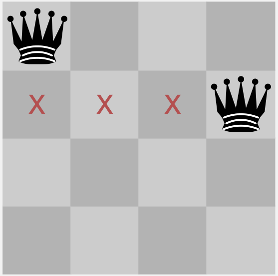


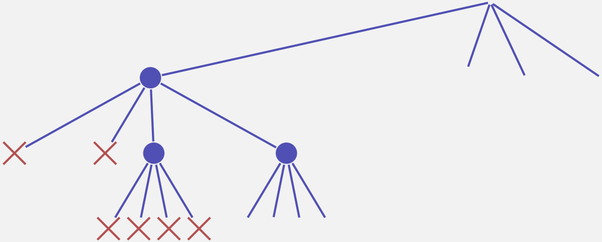
Search Strategy Visualized as a Tree



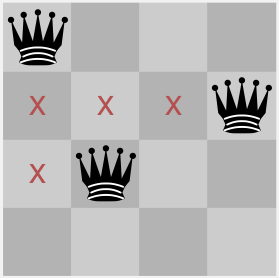


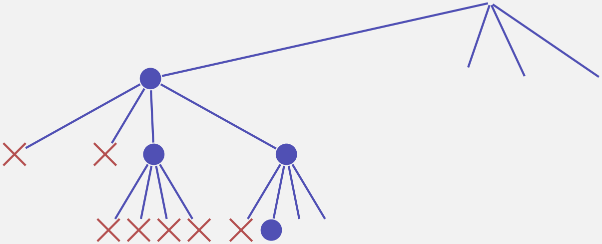
Search Strategy Visualized as a Tree



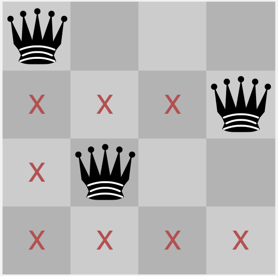


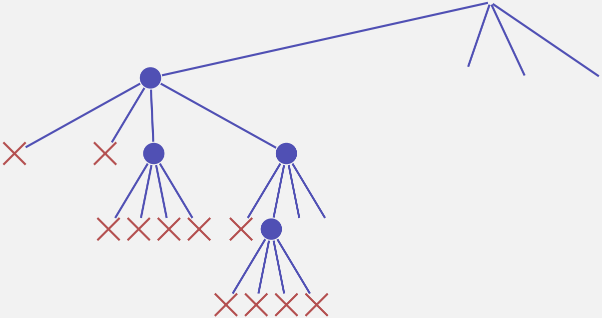
Search Strategy Visualized as a Tree



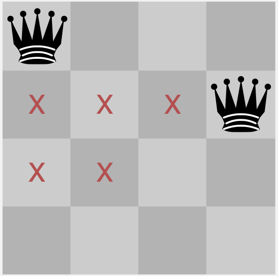


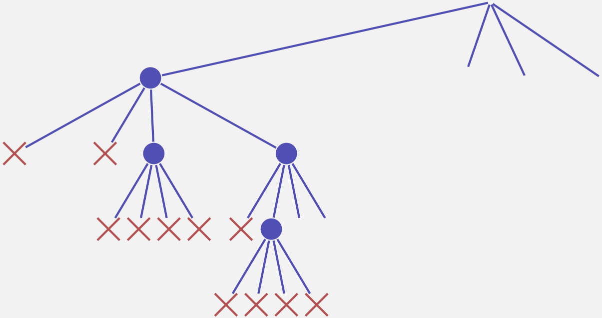
Search Strategy Visualized as a Tree



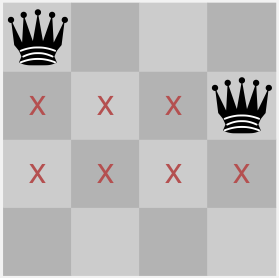


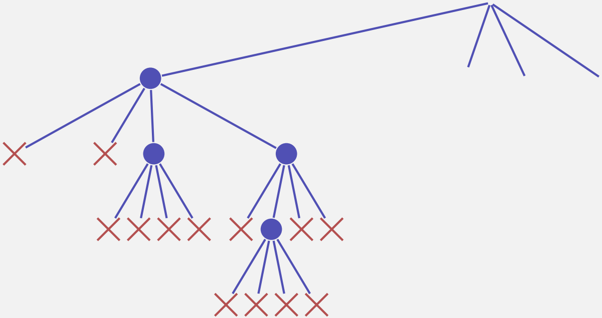
Search Strategy Visualized as a Tree



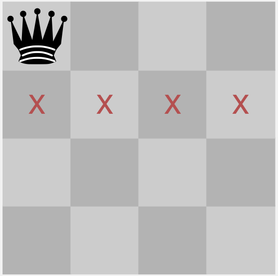


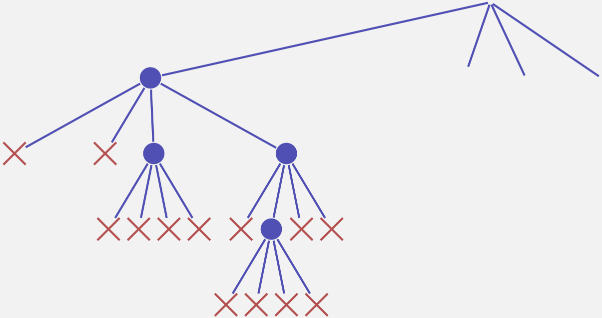
Search Strategy Visualized as a Tree



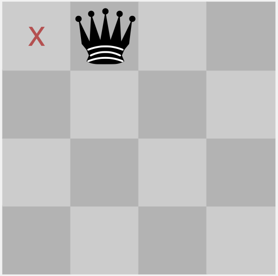


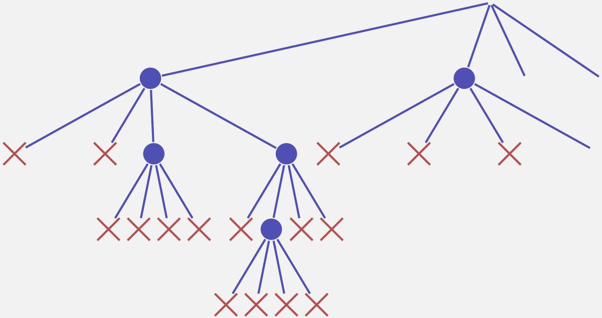
Search Strategy Visualized as a Tree



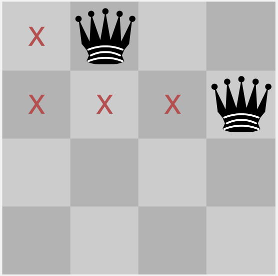


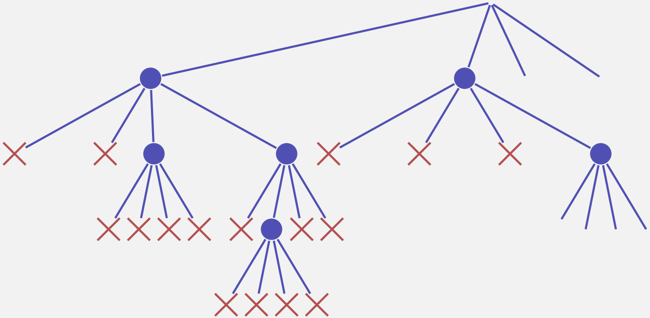
Search Strategy Visualized as a Tree



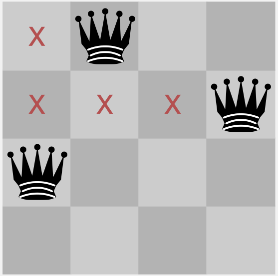


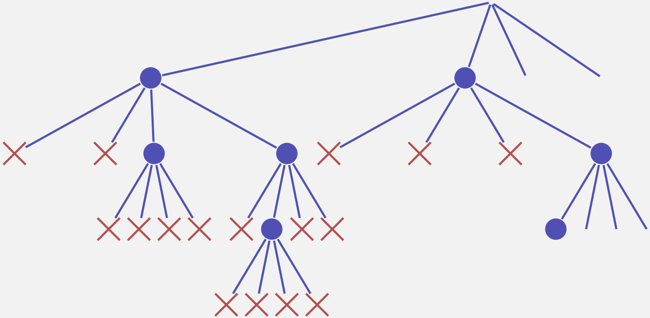
Search Strategy Visualized as a Tree



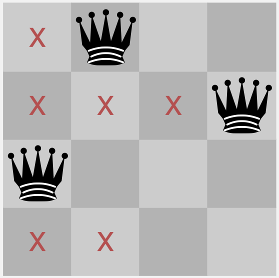


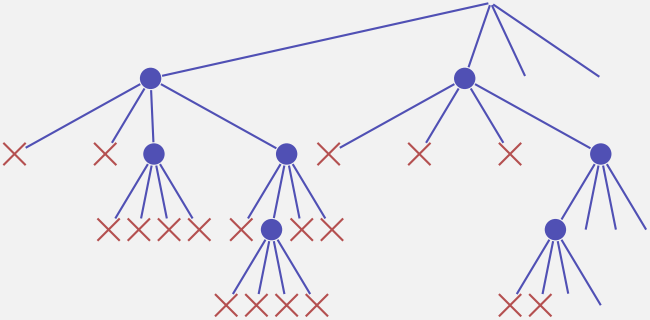
Search Strategy Visualized as a Tree



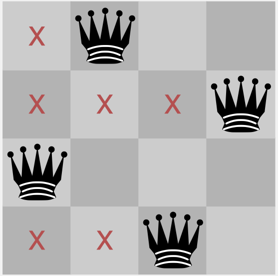


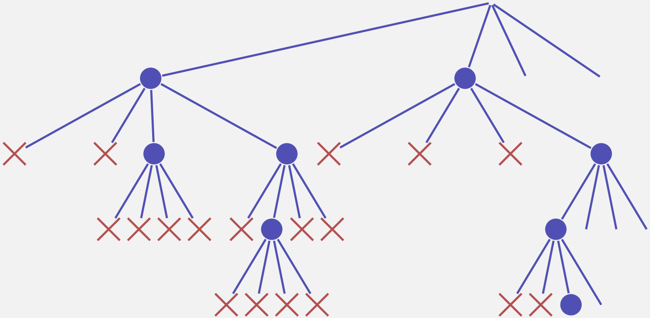
Search Strategy Visualized as a Tree





Search Strategy Visualized as a Tree





Check Queen

```cpp
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 == c0 - r || col + row == c + r)
      return false; // same column or diagonal
  }
  return true; // no shared column or diagonal
}
```

Recursion: Find a Solution

```
// 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
}
```

Recursion: Count all Solutions

```
// 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;
}
```

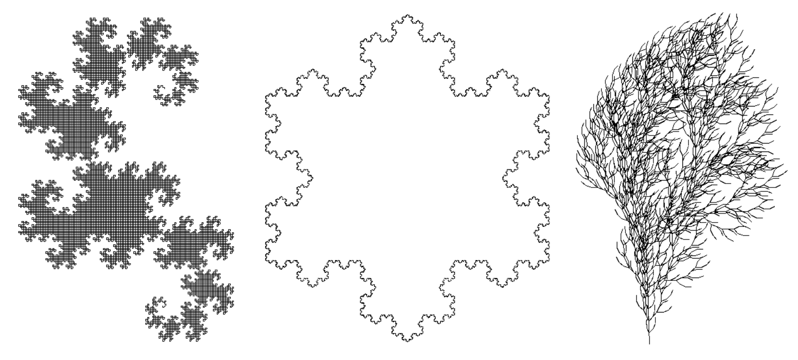
Main Program

```
// 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;
}
```

Lindenmayer-Systems (L-Systems)

Fractals from Strings and Turtles



Definition and Example

- alphabet Σ
- $\{\,\mathrm{F}\,,\,+\,,\,-\,\}$

Definition and Example

- alphabet Σ
- Σ^*: finite words over Σ

- $\{F, +, -\}$

Definition and Example

- alphabet Σ
- Σ^*: finite words over Σ
- production $P : \Sigma \to \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

Definition and Example

- alphabet Σ
- Σ^*: finite words over Σ
- production $P : \Sigma \to \Sigma^*$
- initial word $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	$\mathrm{F} + \mathrm{F} +$
$+$	$+$
$-$	$-$

- F

Definition and Example

- alphabet Σ
- Σ^*: finite words over Σ
- production $P : \Sigma \to \Sigma^*$
- initial word $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
c	$P(c)$
F	$\mathrm{F} + \mathrm{F} +$
$+$	$+$
$-$	$-$
- F

Definition

The triple $\mathcal{L} = (\Sigma, P, s_0)$ is an L-System.

The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$$

$$w_0 \quad := \quad s_0 \qquad\qquad w_0 \quad := \quad \mathrm{F}$$

The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$w_0 := s_0 \qquad w_0 := \mathrm{F}$$

$$w_1 := P(w_0) \qquad w_1 := \mathrm{F} + \mathrm{F} +$$

The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$w_0 := s_0 \qquad w_0 := \mathrm{F}$$

$$w_1 := P(w_0) \qquad w_1 := \mathrm{F} + \mathrm{F} +$$

$$w_2 := P(w_1) \qquad w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$$

Definition

$$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$$

The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$$

$$\begin{aligned} w_0 &:= s_0 \\ w_1 &:= P(w_0) \\ w_2 &:= P(w_1) \end{aligned}$$

$$\begin{aligned} w_0 &:= \mathrm{F} \\ & \quad\ \mathrm{F} + \mathrm{F} + \\ w_1 &:= \boxed{\mathrm{F}}\,\boxed{+}\,\boxed{\mathrm{F}}\,\boxed{+} \\ w_2 &:= \underbrace{\boxed{\mathrm{F} + \mathrm{F} +}}_{P(\mathrm{F})}\,\underbrace{\boxed{+}}_{P(+)}\,\underbrace{\boxed{\mathrm{F} + \mathrm{F} +}}_{P(\mathrm{F})}\,\underbrace{\boxed{+}}_{P(+)} \end{aligned}$$

Definition

$$P(c_1 c_2 \ldots c_n) := P(c_1)P(c_2)\ldots P(c_n)$$

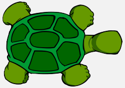
The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $\quad P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$
\begin{aligned}
w_0 &:= s_0 & \qquad w_0 &:= \mathrm{F} \\
w_1 &:= P(w_0) & \qquad w_1 &:= \mathrm{F} + \mathrm{F} + \\
w_2 &:= P(w_1) & \qquad w_2 &:= \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + + \\
&\vdots & &\vdots
\end{aligned}
$$

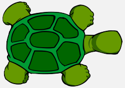
Turtle Graphics

Turtle with position and direction



Turtle Graphics

Turtle with position and direction

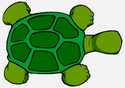


Turtle understands 3 commands:

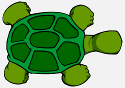
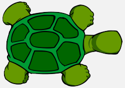
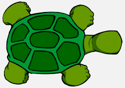
F: move one step forwards	$+$: rotate by 90 degrees	$-$: rotate by -90 degrees

Turtle Graphics

Turtle with position and direction

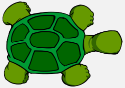


Turtle understands 3 commands:

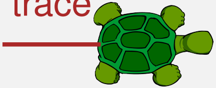
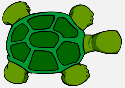
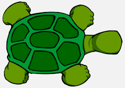
F : move one step forwards	$+$: rotate by 90 degrees	$-$: rotate by -90 degrees
		

Turtle Graphics

Turtle with position and direction

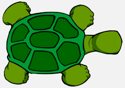


Turtle understands 3 commands:

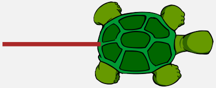
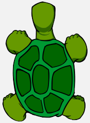
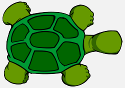
F : move one step forwards $\checkmark$	$+$: rotate by 90 degrees	$-$: rotate by -90 degrees
		

Turtle Graphics

Turtle with position and direction

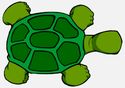


Turtle understands 3 commands:

F : move one step forwards $\checkmark$	$+$: rotate by 90 degrees $\checkmark$	$-$: rotate by -90 degrees
		

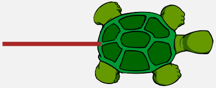
Turtle Graphics

Turtle with position and direction

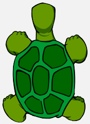


Turtle understands 3 commands:

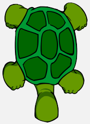
F : move one step forwards ✓



$+$: rotate by 90 degrees ✓

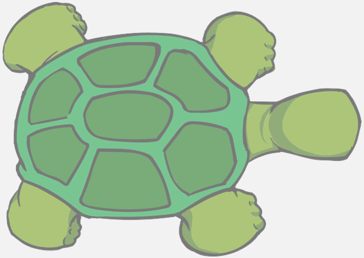


$-$: rotate by -90 degrees ✓



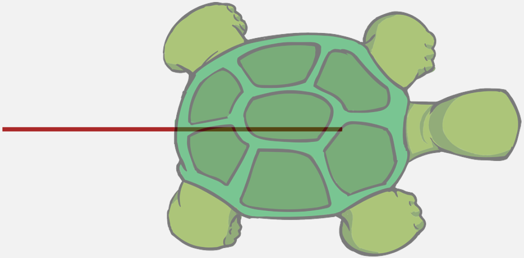
Draw Words!

$$w_1 = \mathrm{F} + \mathrm{F} +$$



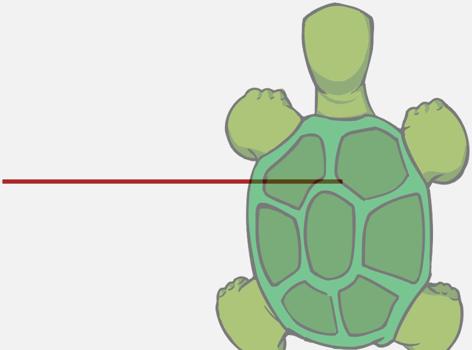
Draw Words!

$w_1 = \textcolor{red}{\mathrm{F}} + \mathrm{F} +$



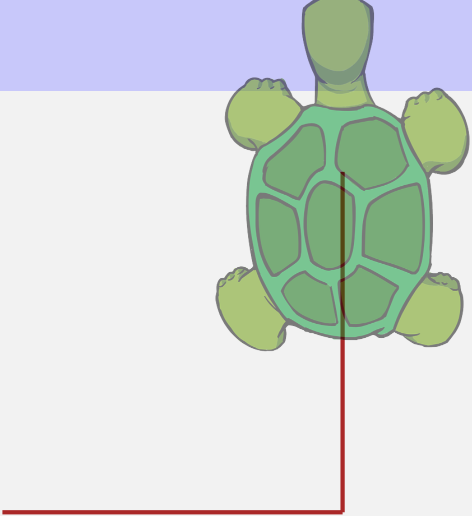
Draw Words!

$$w_1 = \mathrm{F} + \mathrm{F} +$$

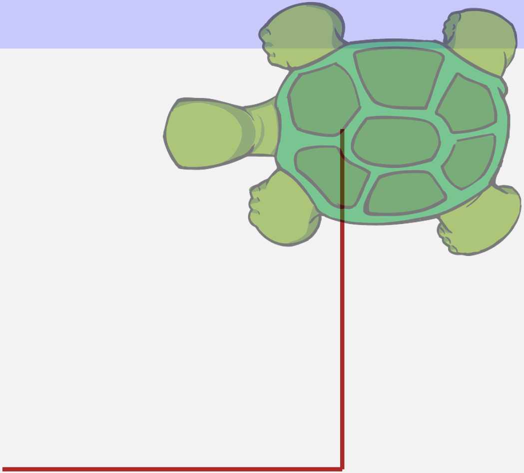


Draw Words!

$$w_1 = \mathrm{F} + {\color{red}\mathrm{F}} +$$



Draw Words!



$$w_1 = \mathrm{F} + \mathrm{F}\, {\color{red}+}$$

Draw Words!

$$w_1 = \mathrm{F} + \mathrm{F} + \checkmark$$

word $w_0 \in \Sigma^*$:

```
int main () {
  std::cout << "Maximal Recursion Depth =? ";
  unsigned int n;
  std::cin >> n;

  std::string w = "F"; // w_0
  produce(w,n);

  return 0;
}
```

`lindenmayer:` Main Program

word $w_0 \in \Sigma^*$:

```
int main () {
  std::cout << "Maximal Recursion Depth =? ";
  unsigned int n;
  std::cin >> n;

  std::string w = "F"; // w_0
  produce(w,n);

  return 0;
}
```

$w = w_0 = \mathrm{F}$

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
  if (depth > 0){
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth-1);
  } else {
    draw_word(word);
  }
}
```

lindenmayer: production

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
  if (depth > 0){
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth-1);
  } else {
    draw_word(word);
  }
}
```

$w = w_i \rightarrow w = w_{i+1}$

lindenmayer: production

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
  if (depth > 0){
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth-1);
  } else {
    draw_word(word);
  }
}
```

lindenmayer: production

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
  if (depth > 0){
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth-1);
  } else {
    draw_word(word);
  }
}
```

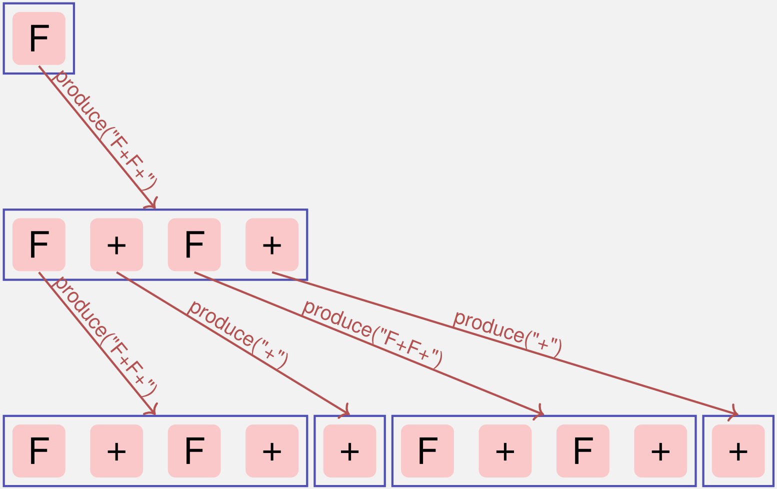
draw $w = w_n$!

```
// POST: returns the production of c
std::string replace (const char c)
{
  switch (c) {
  case 'F':
    return "F+F+";
  default:
    return std::string (1, c); // trivial production c -> c
  }
}
```

lindenmayer: draw

```
// POST: draws the turtle graphic interpretation of word
void draw_word (const std::string& word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward(); // move one step forward
      break;
    case '+':
      turtle::left(90); // turn counterclockwise by 90 degrees
      break;
    case '-':
      turtle::right(90); // turn clockwise by 90 degrees
    }
}
```

The Recursion



L-Systeme: Erweiterungen

- arbitrary symbols without graphical interpetation
- arbitrary angles (snowflake)
- saving and restoring the state of the turtle $\rightarrow$ plants (bush)