Freie Universität Berlin

Analyse von Algorithmen

Die O-Notation

Prof. Dr. Margarita Esponda
Freie Universität Berlin

Analyse von Algorithmen

Wichtigste Eigenschaften bei der Analyse von Algorithmen

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

Analyse von Algorithmen

Rechenzeit Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der Eingabegröße.

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

Bandbreite Wie groß ist die erforderliche Datenübertragung.

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 **beliebige** Eingabedaten- die Laufzeit unseres Algorithmus immer kleiner oder gleich dieser Schranke ist.

Beispiel: Die Anzahl der Objekte, die wir sortieren wollen

Die Anzahl der Bytes, 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.

Beispiel:

Bei Sortieralgorithmen messen wir Vergleiche

Bei anderen Algorithmen:

- Speicherzugriffe
- Anzahl der Multiplikationen
- Anzahl der Bitoperationen
- Anzahl der Schleifendurchgänge
- Anzahl der Funktionsaufrüfe
- u.s.w.

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**).

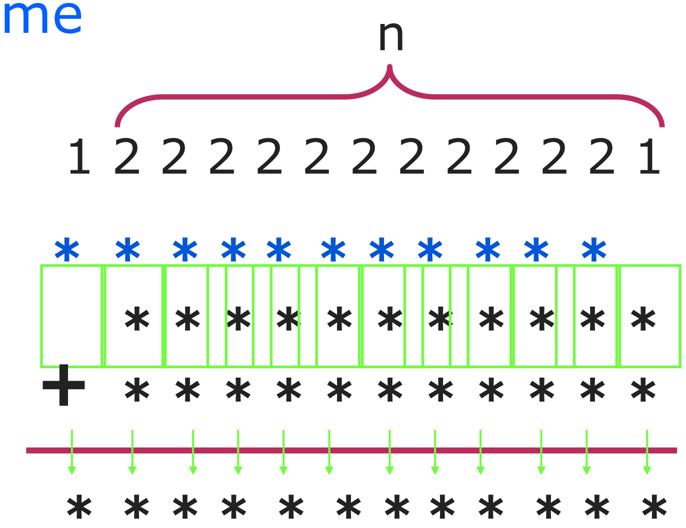
Die meisten Algorithmen besitzen einen Hauptparameter N, der die Anzahl der zu verarbeitenden Datenelemente angibt.

Die obere Schranke ist eine Funktion, die das Wachstum der Laufzeit in Abhängigkeit der Eingabegröße beschreibt.

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

Summe und Multiplikation in der Schule

Summe



Eingabegröße:

n = Zahlenbreite

Berechnungsschritt:

Addition von zwei Ziffern

Komplexitätsanalyse:

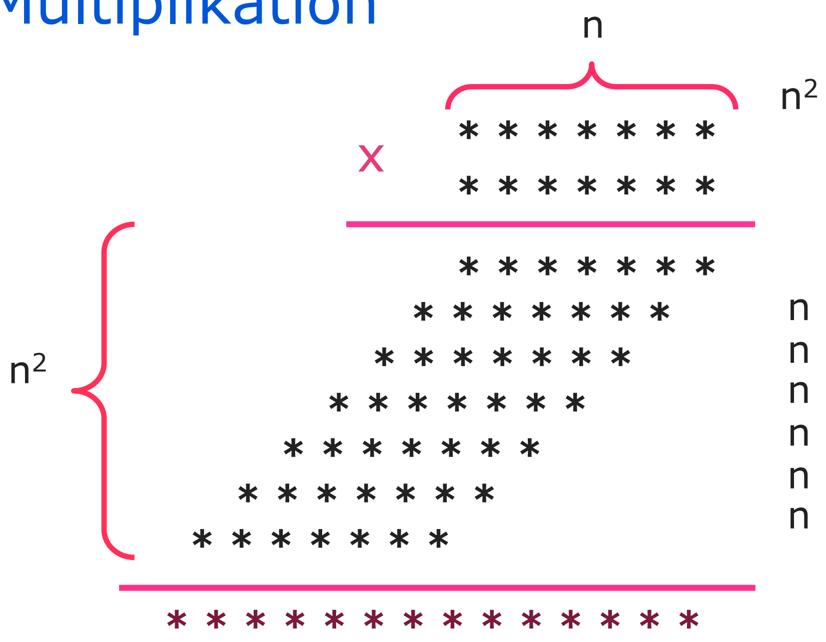
T(n) = Anzahl der Berechnungsschritte, um zwei Zahlen mit **n** Ziffern zu addieren

Im schlimmsten Fall:

T(n) = 2n

T(n) ist eine lineare Funktion

Multiplikation



Eingabegröße:

n = Anzahl der Ziffern

Berechnungsschritt:

Multiplikation von zwei Ziffern

Komplexitätsanalyse:

$T(n) = n^2$

Multiplikation und Summen von Ziffern

Im schlimmsten Fall

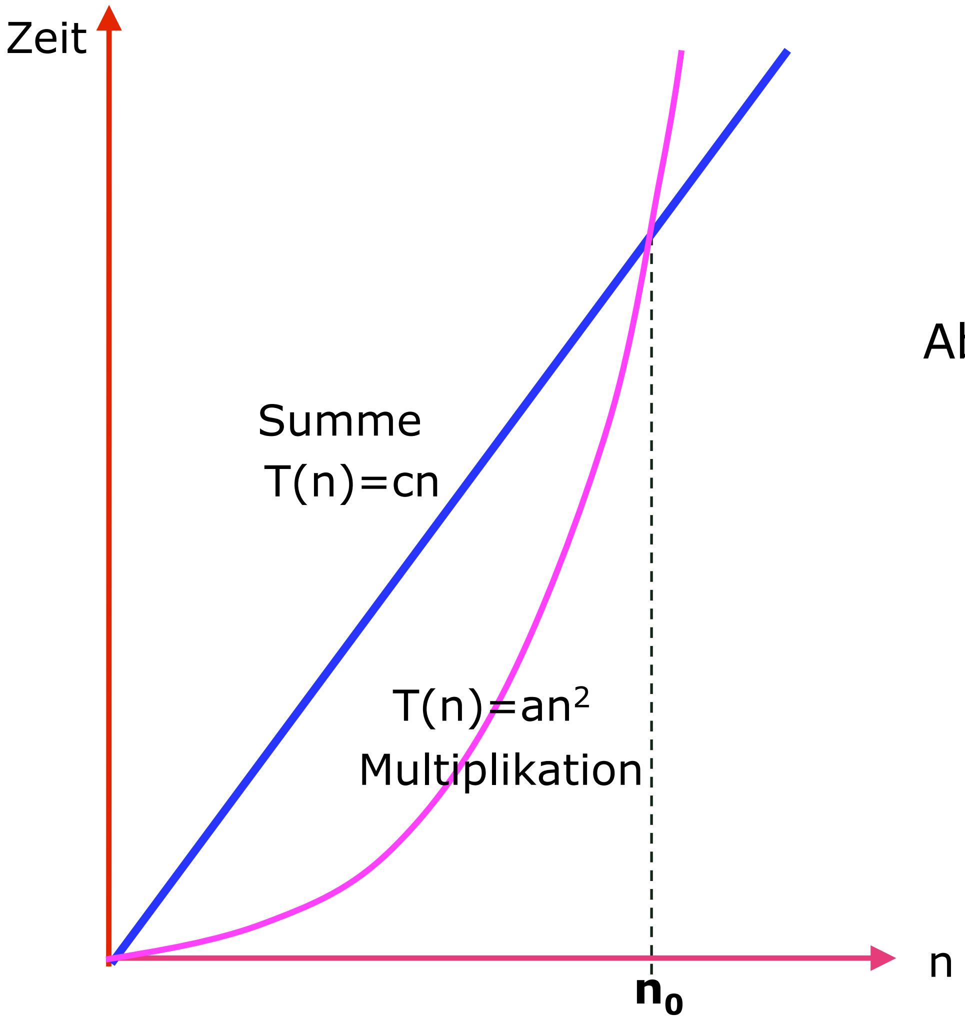
keine Nullen

immer ein Übertrag

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

T(n) ist eine quadratische Funktion

Summe und Multiplikation



Ab einem bestimmten n_0 gilt $\mathbf{c}n < \mathbf{a}n^2$

O-Notation

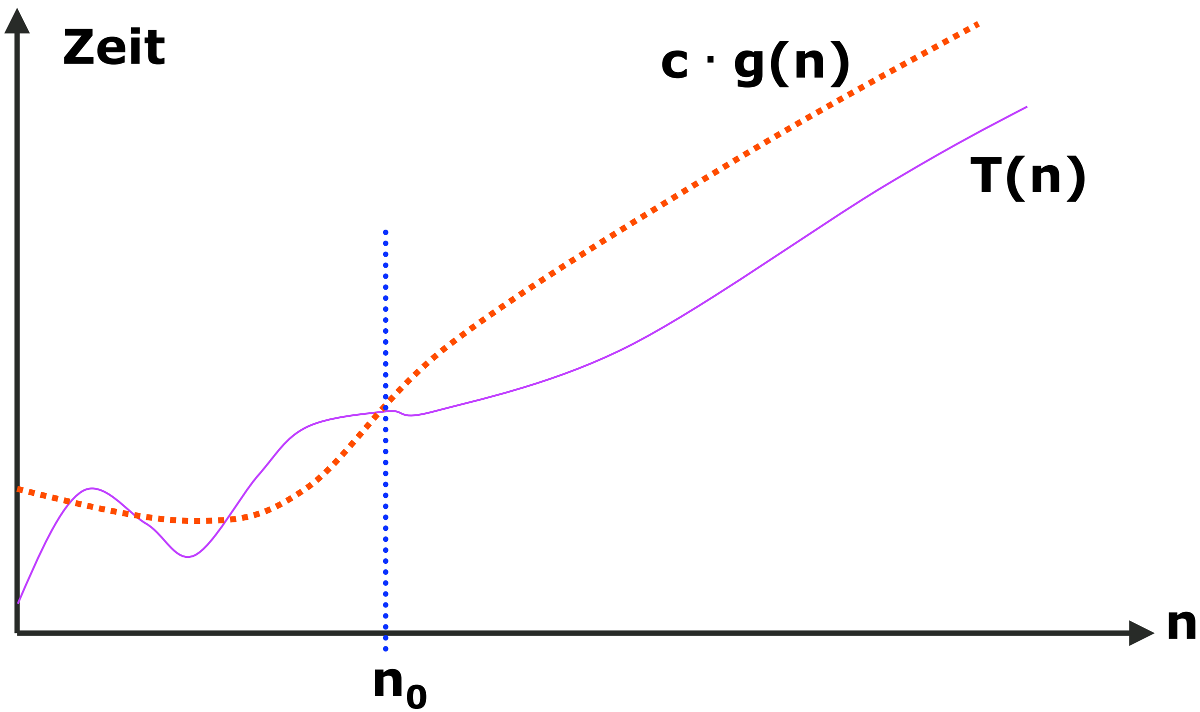
Für die Effizienzanalyse von Algorithmen wird eine spezielle mathematische Notation verwendet, die als **O-Notation** bezeichnet wird.

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 **n_0** gibt, so dass **$T(n) \leq c \cdot g(n)$** für alle **$n \geq n_0$**



O-Notation

Beispiel: für $\mathbf{T(n) = n^2 + 2n}$ gilt $\mathbf{n^2 + 2n = O(n^2)}$

denn mit $\mathbf{c = 3}$ und ab $\mathbf{n_0 = 1}$ gilt

$$\mathbf{2n + n^2 \leq c \cdot n^2}$$

Begründung:

$$2n + n^2 \leq 3n^2$$

$$2n + n^2 \leq 2n^2 + n^2$$

$$2n \leq 2n^2$$

$$1 \leq n$$

Die Funktion **T(n)** liegt in der Komplexitätsklasse $\mathbf{O}(n^2)$

oder

Die Größenordnung der Funktion **T(n)** ist $\mathbf{O}(n^2)$

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

Freie Universität Berlin

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:

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

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.

Bedeutung der O-Notation

Beispiel: $100n^3 + 15n^2 + 15 = \mathbf{O}(n^3)$?

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

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

Begründung:

$$\lim_{n \to \infty} \frac{3n + 7}{n} = 3$$

$$\lim_{n \to \infty} \frac{3n + 7}{n^2} = 0$$

$$\lim_{n \to \infty} \frac{3n + 7}{n^3} = 0$$

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

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	$\mathbf{O}(n)$	
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!

Komplexität eines Algorithmus

O-Notation

Obere Komplexitätsgrenze (höchstens)

Ω–Notation

Untere Komplexitätsgrenze (mindestens)

θ-Notation

Genaue Komplexität (genau)

Komplexität eines Algorithmus

Definitionen:

Ω–Notation

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

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

$T(n) = c_1 + c_2 n$

c_2 = Zeitkosten eines Schleifendurchgangs

$T(n) = O(n)$

Rekursiv

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

$T(n) = c_1 + c_2 n$

c_2 = Zeitkosten eines Funktionsaufrufs

$T(n) = O(n)$

direkt

Formel von Gauß

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

$T(n) = O(1)$

Freie Universität Berlin

Implementierung der Funktion Fakultät

Fakultät (0) = 1

Fakultät (n) = n · Fakultät (n-1)

für alle n>o

Rekursive Implementierung

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

Rechenzeit: T(n) = **O**(n)

Speicherplatz: T(n) = **O**(n)

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
```

Rechenzeit: T(n) = **O**(n)

Speicherplatz: T(n) = **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**

Freie Universität Berlin

Eine rekursive Funktion verursacht eine Kette von Funktionsaufrufen

main ()

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 main

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 main
lokaler Variablen von main

Laufzeitkeller

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.

Endrekursion

Freie Universität Berlin

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

```
factorial 0 = 1
factorial n = n*factorial (n-1)
```

Ablauf einer Berechnung:

```
factorial 6 ⇒ 6 * factorial 5
            ⇒ 6 * (5 * factorial 4)
            ⇒ 6 * (5 * (4 * factorial 3))
            ⇒ 6 * (5 * (4 * (3 * factorial 2)))
            ⇒ 6 * (5 * (4 * (3 * (2 * factorial 1))))
            ⇒ 6 * (5 * (4 * (3 * (2 * (1 * factorial 0)))))
            ⇒ 6 * (5 * (4 * (3 * (2 * (1 * 1)))))
            ⇒ 6 * (5 * (4 * (3 * (2 * 1))))
            ⇒ 6 * (5 * (4 * (3 * 2)))
            ⇒ 6 * (5 * (4 * 6))
            ⇒ 6 * (5 * 24)
            ⇒ 6 * 120
            ⇒ 720
```

Der Ausführungsstapel wächst bei jeder rekursive Aufruf und Teilausdrücke müssen ständig zwischen-gespeichert werden.

Die Endberechnungen finden erst beim Abbau des Ausführungsstapels statt.

Endrekursion

Beispiel einer endrekursiven Definition der Fakultätsfunktion

Haskell:

```
factorial n = factorial_helper 1 n
                    where
                        factorial_helper a 0 = a
                        factorial_helper a n = factorial_helper (a*n) (n-1)
```

Python:

```
def factorial (n):

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

        return factorial_helper (1, n)
```

Endrekursion

Ablauf einer Berechnung:

factorial (6) ⇒ factorial_helper (1, 6)

⇒ factorial_helper (6, 5)

⇒ factorial_helper (30, 4)

⇒ factorial_helper (120, 3)

⇒ factorial_helper (360, 2)

⇒ factorial_helper (720, 1)

⇒ factorial_helper (720, 0)

⇒ 720

keine Zwischenausdrücke müssen gespeichert werden.

Endrekursive Funktionen können aus diesem Grund oft vom Übersetzer (Compiler) optimiert werden, indem diese in einfache Schleifen verwandelt werden.

Berechnung der Fakultätsfunktion

Endrekursiv

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

    return factorial_helper (1, n)
```

Endrekursive Funktionen werden in Python nicht optimiert!

"Tail Recursion Elimination

TRE is incompatible with nice stack traces… and makes debugging hard."

Guido van Rossum

```
>>> factorial (993)
. . .
RuntimeError: maximum recursion depth exceeded in comparison
```

Berechnung der Fibonacci-Zahlen

Rekursiv

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

$$\text{Fib}(\ n\ ) = \theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n}\right)$$

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

Freie Universität Berlin

Berechnung der Fibonacci-Zahlen

Wie viele Schritte brauchen wir, um **fib n** zu berechnen?

fib 0 fib 1 fib 2 fib 3 fib 4 fib 5 fib 6

1 + 1 = 2 3 5 8 13

1 + 2 = 3

2 + 3 = 5

3 + 5 = 8

3 + 5 = 13

Die Anzahl der Reduktionen für **fib n** ist gleich **fib (n+1)**

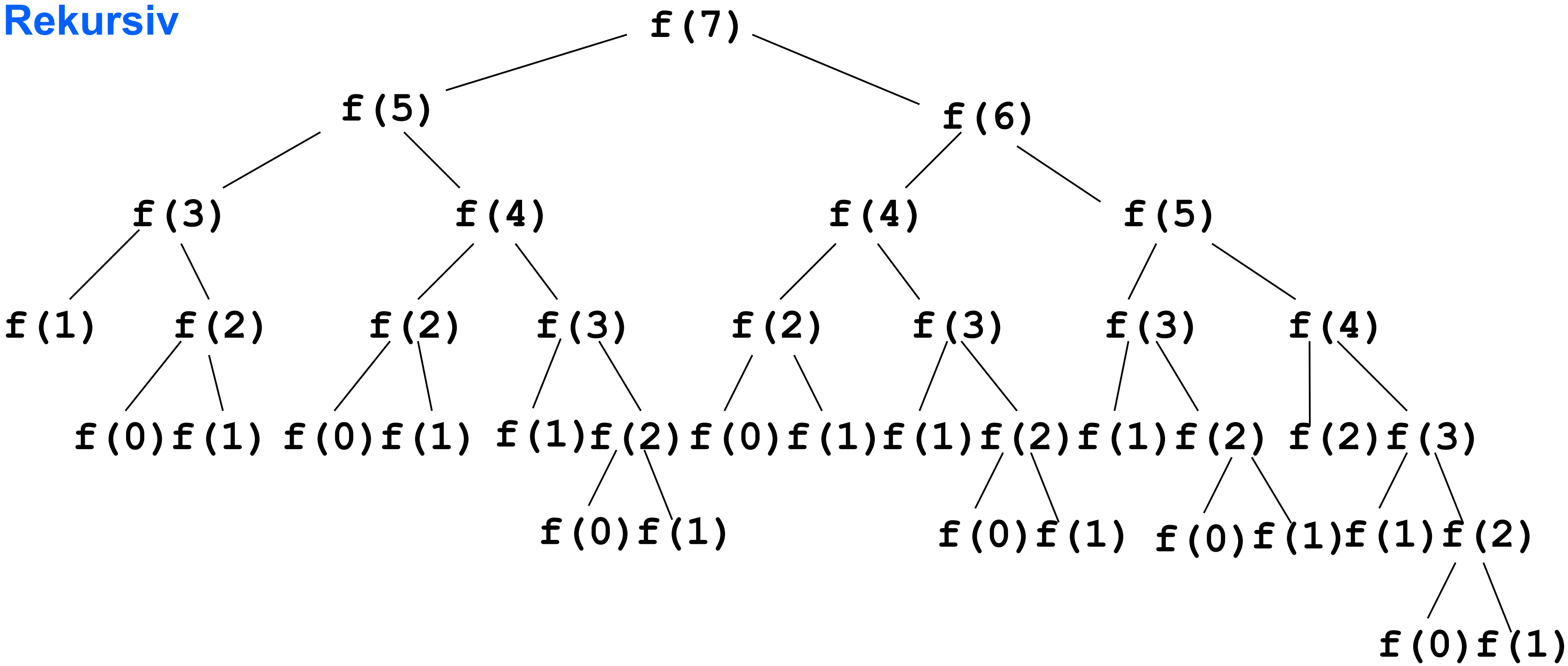
Die rekursive Berechnung der Fibonacci-Zahlen hat eine exponentielle Komplexität

$$\mathbf{O}((1{,}618...)^n)$$

Freie Universität Berlin

Berechnung der Fibonacci-Zahlen

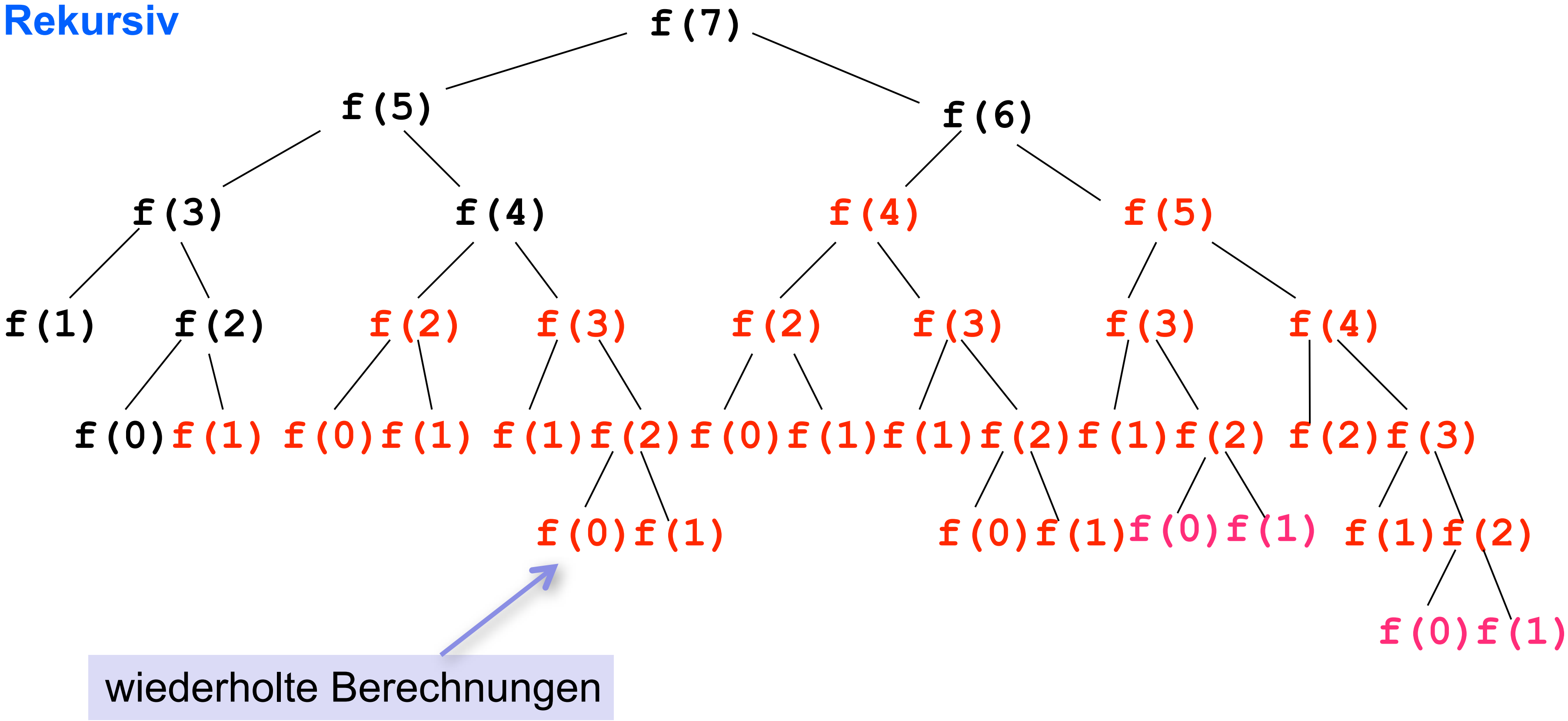
Rekursiv



Freie Universität Berlin

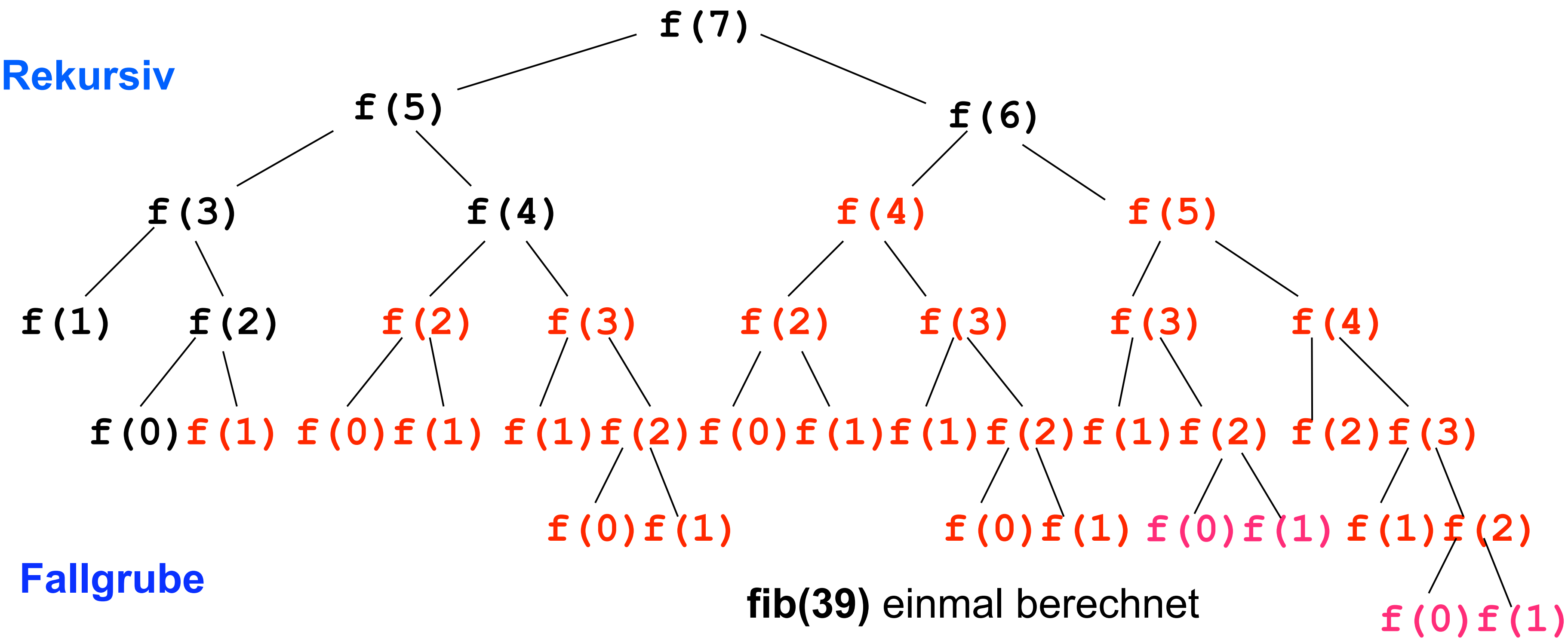
Berechnung der Fibonacci-Zahlen

Rekursiv



Berechnung der Fibonacci-Zahlen

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**

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

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 (992)
9252394159943865548695885305267321133917910271071460896757822139976047281321590991446751768798293528186087306538837695052158186157009963747932427410224440709142685670040412619319700044602587378855210823082290
>>> fib_end_rek (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**

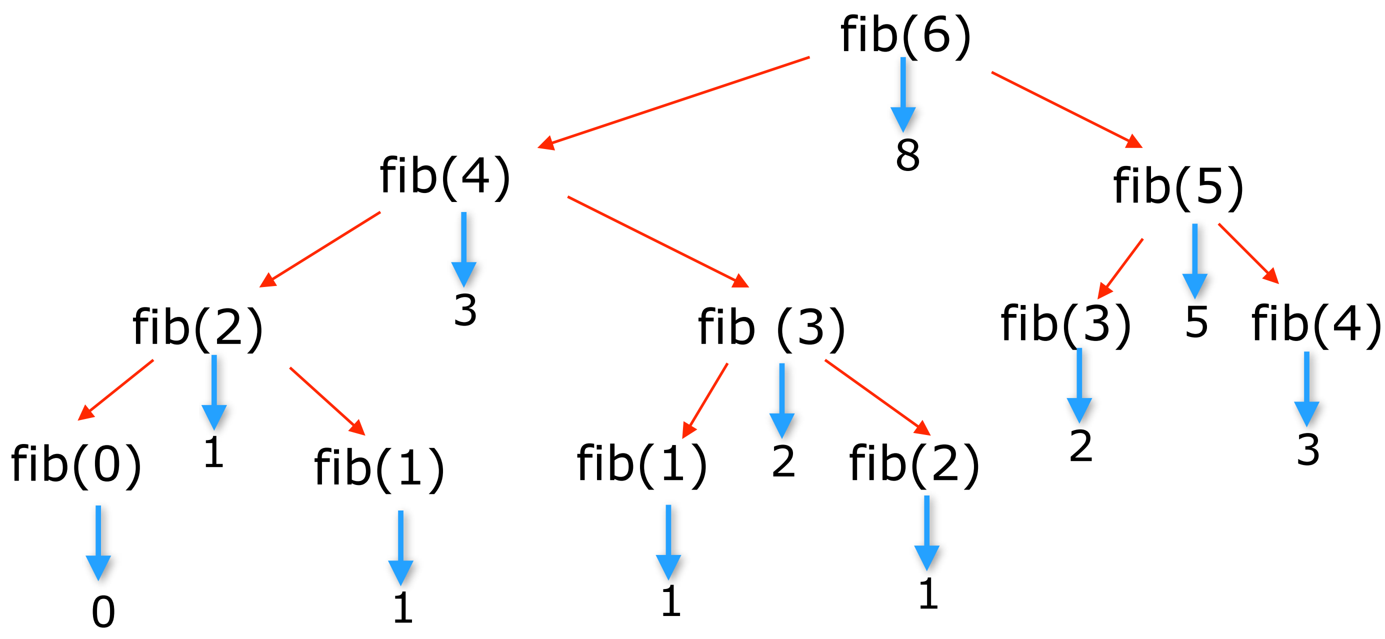
Bei dynamischer Programmierung werden Zwischenberechnungen in einer Tabelle gespeichert, damit diese später wieder verwendet werden können.

Berechnung der Fibonacci-Zahlen

Lösung mit **Dynamischer Programmierung**

fibs-Tabelle

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



Berechnung der Fibonacci-Zahlen

mit **Dynamischer Programmierung**

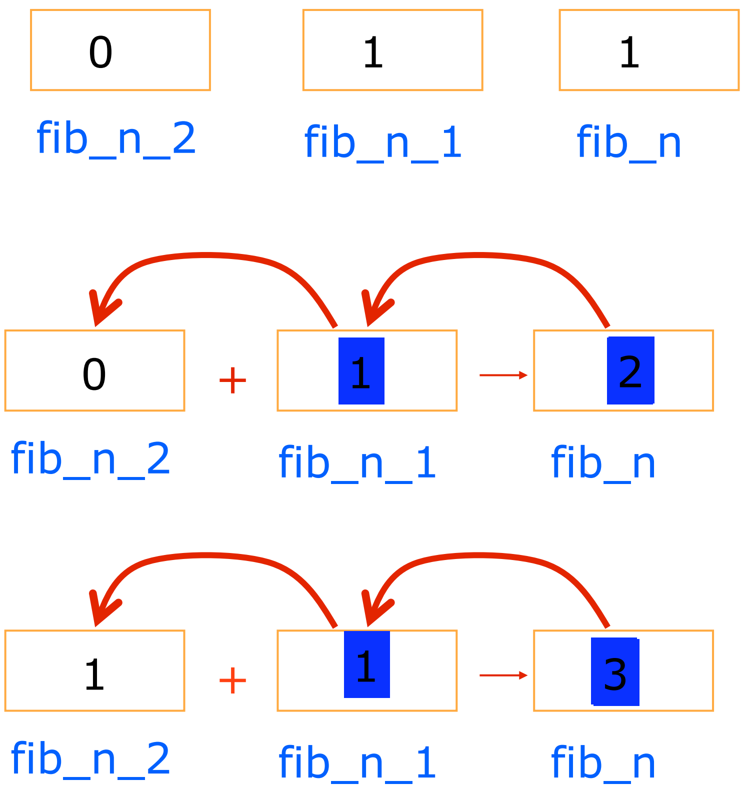
```
def fib_dyn(n):
    if n==0 or n==1:
        return n
    else:
        fibs = [0 for i in range(n+1)]
        fibs[1] = 1
        return fib_help(fibs,n)

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]
```

$T(n) = O(n)$

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+1):
            fib_n_2 = fib_n_1
            fib_n_1 = fib_n
            fib_n = fib_n_1 + fib_n_2
        return fib_n
```

$T(n) = O(n)$

Freie Universität Berlin

Berechnung der Fibonacci-Zahlen

```
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

Iterativ

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

$c_2 =$ Zeitkosten eines Schleifedurchgangs

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

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

Berechnung der Fibonacci-Zahlen

Direkt

$$Fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

```
def fib_const(n):
        sq5 = sqrt(5)
        return  int( ( ((1+sq5)/2)**n - ((1-sq5)/2)**n )/sq5 )
```

O(1)? T(n) = O(n)?

Rekursion vs. Iteration

Jede rekursive Funktion kann als Iteration umgeschrieben werden.

Der Compiler implementiert Rekursion mit Hilfe des Stapels

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

Jede interative Funktion kann in eine rekursive Funktion umgeschrieben werden.

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

Wann sollen wir Iteration und wann Rekursion verwenden?

Einfache und übersichtliche Implementierung	Rekursion
Effiziente Implementierung	Iteration

Beispiel:

Array

3
5
7
11
17
19
23
29
31
34
37
57

Sortierte Menge

Eingabegröße:

n = Anzahl der sortierte Zahlen

Berechnungsschritt:

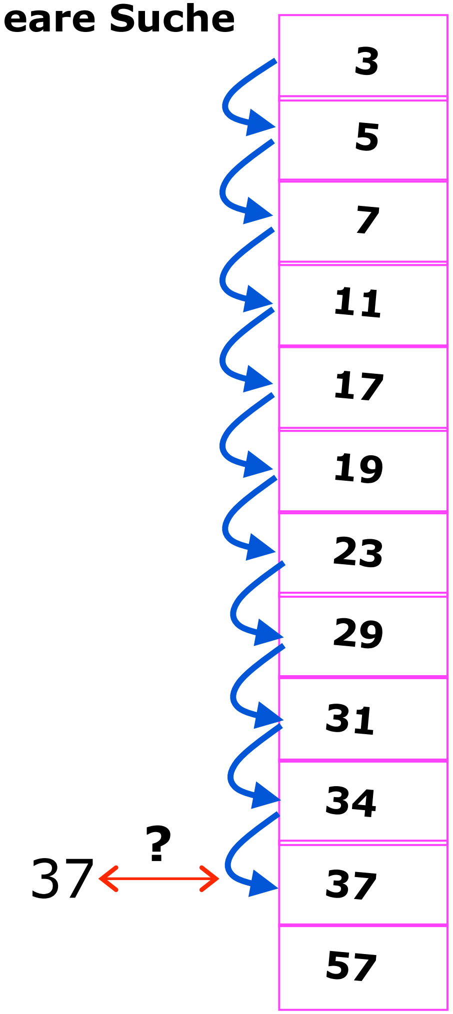
Vergleichsoperation

Komplexitätsanalyse:

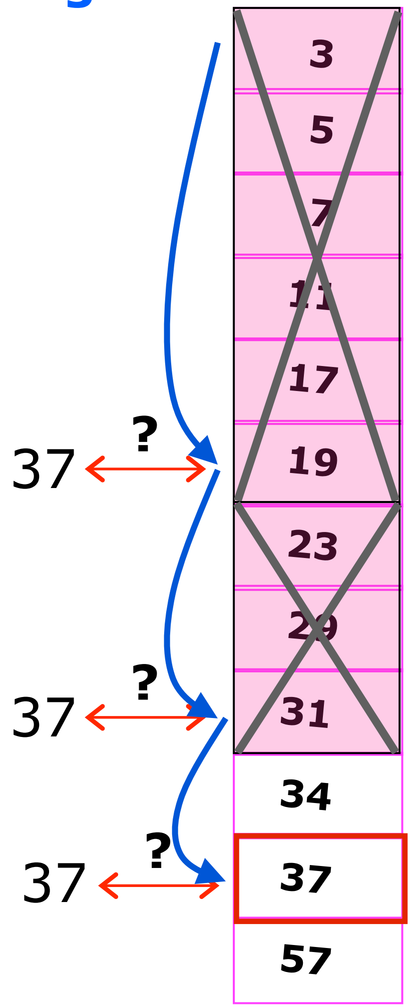
T(n) = Anzahl der Berechnungsschritte, um eine Zahl zu finden.

Sortierte Menge

Lineare Suche



Im schlimmsten Fall **n** Schritte



nur **3** Schritte

Freie Universität Berlin

Maximale Schrittanzahl

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

$7 = \log_2 (128)$

Im schlimmsten Fall

$\mathbf{log_2(n)}$ Schritte