14. Characters and Texts II

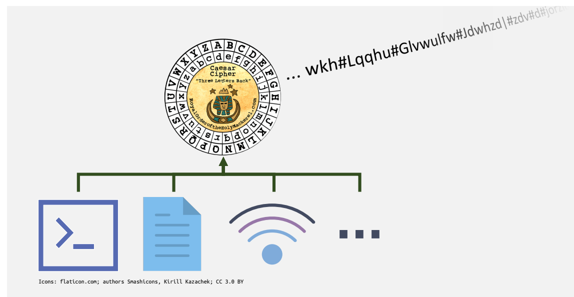
Caesar Code with Streams, Text as Strings, String Operations

Caesar-Code: Generalisation

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

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

- Currently only from **std::cin** to **std::cout**

- Better: from arbitrary character source (console, file, ...) to arbitrary character sink (console, ...)



Interlude: Abstract vs. Concrete Types

`DestroyBox`

$\left(\begin{array}{c}\text{abstract,}\\ \text{generic}\end{array}\right)$

Interlude: Abstract vs. Concrete Types

`DestroyBox`

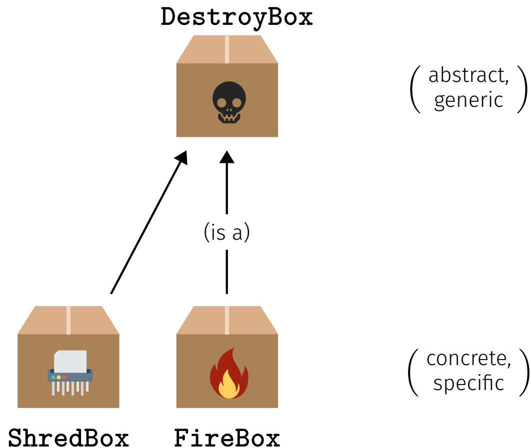
(abstract, generic)

(is a)

(concrete, specific)

`FireBox`

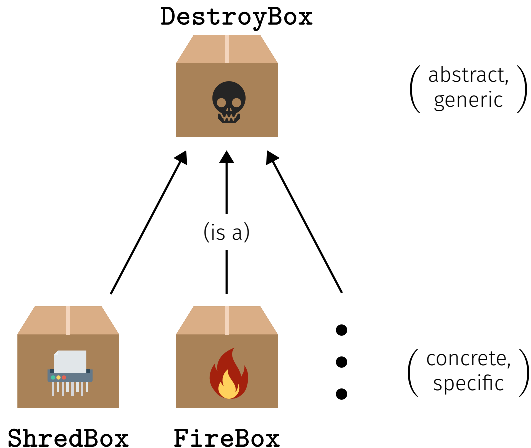
Interlude: Abstract vs. Concrete Types



(abstract, generic)

(concrete, specific)

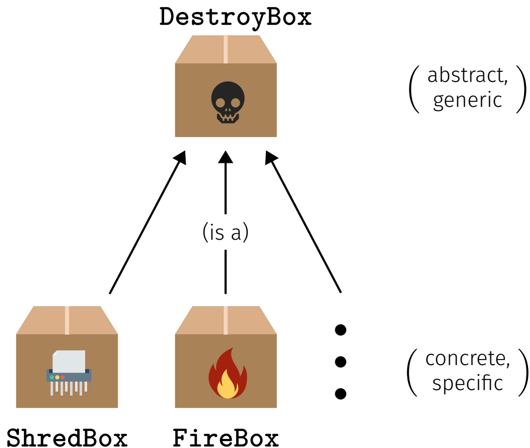
Interlude: Abstract vs. Concrete Types



(abstract, generic)

(concrete, specific)

Interlude: Abstract vs. Concrete Types

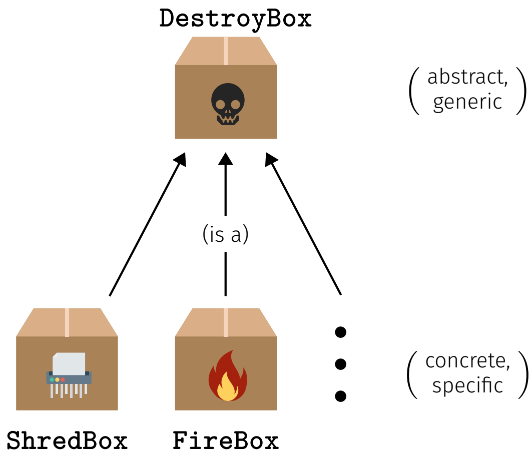


(abstract, generic)

(concrete, specific)

```
void move_house(DestroyBox& db) {
  // any destroy box will do
  db.dispose(old_ikea_couch);
  db.dispose(cheap_wine);
  ...
}
```

Interlude: Abstract vs. Concrete Types



(abstract, generic)

```
void move_house(DestroyBox& db) {
  // any destroy box will do
  db.dispose(old_ikea_couch);
  db.dispose(cheap_wine);
  ...
}
```

(concrete, specific)

```
FireBox fb(5000°C);
move_house(fb);
```

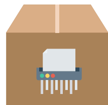
```
ShredBox sb;
move_house(sb);
```

Abstract and Concrete Character Streams

`DestroyBox`

(abstract, generic)

(is a)



`ShredBox` `FireBox`

(concrete, specific)

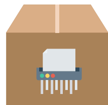
`std::ostream`

Abstract and Concrete Character Streams

`DestroyBox`

(abstract, generic)

(is a)



`ShredBox`

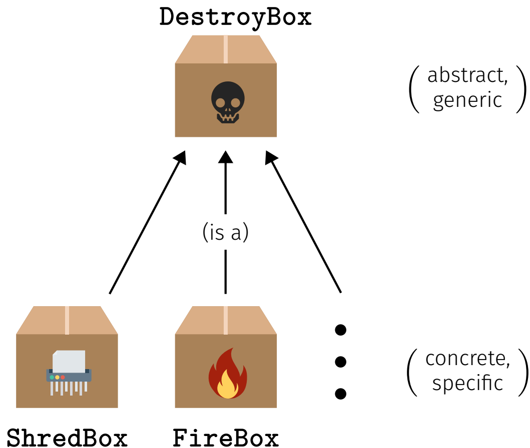
`FireBox`

(concrete, specific)

`std::ostream`

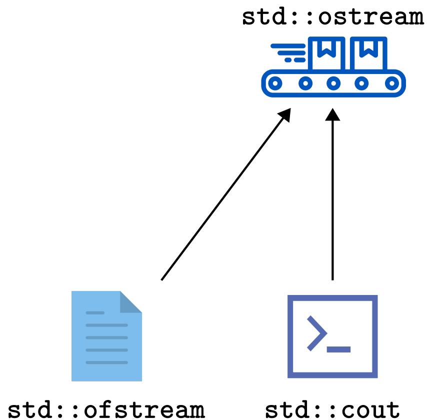
`std::cout`

Abstract and Concrete Character Streams

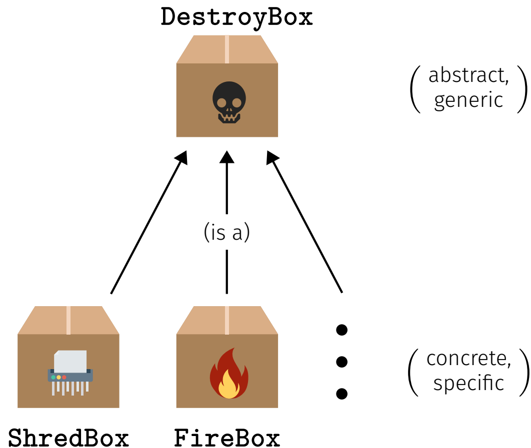


(abstract, generic)

(concrete, specific)

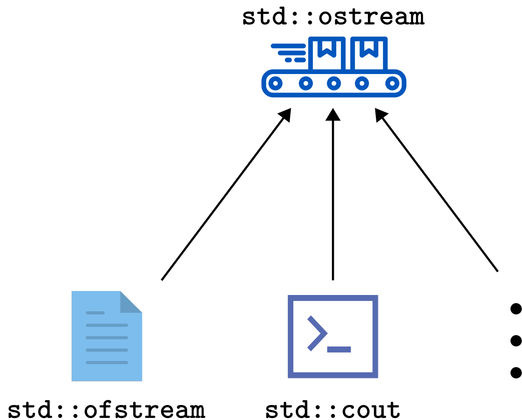


Abstract and Concrete Character Streams



(abstract, generic)

(concrete, specific)



Caesar-Code: Generalisation

```
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` is an *abstract input/output stream* of `char`s

Caesar-Code: Generalisation

```
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` is an *abstract input/output stream* of `char`s
- Function is called with *concrete* streams, e.g.:
  - Console: `std::cin`/`cout`
  - Files: `std::ifstream`/ `ofstream`

Caesar-Code: Generalisation, Example 1

```
#include <iostream>
...

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

Calling the generalised `caesar` function: from `std::cin` to `std::cout`

Caesar-Code: Generalisation, Example 2

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

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

caesar(std::cin, to, s);
```

Calling the generalised **`caesar`** function: from **`std::cin`** to file

Caesar-Code: Generalisation, Example 3

```
#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);
```

Calling the generalised `caesar` function: from file to file

Streams: Final Words

Note: You only need to be able to *use* streams

- *User knowledge*, on the level of the previous slides, suffices for exercises and exam
- I.e. you do not need to know how streams work internally
- At the end of this course, you'll hear how you can define *abstract*, and corresponding *concrete*, types yourself

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`

15. Vectors II

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

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

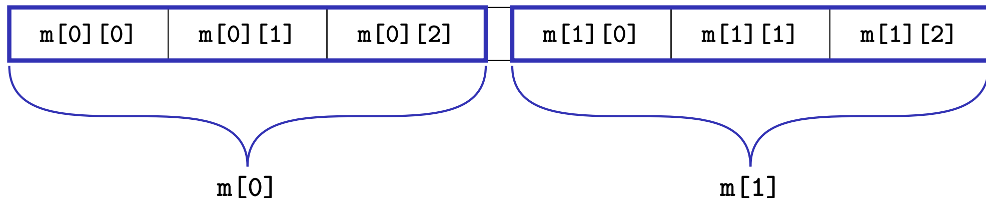
In memory: flat

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

m[0] (covers m[0][0], m[0][1], m[0][2]) — m[1] (covers m[1][0], m[1][1], m[1][2])

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

Using initialisation lists:

```
// 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

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

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));
```

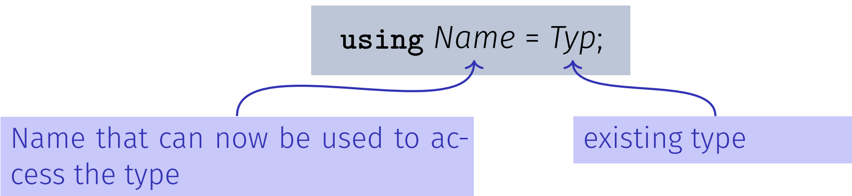
(Many further ways of initialising a vector exist)

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:



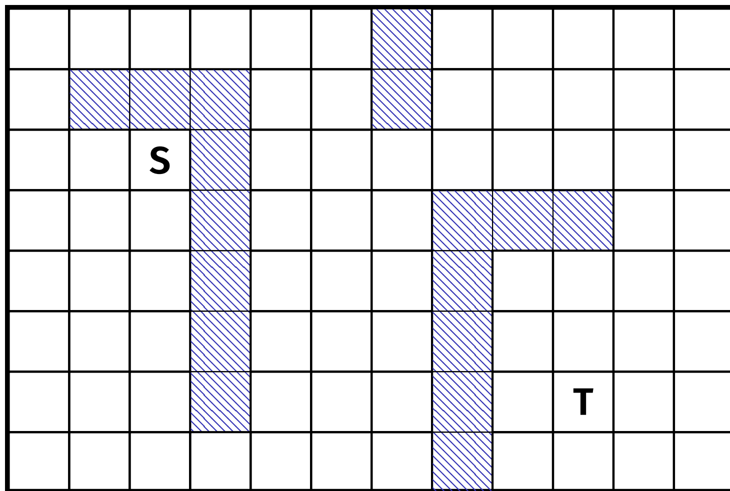
Type Aliases: Example

```
#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);
}
```

Recall: `const` reference for enfficiency (no copy) and safety (immutable)

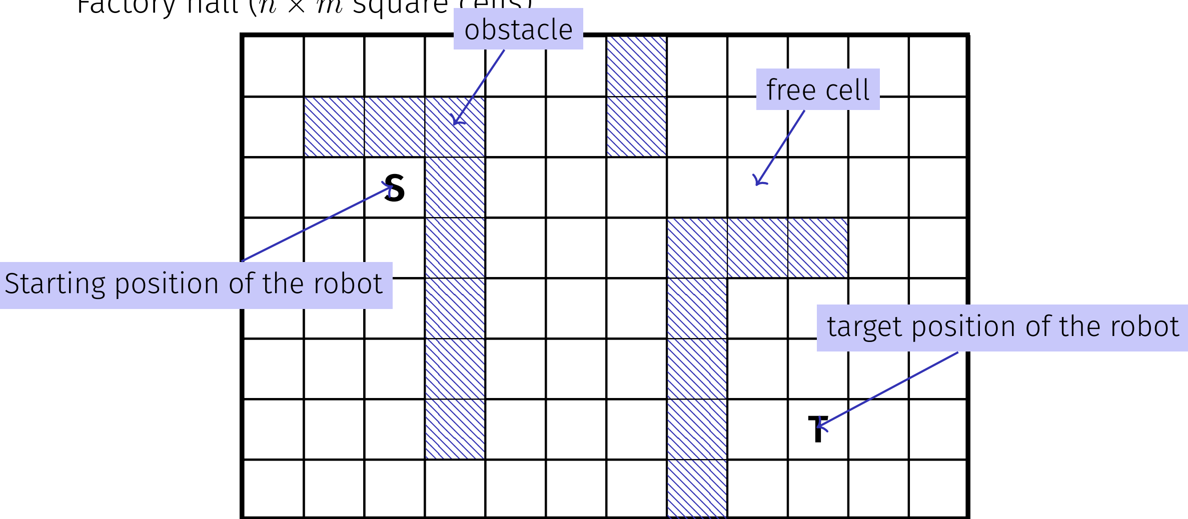
Application: Shortest Paths

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



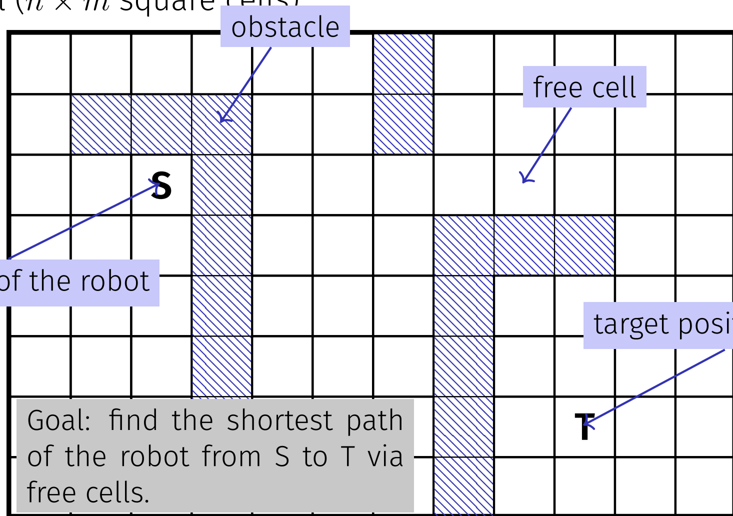
Application: Shortest Paths

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



Application: Shortest Paths

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



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

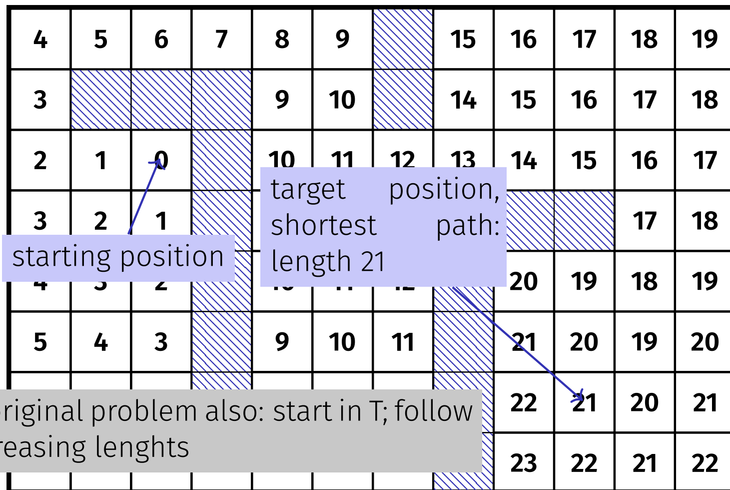
Find the *lengths* of the shortest paths to *all* possible targets.

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

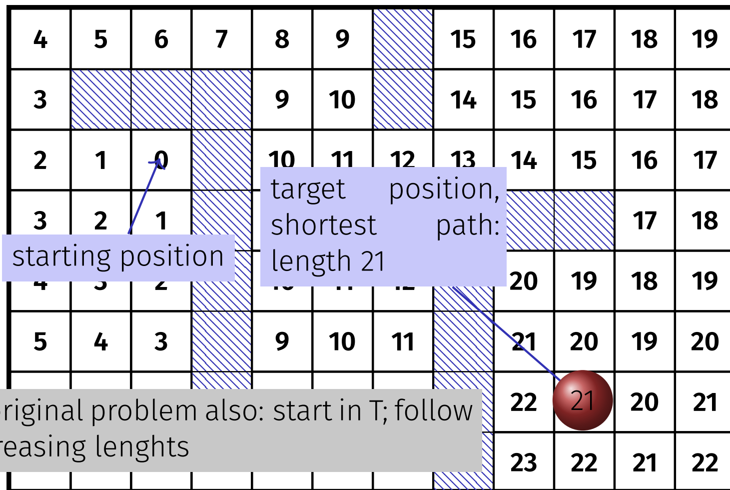
Find the *lengths* of the shortest paths to *all* possible targets.



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

This problem appears to be different

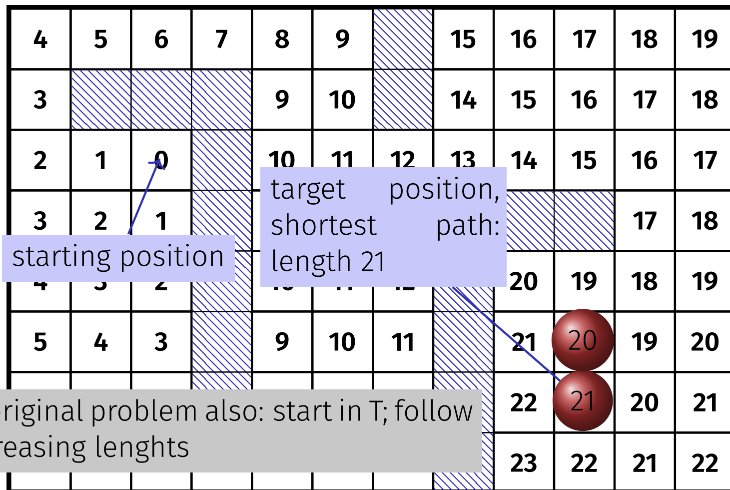
Find the *lengths* of the shortest paths to *all* possible targets.



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

This problem appears to be different

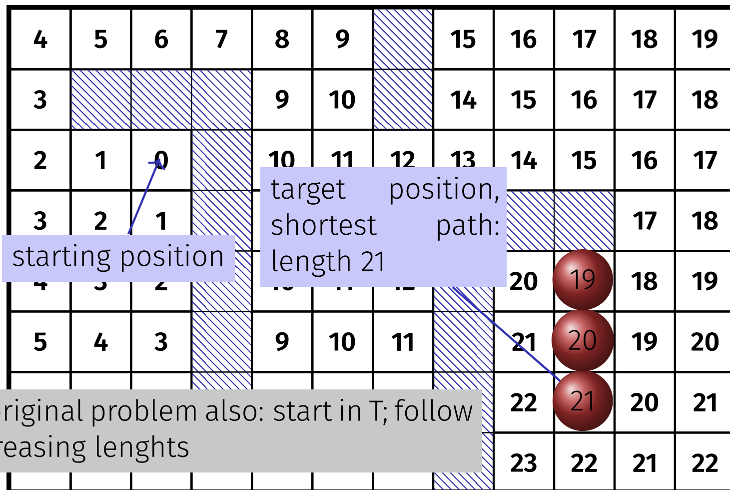
Find the *lengths* of the shortest paths to *all* possible targets.



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

This problem appears to be different

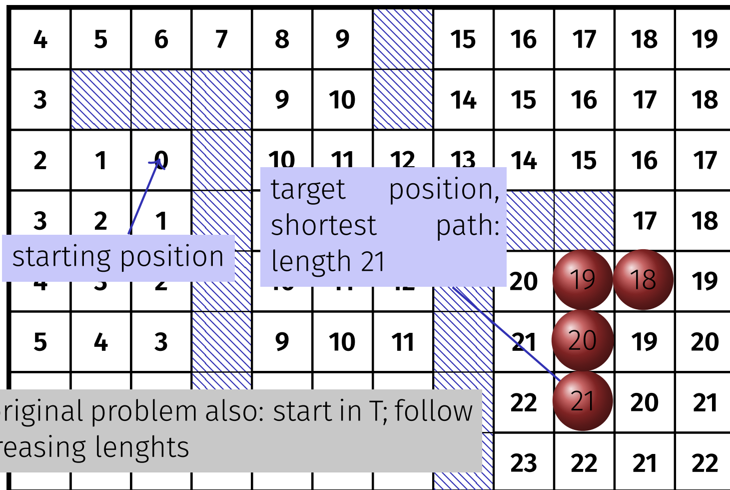
Find the *lengths* of the shortest paths to *all* possible targets.



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

This problem appears to be different

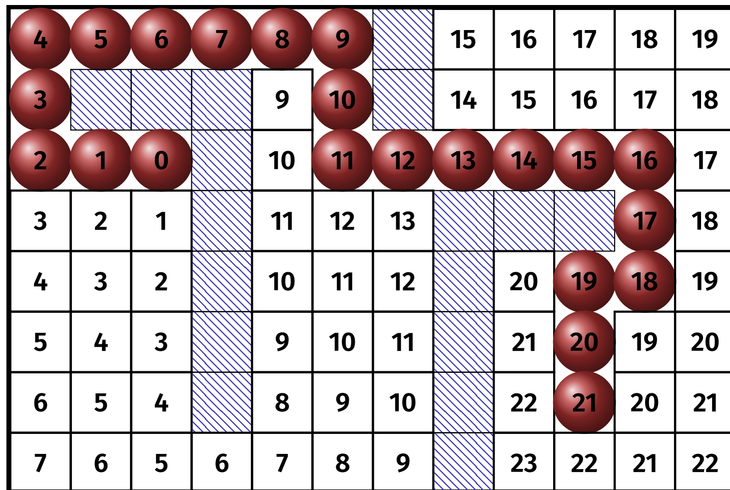
Find the *lengths* of the shortest paths to *all* possible targets.



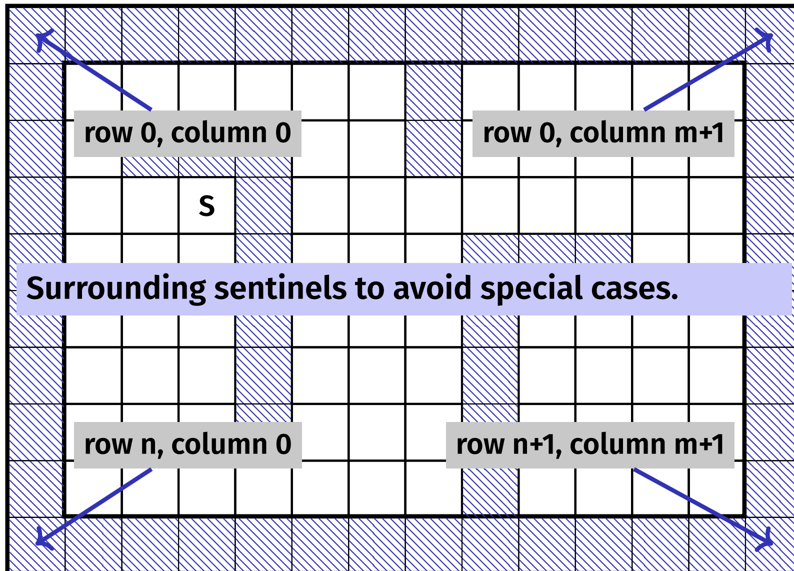
This solves the original problem also: start in T; follow a path with decreasing lenghts

This problem appears to be different

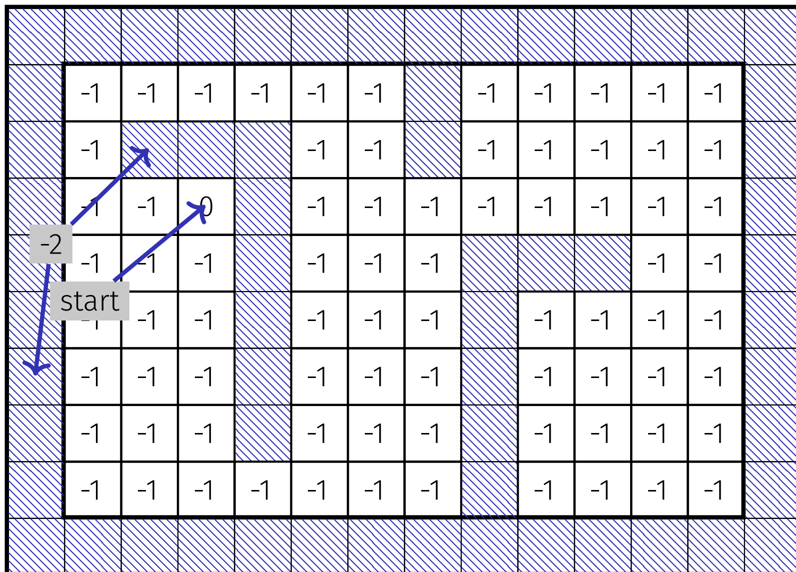
Find the *lengths* of the shortest paths to *all* possible targets.



Preparation: Sentinels

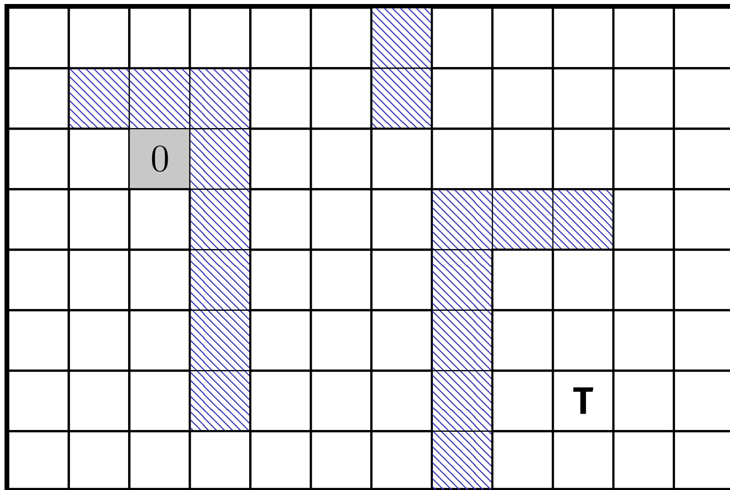


Preparation: Initial Marking



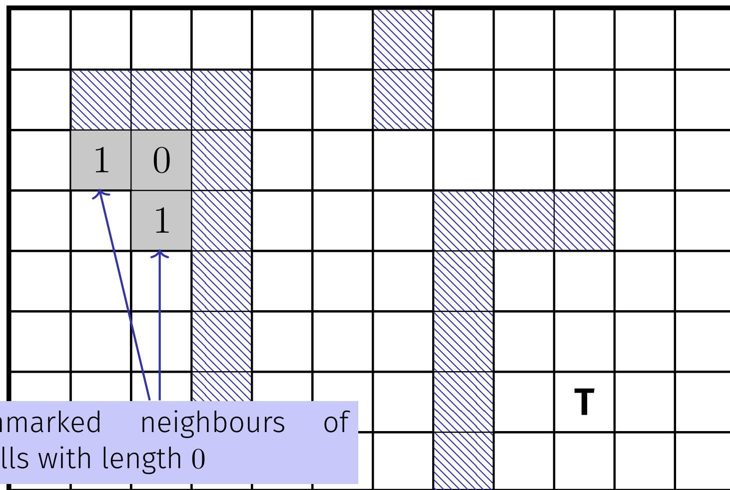
Mark all Cells with their Path Lengths

Step 0: all cells with path length 0



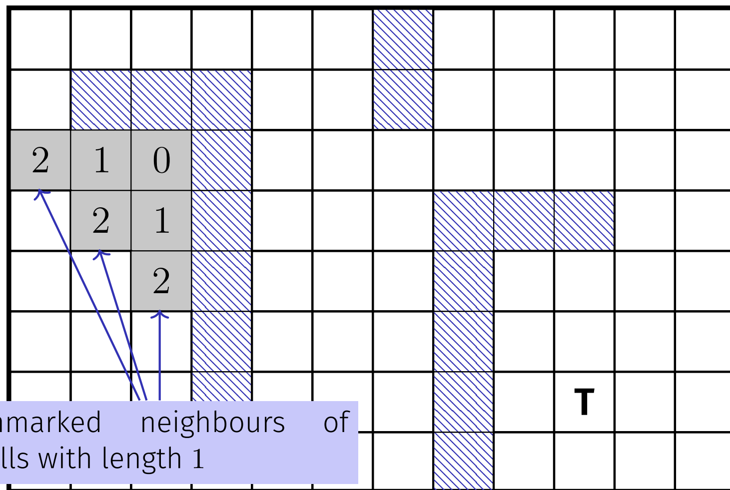
Mark all Cells with their Path Lengths

Step 1: all cells with path length 1



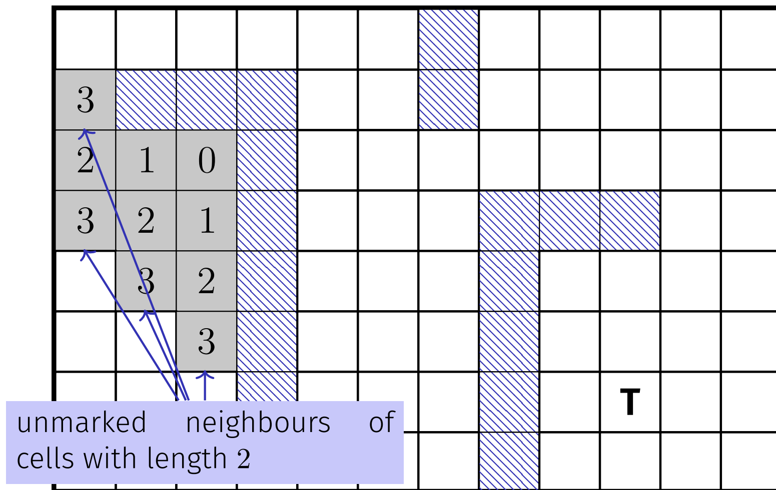
Mark all Cells with their Path Lengths

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 $i = 1, 2, 3 \ldots$

```
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 $i = 1, 2, 3...$

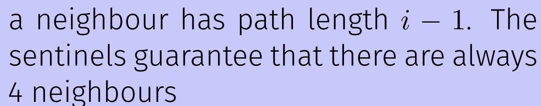
```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != -1) continue;
      if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
          floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
        floor[r][c] = i; // label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

cell already marked or obstacle

Main Loop

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

```
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* (Breadth-first vs. *depth-first* search is typically discussed in lectures on algorithms)

The Shortest Paths Program

- Algorithm: *Breadth-First Search* (Breadth-first vs. *depth-first* search is typically discussed in lectures on algorithms)
- The program can become pretty slow because for each i all cells are traversed

The Shortest Paths Program

- Algorithm: *Breadth-First Search* (Breadth-first vs. *depth-first* search is typically discussed in lectures on algorithms)
- 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

16. Recursion 1

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

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!

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

```
// 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 **guaranteed progress towards an exit condition ($\approx$ base case)**

Example `fac(n)`:

- Recursion ends if $n \leq 1$
- Recursive call with new argument $< n$
- Exit condition will thus be reached eventually

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

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

Recursive Functions: Evaluation

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

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

Recursive Functions: Evaluation

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

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

`fac(4)`

Calling `fac(4)`

Recursive Functions: Evaluation

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

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

`fac(4)` ⇝ `int n = 4`

Calling `fac(4)` ⇝ Initialisation of formal argument `n`

Recursive Functions: Evaluation

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

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

`fac(4)` $\leadsto$ `int n = 4`

Evaluation of return expression

Recursive Functions: Evaluation

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

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

`fac(4)` $\rightsquigarrow$ `int n = 4`

$\hookrightarrow$ `fac(n - 1)`

Recursive call with argument `n - 1` $= 4 - 1 = 3$

Recursive Functions: Evaluation

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

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

`fac(4)` $\rightsquigarrow$ `int n = 4`

$\hookrightarrow$ `fac(n - 1)` $\rightsquigarrow$ `int n = 3`

Initialisation of formal argument `n`

Recursive Functions: Evaluation

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

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

`fac(4)` $\rightsquigarrow$ `int n = 4`

$\hookrightarrow$ `fac(n - 1)` $\rightsquigarrow$ `int n = 3`

$\vdots$

Every call of **`fac`** *operates on its own* **`n`**

The Call Stack

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

The Call Stack

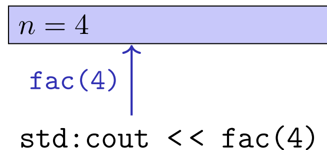
For each function call:

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

The Call Stack

For each function call:

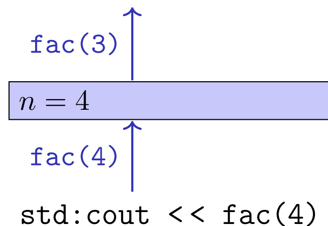
- push value of the call argument onto the stack



The Call Stack

For each function call:

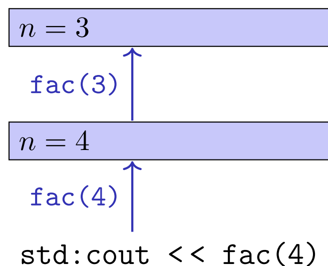
- push value of the call argument onto the stack



The Call Stack

For each function call:

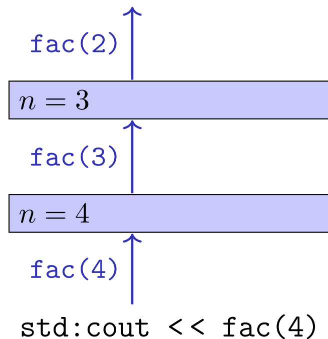
- push value of the call argument onto the stack



The Call Stack

For each function call:

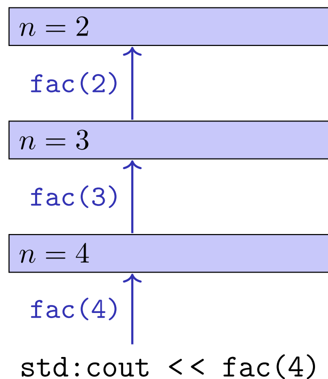
- push value of the call argument onto the stack



The Call Stack

For each function call:

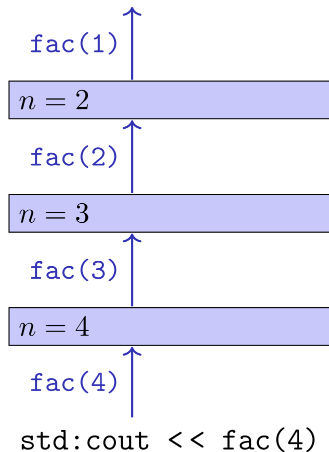
- push value of the call argument onto the stack



The Call Stack

For each function call:

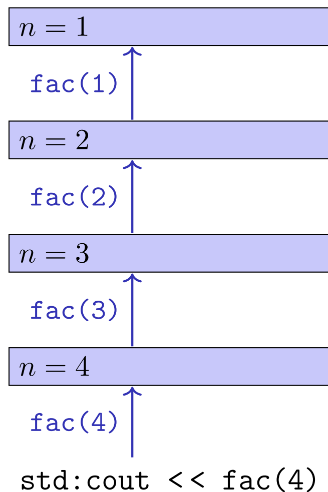
- push value of the call argument onto the stack



The Call Stack

For each function call:

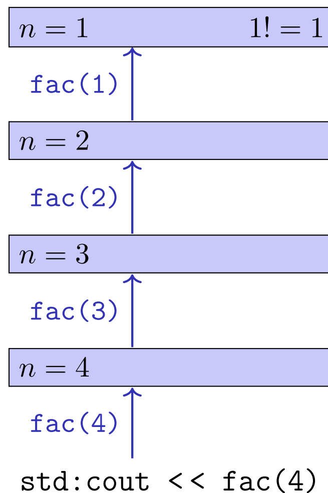
- push value of the call argument onto the stack



The Call Stack

For each function call:

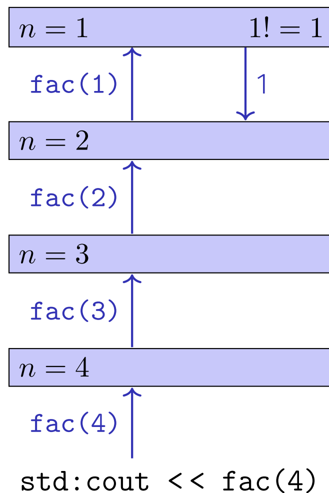
- push value of the call argument onto the stack
- always work with the top value



The Call Stack

For each function call:

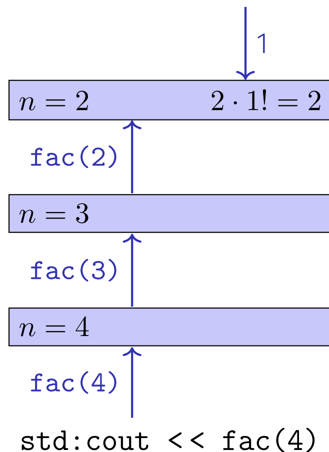
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

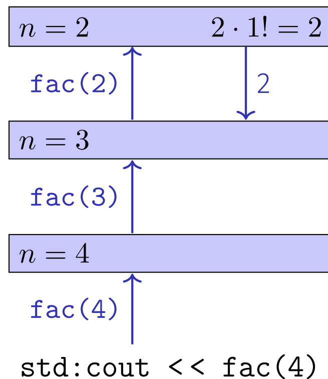
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

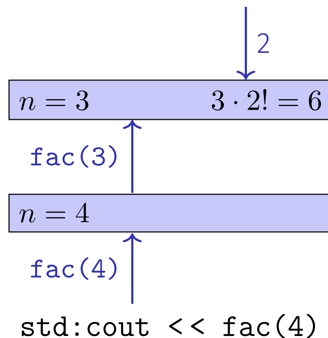
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

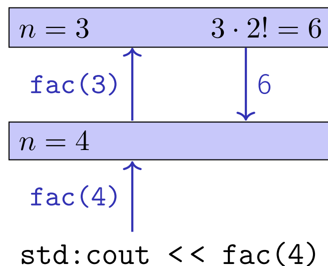
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

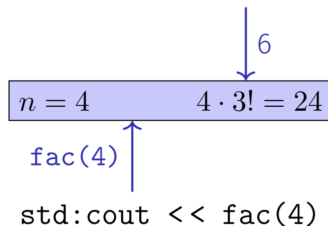
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

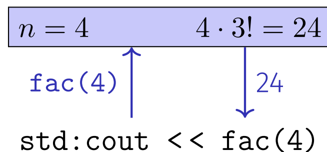
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



The Call Stack

For each function call:

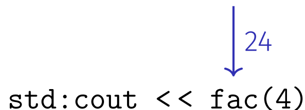
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



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



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 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 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 $\dots F_1$ 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 Fibonacci 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 Fibonacci numbers (variables `a` and `b`)
- Compute the next number as a sum of `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 Fibonacci numbers (variables `a` and `b`)
- Compute the next number as a sum of `a` and `b`

Can be implemented recursively and iteratively, the latter is easier/more direct

Fast Fibonacci Numbers 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; // a becomes F_{i-1}
    b += a_old; // b becomes F_{i-1} + F_{i-2}, i.e. 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 == 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; // a becomes F_{i-1}
    b += a_old; // b becomes F_{i-1} + F_{i-2}, i.e. 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 == 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; // a becomes F_{i-1}
    b += a_old; // b becomes F_{i-1} + F_{i-2}, i.e. 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 == 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; // a becomes F_{i-1}
    b += a_old; // b becomes F_{i-1} + F_{i-2}, i.e. F_i
  }

  return b;
}
```

very fast, also for **`fib(50)`**

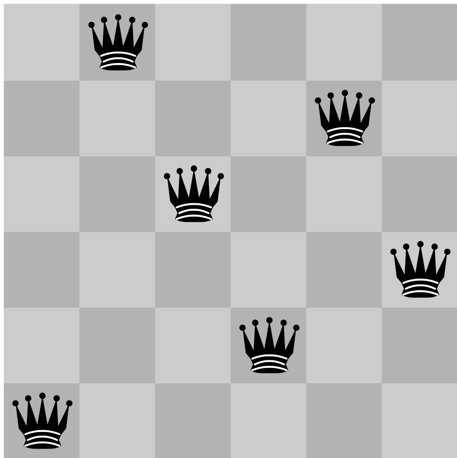
$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a   b

The Power of Recursion

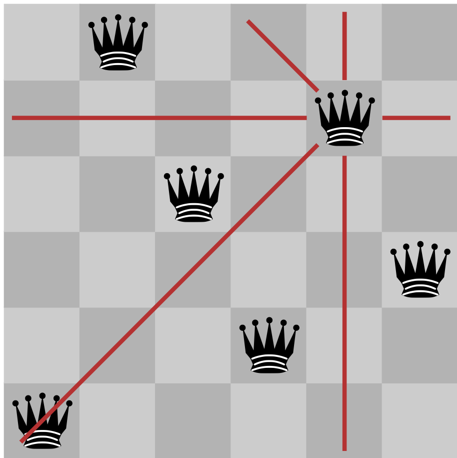
- Some problems appear to be hard to solve without recursion. With recursion they become significantly simpler.
- Examples: *The n-Queens-Problem*, The towers of Hanoi, Sudoku-Solver, Expression Parsers, Reversing In- or Output, Searching in Trees, Divide-And-Conquer (e.g. sorting) , ...
- ...and the 2. bonus exercise: Nonograms

The n-Queens Problem



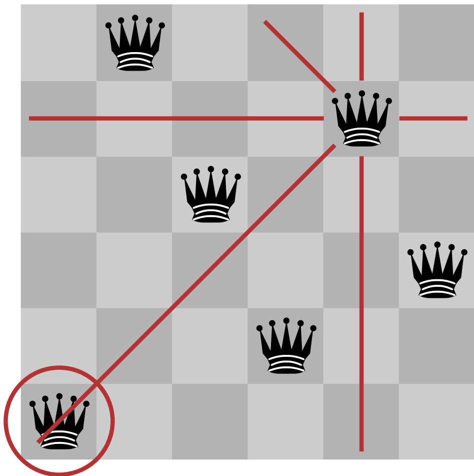
- Provided is a $n timesn$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?

The n-Queens Problem



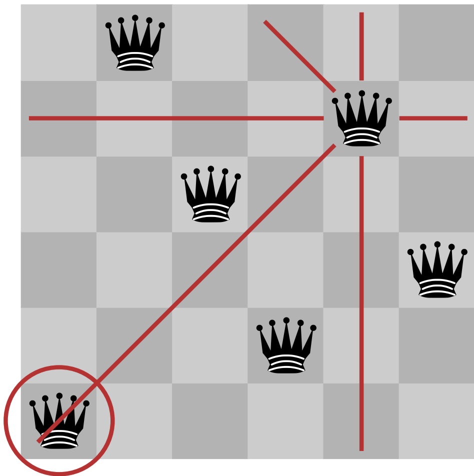
- Provided is a $n \ timesn$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?

The n-Queens Problem



- Provided is a $n\ timesn$ 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!

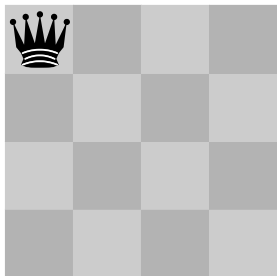
Solution?

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

Solution?

- Try all possible placements?
- $\binom{n^2}{n}$ possibilities. Too many!
- Only ne queen per row: n^n possibilities. Better – but still too many.
- Idea: don't proceed with futile attempts, retract incorrect moves instead $\Rightarrow$ *Backtracking*

Solution with Backtracking

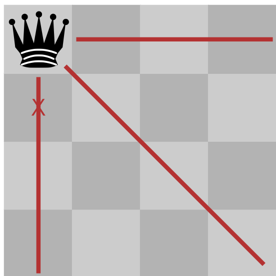


First Queen

queens

0
0
0
0

Solution with Backtracking

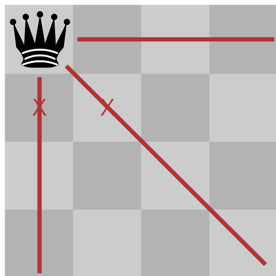


Forbidden Squares: no other queens may be here.

queens

0
0
0
0

Solution with Backtracking

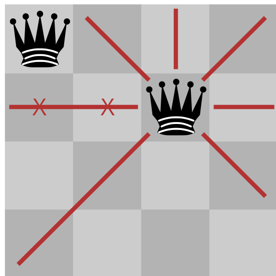


Forbidden Squares: no other queens may be here.

queens

0
1
0
0

Solution with Backtracking

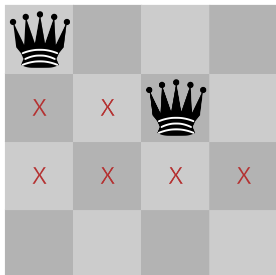


Second Queen in next row (no collision)

queens

0
2
0
0

Solution with Backtracking

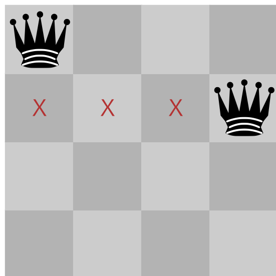


All squares in next row forbiden. Track back !

queens

0
2
4
0

Solution with Backtracking

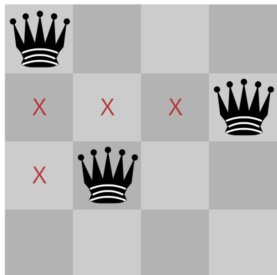


Move queen one step further and try again

queens

0
3
0
0

Solution with Backtracking

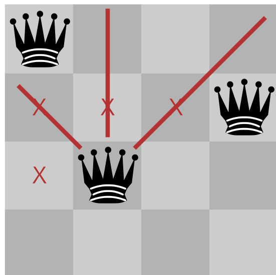


next row

queens

0
3
1
0

Solution with Backtracking

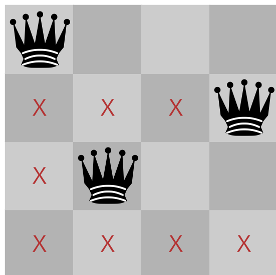


Ok (only previous queens have to be tested)

queens

0
3
1
0

Solution with Backtracking

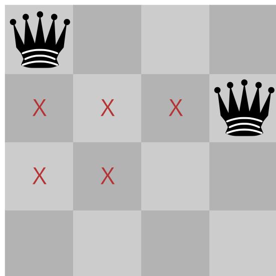


All squares of the next row forbidden. Track back.

queens

0
3
1
4

Solution with Backtracking

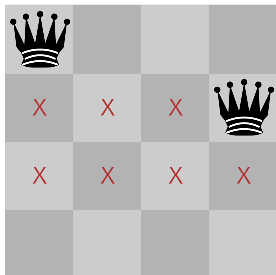


Continue in previous row.

queens

0
3
1
0

Solution with Backtracking

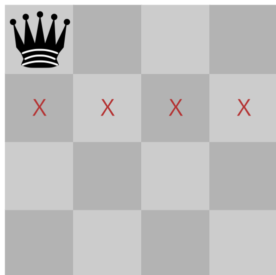


Remaining squares also forbidden. Track back!

queens

0
3
4
0

Solution with Backtracking

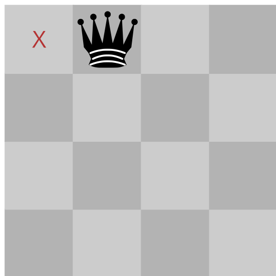


All squares of this row did not yield a solution. Track back!

queens

0
4
0
0

Solution with Backtracking

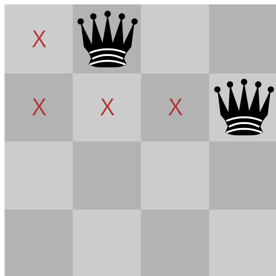


again advance queen
by one square

queens

1
0
0
0

Solution with Backtracking

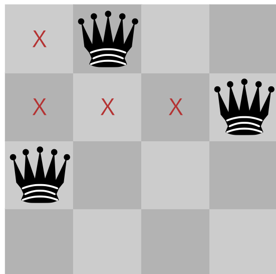


next row

queens

1
3
0
0

Solution with Backtracking

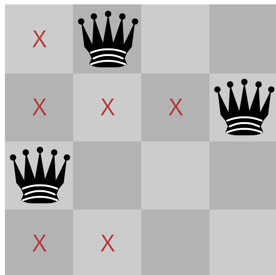


next row

queens

1
3
0
0

Solution with Backtracking

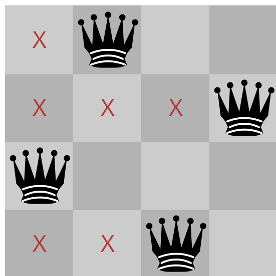


next row

queens

1
3
0
1

Solution with Backtracking

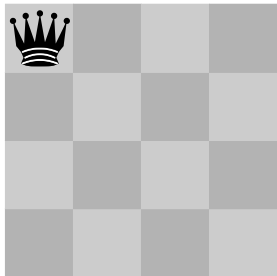


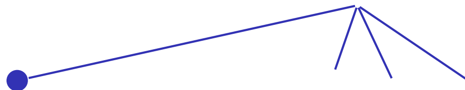
Found a solution

queens

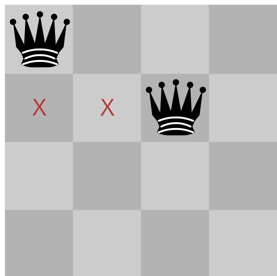
1
3
0
2

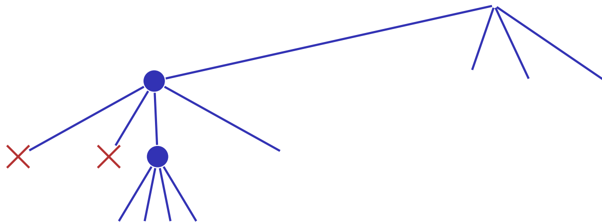
Search Strategy Visualized as a Tree



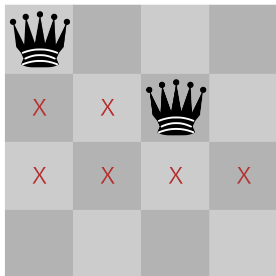


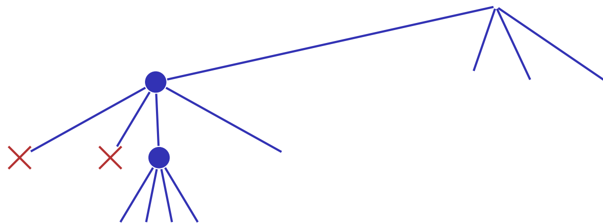
Search Strategy Visualized as a Tree



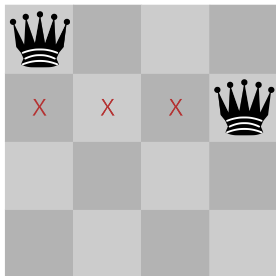


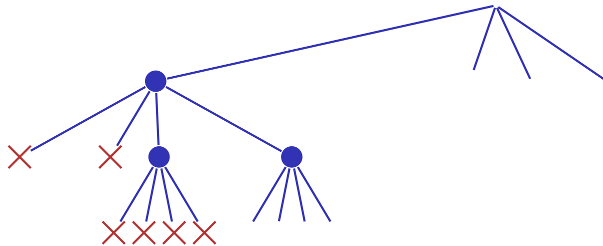
Search Strategy Visualized as a Tree



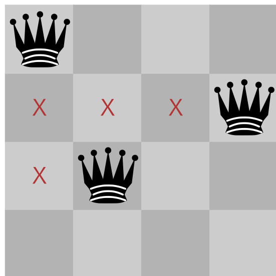


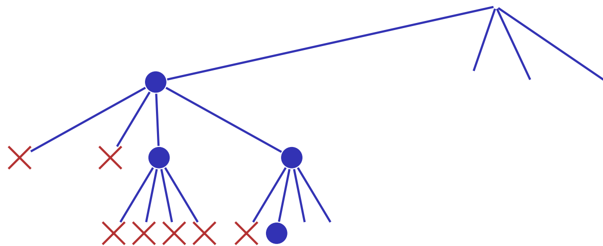
Search Strategy Visualized as a Tree



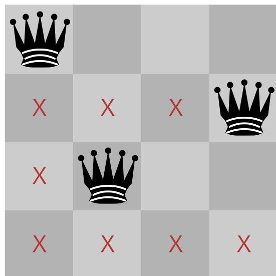


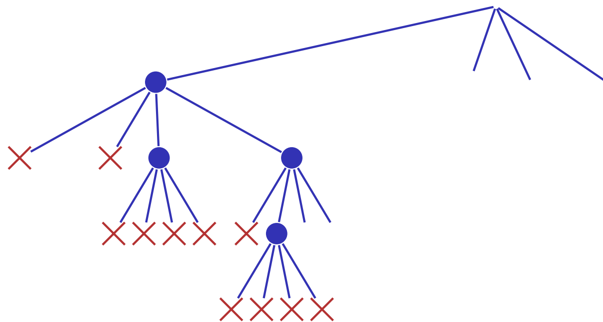
Search Strategy Visualized as a Tree



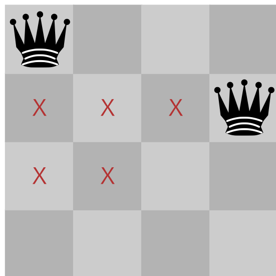


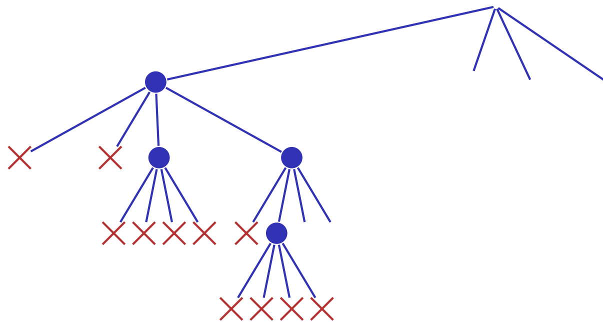
Search Strategy Visualized as a Tree



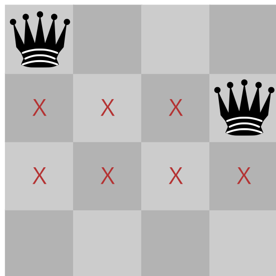


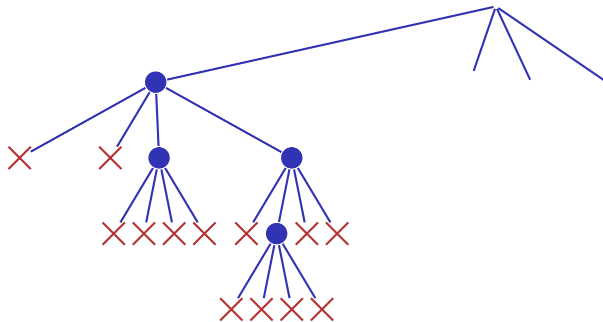
Search Strategy Visualized as a Tree



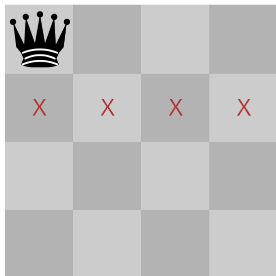


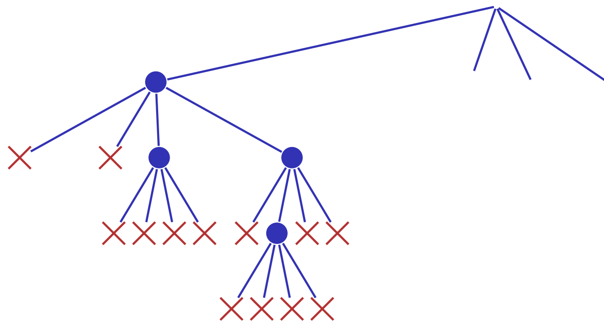
Search Strategy Visualized as a Tree



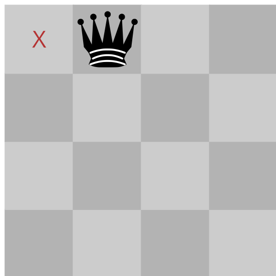


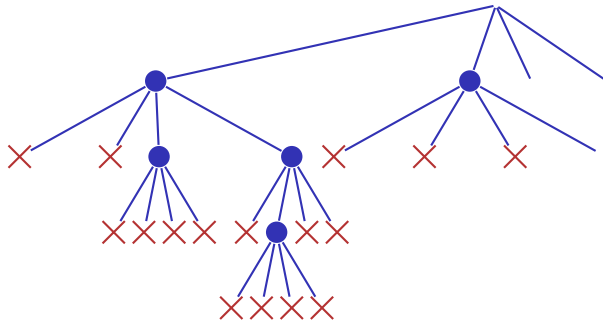
Search Strategy Visualized as a Tree



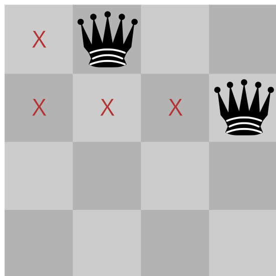


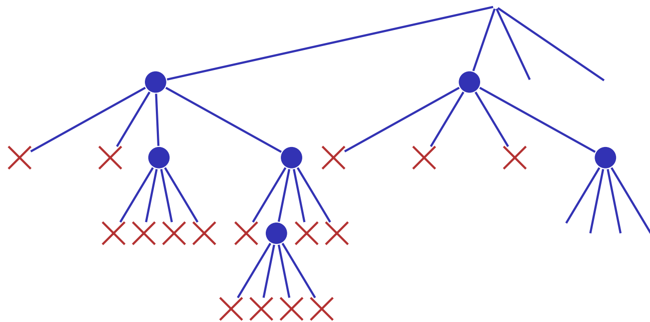
Search Strategy Visualized as a Tree



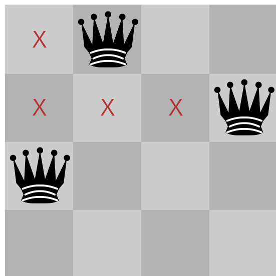


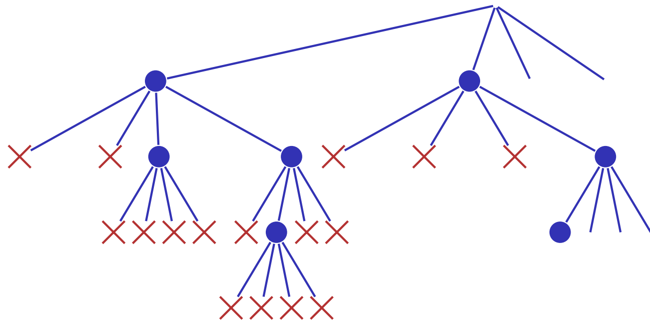
Search Strategy Visualized as a Tree



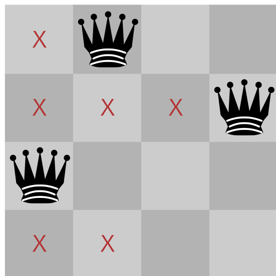


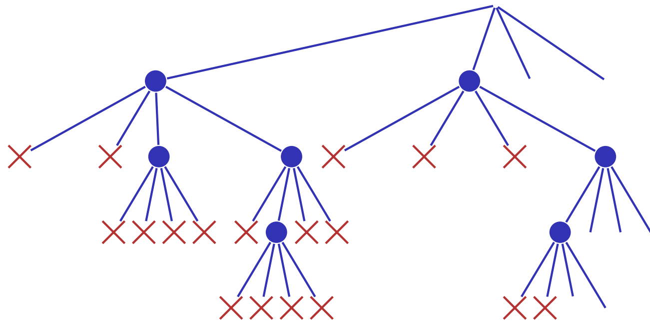
Search Strategy Visualized as a Tree





Search Strategy Visualized as a Tree





Search Strategy Visualized as a Tree

Check Queen

```
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
}
```

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;
}
```