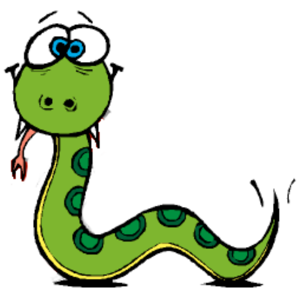
Freie Universität Berlin

Algorithmen und Programmieren II

Python (Teil 5)



SoSe 2020

Prof. Dr. Margarita Esponda

Verschachtelte Funktionen

Funktionen können innerhalb anderer Funktionen definiert werden.

```
def percent (a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    return (pc(a), pc(b), pc(c))

print (percent (2, 4, 4))
print (percent (1, 1, 1))
```

```
>>>
(20.0, 40.0, 40.0)
(33.3333333333, 33.3333333333, 33.3333333333)
>>>
```

Funktionen als Objekte

```
def myMap(ls, func):
    """assumes ls is a list and f is a function"""
    for i in range(len(ls)):
        ls[i] = func(ls[i])

list1 = [2, 3, 4, 5, 0, 1]
myMap(list1, factorial)
print ( list1 )
```

```
def factorial (n):
  if  n<=0:
    return 1
  else:
    return n * factorial(n-1)
```

Ausgabe:

```
>>>
[2, 6, 24, 120, 1, 1]
```

Funktionen höherer Ordnung

```
myMap (f, [])     = []
myMap (f, (x:xs)) = (f x) : myMap (f, xs)
```

```python
def myMap (f, xs):
    """ assumes f is a function and xs is a list """
    result = []
    for x in xs:
        result.append(f(x))
    return result

nums = [2,3,4,5,0,1]
result_list = myMap (factorial, nums)
print(nums)
print(result_list)
```

Ausgabe:

```
>>>
[2, 3, 4, 5, 0, 1]
[2, 6, 24, 120, 1, 1]
```

test-Funktionen

```
import random

def sum(nums):
    summe = 0
    for i in range(len(nums)):
        summe = summe + int(nums[i])
    return summe

def test_sum():
    print(sum([]))
    print(sum([9, 0, 3, 2, 5, 4, 9, 0, 9, 110]))

def test_sum1():
    array = []
    for i in range(10):
        array.append( random.randint(1,100) )
    print("sum( ", array, ")= ", sum(array))

def test_sum2():
    print("type the numbers to be added separated with spaces: ")
    array = list(map(int, input().split()))
    print("sum( ", array, ")= ", sum(array))
```

```
>>>
 0
151
sum(  [17, 15, 6, 79, 85, 52, 75, 76, 35, 17] )=  457
type the numbers to be added separated with spaces:
17 15 6 79 85 52 75 76 35 17
sum(  [17, 15, 6, 79, 85, 52, 75, 76, 35, 17] )=  457
```

Funktionen als Objekte

```
def print_char_picture( decide_char_func ):
    size = 40
    for i in range( 0, size):
        for j in range( 0, size):
            print( decide_char_func( j, i, size), end='')
        print()

def diagonal( x, y, size):
    if x==y:
        return '@'
    else:
        return '.'

def grid( x, y, size):
    if (x%4==0) or (y%4==0):
        return '.'
    else:
        return ' '

print_char_picture(diagonal)
print_char_picture(grid)
```

yield-Anweisung

Die **yield**-Anweisung innerhalb einer Funktion **f** verursacht einen Rücksprung in die aufrufende Funktion und der Wert hinter der **yield**-Anweisung wird als Ergebnis zurückgegeben.

Im Unterschied zur return-Anweisung werden die aktuelle Position innerhalb der Funktion **f** und ihre lokalen Variablen zwischengespeichert.

Beim nächsten Aufruf der Funktion **f** springt Python hinter dem zuletzt ausgeführten yield weiter und kann wieder auf die alten lokalen Variablen von **f** zugreifen.

Wenn das Ende der Funktion **f** erreicht wird, wird diese endgültig beendet.

Freie Universität Berlin

yield-Anweisung

```
def myRange(n):
    i = 0
    while (i<n):
        yield i
        i += 1

for i in myRange(5):
    print(i)
```

```
>>>
0
1
2
3
4
>>>
```

```
def genConstants():
    yield 3
    yield 5
    yield 11

def testMyRange():
    for x in genConstants():
        print(x)

testMyRange()
```

```
>>>
3
5
11
>>>
```

Listen-Generatoren

Python:

```
[ x*x for x in range (5) ]
```

Eine Liste mit den Quadratzahlen von 0 bis 4 wird generiert.

```
>>> [ x%3 for x in [1,5,-3,-6,4,7,6,0] if (x>0) ]
[1, 2, 1, 1, 0]
```

Online **Python** Tutor - Visualize program execution

http://www.pythontutor.com/visualize.html

Neue Anweisungen in Python

Wort	kurze Erläuterung
from	Teil einer import-Anweisung
global	Verlegung einer Variablen in den globalen Namensraum
is	test auf Identität
pass	Platzhalter, führt nichts aus

exec	Ausführung von Programmcode
yield	Ausführung von Programmcode

Analyse von Algorithmen

Die O-Notation

Freie Universität Berlin

Korrekte und effiziente Lösung von Problemen

Problem

Algorithmus

Wesentlicher Teil der Lösung eines Problems.

Sehr **kreative Phase**

Programm in einer höheren Programmiersprache

Programmierung oder Codierung des Algorithmus in einer bestimmten Programmiersprache.

Einfache Phase

Analyse von Algorithmen

Wichtigste Eigenschaften bei der Analyse von Algorithmen

- **Korrektheit**
- Terminierbarkeit
- Komplexität
  - Rechenzeit
  - Speicherplatz
  - Bandbreite oder Datentransfer

Freie Universität Berlin

Analyse von Algorithmen

Rechenzeit Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der Eingabegröße.

Speicherplatz Maximaler Speicherverbrauch während der Ausführung des Algorithmus in Abhängigkeit von der Komplexität der Eingabe.

Bandbreite Wie groß ist die erforderliche Datenübertragung.

Freie Universität Berlin

Analyse von Algorithmen

Zeitanalyse

- Charakterisierung unserer Daten
  (**Eingabegröße**)
- Bestimmung der abstrakten Operationen
  (**Berechnungsschritte** in unserem Algorithmus)
- Eigentliche mathematische Analyse, um eine Funktion in Abhängigkeit der Eingabegröße zu finden.
  Komplexitätsanalyse

Eingabedaten

Zuerst müssen wir unsere Eingabedaten charakterisieren.

Meistens ist es sehr schwer eine genaue Verteilung der Daten zu finden, die dem realen Fall entspricht. Deswegen müssen wir in der Regel den **schlimmsten Fall** betrachten und auf diese Weise eine obere Schranke für die Laufzeit finden.

Wenn diese obere Schranke korrekt ist, garantieren wir, dass -für eine **vorgegebene** Datenmenge- die Laufzeit unseres Algorithmus immer kleiner oder gleich dieser Schranke ist.

Beispiel: Die Anzahl der Objekte, die wir sortieren wollen

Die Anzahl der Listen, die wir verarbeiten wollen

u.s.w.

Die zu messenden Operationen

Der zweite Schritt unserer Analyse ist die **Bestimmung der abstrakten Operationen**, die wir messen wollen. D.h., Operationen, die mehrere kleinere Operationen zusammenfassen, welche einzeln in konstanter Zeit ausgeführt werden, aber den gesamten Zeitaufwand des Algorithmus durch ihr häufiges Vorkommen wesentlich mitbestimmen.

Die zu messenden Operationen

Beispiel: Bei Sortieralgorithmen messen wir Vergleiche

Bei anderen Algorithmen in imperativen Sprachen:

Speicherzugriffe
Anzahl der Multiplikationen
Anzahl der Bitoperationen
Anzahl der Schleifen-Durchgänge
Anzahl der Funktionsaufrufe
u.s.w.

In Funktionalen Programmiersprachen

Anzahl der Reduktionen

Die eigentliche Analyse

Hier wird eine mathematische Analyse durchgeführt, um die Anzahl der Operationen zu bestimmen.

Das Problem besteht darin, die beste obere Schranke zu finden, d.h. eine Schranke, die tatsächlich erreicht wird, wenn die ungünstigsten Eingabedaten vorkommen (**worst case**).

Die meisten Algorithmen besitzen einen Hauptparameter **n**, der die **Anzahl der zu verarbeitenden Datenelemente** angibt.

Die obere Schranke ist eine Funktion, die das Wachstum der Laufzeit in Abhängigkeit der Eingabegröße beschreibt (**T(n)**).

Oft ist es für die Praxis sehr nützlich, den mittleren Fall zu finden, aber meistens ist diese Berechnung sehr aufwendig.

O-Notation

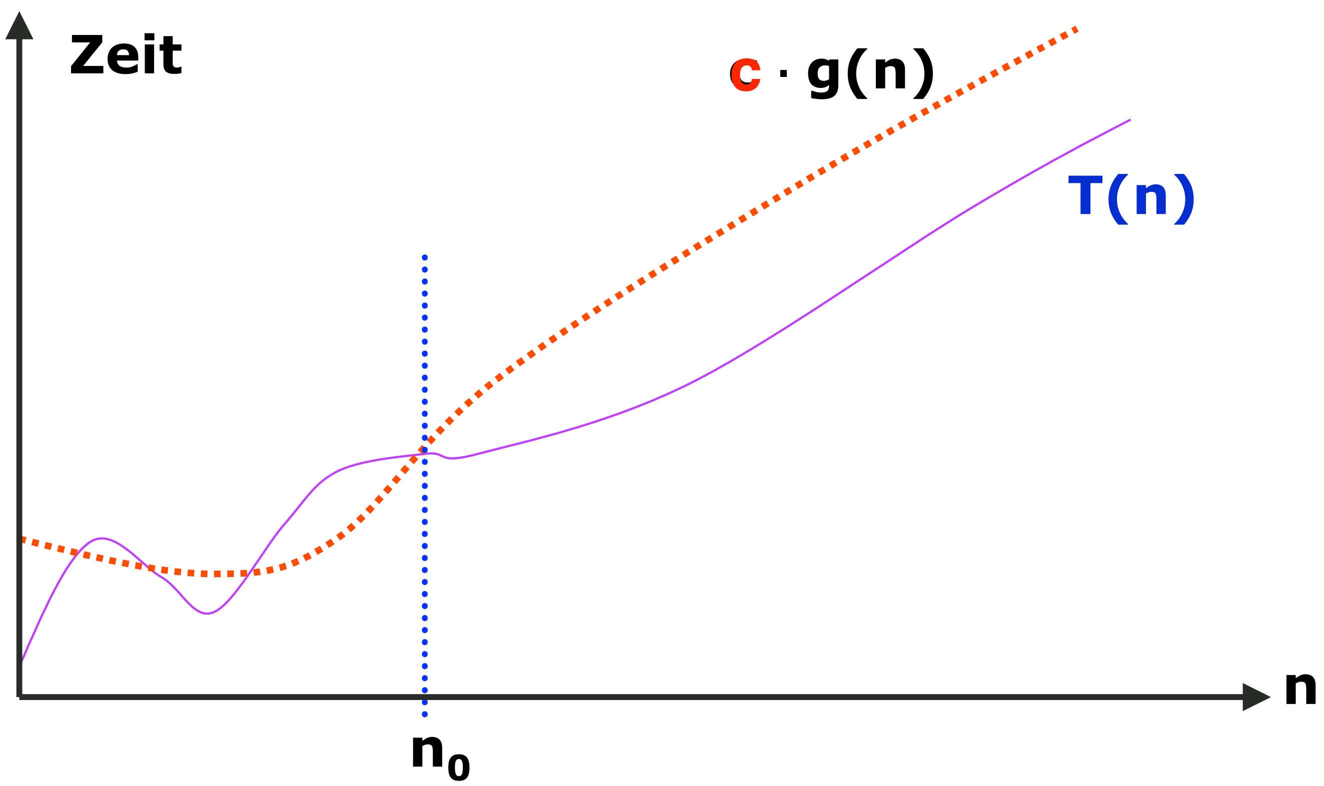
Für die Effizienzanalyse von Algorithmen wird eine spezielle mathematische Notation verwendet, die als **O-Notation** bezeichnet wird.

Die **O-Notation** erlaubt es, Algorithmen auf einer höheren Abstraktionsebene miteinander zu vergleichen.

Algorithmen können mit Hilfe der **O-Notation** unabhängig von Implementierungsdetails, wie Programmiersprache, Compiler und Hardware-Eigenschaften, verglichen werden.

Definition:

Die Funktion **T(n) = O(g(n))**, wenn es positive Konstanten **c** und $\mathbf{n_0}$ gibt, so dass $\mathbf{T(n) \leq c \cdot g(n)}$ für alle $\mathbf{n \geq n_0}$



Bedeutung der O-Notation

Wichtig ist, dass $\mathbf{O}(n^2)$ eine Menge darstellt, weshalb die Schreibweise $2n + n^2 \in \mathbf{O}(n^2)$ besser ist als die Schreibweise

$n^2 + 2n = \mathbf{O}(n^2)$

n^2 beschreibt die allgemeine Form der Wachstumskurve

$$n^2 + 2n = \mathbf{O}(n^2)$$

Bedeutet nicht "**=**" im mathematischen Sinn

deswegen darf man die
Gleichung nicht drehen!

$\boxed{\mathbf{O}(n^2) = n^2 + 2n}$ → **FALSCH!**

Freie Universität Berlin

Eigenschaften der O-Notation

Die **O**-Notation betont die dominante Größe

Beispiel: Größter Exponent

$$3n^3 + n^2 + 1000n + 500 = \mathbf{O}(n^3)$$

Ignoriert Proportionalitätskonstante

Ignoriert Teile der Funktion mit kleinerer Ordnung

Beispiel:

$$5n^2 + \log_2(n) = \mathbf{O}(n^2)$$

Teilaufgaben des Algorithmus mit kleinem Umfang

Bedeutung der O-Notation

Die Definition der **O**-Notation besagt, dass wenn **$T(n) = O(g(n))$**, ab irgendeinem n_0 die Gleichung $T(n) \leq c \cdot g(n)$ gilt.

Weil **T(n)** und **g(n)** Zeitfunktionen sind, ihre Werte also immer positiv sind, gilt:

$$\frac{T(n)}{g(n)} \leq c \quad \text{ab irgendeinem } n_0$$

Beide Funktionen können besser verglichen werden, wenn man den Grenzwert berechnet.

$$\lim_{n \to \infty} \frac{T(n)}{g(n)}$$

- Wenn der Grenzwert existiert, dann gilt: **$T(n) = O(g(n))$**
- Wenn der Grenzwert gleich **0** ist, dann bedeutet dies, dass **g(n)** sogar schneller wächst als **T(n)**. Dann wäre **g(n)** eine zu große Abschätzung der Laufzeit.

Freie Universität Berlin

Bedeutung der O-Notation

Beispiel: $100n^3 + 15n^2 + 15 = O(n^3)$?

$$\lim_{n \to \infty} \frac{100n^3 + 15n^2 + 15}{n^3} = 100$$

dann gilt: $\boxed{100n^3 + 15n^2 + 15 = \mathbf{O(n^3)}}$

Beispiel: $3n + 7 = \mathbf{O(n)} = O(n^2) = O(n^3)$

Ziel unserer Analyse ist es, die kleinste obere Schranke zu finden, die bei der ungünstigsten Dateneingabe vorkommen kann.

Klassifikation von Algorithmen

nach Wachstumsgeschwindigkeit

Komplexitätsklassen

konstant	$\mathbf{O}(1)$	
logarithmisch	$\mathbf{O}(\log_2 n)$	$\mathbf{O}(\log_e n)$
linear	$\mathbf{O}(n)$	
	$\mathbf{O}(n \log_2 n)$	
quadratisch	$\mathbf{O}(n^2)$	
kubisch	$\mathbf{O}(n^3)$	
exponentiell	$\mathbf{O}(2^n)$	$\mathbf{O}(10^n)$

O(1)

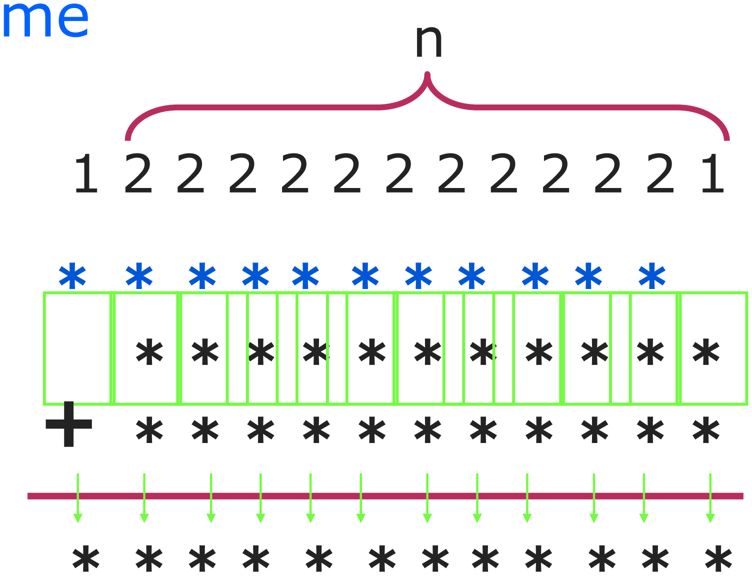
Die meisten Anweisungen in einem Programm werden nur einmal oder eine konstante Anzahl von Malen wiederholt.

Wenn alle Anweisungen des Programms diese Eigenschaft haben, spricht man von konstanter Laufzeit.

Beste Zielzeit bei der Entwicklung von Algorithmen!

Summe und Multiplikation in der Schule

Summe



Eingabegröße:

n = Zahlenbreite

Berechnungsschritt:

Addition von zwei Ziffern

Komplexitätsanalyse:

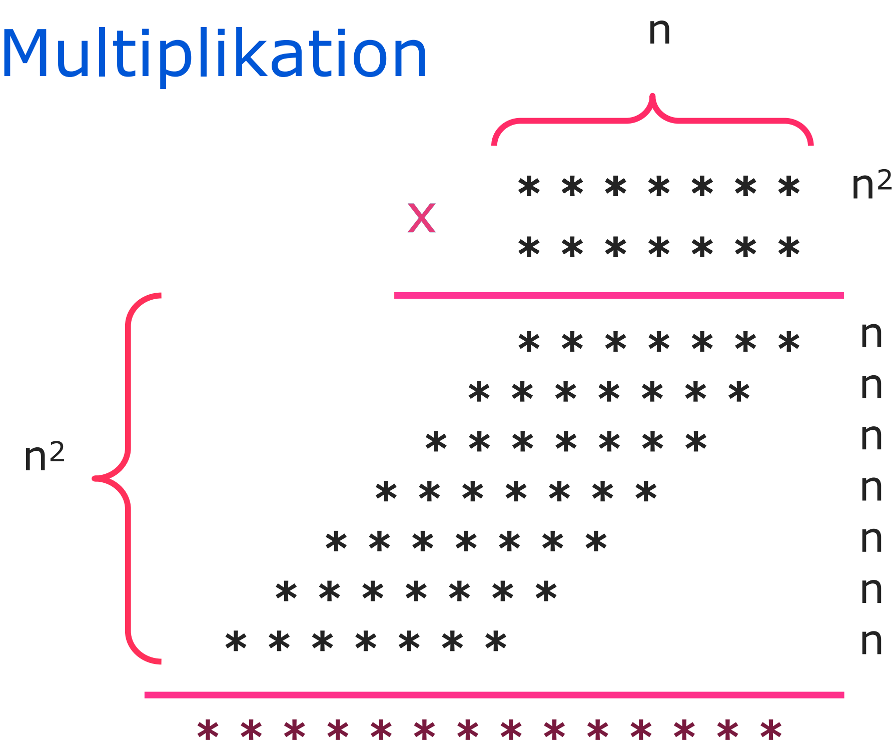
T(n) = Anzahl der Berechnungsschritte, um zwei Zahlen mit **n** Ziffern zu addieren

Im schlimmsten Fall:

T(n) = 2n **T(n)** ist eine lineare Funktion

$\boxed{O(n)}$

Multiplikation



Multiplikation und Addition von Ziffern

Eingabegröße:

n = Anzahl der Ziffern

Berechnungsschritt:

Multiplikation von zwei Ziffern

Komplexitätsanalyse:

$T(n) = n^2$

Im schlimmsten Fall

keine Nullen

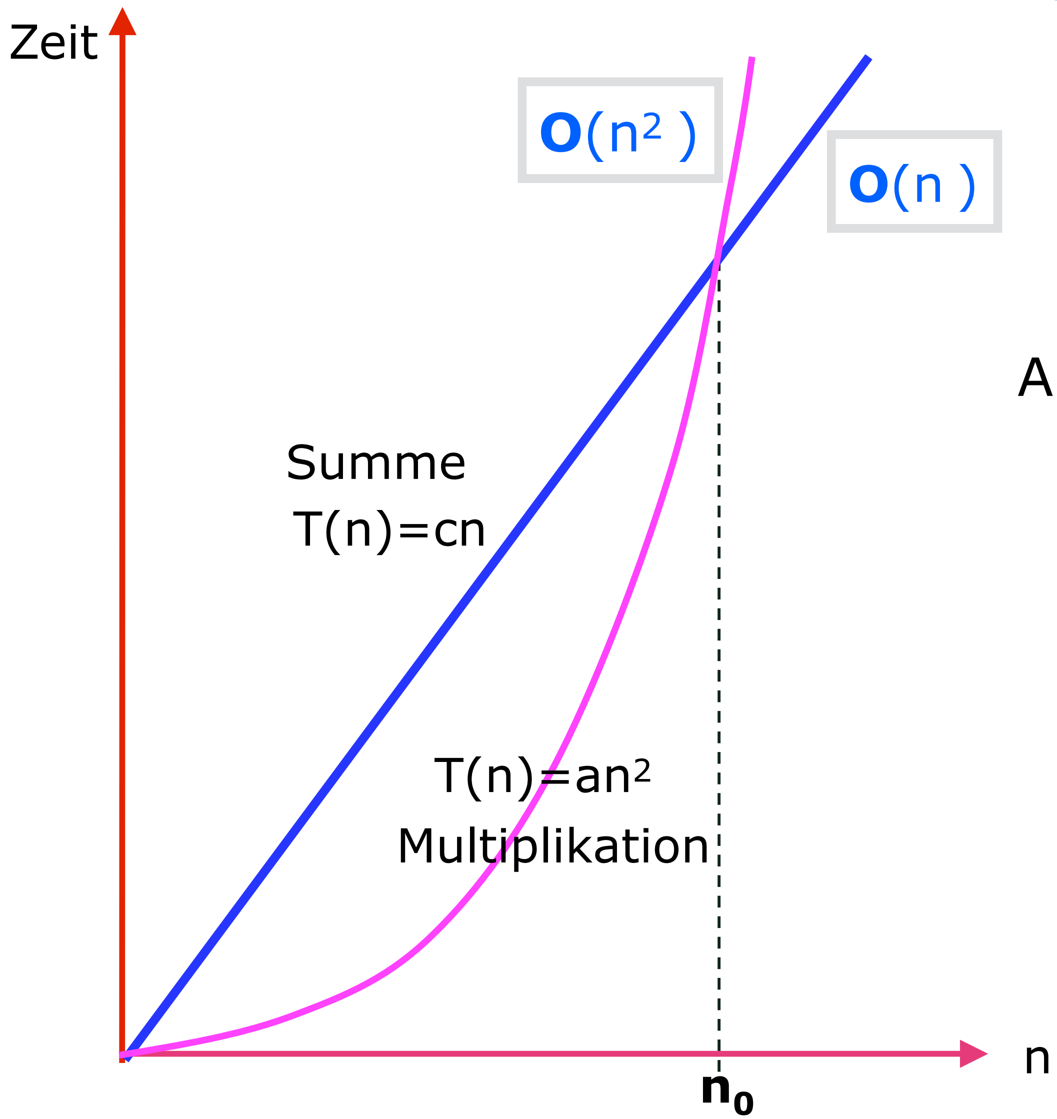
immer ein Übertrag

$T(n) = n^2 + cn^2 = (1+c)\, n^2 = an^2$

T(n) ist eine quadratische Funktion

$\boxed{\mathbf{O}(n^2)}$

Summe und Multiplikation



Ab einem bestimmten n_0 gilt **c**n < **a**n²

Freie Universität Berlin

Multiplikation komplexer Zahlen

$$(a + bi)\ (c + di) = (ac + adi + cbi + bdi^2)$$
$$= (ac - bd) + (ad + cb)i$$

Eingabe: a, b, c, d

2 Summen

Ausgabe: (ac – bd), (ad + cb)

4 Multiplikationen

Eine Multiplikation kostet **1 Euro**

Eine Summe kostet **1 Cent**

Zwei Komplexe Zahlen zu multiplizieren kostet

4,02 €

Algorithmus von Gauß

$$(a + b\mathit{i}) \; (c + d\mathit{i}) \;\; = (ac - bd) + (ad + cb)\mathit{i}$$

$$\underbrace{\overset{1}{a + b}}_{r_1} \qquad \underbrace{\overset{2}{c + d}}_{r_2} \qquad \overset{2}{ac} \xrightarrow{} r_4 \qquad \overset{3}{bd} \xrightarrow{} r_5$$

$$\underbrace{r_1 \overset{1}{\;} r_2}_{r_3}$$

$$\underbrace{r_4 \overset{3}{-} r_5}_{\parallel} \qquad \underbrace{r_3 \overset{4}{-} r_4 \overset{5}{-} r_5}_{\parallel}$$

$$\overbrace{(ac - bd)}, \qquad \overbrace{(ad + cb)}$$

$$r_3 \;=\; r_1 \, r_2$$
$$= ac + ad + cb + bd$$

$$r_3 - r_4 - r_5 = \cancel{ac} + ad + cb + \cancel{bd} - \cancel{ac} - \cancel{bd}$$
$$= ad + cb$$

5 Summen

3 Multiplikationen

Zwei Komplexe Zahlen zu multiplizieren kostet

3,05 €

Die Funktion **sum** berechnet für ein gegebenes **n>0** die Summe aller Zahlen von **1** bis **n**

Iterativ (nicht rekursiv)

```
def sum(n):
    sum = 0
    for i in range(n+1):
        sum = sum + i
    return sum
```

c_2

c_1

c_1 = Zeit (**sum=0** und **return sum**)

c_2 = Zeitkosten eines Schleifendurchgangs

$$T(n) = c_1 + c_2(n+1)$$

$$T(n) = \cancel{c_1} + \cancel{c_2} + \cancel{c_2}n$$

$$T(n) = O(n)$$

Rekursiv

```
def sum_rec(n):
    if n==0:
        return 0
    else:
        return n + sum_rec(n-1)
```

c_1

c_1 = Zeit (**return 0**)

c_2 = Zeitkosten eines Funktionsaufrufs

$$T(n) = c_1 + c_2(n-1)$$

$$T(n) = c_1 - c_2 + c_2n$$

$$T(n) = O(n)$$

Komplexität eines Algorithmus

O-Notation

Obere Komplexitätsgrenze (höchstens)

Ω–Notation

Untere Komplexitätsgrenze (mindestens)

θ-Notation

Genaue Komplexität (genau)

Freie Universität Berlin

Komplexität eines Algorithmus

Definitionen:

Ω–Notation

Die Funktion $\mathbf{T(n)} = \Omega\mathbf{(g(n))}$, wenn es positive Konstanten $\mathbf{c}$ und $\mathbf{n_0}$ gibt, so dass $\mathbf{T(n)} \geq \mathbf{c} \times \mathbf{g(n)}$ für alle $\mathbf{n} \geq \mathbf{n_0}$

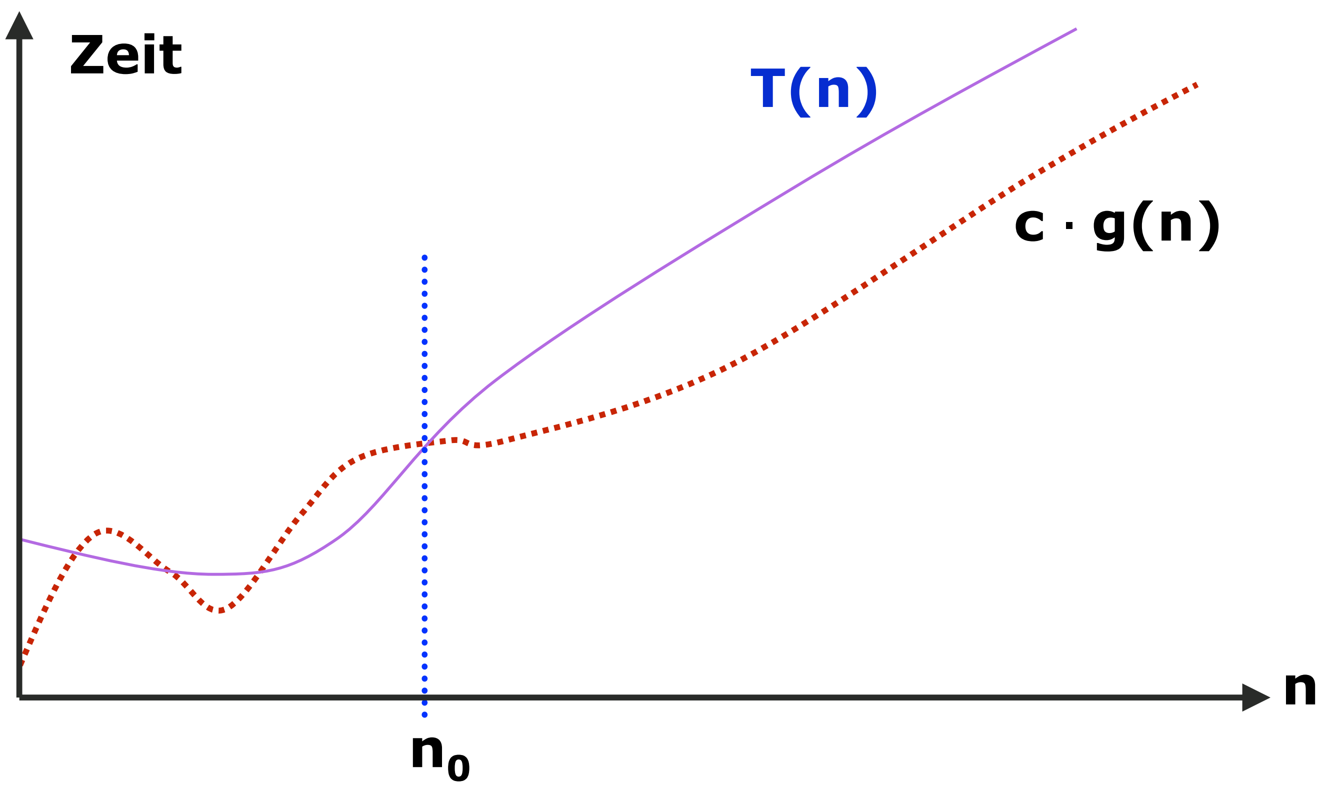
θ-Notation

Die Funktion $\mathbf{T(n)} = \theta\mathbf{(g(n))}$ genau dann, wenn

$$\mathbf{T(n)} = \mathbf{O(g(n))} \text{ und } \mathbf{T(n)} = \Omega\mathbf{(g(n))}$$

Ω–Notation

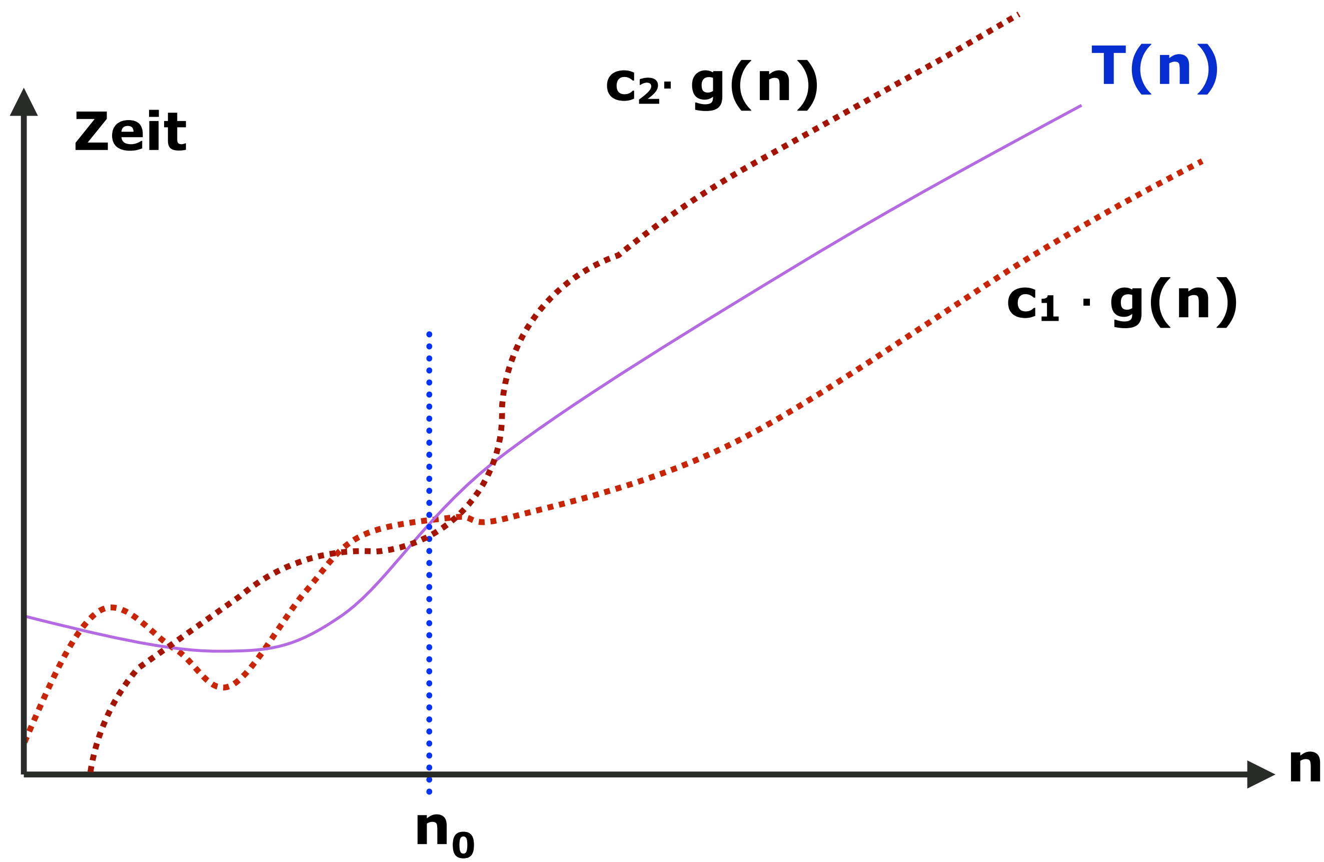
Die Funktion **$T(n) = \Omega(g(n))$**, wenn es positive Konstanten **c** und **n_0** gibt, so dass **$T(n) \geq c \times g(n)$** für alle **$n \geq n_0$**



θ-Notation

Die Funktion **$T(n) = \theta(g(n))$** genau dann, wenn

$T(n) = O(g(n))$ und **$T(n) = \Omega(g(n))$**



Die Funktion **sum** berechnet für ein gegebenes **n>0** die Summe aller Zahlen von **1** bis **n**

Iterativ (nicht rekursiv)

```python
def sum(n):
    sum = 0                      # c1
    for i in range(n+1):         # c2
        sum = sum + i            # c2
    return sum                   # c1
```

c_1 = Zeit (**sum**=0 und **return sum**)

c_2 = Zeitkosten eines Schleifendurchgangs

$T(n) = c_1 + c_2(n+1)$

$T(n) = \cancel{c_1} + \cancel{c_2} + \cancel{c_2}n$

$T(n) = O(n)$

Rekursiv

```python
def sum_rec(n):
    if n==0:                     # c1
        return 0                 # c1
    else:
        return n + sum_rec(n-1)
```

c_1 = Zeit (**if** n==0: **return 0**)

c_2 = Zeit (**if** n!=0: **return n + sum_rec(n-1)**)

$T(n) = c_1 + c_2(n-1)$

$T(n) = c_1 - c_2 + c_2 n$

$T(n) = O(n)$

Die Funktion **sum** berechnet für ein gegebenes **n**>**0** die Summe aller Zahlen von **1** bis **n**

direkt

Formel von Gauß

```
def sum(n):
    return n*(n+1)//2
```

$T(n) = \mathbf{O}(1)$

Warum ist Rekursion ineffizient?

Eine rekursive Funktion verursacht eine Kette von Funktionsaufrufen

Eine Funktion arbeitet in ihrer eigenen **lokalen Umgebung**

- **Werte aller lokaler Variablen**
- **Stelle, an der die Ausführung der Funktion sich gerade befindet**

Wenn innerhalb einer Funktion **f (...)** eine Funktion **g (...)** aufgerufen wird:

- **die gesamte lokale Umgebung von f wird gespeichert**
- **die Werte der Parameter von g werden gesetzt**
- **das Programm springt zum Anfang der Funktion g und die Funktion g wird entsprechend ausgeführt**
- **das Programm springt zurück zu f und das Ergebnis der Funktion g wird an f übergeben**
- **die gesamte Umgebung von f wird zurückgesetzt**
- **und die Ausführung der Funktion f wird fortgesetzt**

Implementierung der Funktion Fakultät

Fakultät (0) = 1
Fakultät (n) = n · Fakultät (n-1)

Rekursive Implementierung

```
def factorial (n):
    if  n<=0:
        return 1
    else:
        return n * factorial(n-1)
```

Iterative Implementierung

```
def factorial (n ):
    if (n<=0):
        return 1
    else:
        factor = 1
        for  i in range(2, n+1):
            factor = factor * i
        return factor
```

Rekursionsarten

Lineare Rekursion

Rekursive Funktionen, die in jedem Zweig ihre Definition maximal einen rekursiven Aufruf beinhalten, werden als **linear rekursiv** bezeichnet.

Beispiel:

$$factorial(n) = \begin{cases} 1 & , falls \quad n \leq 1 \\ n \cdot factorial(n-1) & , \quad sonst \end{cases}$$

Endrekursion (*tail recursion*)

Linear rekursive Funktionen werden als endrekursive Funktionen klassifiziert, wenn der rekursive Aufruf in jedem Zweig der Definition die letzte Aktion zur Berechnung der Funktion ist. D.h. keine weiteren Operationen müssen nach der Auswertung der Rekursion berechnet werden.

Eine **nicht** endrekursive Funktion ist folgende Definition der Fakultätsfunktion:

start ()

factorial (5)

factorial (4)*5

factorial (3)*4*5

factorial (2)*3*4*5

factorial (1)*2*3*4*5

factorial (0)*1*2*3*4*5

1*1*2*3*4*5

1*2*3*4*5

2*3*4*5

6*4*5

24*5

120

zurück in start

Die Endberechnungen finden erst beim Abbau des Ausführungsstapels statt.

return-Adresse in factorial n = 1
return-Adresse in factorial n = 2
return-Adresse in factorial n = 3
return-Adresse in factorial n = 4
return-Adresse in factorial n = 5
return-Adresse in start lokaler Variablen von start

Laufzeitkeller

Endrekursive Definition der Fakultätsfunktion

Python:

```
def factorial (n):

        def factorial_helper (a, n):
                if n==0:
                        return a
                else:
                        return factorial_helper (a*n, n-1)

        return factorial_helper (1, n)
```

Berechnung der Fibonacci-Zahlen

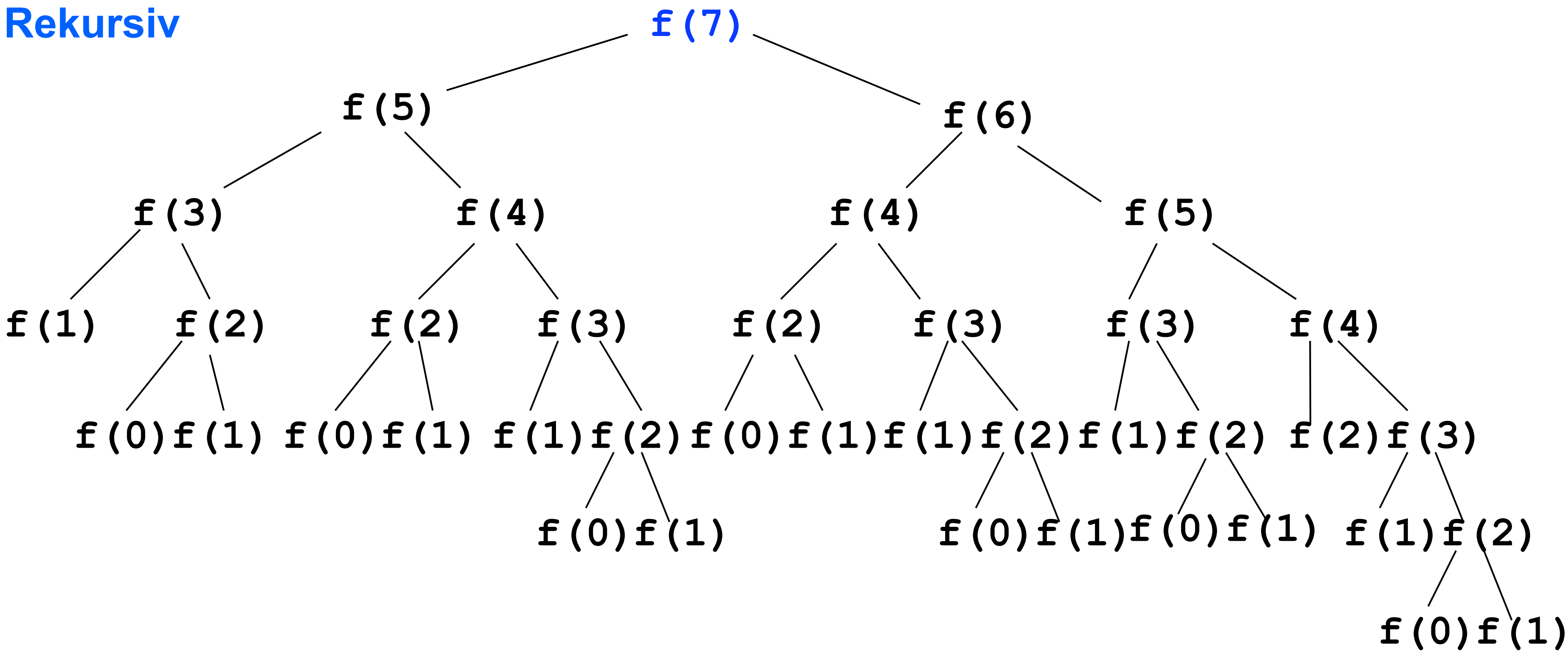
Rekursiv

```
def  fib (n):
        if n==0  or  n==1:
             return n
        else:
             return  fib(n-2) + fib(n-1)
```

Die rekursive Berechnung der Fibonacci-Zahlen hat eine exponentielle Komplexität $\mathbf{O}((1{,}618...)^n)$

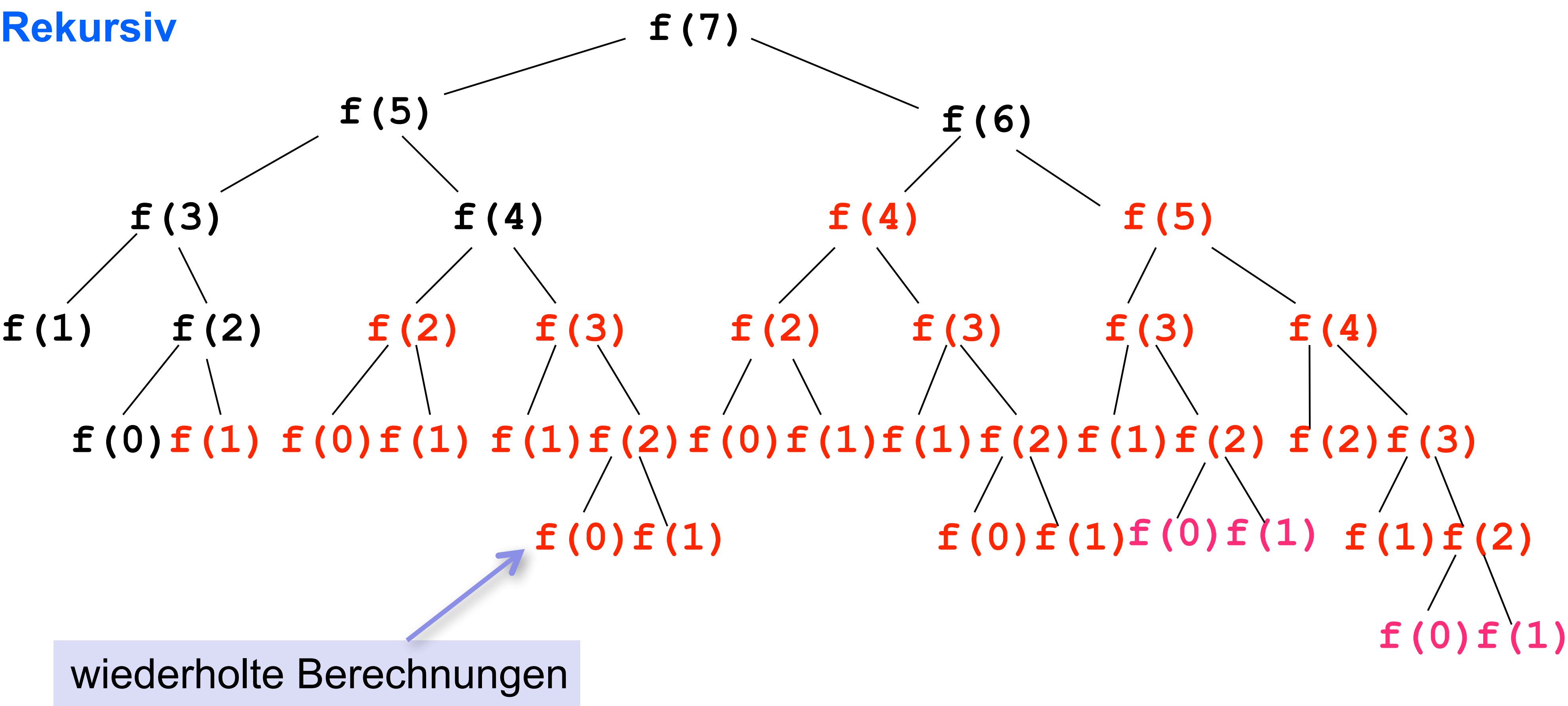
Berechnung der Fibonacci-Zahlen

Rekursiv



Berechnung der Fibonacci-Zahlen

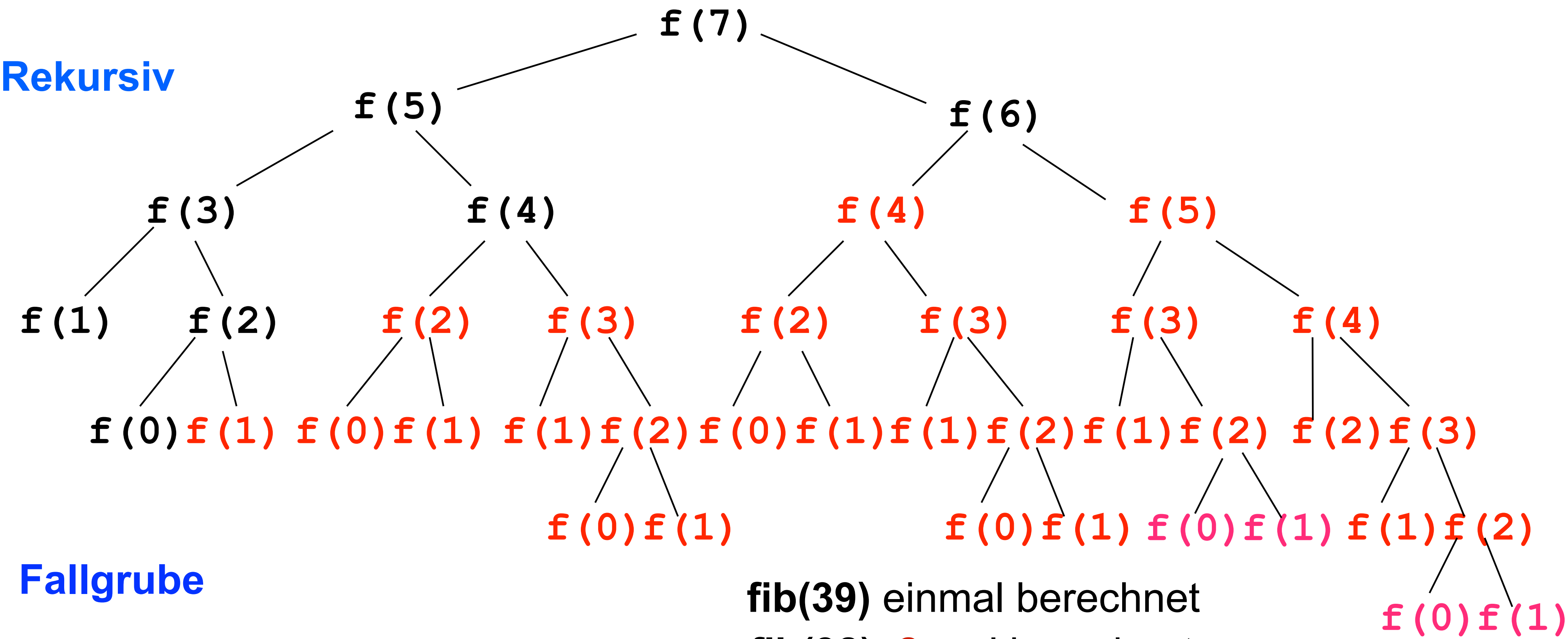
Rekursiv



Berechnung der Fibonacci-Zahlen

Freie Universität Berlin

Rekursiv



Fallgrube

Wenn wir **fib(40)** mit unserer rekursiven Implementierung berechnen,

wird:

fib(39) einmal berechnet

fib(38) **2** mal berechnet

fib(37) **3** mal berechnet

fib(36) **5** mal berechnet

fib(35) **8** mal berechnet

. . .

fib(0) **165 580 141** mal berechnet

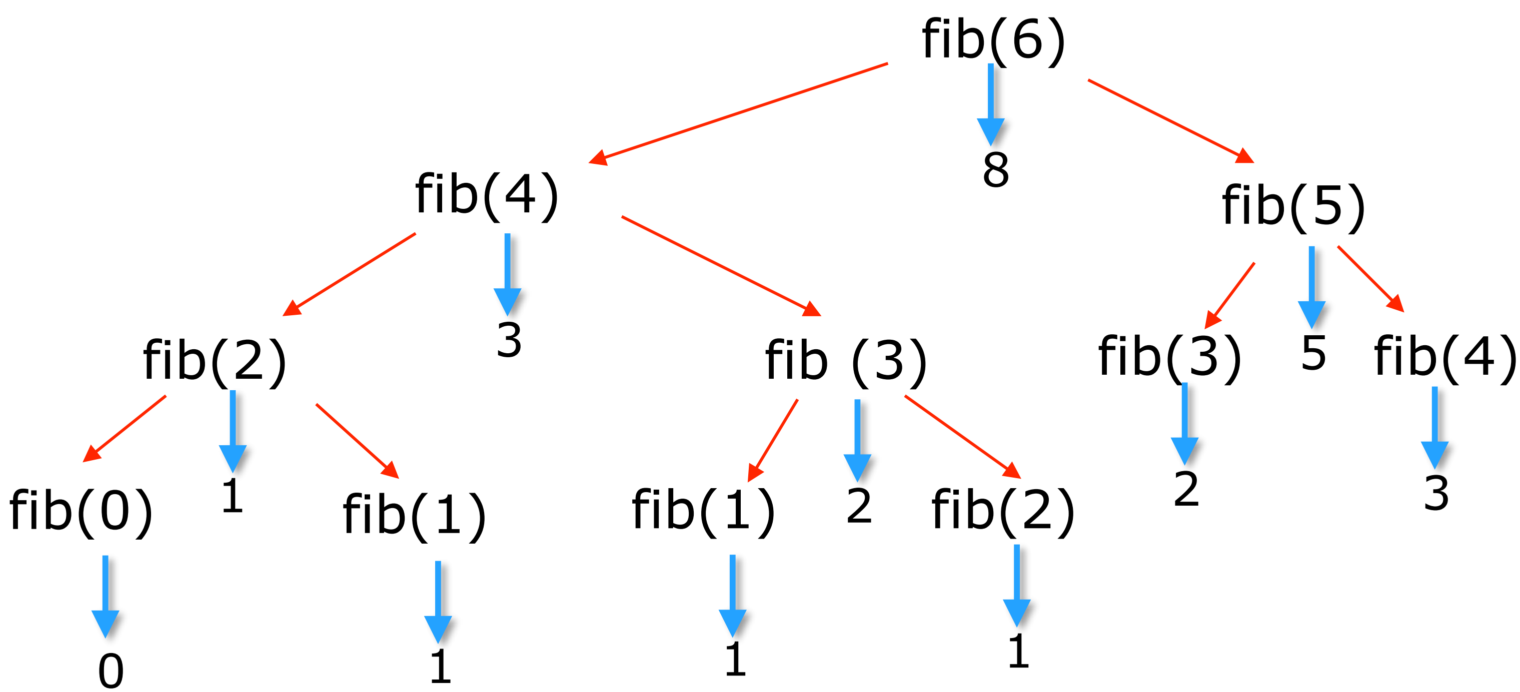
Beim Aufruf von **fib(40)** werden **331 160 281** Funktionsaufrufe gemacht

Berechnung der Fibonacci-Zahlen

Lösung mit **Dynamischer Programmierung**

fibs-Tabelle

0	0
1	1
2	1
3	2
4	3
5	5
6	8



Berechnung der Fibonacci-Zahlen

```
def fib_dyn(n):
    if n==0 or n==1:
        return n
    else:
        fibs = [0 for i in range(n+1)]
        fibs[1] = 1
        return fib_help(fibs, n)

def fib_help (fibs, n):
    if  fibs[n] != 0:
        return  fibs[n]
    elif n==0:
        return 0
    else:
        fibs[n] = fib_help(fibs, n-1) + fib_help(fibs, n-2)
        return  fibs[n]
```

mit **Dynamischer Programmierung**

$T(n) = \mathbf{O}(n)$

```
>>> fib_dyn(992)
9252394159943865548695885305267321133917910271071460896757822139976047281321590991446751768798293528186087306538837695052181861570099637479324274102244407091426856700404126193197000446025873788552108230822 9
>>>  fib_dyn(993)
. . .
RuntimeError: maximum recursion depth exceeded in comparison
```

Berechnung der Fibonacci-Zahlen

Eine rekursive Implementierung kann extrem ineffizient werden, wenn die gleichen Berechnungen wiederholt berechnet werden.

Eine Lösung ist **Dynamische Programmierung**

Bei dynamischer Programmierung werden Zwischenberechnungen in einer Tabelle gespeichert, damit diese später wieder verwendet werden können.

Berechnung der Fibonacci-Zahlen

```
def fib_end_rek(n):
    def quick_fib(a, b, n):
        if n==0:
            return a
        else:
            return quick_fib(b, a+b, n-1)
    return quick_fib(0, 1, n)
```

Endrekursiv

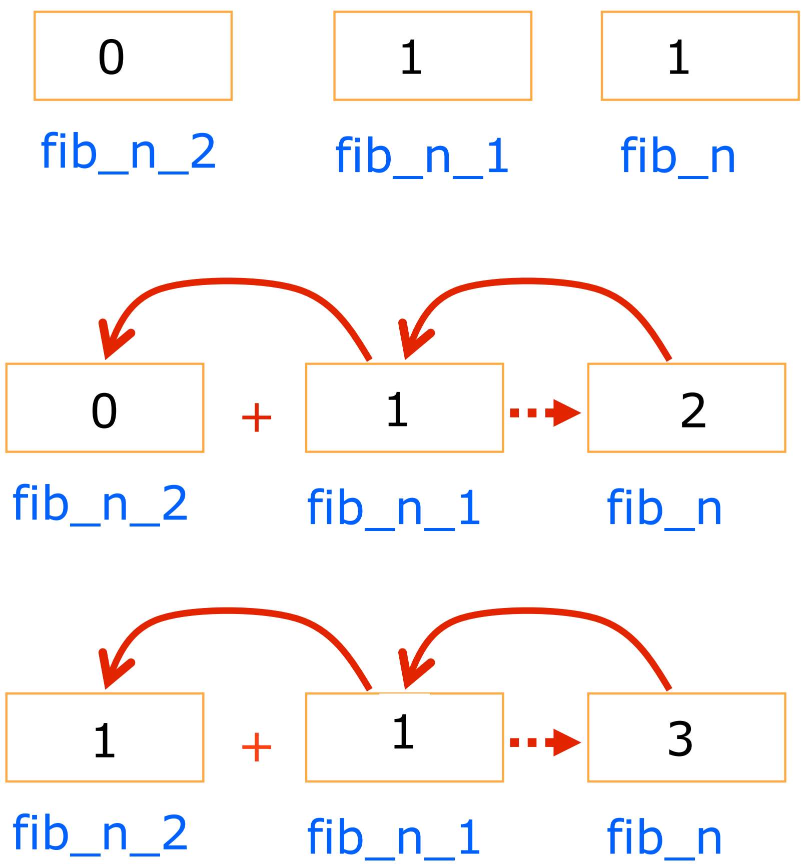
$T(n) = \mathbf{O}(n)$

```
>>> fib_end_rek (991)
5718294068156339795296436970062730451068459807489911120716730387437140314978877390230916107697646273077726548022987843618034217471145712656905194499158734521641931742934079402019778977169370976041642881309090
>>> fib_end_rek (992)
. . .
RuntimeError: maximum recursion depth exceeded in comparison
```

Freie Universität Berlin

Berechnung der Fibonacci-Zahlen

Iterativ




```
def fib_iter(n):
    if n==0 or n==1:
        return n
    elif n==2:
        return 1
    else:
        fib_n_2 = 0
        fib_n_1 = 1
        fib_n = 1
        for i in range(2, n):
            fib_n_2 = fib_n_1
            fib_n_1 = fib_n
            fib_n = fib_n_1 + fib_
        return fib_n
```

```
>>> fib_iter(5000)
3878968454388325633701916308325
9053120821277146462451061605972
1489555013904403709701082291646
2210669479293452858882973813483
1020089549829403614301569114789
3836421656394410691021450563413
3706558656238254656700712525929
9038549338139288363783475189087
6297071203333705292310769300851
8093849801803847813996748881765
5546537882916442689129803846137
7896902150229308247566634622492
3071883324803280375039130352903
3045058427011476352422702109346
3769910400671417488329842289149
1273104054328753298044273676822
9772449877498745556919077038806
3704683279481135897373999311010
6219308149018570815397854379195
3056175107610530756887837660336
6735544525884488624161921055345
7493675897849027988234351023599
8446639348532564119522218595630
6047536464547076033090242080638
2584929156452876291575759142343
8091423029174910889841552098544
3248659407979357131684169286803
9545309545388698114665082066862
8974206393234384884652409887423
9587380197699382031717420893226
5468879364002630797780058759129
6713896342142525791168727556003
6031137054775472460463998758804
6985178408674382863125
>>>
```

```
def fib_iter(n):
    if n==0 or n==1:
        return n
    elif n==2:
        return 1
    else:
        fib_n_2 = 0
        fib_n_1 = 1
        fib_n = 1
        for i in range(2, n):
            fib_n_2 = fib_n_1
            fib_n_1 = fib_n
            fib_n = fib_n_1 + fib_n_2
        return fib_n
```

c_1

c_2

Komplexitätsanalyse

Iterativ

$$T(n) = c_1 + c_2\,(n-3)$$

c_2 = Zeitkosten eines Schleifedurchgangs

$$T(n) = c_1 - 3c_2 + c_2 n$$

$$T(n) = O(n)$$

Freie Universität Berlin

yield-Anweisung

```
def fibonacci():
    """Unendlicher Fibonacci-Zahlen-Generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def getFibonacci(n):
    counter = 0
    for x in fibonacci():
        counter += 1
        if (counter > n):
            break
    return x

print(getFibonacci(10))
```

Rekursion vs. Iteration

Jede interative Funktion kann in eine rekursive Funktion umgeschrieben werden.

Sehr wichtig ist es, bei rekursiven Funktionen die Abbruchbedingung korrekt zu programmieren

Endrekursive Funktionen können als Iteration umgeschrieben werden.

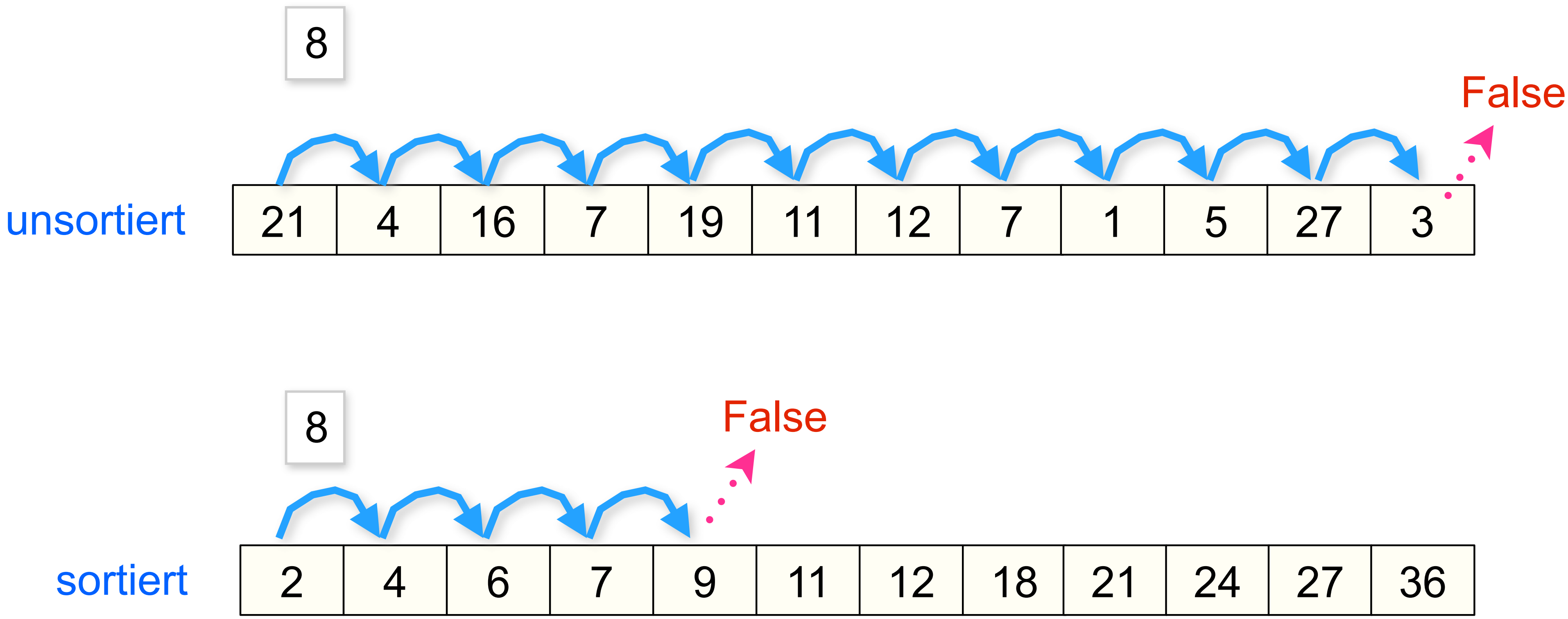
Der Compiler implementiert Rekursion mit Hilfe des Stapels

Die Rekursion kann in eine Iteration verwandelt werden, wenn ein Stapel explizit verwendet wird

Wann sollen wir Iteration und wann Rekursion verwenden?

Einfache und übersichtliche Implementierung	**Rekursion**
Effiziente Implementierung	**Iteration**

Suchen


8
False
unsortiert
21 4 16 7 19 11 12 7 1 5 27 3
8
False
sortiert
2 4 6 7 9 11 12 18 21 24 27 36

Lineare Suche in Python

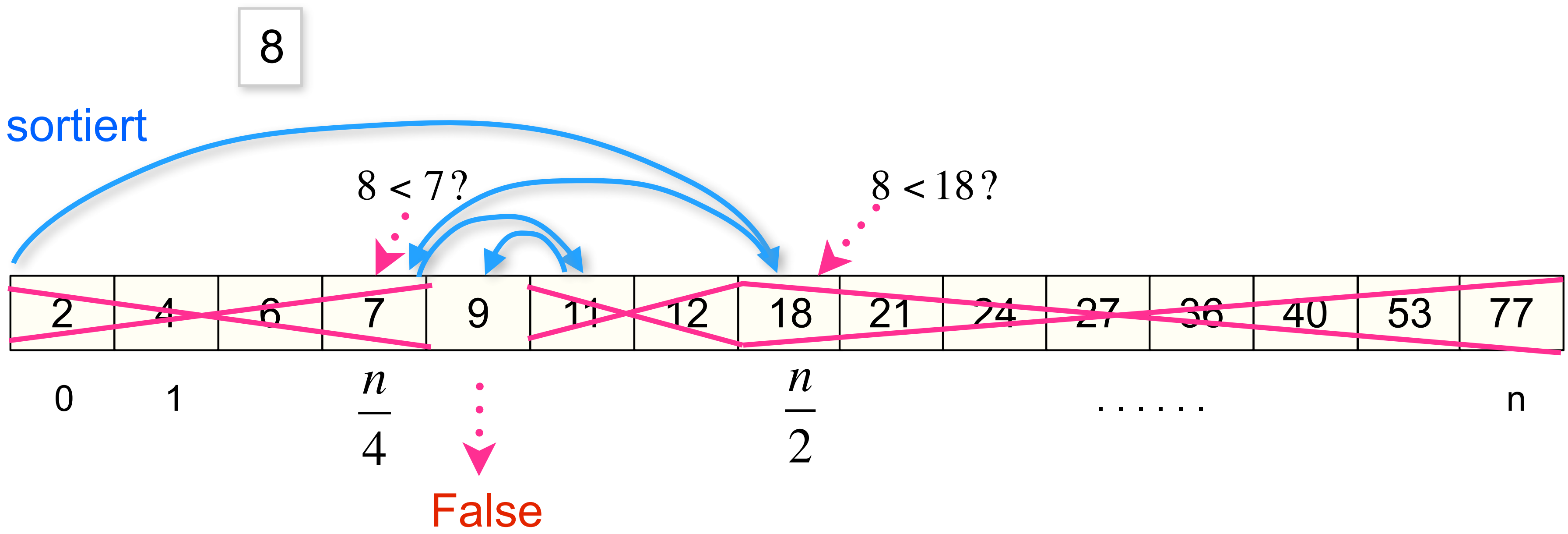
```
def linear_search(key, seq ):
        for i in seq:
                if i==key:
                        return True
        return False
```

Rekursive lineare Suche in Python

```
def linear_search(key, seq ):

    if len(seq)==0:

        return False

    elif seq[0]==key:

        return True

    else:

        return linear_search(key, seq[1:])
```

Suchen in sortierten Listen

Binärsuche



nur **4** Schritte

Gut für imperative Programmiersprachen!

Binäre Suche

Rekursiv

```
def bin_search(key,seq):
    if len(seq)>1:
        m = len(seq)//2
        if seq[m]==key:
            return True
        elif key<seq[m]:
            return bin_search(key, seq[0:m])
        else:
            return bin_search(key, seq[(m+1):])
    elif len(seq)==1:
        return seq[0]==key
    else:
        return False
```

Maximale Schrittanzahl mit Arrays

$n = 128$	$= 2^7$
64	$= 2^6$
32	$= 2^5$
16	$= 2^4$
8	$= 2^3$
4	$= 2^2$
2	$= 2^1$
1	$= 2^0$

$7 = \log_2(128)$

Im schlimmsten Fall

$\approx$ **$\log_2(n)$**

Anzahl der Elemente	Lineare Suche	Binäre Suche
3	3	2
...	...	...
1,023	1,023	10
...	...	...
1,073,741,824	1,073,741,824	30

Iterativ

2 ← lowerBound = 0
3
4
6
7
8
9
8 $\stackrel{?}{==}$ 10 ← current = (lowerBound+upperBound)/2
12
14
15
19
27
28
33
36 ← upperBound = size-1

Iterativ

2
3
4
6
7
8
9
~~10~~
~~12~~
~~14~~
~~15~~
~~19~~
~~27~~
~~28~~
~~33~~
~~36~~

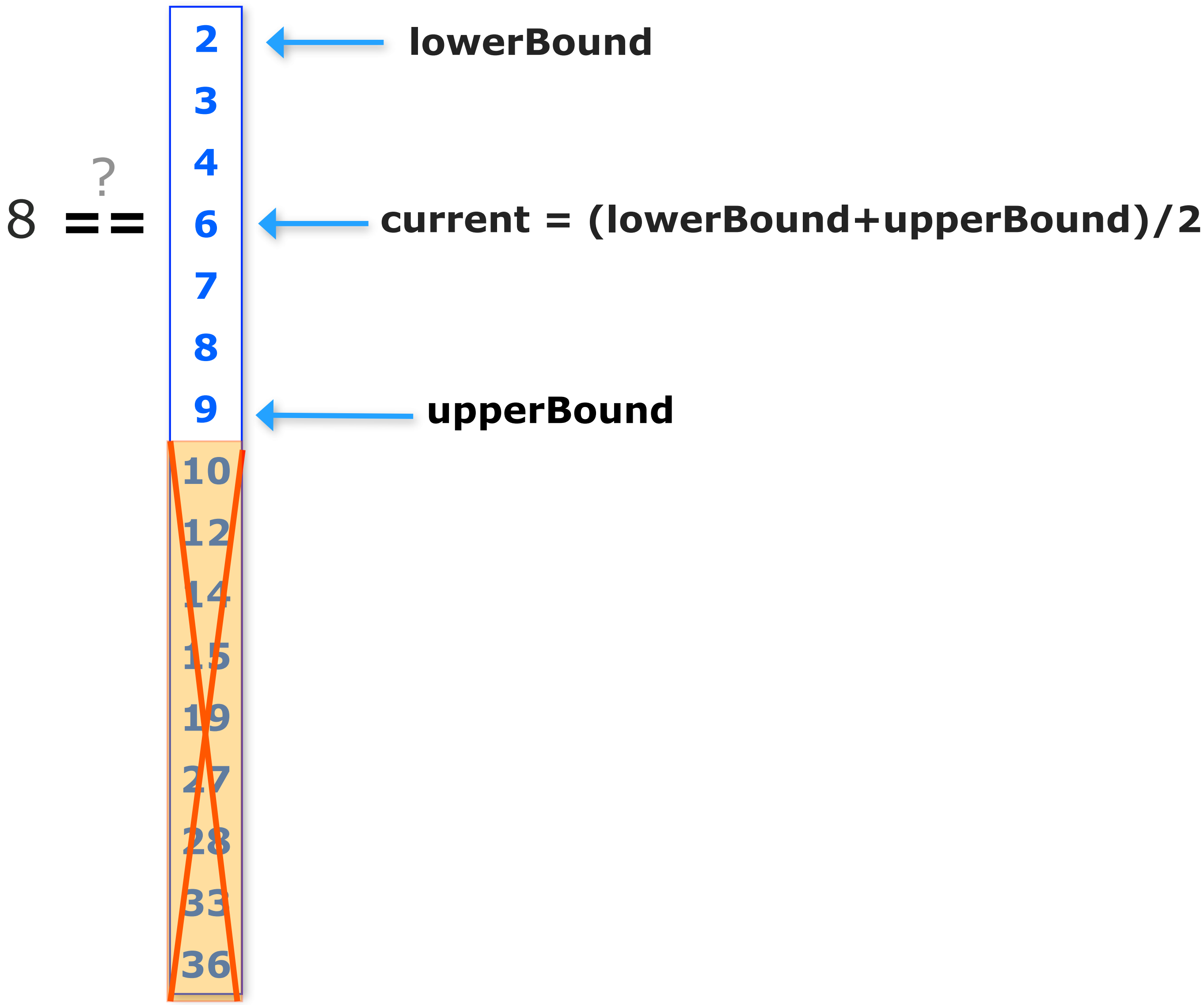
lowerBound = 0

$8 \stackrel{?}{==}$ 10

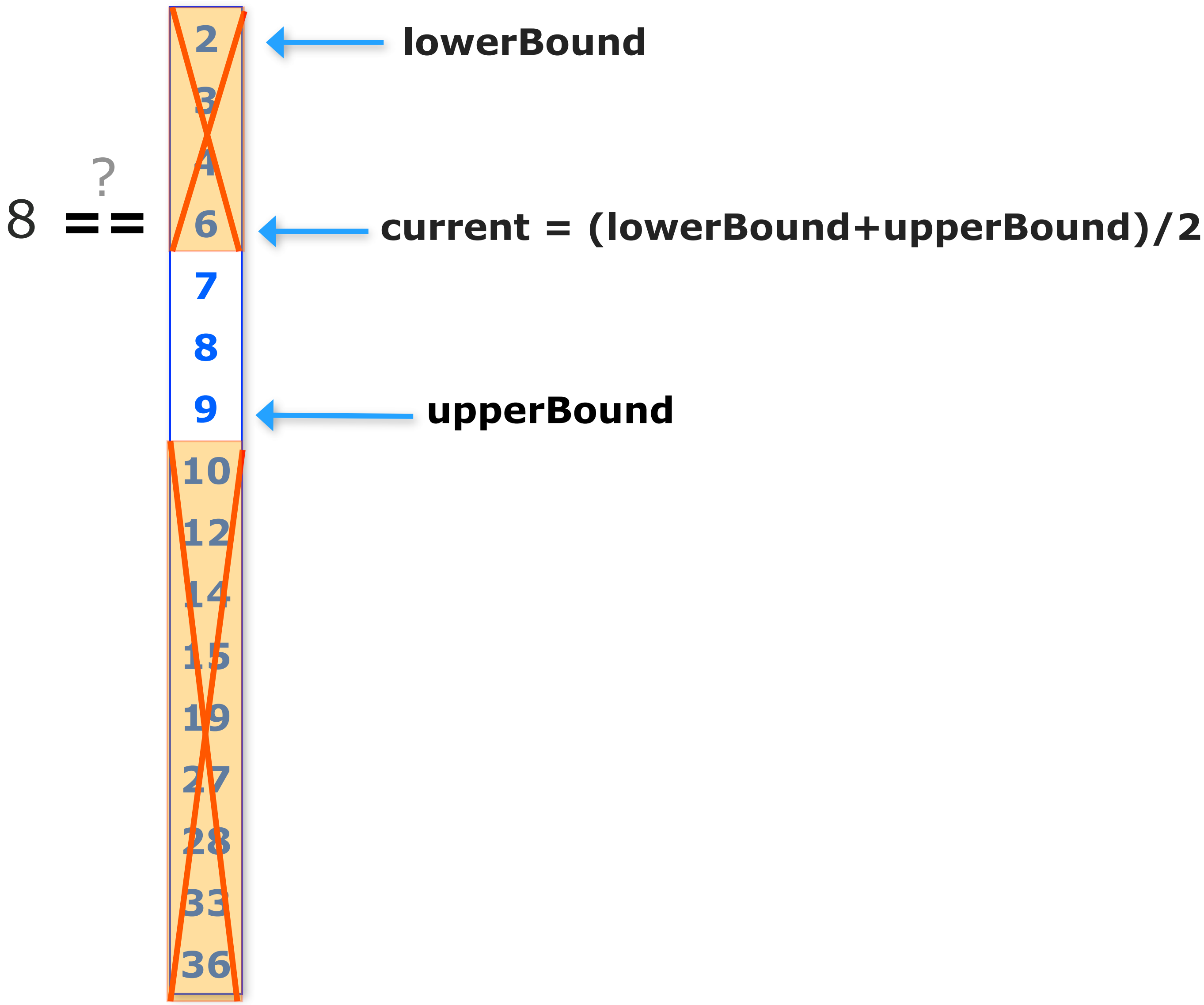
current = (lowerBound+upperBound)/2

upperBound = size-1

Iterativ

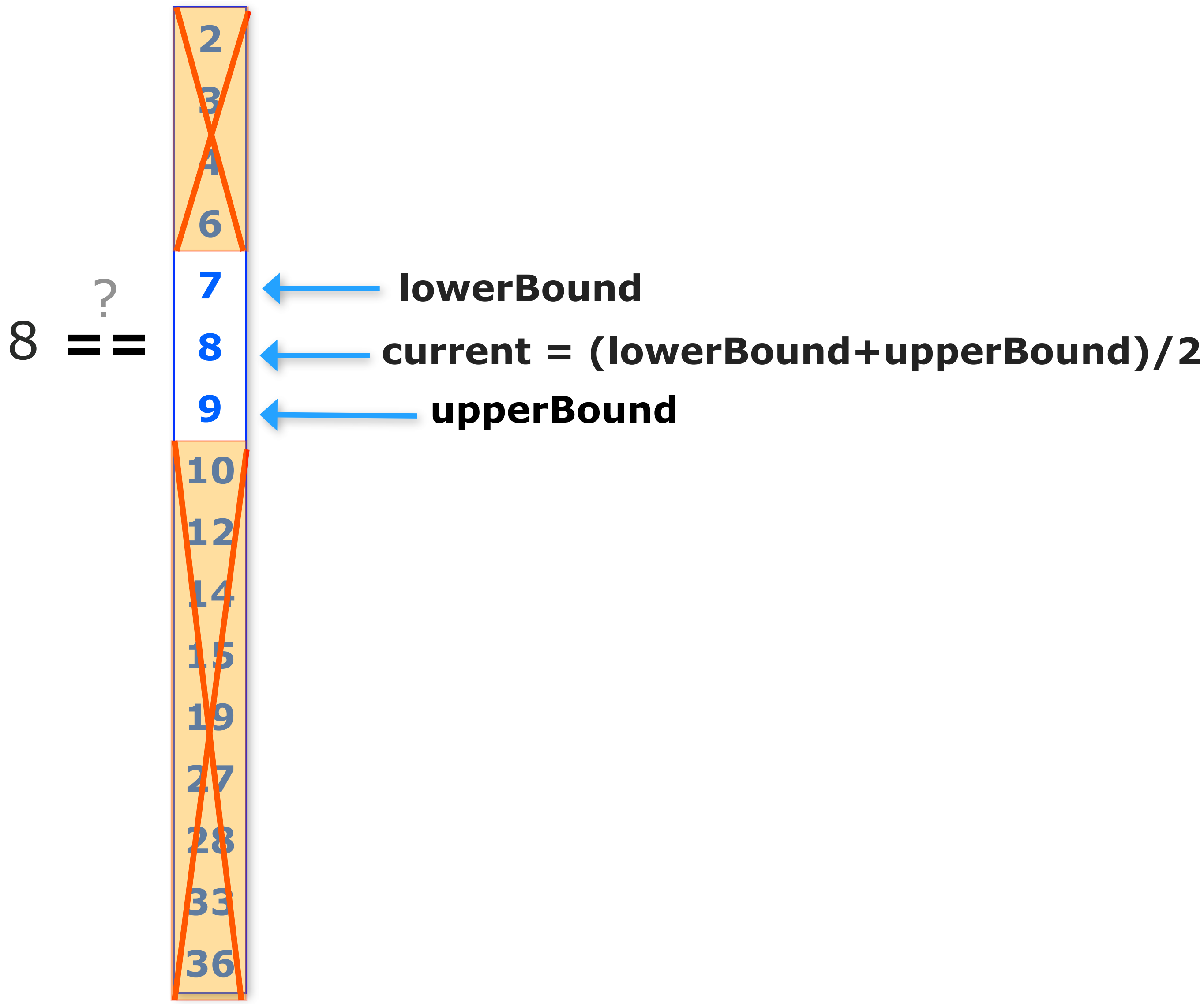

8 ==?
2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ

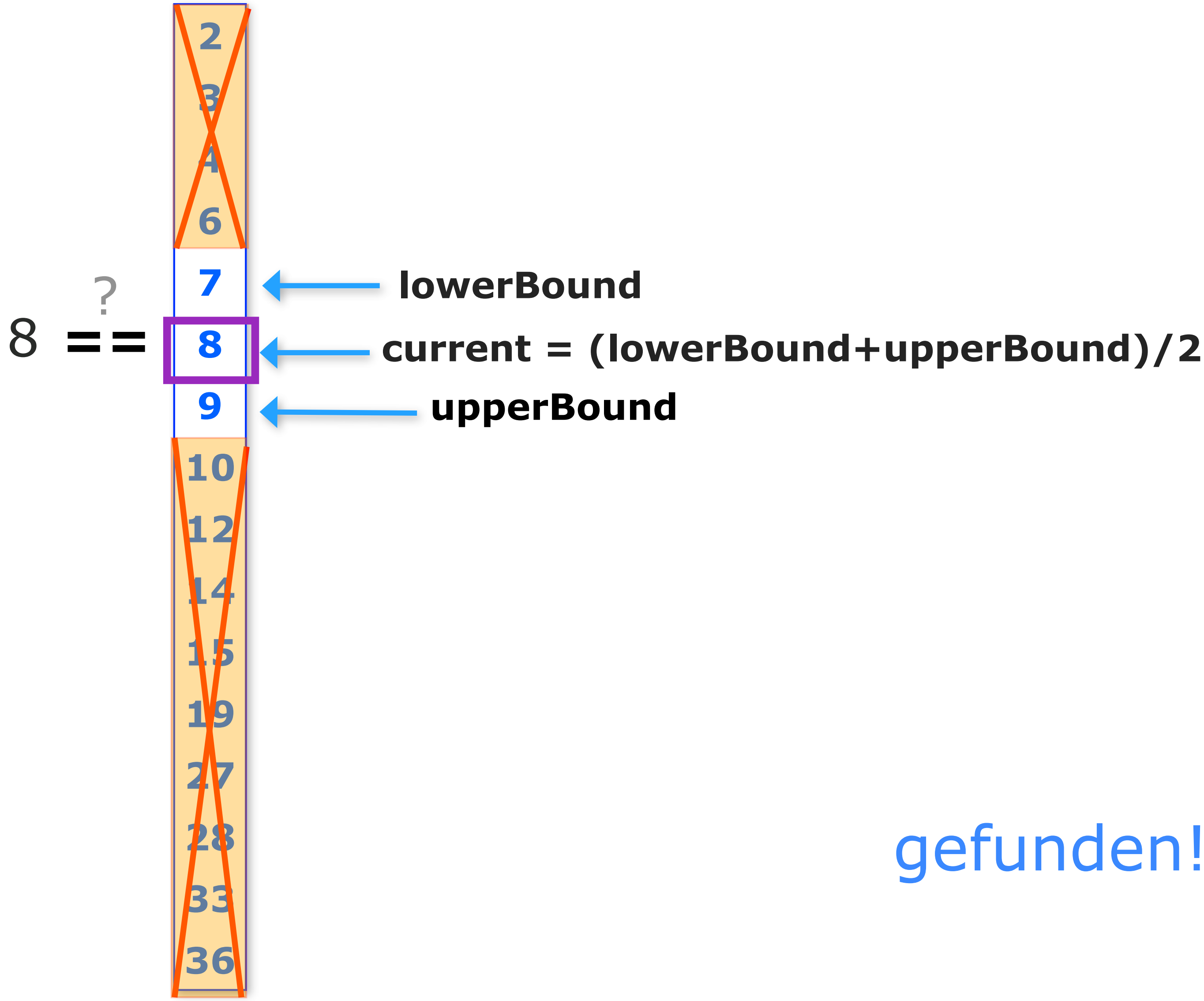


8 ?==
2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ


2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
8 ==
?
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ



gefunden!

Iterativ

16 $\overset{?}{==}$

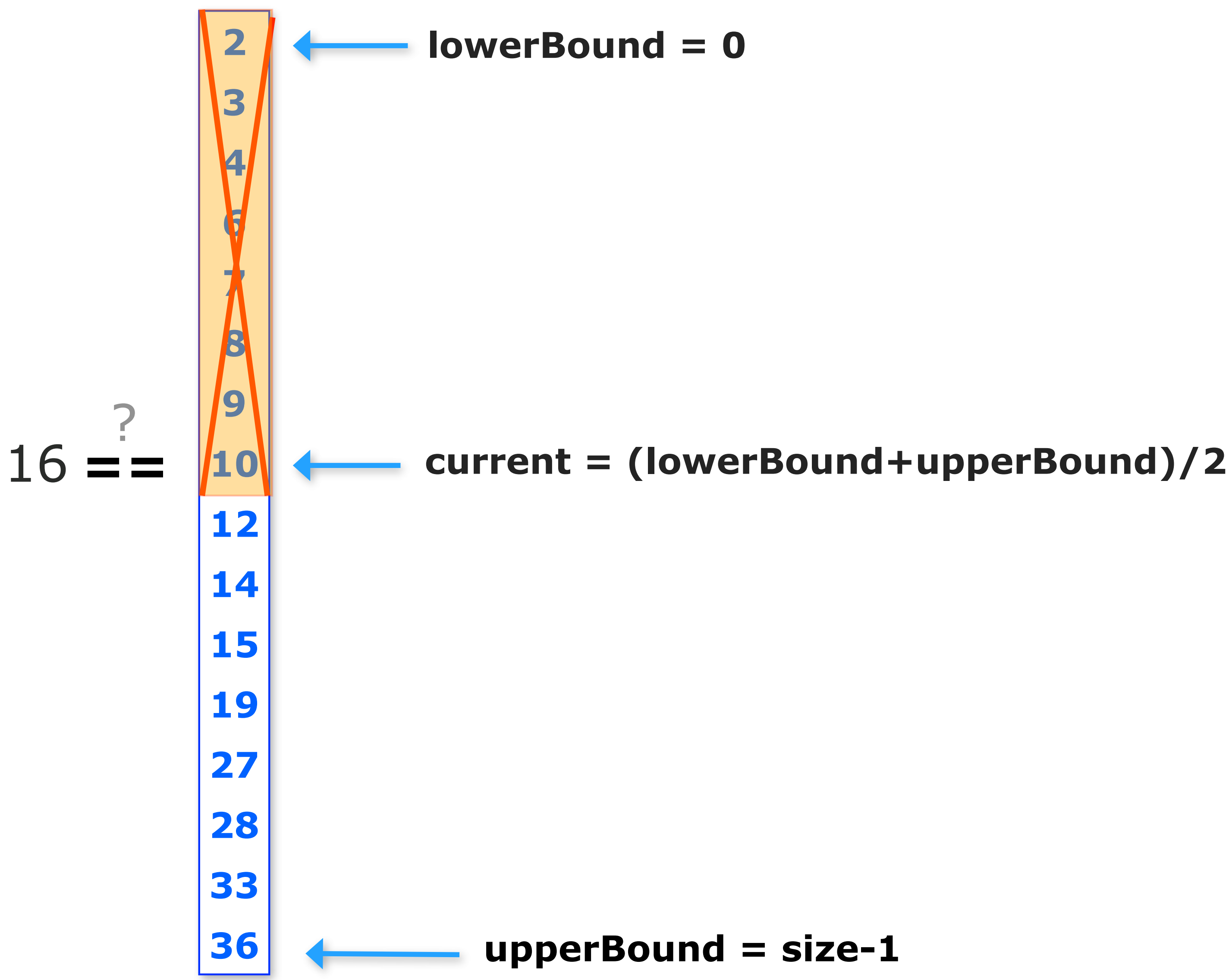
2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36

lowerBound = 0 (at 2)

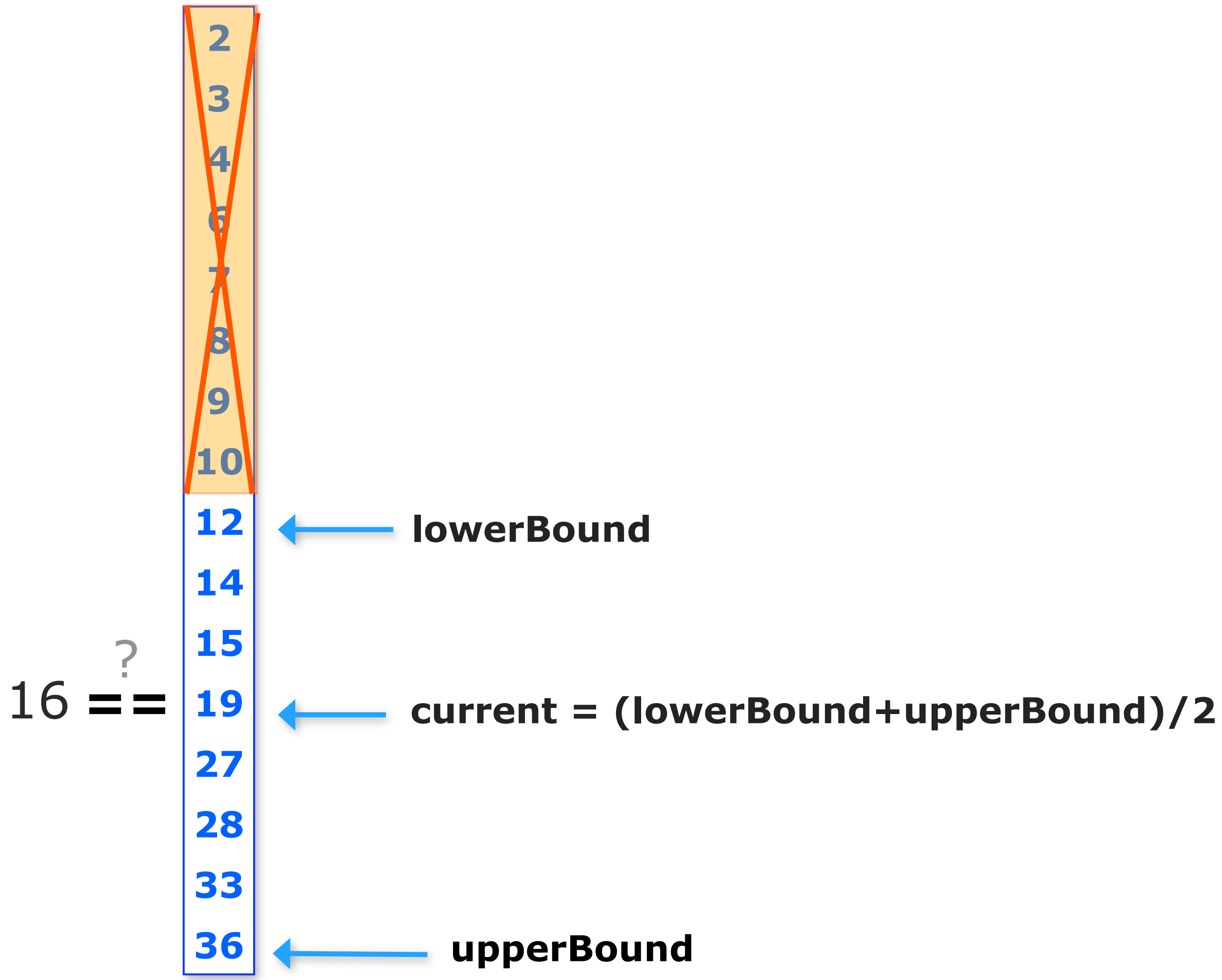
current = (lowerBound+upperBound)/2 (at 10)

upperBound = size-1 (at 36)

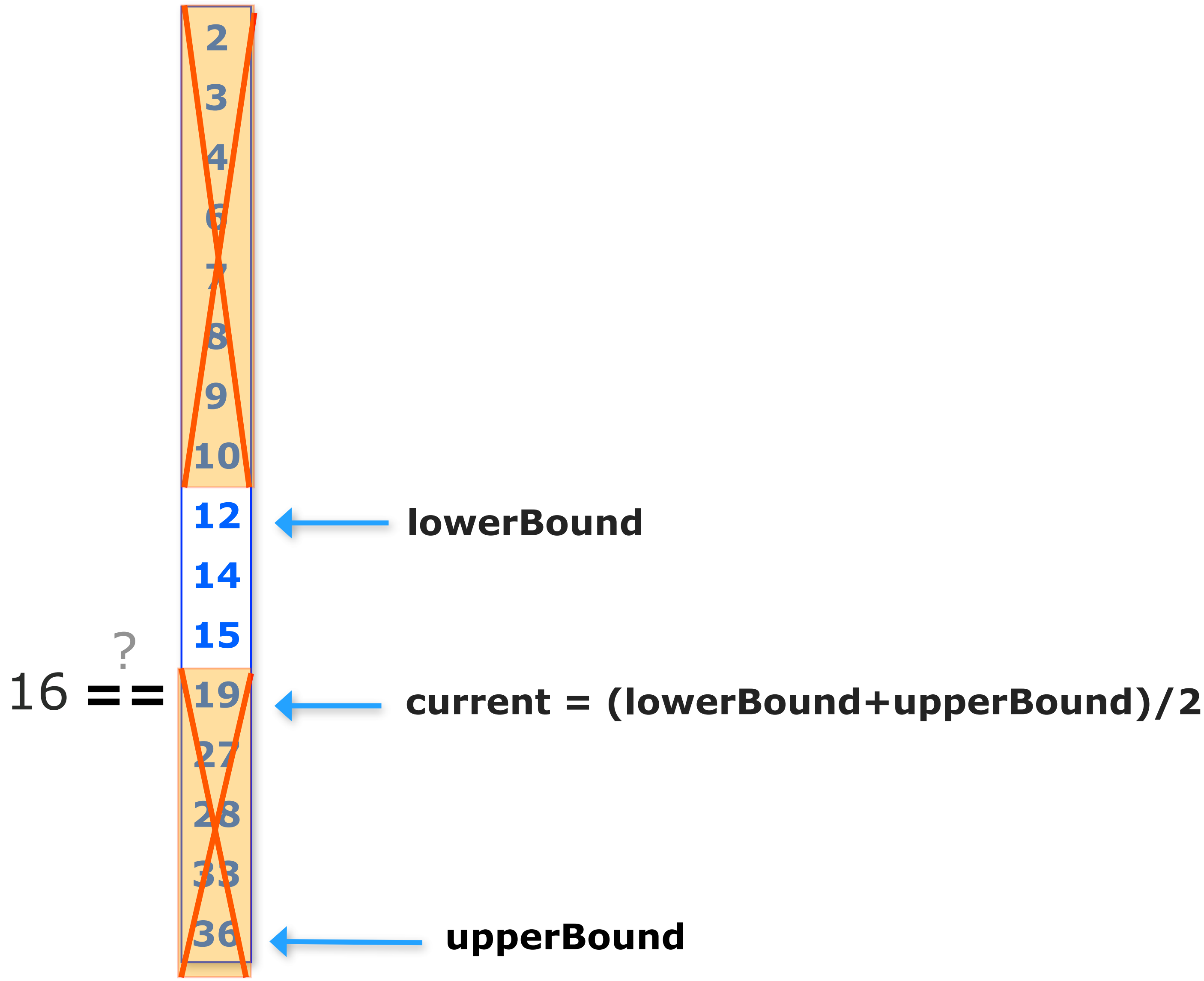
Iterativ


2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
lowerBound = 0
16 ==
?
current = (lowerBound+upperBound)/2
upperBound = size-1

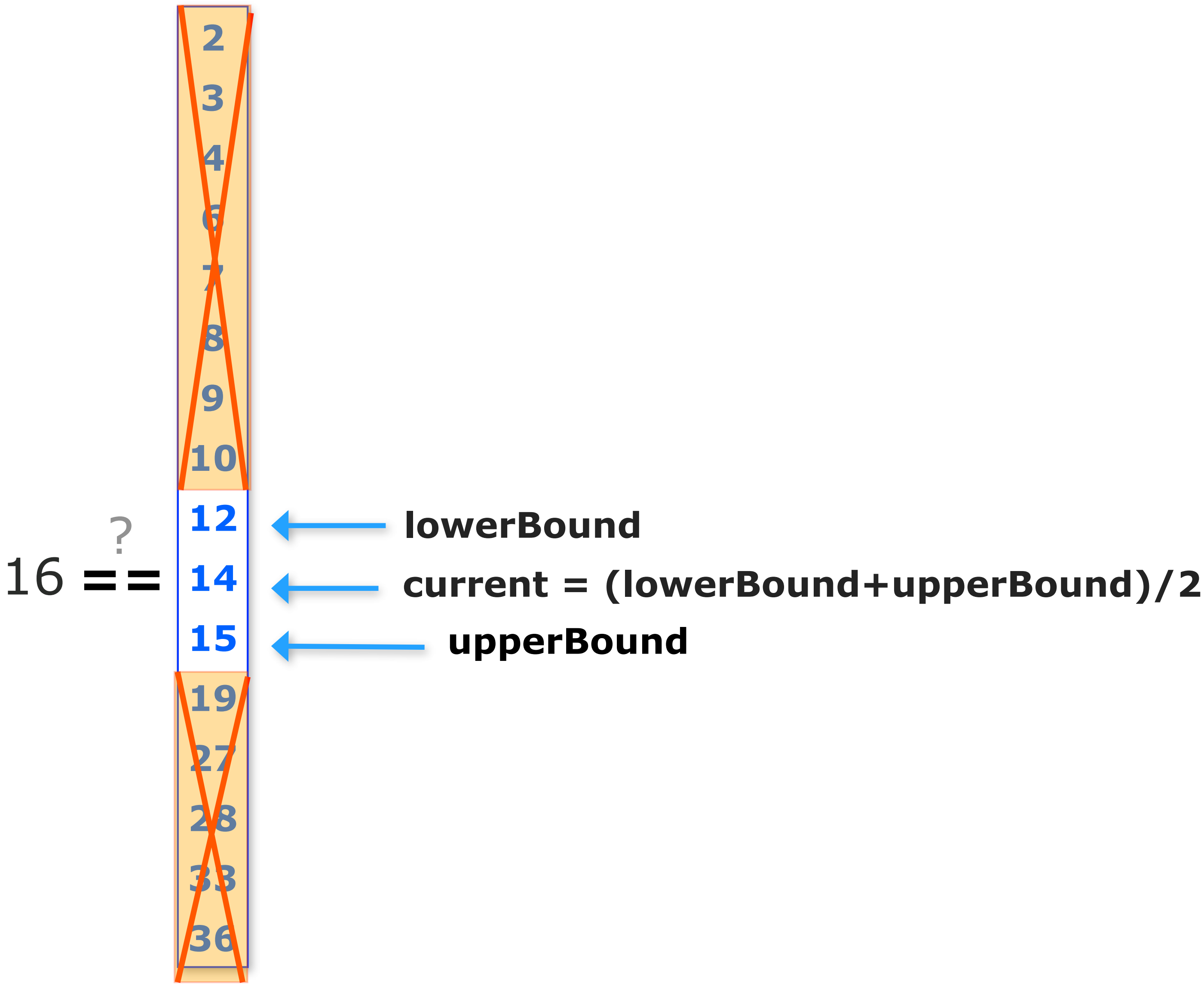
Iterativ


2
3
4
6
7
8
9
10
12
14
15
16 ==?
19
27
28
33
36
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ

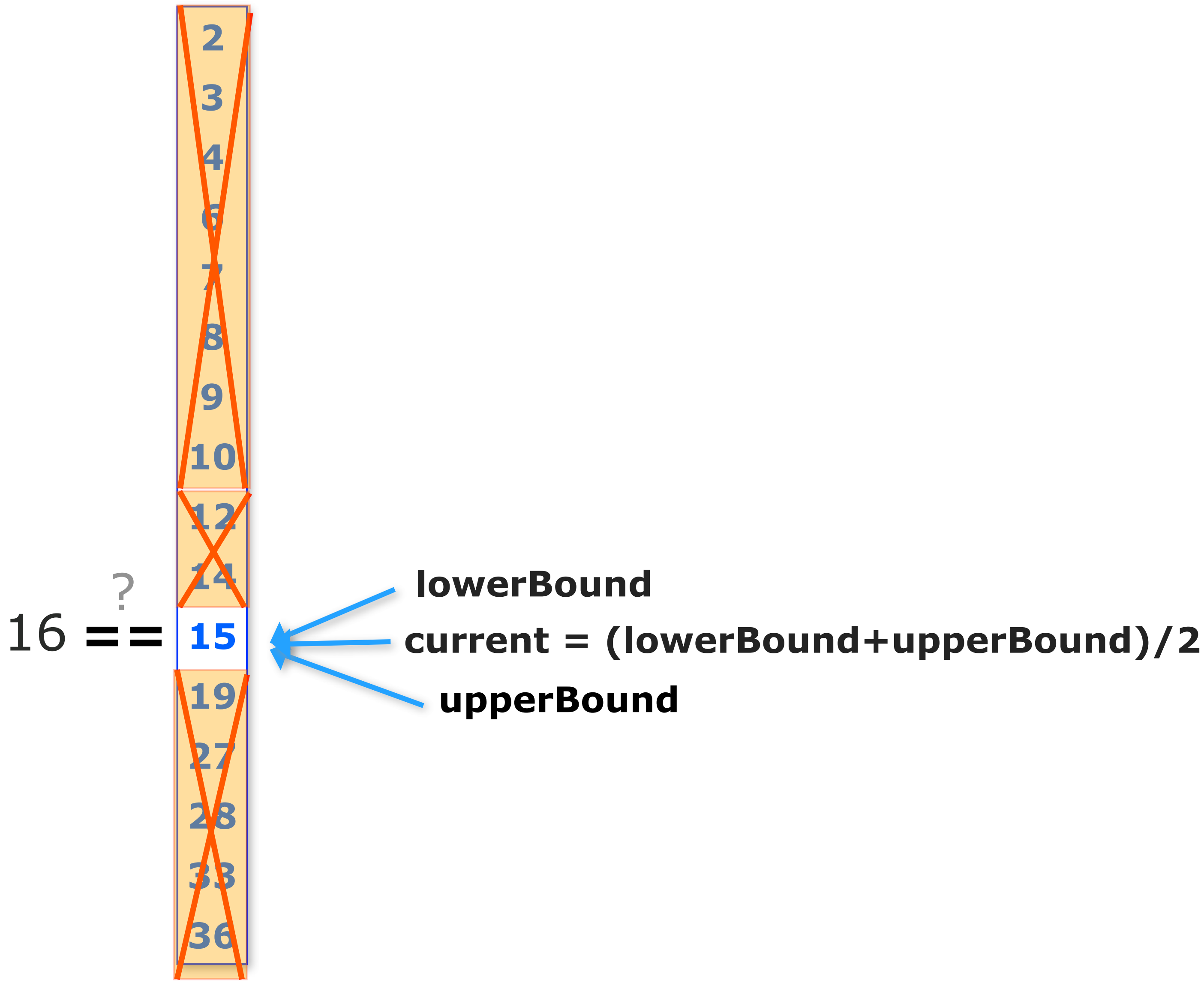

2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
16 == ?
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ

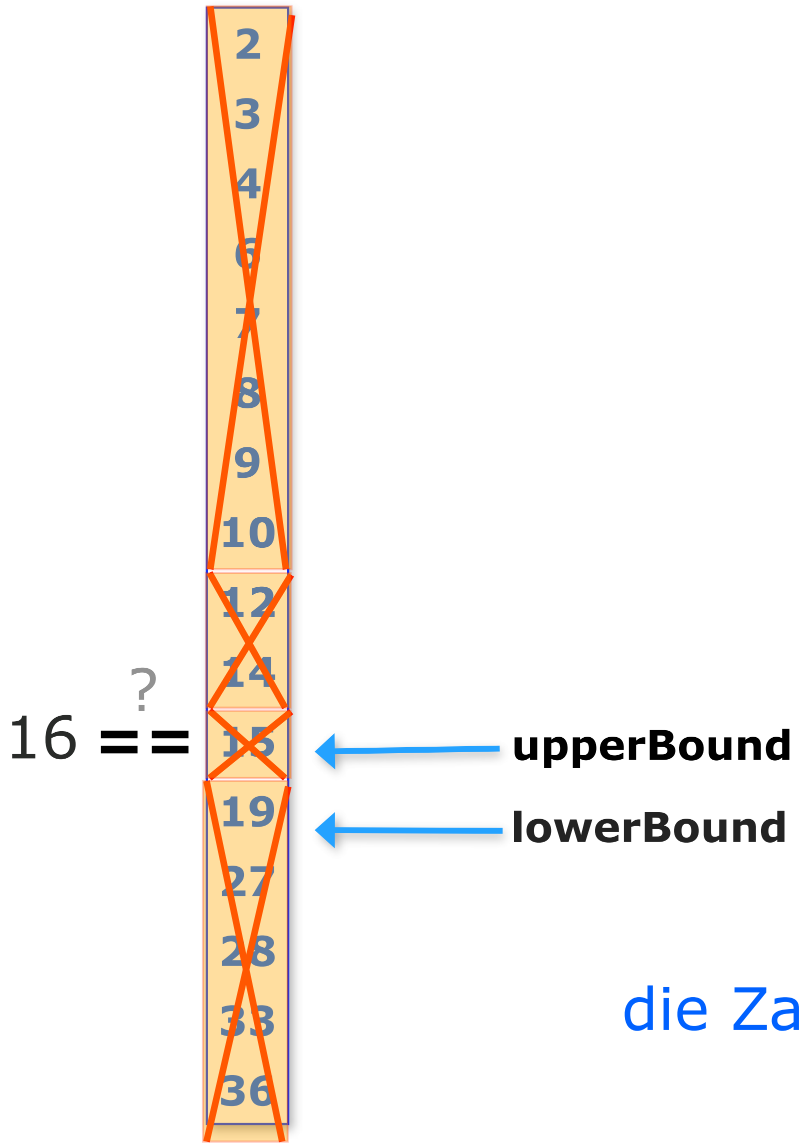


2
3
4
6
7
8
9
10
12
14
15
19
27
28
33
36
16 == ?
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ


2
3
4
6
7
8
9
10
12
14
?
16 ==
15
19
27
28
33
36
lowerBound
current = (lowerBound+upperBound)/2
upperBound

Iterativ



Wenn **lowerBound>upperBound**

die Zahl ist **nicht** gefunden worden!

Binärsuche in einem Feld

Iterativ

```
def binary_search (nums, key):

    lowerBound = 0
    upperBound = len(nums) - 1

    while  lowerBound <= upperBound:
           current = (lowerBound + upperBound)//2
           if   nums[current] == key:
                    return True
           else:
                    if  nums[current] < key:
                            lowerBound = current + 1
                    else:
                            upperBound = current - 1
    return False
```