

Algorithms

Recursion I

Emanuele Rodolà
rodola@di.uniroma1.it



Recursive algorithms

MERGE-SORT(A, p, r)

1 **if** $p < r$

2 **then** $q \leftarrow \lfloor (p + r)/2 \rfloor$

3 MERGE-SORT(A, p, q)

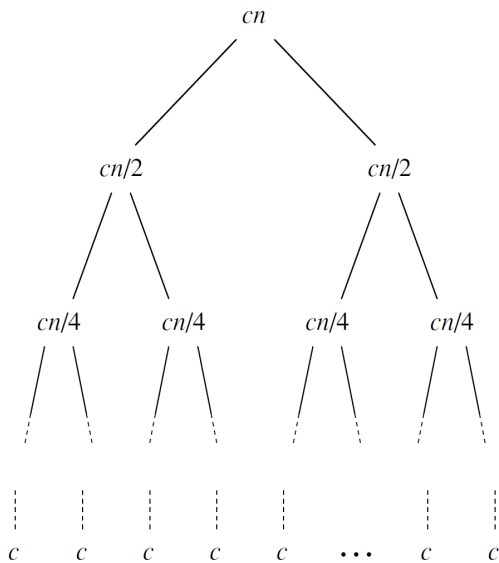
4 MERGE-SORT($A, q + 1, r$)

5 MERGE(A, p, q, r)

Recursive algorithms

```
9 def binsearch(S, x, i, f):
10     if i < f:
11         q = int((i + f) / 2)
12         if x < S[q]:
13             return binsearch(S, x, i, q)
14         elif x > S[q]:
15             return binsearch(S, x, q + 1, f)
16         elif x == S[q]:
17             return q + 1
18     else:
19         return -1
20
21
22 A = [11, 21, 31, 41, 51, 61, 71, 91]
23 v = 71
24 index = binsearch(A, v, 0, len(A))
25 print(index)
```

Recursion tree



The factorial

$$4! = 4 \times 3 \times 2 \times 1$$

The factorial

$$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \cdots$$

The factorial

$$n! = n \times (n - 1)!$$

The factorial

$$n! = n \times (n - 1)!$$

$$(n - 1)! = (n - 1) \times (n - 2)!$$

The factorial

$$n! = n \times (n - 1)!$$

$$(n - 1)! = (n - 1) \times (n - 2)!$$

$$(n - 2)! = (n - 2) \times (n - 3)!$$

The factorial

$$n! = n \times (n - 1)!$$

$$(n - 1)! = (n - 1) \times (n - 2)!$$

$$(n - 2)! = (n - 2) \times (n - 3)!$$

$$\vdots$$

$$1! = 1$$

The factorial

$$n! = n \times (n - 1)!$$

$$(n - 1)! = (n - 1) \times (n - 2)!$$

$$(n - 2)! = (n - 2) \times (n - 3)!$$

$$\vdots$$

$$1! = 1 \quad \text{base case}$$

The factorial

```
1 factorial(n):  
2     ....if n == 1:  
3     ....|.....return 1  
4     ....else:  
5     ....|.....return n*factorial(n-1)  
6     ....end
```

The factorial

```
1 factorial(n):  
2     ....if n == 1:  
3     ....|.....return 1  
4     ....else:  
5     ....|.....return n*factorial(n-1)  
6     ....end
```

Without the **base case**, we would get an infinite loop.

The factorial

```
1 factorial(n):  
2     ....if n == 1:  
3     ....|....return 1  
4     ....else:  
5     ....|....return n*factorial(n-1)  
6     ....end
```

Without the **base case**, we would get an infinite loop.

Going through the recursions traces a path along the recursion tree.

The factorial

```
1 factorial(n):  
2     ....if n == 1:  
3     ....|.....return 1  
4     ....else:  
5     ....|.....return n*factorial(n-1)  
6     ....end
```

Without the **base case**, we would get an infinite loop.

Going through the recursions traces a path along the recursion tree.

Each function call generates a **context**, containing all the data associated with that function call.

The recursion stack

Recursion involves **nested function calls**.

The contexts are memorized one on top of the other, in a **stack**.

The recursion stack

Recursion involves **nested function calls**.

The contexts are memorized one on top of the other, in a **stack**.

```
1 factorial(n):  
2   ... if n == 1:  
3   ...     return 1  
4   ... else:  
5   ...     return n*factorial(n-1)  
6   ... end
```

The recursion stack

Recursion involves **nested function calls**.

The contexts are memorized one on top of the other, in a **stack**.


```
1 factorial(3n):  
2   ... if n == 1:  
3   ...     return 1  
4   ... else:  
5   ...     return 3n*factorial(2n-1)  
6   ... end
```

The recursion stack

Recursion involves **nested function calls**.

The contexts are memorized one on top of the other, in a **stack**.

```
1 factorial(3n):  
2     ... if n == 1:  
3     ...     return 1  
4     ... else:  
5     ...     return 3n*factorial(2n-1)  
6     ... end
```

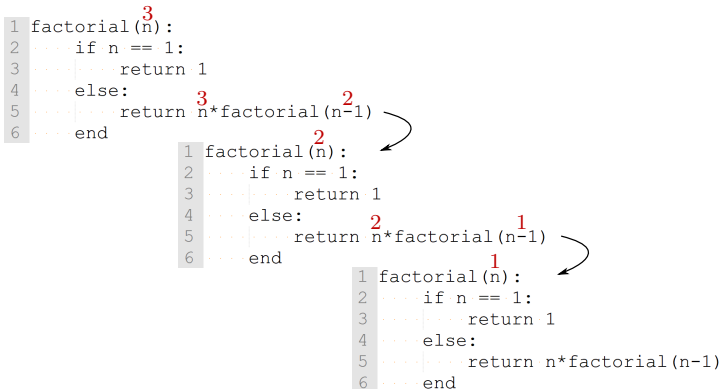


```
1 factorial(2n):  
2     ... if n == 1:  
3     ...     return 1  
4     ... else:  
5     ...     return 2n*factorial(1n-1)  
6     ... end
```

The recursion stack

Recursion involves **nested function calls**.

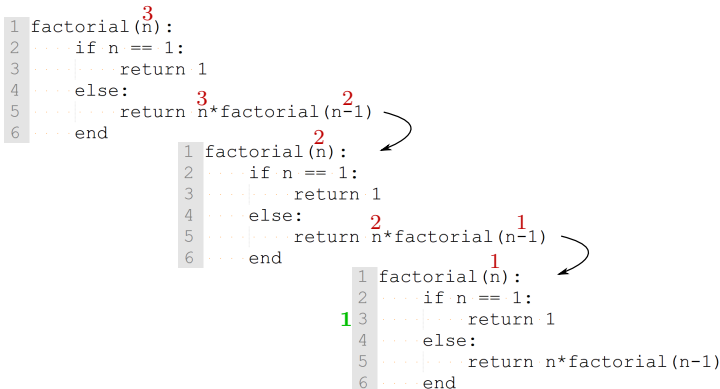
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

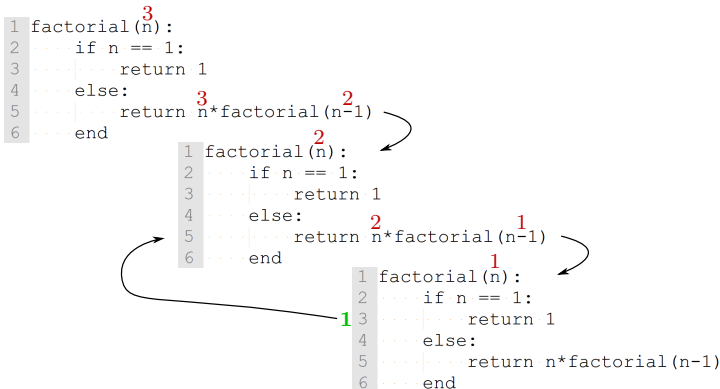
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

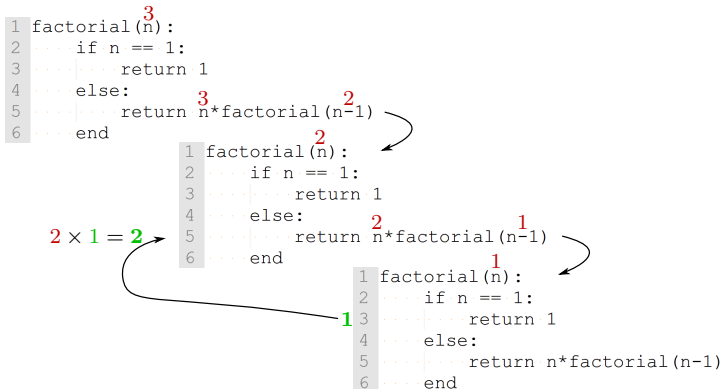
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

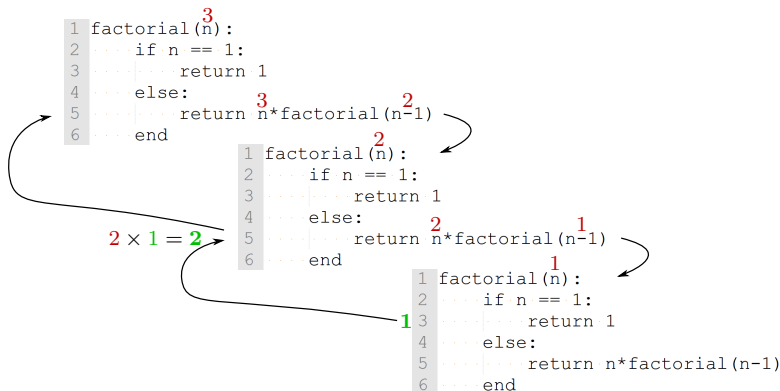
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

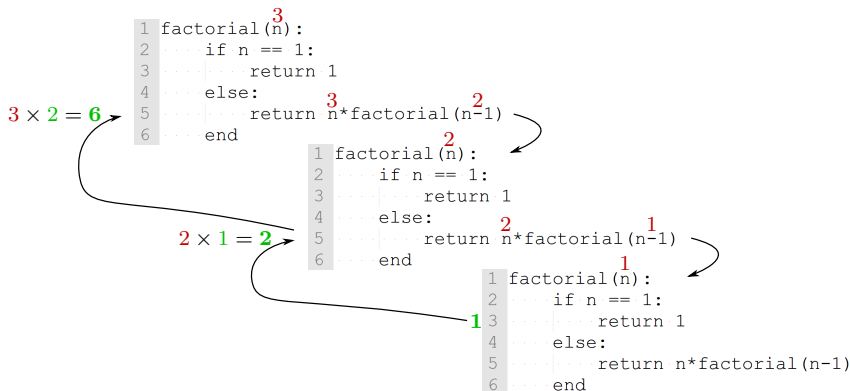
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

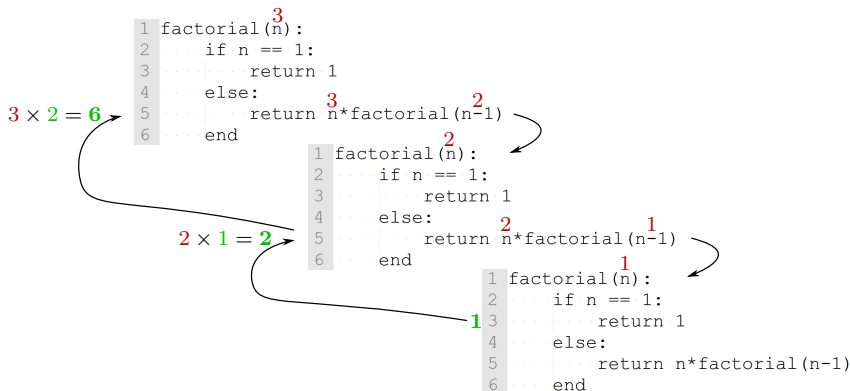
The contexts are memorized one on top of the other, in a **stack**.



The recursion stack

Recursion involves **nested function calls**.

The contexts are memorized one on top of the other, in a **stack**.



Tip: Use the **debugger** to trace recursive calls step by step.

The recursion stack

```
1 factorial(3n):  
2     ... if n == 1:  
3     ...     return 1  
4     ... else:  
5     ...     return 3n*factorial(2n-1)  
6     ... end
```

$$3 \times 2 = 6$$

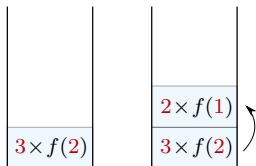
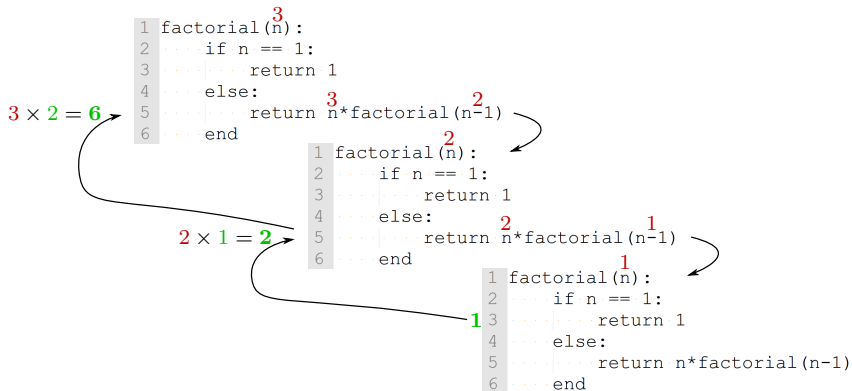
```
1 factorial(2n):  
2     ... if n == 1:  
3     ...     return 1  
4     ... else:  
5     ...     return 2n*factorial(1n-1)  
6     ... end
```

$$2 \times 1 = 2$$

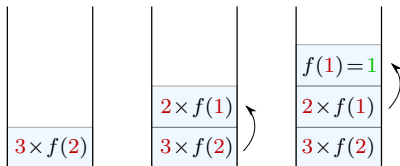
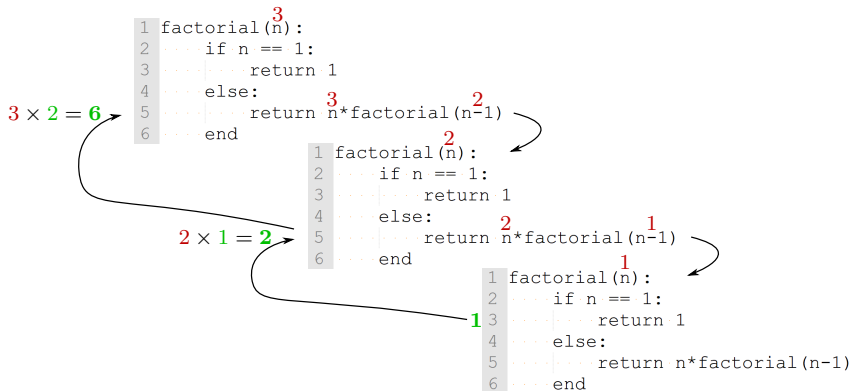
```
1 factorial(1n):  
2     ... if n == 1:  
3     ...     return 1  
4     ... else:  
5     ...     return n*factorial(n-1)  
6     ... end
```

$$3 \times f(2)$$

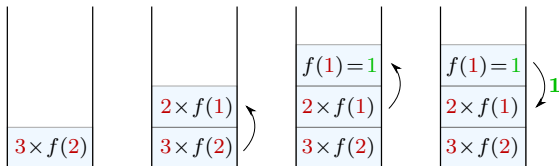
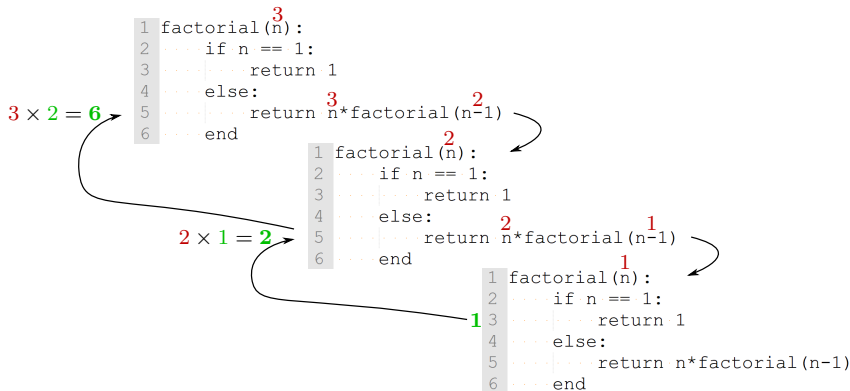
The recursion stack



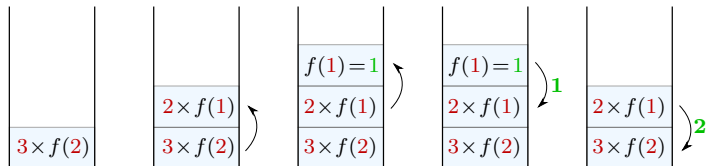
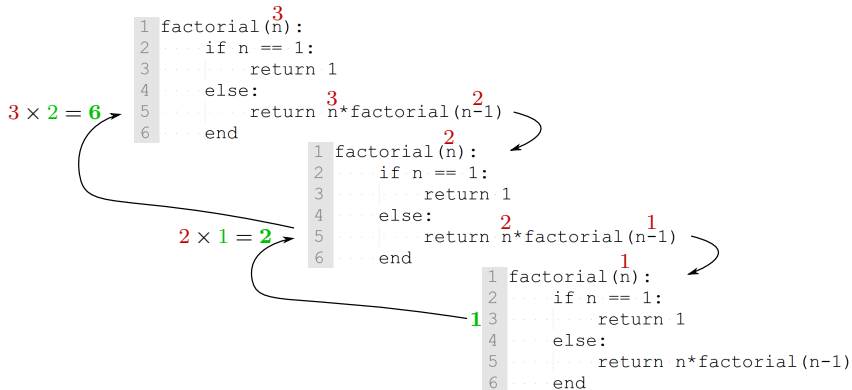
The recursion stack



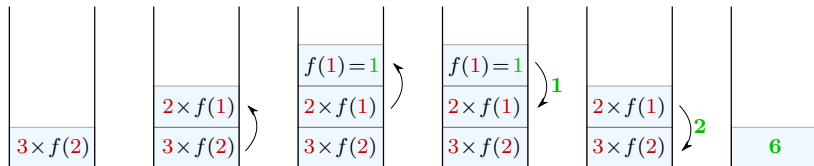
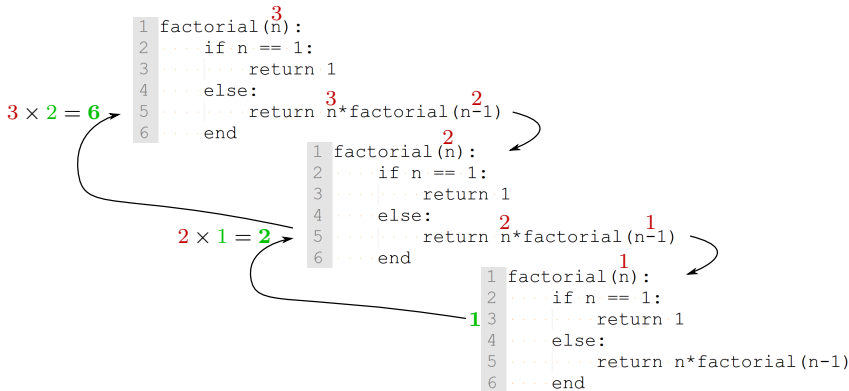
The recursion stack



The recursion stack



The recursion stack



Backtracking

General algorithm for a wide family of computational problems.

- Incrementally **build** a possible solution.
- Abandon the solution as soon as it does not meet the constraints.
(**backtrack**)

Backtracking

General algorithm for a wide family of computational problems.

- Incrementally **build** a possible solution.
- Abandon the solution as soon as it does not meet the constraints.
(**backtrack**)

Each intermediate step is a node of a **search tree**.

Backtracking traverses the tree **recursively**.

Backtracking

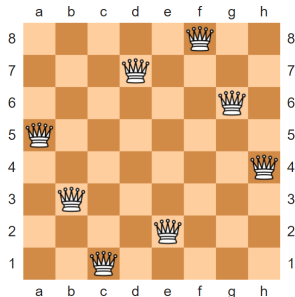
General algorithm for a wide family of computational problems.

- Incrementally **build** a possible solution.
- Abandon the solution as soon as it does not meet the constraints.
(**backtrack**)

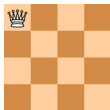
Each intermediate step is a node of a **search tree**.

Backtracking traverses the tree **recursively**.

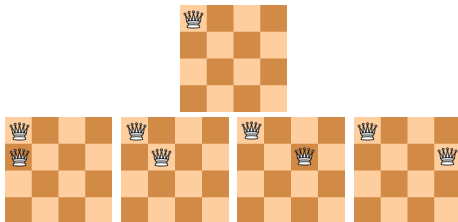
Classical example: **8 queens problem**.



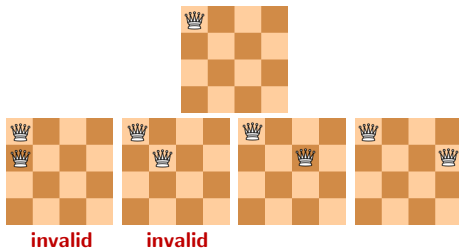
Example: 4 queens



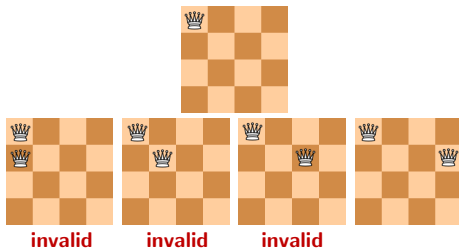
Example: 4 queens



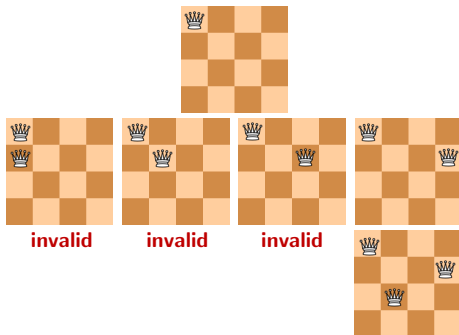
Example: 4 queens



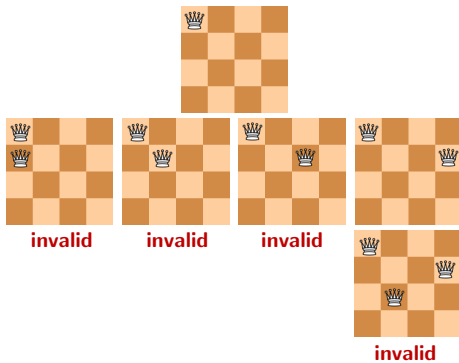
Example: 4 queens



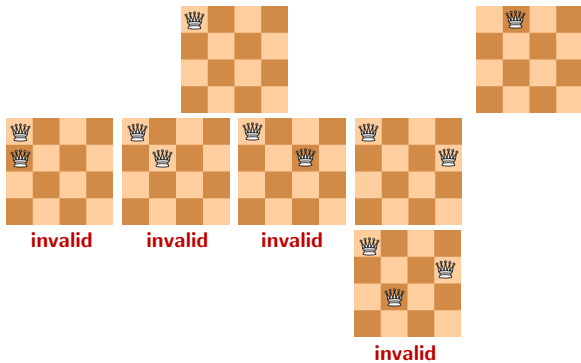
Example: 4 queens



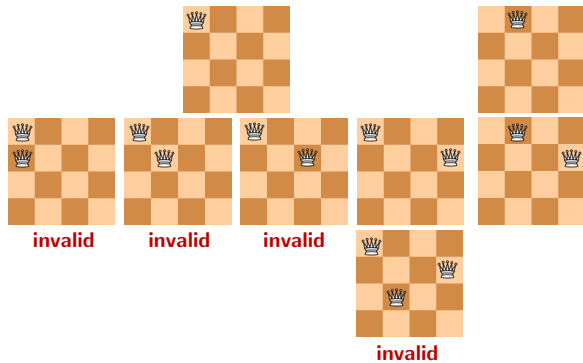
Example: 4 queens



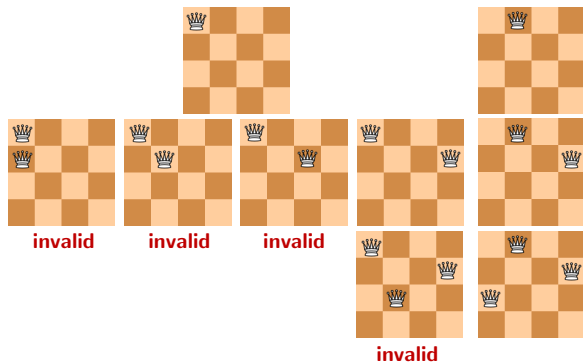
Example: 4 queens



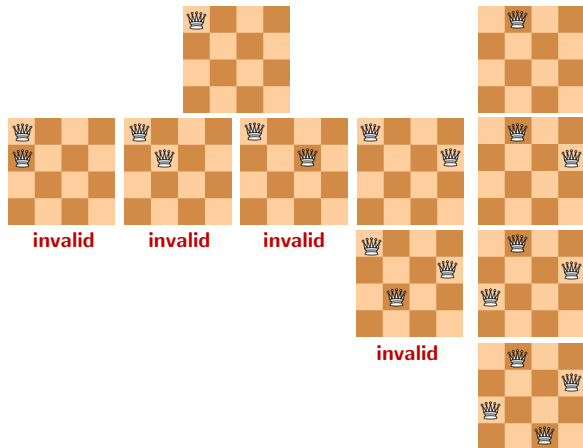
Example: 4 queens



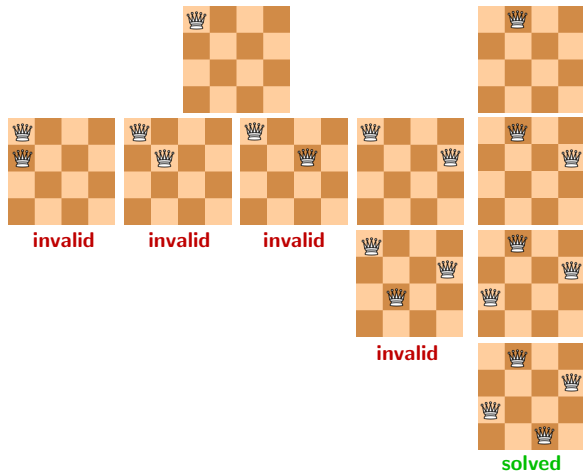
Example: 4 queens



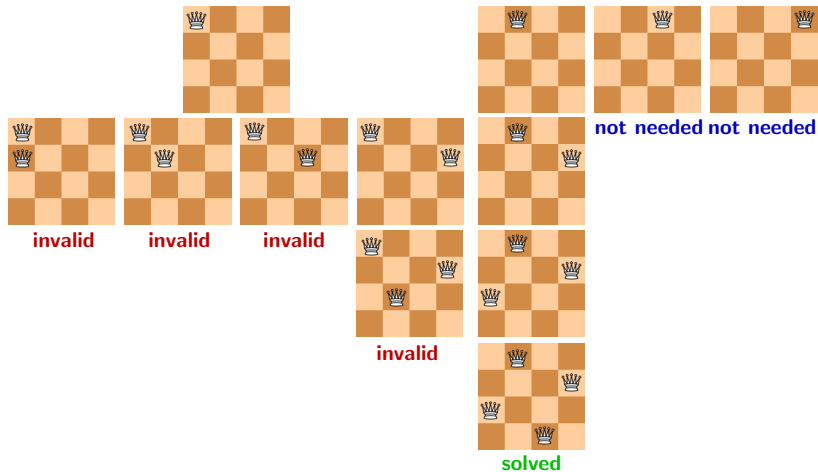
Example: 4 queens



Example: 4 queens



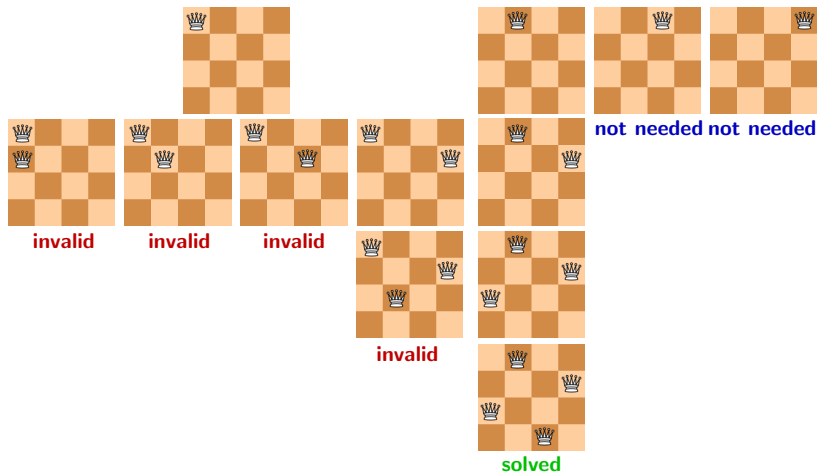
Example: 4 queens





9/10

Example: 4 queens



We only explore a small part of the **potential search tree**.

This is an example of **depth-first search** of a tree.

Exercises

Write the following **recursive** functions in Python:

- 1 The factorial.
- 2 The Fibonacci function, defined as:

$$F_n = F_{n-1} + F_{n-2}$$

with the base case $F_0 = 0, F_1 = 1$.

- 3 Write all the possible permutations of the nucleobase sequence *GATC*, that is: *GTAC*, *GCAT*, *ATCG*, *ACGT*, ...