Algorithms

Recursion I

Emanuele Rodolà
rodola@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

2nd semester a.y. 2020/2021 · March 17, 2021

Recursive algorithms

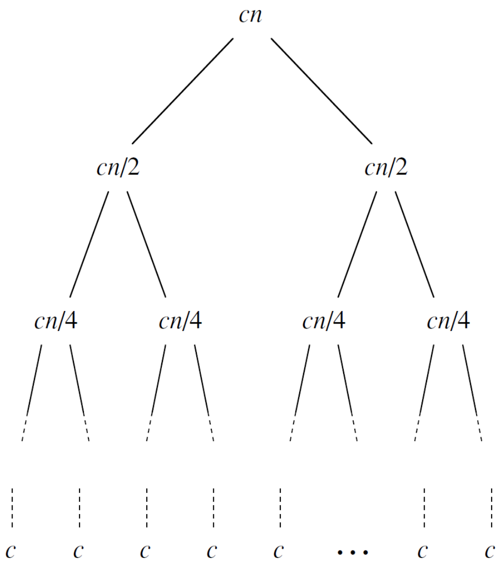
```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Recursive algorithms

```python
def binsearch(S, x, i, f):
    if i < f:
        q = int((i + f) / 2)
        if x < S[q]:
            return binsearch(S, x, i, q)
        elif x > S[q]:
            return binsearch(S, x, q + 1, f)
        elif x == S[q]:
            return q + 1
    else:
        return -1


A = [11, 21, 31, 41, 51, 61, 71, 91]
v = 71
index = binsearch(A, v, 0, len(A))
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

$$
\begin{aligned}
n! &= n \times (n-1)! \\
(n-1)! &= (n-1) \times (n-2)! \\
(n-2)! &= (n-2) \times (n-3)!
\end{aligned}
$$

The factorial

$$
\begin{aligned}
n! &= n \times (n-1)! \\
(n-1)! &= (n-1) \times (n-2)! \\
(n-2)! &= (n-2) \times (n-3)! \\
&\vdots \\
1! &= 1
\end{aligned}
$$

The factorial

$$
\begin{aligned}
n! &= n \times (n-1)! \\
(n-1)! &= (n-1) \times (n-2)! \\
(n-2)! &= (n-2) \times (n-3)! \\
&\vdots \\
1! &= 1 \qquad \textbf{base case}
\end{aligned}
$$

The factorial

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The factorial

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

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

The factorial

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

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

Q: What is the complexity $T(n)$ of this algorithm?

The factorial

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

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

Q: What is the complexity $T(n)$ of this algorithm?

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.

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

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

```
          2
factorial(n):
    if n == 1:
        return 1
    else:
               2            1
        return n*factorial(n-1)
    end
```

```
          1
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

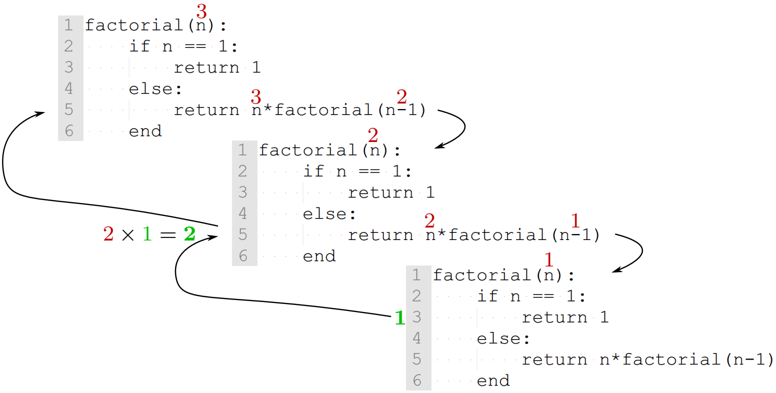
```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

1

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.

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$3 \times 2 = \mathbf{6}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$\mathbf{1}$

The recursion stack

Recursion involves nested function calls.

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

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$3 \times 2 = \mathbf{6}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$\mathbf{1}$

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

The recursion stack

$3 \times 2 = \mathbf{6}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$\mathbf{1}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$3 \times f(2)$

The recursion stack

$3 \times 2 = \mathbf{6}$

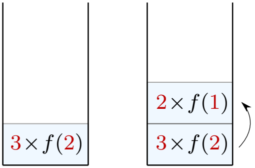
```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

1

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```



The recursion stack

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

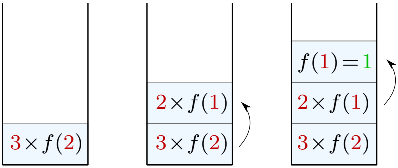
$3 \times 2 = \mathbf{6}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$\mathbf{1}$



The recursion stack

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

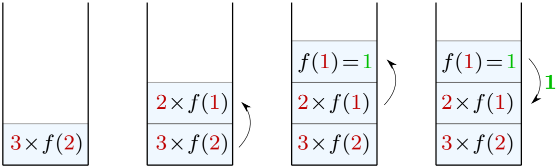
$3 \times 2 = \mathbf{6}$

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = \mathbf{2}$

1

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```



The recursion stack

$3 \times 2 = 6$

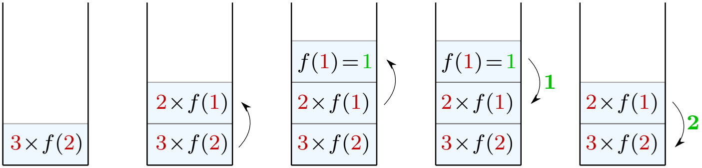
```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

$2 \times 1 = 2$

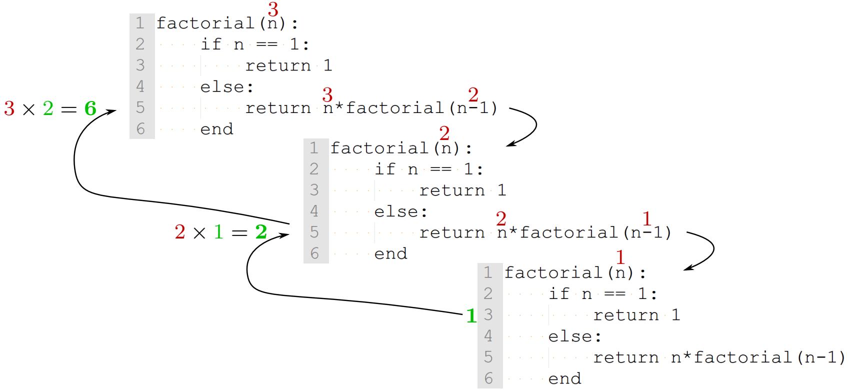
```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```

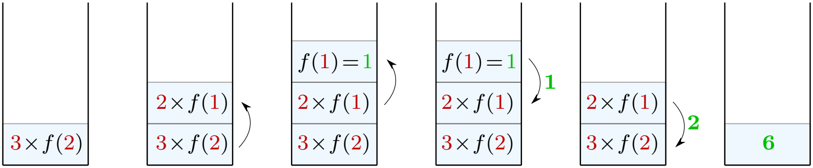
1

```
factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
    end
```



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

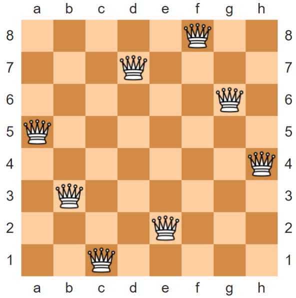
General algorithm for a wide family of computational problems.

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

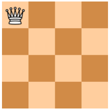
Each intermediate step is a node of a search tree.

Backtracking traverses the tree recursively.

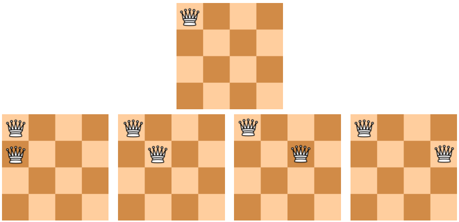
Classical example: 8 queens problem.



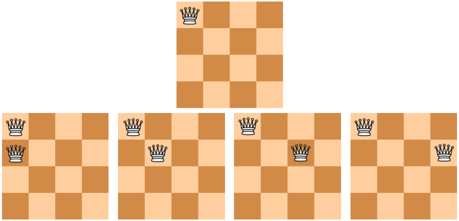
Example: 4 queens



Example: 4 queens

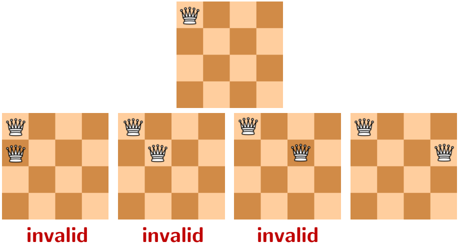


Example: 4 queens

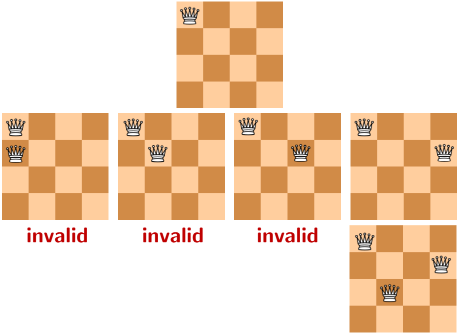


invalid **invalid**

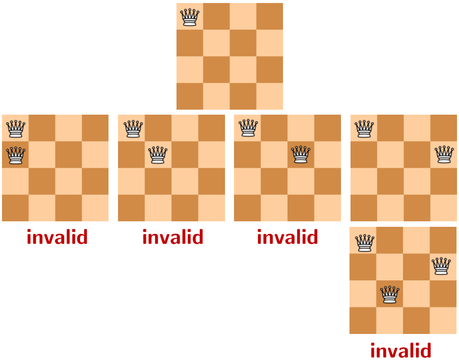
Example: 4 queens



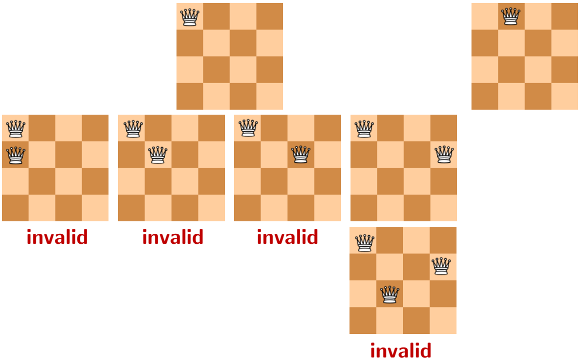
Example: 4 queens



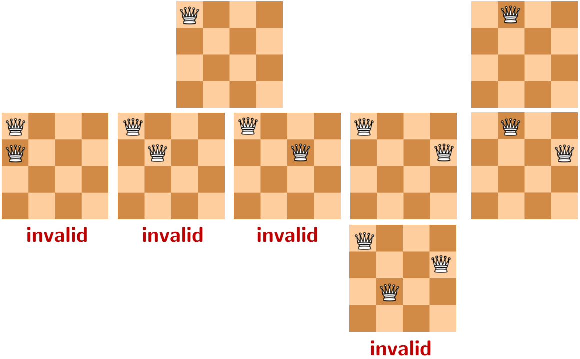
Example: 4 queens



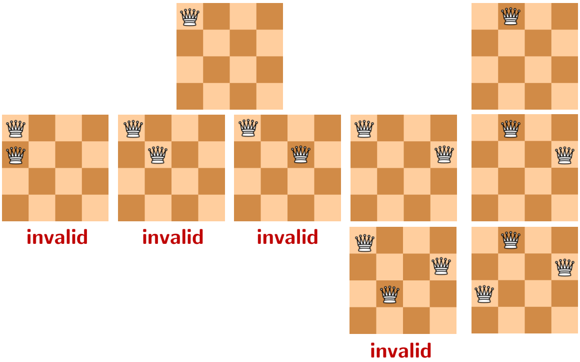
Example: 4 queens



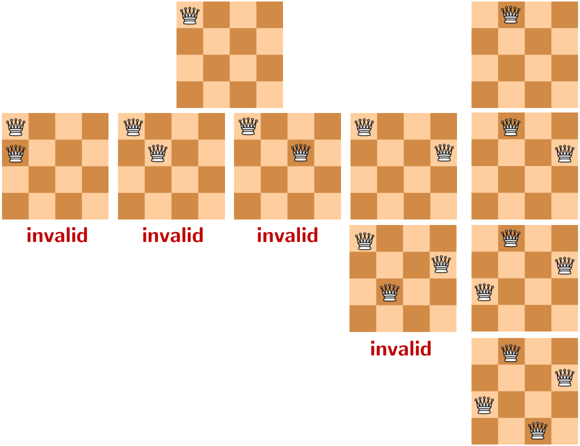
Example: 4 queens



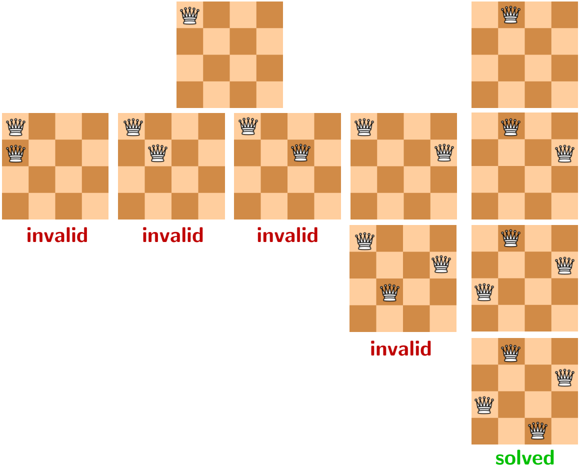
Example: 4 queens



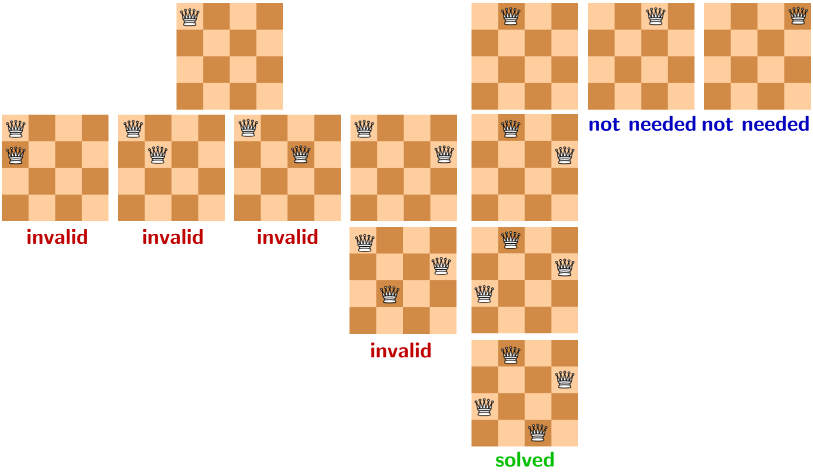
Example: 4 queens



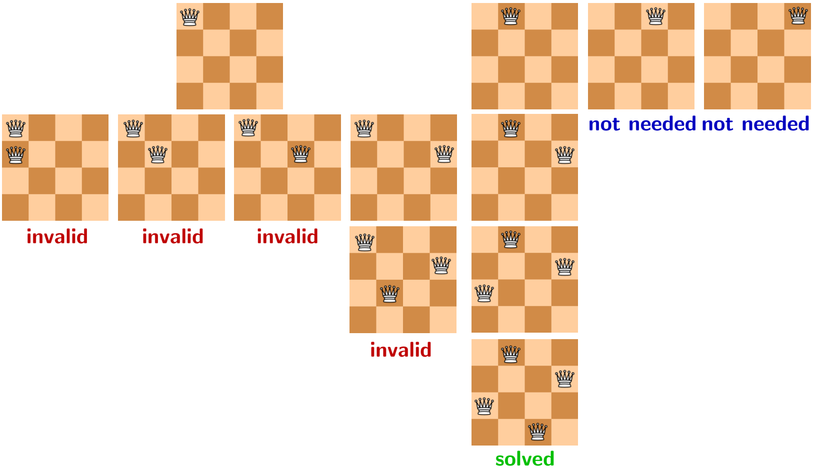
Example: 4 queens



Example: 4 queens

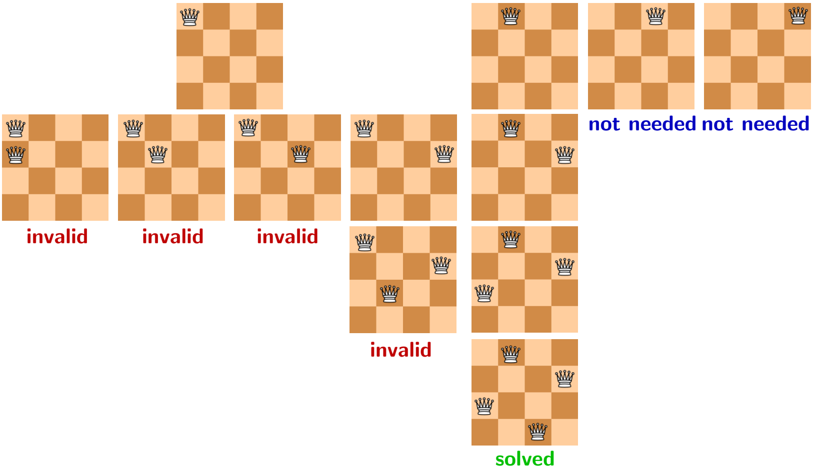


Example: 4 queens



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

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: $GTAC$, $GCAT$, $ATCG$, $ACGT$, ...