ETH zürich

Departement Informatik

Programmieren und Problemlösen

Komplexität und Primzahltests

Dennis Komm

Frühling 2021 – 25. März 2021

Komplexität von Algorithmen

Primzahltest

Aufgabe – Primzahltest

Schreiben Sie eine Funktion, die

- eine ganze Zahl x als Parameter erhält
- berechnet, ob x eine Primzahl ist
- dabei den %-Operator verwendet
- davon abhängig `True` oder `False` zurückgibt

Primzahltest

```python
def primetest(x):
    if x < 2:
        return False
    d = 2
    while d < x:
        if x % d == 0:
            return False
        d += 1
    return True
```

Primzahltest

Wie lange braucht der Algorithmus, um die Ausgabe zu erzeugen?

Primzahltest

Wie lange braucht der Algorithmus, um die Ausgabe zu erzeugen?

- Was ist seine **Laufzeit**?
- Hängt von Anzahl der Schleifendurchläufe ab

Primzahltest

Wie lange braucht der Algorithmus, um die Ausgabe zu erzeugen?

- Was ist seine **Laufzeit**?
- Hängt von Anzahl der Schleifendurchläufe ab
- Absoluter Wert ergibt keinen Sinn
- Schleife wird (ungefähr) `x` Mal durchlaufen (wenn `x` prim ist)

⇨ Laufzeit wächst mit `x`

Primzahltest

Wie lange braucht der Algorithmus, um die Ausgabe zu erzeugen?

- Was ist seine **Laufzeit**?
- Hängt von Anzahl der Schleifendurchläufe ab
- Absoluter Wert ergibt keinen Sinn
- Schleife wird (ungefähr) `x` Mal durchlaufen (wenn `x` prim ist)

⇨ Laufzeit wächst mit `x` ... **aber wie schnell?**

Laufzeit – Funktion der Eingabelänge

- Wir messen Laufzeit als Funktion der **Eingabelänge**

Laufzeit – Funktion der Eingabelänge

- Wir messen Laufzeit als Funktion der **Eingabelänge**
- Für unseren Algorithmus ist die Eingabe eine Zahl `x`
- Zahlen werden im Computer binär dargestellt

Laufzeit – Funktion der Eingabelänge

- Wir messen Laufzeit als Funktion der **Eingabelänge**
- Für unseren Algorithmus ist die Eingabe eine Zahl `x`
- Zahlen werden im Computer binär dargestellt
- Wenn wir führende Nullen ignorieren, gilt für n Bits

$$2^{n-1} \text{ ist } \underbrace{10\ldots00}_{n}, \quad 2^{n-1}+1 \text{ ist } \underbrace{10\ldots01}_{n}, \ldots \quad \text{und } 2^n-1 \text{ ist } \underbrace{11\ldots11}_{n}$$

Laufzeit – Funktion der Eingabelänge

- Wir messen Laufzeit als Funktion der **Eingabelänge**
- Für unseren Algorithmus ist die Eingabe eine Zahl `x`
- Zahlen werden im Computer binär dargestellt
- Wenn wir führende Nullen ignorieren, gilt für n Bits

$$2^{n-1} \text{ ist } \underbrace{10\ldots00}_{n}, \quad 2^{n-1}+1 \text{ ist } \underbrace{10\ldots01}_{n}, \ldots \quad \text{und } 2^n-1 \text{ ist } \underbrace{11\ldots11}_{n}$$

Eine Zahl, die mit n Bits dargestellt wird, hat ungefähr die Grösse 2^n

Laufzeit – Technologiemodell

Random Access Machine

- **Ausführungsmodell:** Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt

Laufzeit – Technologiemodell

Random Access Machine

- **Ausführungsmodell:** Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt
- **Speichermodell:** Konstante Zugriffszeit

Laufzeit – Technologiemodell

Random Access Machine

- **Ausführungsmodell:** Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt
- **Speichermodell:** Konstante Zugriffszeit
- **Elementare Operationen:** Rechenoperationen $(+, -, \cdot, \ldots)$, Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)

Laufzeit – Technologiemodell

Random Access Machine

- **Ausführungsmodell:** Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt
- **Speichermodell:** Konstante Zugriffszeit
- **Elementare Operationen:** Rechenoperationen $(+, -, \cdot, \ldots)$, Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)
- **Einheitskostenmodell:** Jede elementare Operation hat Kosten 1

Laufzeit – Anmerkung

Wir sind an dieser Stelle nicht ganz exakt

Laufzeit – Anmerkung

Wir sind an dieser Stelle nicht ganz exakt

- Zahlen in Python können beliebig grosse Werte beinhalten
- Wir nehmen vereinfacht an, dass arithmetische Operationen in konstanter Zeit ausgeführt werden können

Laufzeit – Anmerkung

Wir sind an dieser Stelle nicht ganz exakt

- Zahlen in Python können beliebig grosse Werte beinhalten
- Wir nehmen vereinfacht an, dass arithmetische Operationen in konstanter Zeit ausgeführt werden können
- Die Zeit der Addition zweier n-Bit-Zahlen hängt von n ab
- Die Kodierung einer Fliesskommazahl kann man nicht sofort aus ihrer Grösse ablesen
- Eine Addition ist sicherlich schneller zu bewältigen als eine Multiplikation

Laufzeit – Anmerkung

Wir sind an dieser Stelle nicht ganz exakt

- Zahlen in Python können beliebig grosse Werte beinhalten
- Wir nehmen vereinfacht an, dass arithmetische Operationen in konstanter Zeit ausgeführt werden können
- Die Zeit der Addition zweier n-Bit-Zahlen hängt von n ab
- Die Kodierung einer Fliesskommazahl kann man nicht sofort aus ihrer Grösse ablesen
- Eine Addition ist sicherlich schneller zu bewältigen als eine Multiplikation
- Logarithmisches Kostenmodell berücksichtigt dies, verwenden wir aber ebenfalls nicht

Laufzeit unseres Primzahltests

- Angenommen, **`x`** ist eine Primzahl und mit n Bits dargestellt

Laufzeit unseres Primzahltests

- Angenommen, `x` ist eine Primzahl und mit n Bits dargestellt
- Anzahl Schleifendurchläufe wächst mit Grösse von `x` $\approx 2^n$
- Die Schleife wird also ca. 2^n Mal ausgeführt

Laufzeit unseres Primzahltests

- Angenommen, `x` ist eine Primzahl und mit n Bits dargestellt
- Anzahl Schleifendurchläufe wächst mit Grösse von `x` $\approx 2^n$
- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen
- Algorithmus führt fünf Operationen pro Iteration aus
- Insgesamt ca. $5 \cdot 2^n$ Operationen

Laufzeit unseres Primzahltests

- Angenommen, `x` ist eine Primzahl und mit n Bits dargestellt
- Anzahl Schleifendurchläufe wächst mit Grösse von `x` $\approx 2^n$
- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen
- Algorithmus führt fünf Operationen pro Iteration aus
- Insgesamt ca. $5 \cdot 2^n$ Operationen
- Uns interessiert, wie sich Laufzeit verhält, wenn n wächst
- Ignoriere Konstante 5

Komplexität von Algorithmen

Asymptotische obere Schranken

Asymptotische obere Schranken

Genaue Laufzeit lässt sich selbst für kleine Eingaben kaum voraussagen

- Uns interessieren **obere Schranken**
- Betrachte das asymptotische Verhalten eines Algorithmus
- Ignoriere alle konstanten Faktoren

Asymptotische obere Schranken

Genaue Laufzeit lässt sich selbst für kleine Eingaben kaum voraussagen

- Uns interessieren **obere Schranken**
- Betrachte das asymptotische Verhalten eines Algorithmus
- Ignoriere alle konstanten Faktoren

Beispiel

- Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1
- Quadratisches Wachstum mit Koeffizient 10 ist genauso gut wie quadratisches Wachstum mit Koeffizient 1

Asymptotische obere Schranken

Gross-$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(2^n)$ enthält alle Funktionen, die nicht schneller wachsen als $c \cdot 2^n$ für eine Konstante c

Asymptotische obere Schranken

Gross-$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(2^n)$ enthält alle Funktionen, die nicht schneller wachsen als $c \cdot 2^n$ für eine Konstante c

Die Menge $\mathcal{O}(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c, wobei f und g positiv sind

Asymptotische obere Schranken

Gross-$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(2^n)$ enthält alle Funktionen, die nicht schneller wachsen als $c \cdot 2^n$ für eine Konstante c

Die Menge $\mathcal{O}(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c, wobei f und g positiv sind

- Verwende die asymptotische Notation für Laufzeit von Algorithmen
- Wir schreiben $\mathcal{O}(n^2)$ und meinen, dass der Algorithmus sich für grosse n (maximal) wie n^2 verhält: verdoppelt sich die Eingabelänge, so vervierfacht sich die Laufzeit (maximal)

Asymptotische obere Schranken – Formale Definition

$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c, wobei f und g positiv sind

Asymptotische obere Schranken – Formale Definition

$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c, wobei f und g positiv sind

$$f(n) \in \mathcal{O}(g(n))$$

$$\Longleftrightarrow$$

$$\exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0\colon\ f(n) \leq c \cdot g(n)$$

Asymptotische obere Schranken – Formale Definition

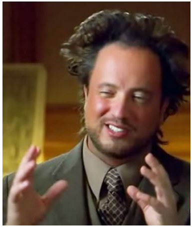
$\mathcal{O}$-Notation

Die Menge $\mathcal{O}(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c, wobei f und g positiv sind

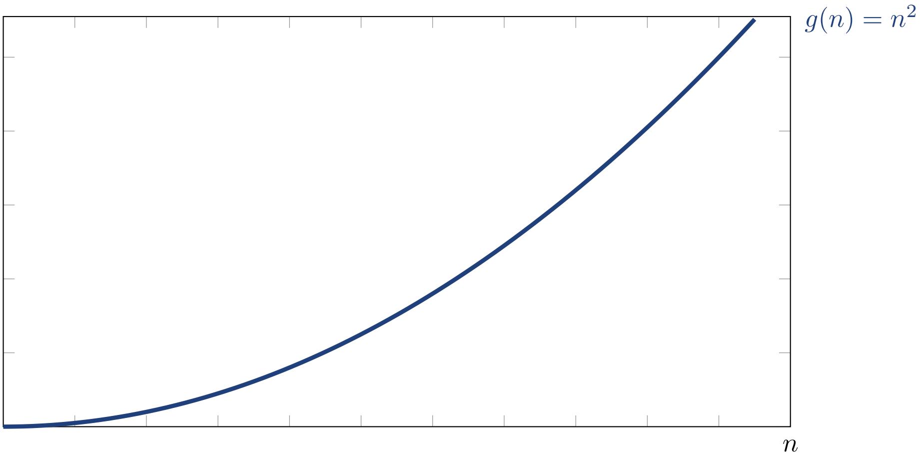
$$f(n) \in \mathcal{O}(g(n))$$

$$\Longleftrightarrow$$

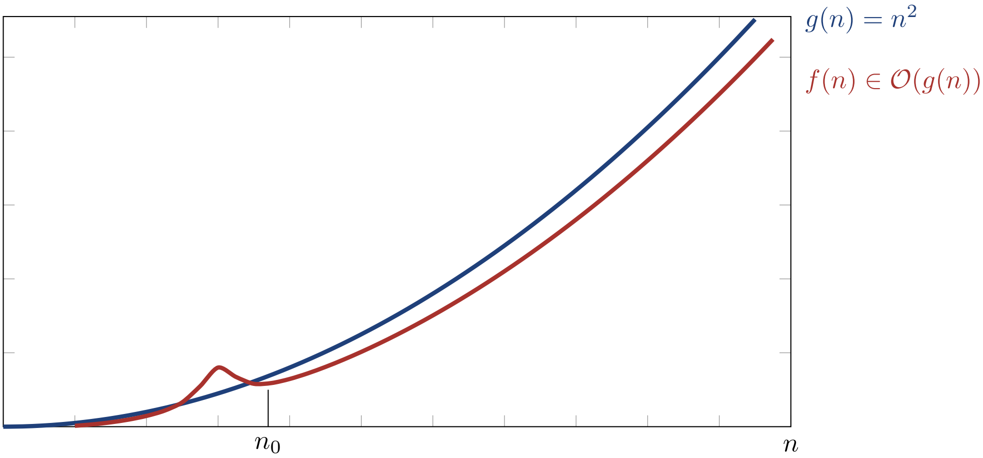
$$\exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0\colon\ f(n) \leq c \cdot g(n)$$



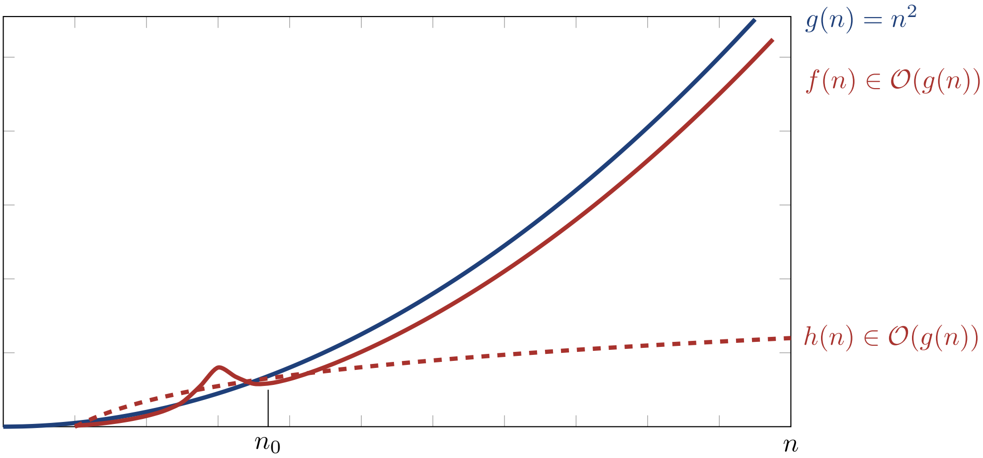
Asymptotische obere Schranken – Anschauung



Asymptotische obere Schranken – Anschauung



Asymptotische obere Schranken – Anschauung



Asymptotische obere Schranken – Beispiele

$$\mathcal{O}(g(n)) = \{f\colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	**Beispiel**
$3n + 4$		
$2n$		
$n^2 + 100n$		
$n + \sqrt{n}$		

Asymptotische obere Schranken – Beispiele

$$\mathcal{O}(g(n)) = \{f\colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	**Beispiel**
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$		
$n^2 + 100n$		
$n + \sqrt{n}$		

Asymptotische obere Schranken – Beispiele

$$\mathcal{O}(g(n)) = \{f\colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	**Beispiel**
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$		
$n + \sqrt{n}$		

Asymptotische obere Schranken – Beispiele

$$\mathcal{O}(g(n)) = \{f\colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	**Beispiel**
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$		

Asymptotische obere Schranken – Beispiele

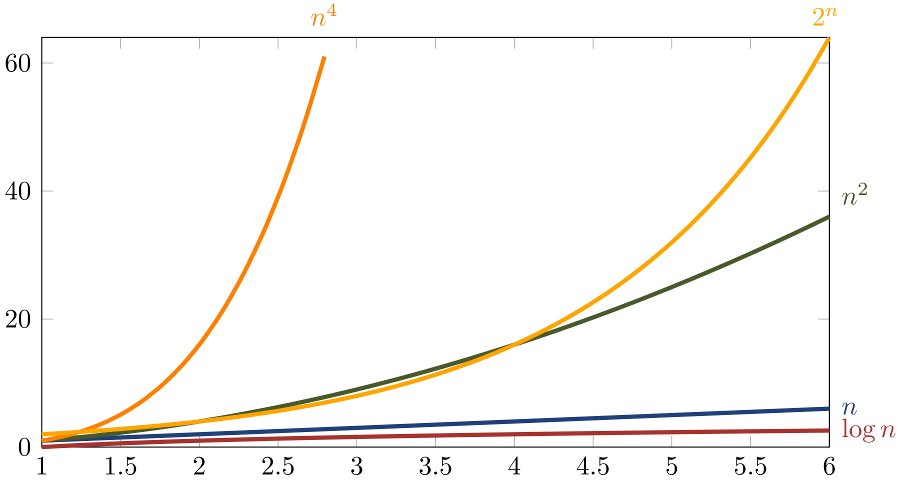
$$\mathcal{O}(g(n)) = \{f\colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	**Beispiel**
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

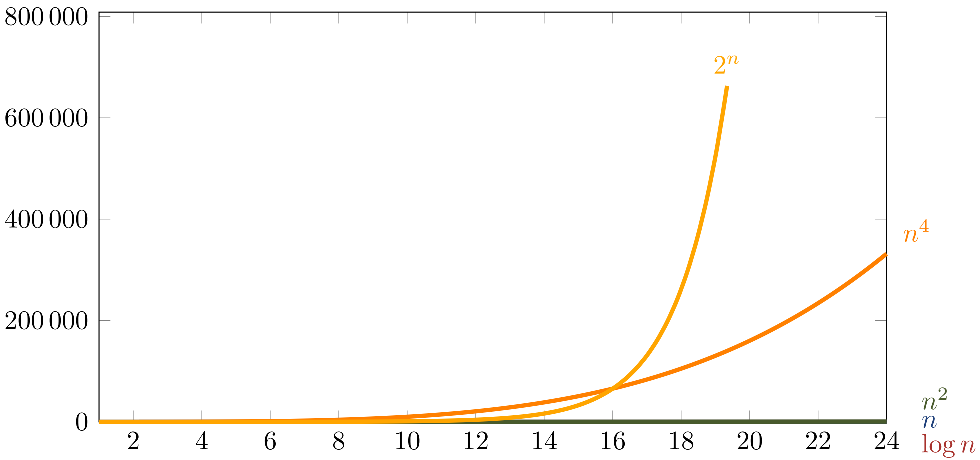
Komplexität von Algorithmen

Laufzeit-Analyse

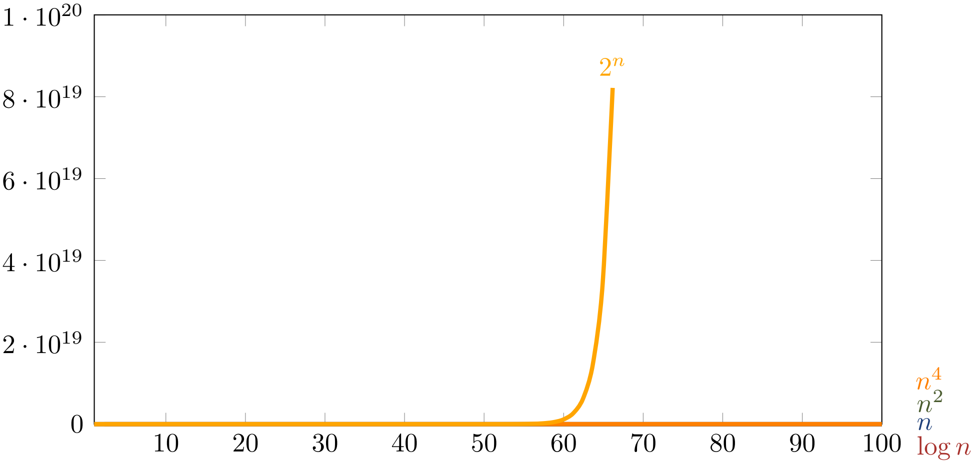
Kleine n



Grössere n


800 000
600 000
400 000
200 000
0
2
4
6
8
10
12
14
16
18
20
22
24
2^n
n^4
n^2
n
$\log n$

„Grosse“ n


$1 \cdot 10^{20}$
$8 \cdot 10^{19}$
$6 \cdot 10^{19}$
$4 \cdot 10^{19}$
$2 \cdot 10^{19}$
0
10
20
30
40
50
60
70
80
90
100
2^n
n^4
n^2
n
$\log n$

Ein schnellerer Primzahltest

Erster Versuch

Ein schnellerer Primzahltest

Ziel

Laufzeit echt besser als $\mathcal{O}(2^n)$

Ein schnellerer Primzahltest

Ziel

Laufzeit echt besser als $\mathcal{O}(2^n)$

Beobachtung

- Wenn eine Zahl $\mathtt{x}$ nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar

Ein schnellerer Primzahltest

Ziel

Laufzeit echt besser als $\mathcal{O}(2^n)$

Beobachtung

- Wenn eine Zahl $\mathtt{x}$ nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar
- Es reicht dann aus, nur ungerade Zahlen zu testen
- Algorithmus muss nur noch die Hälfte der Zahlen testen
- Schleife wird nur noch $\mathtt{x}/2$ Mal durchlaufen

Ein schnellerer Primzahltest

```python
def primetest2(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d < x:
        if x % d == 0:
            return False
        d += 2

    return True
```

Ein schnellerer Primzahltest

```python
def primetest2(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d < x:
        if x % d == 0:
            return False
        d += 2

    return True
```

Ein schnellerer Primzahltest

Wie gross ist die Verbesserung?

Ein schnellerer Primzahltest

Wie gross ist die Verbesserung?

- Schleife wird ca. $\mathtt{x}/2$ Mal durchlaufen anstatt $\mathtt{x}$ Mal
- Laufzeit verbessert sich um Faktor 2

Ein schnellerer Primzahltest

Wie gross ist die Verbesserung?

- Schleife wird ca. $\texttt{x}/2$ Mal durchlaufen anstatt $\texttt{x}$ Mal
- Laufzeit verbessert sich um Faktor 2
- Sei $\texttt{x}$ wieder mit n Bits dargestellt
- Insgesamt ca. $5 \cdot 2^n/2 = 2.5 \cdot 2^n$ elementare Operationen

Ein schnellerer Primzahltest

Wie gross ist die Verbesserung?

- Schleife wird ca. $\mathtt{x}/2$ Mal durchlaufen anstatt $\mathtt{x}$ Mal
- Laufzeit verbessert sich um Faktor 2
- Sei $\mathtt{x}$ wieder mit n Bits dargestellt
- Insgesamt ca. $5 \cdot 2^n/2 = 2.5 \cdot 2^n$ elementare Operationen
- Laufzeit noch immer in $\mathcal{O}(2^n)$

Ein schnellerer Primzahltest

Wie gross ist die Verbesserung?

- Schleife wird ca. $\mathtt{x}/2$ Mal durchlaufen anstatt $\mathtt{x}$ Mal
- Laufzeit verbessert sich um Faktor 2
- Sei $\mathtt{x}$ wieder mit n Bits dargestellt
- Insgesamt ca. $5 \cdot 2^n/2 = 2.5 \cdot 2^n$ elementare Operationen
- Laufzeit noch immer in $\mathcal{O}(2^n)$

$\Rightarrow$ Keine asymptotische Verbesserung

Ein schnellerer Primzahltest

Zweiter Versuch

Ein schnellerer Primzahltest

Beobachtung

- Wenn $\mathtt{x}$ mit $\mathtt{x} > 2$ keine Primzahl ist, dann ist $\mathtt{x}$ teilbar durch eine Zahl a mit

$$1 < a < \mathtt{x}$$

Ein schnellerer Primzahltest

Beobachtung

- Wenn $\mathtt{x}$ mit $\mathtt{x} > 2$ keine Primzahl ist, dann ist $\mathtt{x}$ teilbar durch eine Zahl a mit

$$1 < a < \mathtt{x}$$

- Dann ist $\mathtt{x}$ auch durch Zahl b teilbar mit

$$a \cdot b = \mathtt{x} \quad \text{und} \quad 1 < b < \mathtt{x}$$

Ein schnellerer Primzahltest

Beobachtung

- Wenn $\mathtt{x}$ mit $\mathtt{x} > 2$ keine Primzahl ist, dann ist $\mathtt{x}$ teilbar durch eine Zahl a mit

$$1 < a < \mathtt{x}$$

- Dann ist $\mathtt{x}$ auch durch Zahl b teilbar mit

$$a \cdot b = \mathtt{x} \quad \text{und} \quad 1 < b < \mathtt{x}$$

- Es kann nicht sein, dass

$$a > \sqrt{\mathtt{x}} \quad \text{und} \quad b > \sqrt{\mathtt{x}},$$

denn sonst wäre

$$a \cdot b > \mathtt{x}$$

Ein schnellerer Primzahltest

Module einbinden

Module einbinden

Bislang wurden alle Funktionen in einer Datei definiert

Module einbinden

Bislang wurden alle Funktionen in einer Datei definiert

Module

- Teile Funktionen auf mehrere Dateien auf
- Dateien können sich nicht „sehen“
- Funktionen können **importiert werden**
- Strukturierter Code

Module einbinden

Datei `funktionen.py`

```
def wurzel_ziehen(n):
    i = 1
    while i * i < n: # Gibt Wurzel der naechstgroesseren Quadratzahl aus
        i += 1
    return i
```

Module einbinden

Datei `funktionen.py`

```
def wurzel_ziehen(n):
    i = 1
    while i * i < n: # Gibt Wurzel der naechstgroesseren Quadratzahl aus
        i += 1
    return i
```

Datei `anwendung.py`

```



print(wurzel_ziehen(81))
```

Module einbinden

Datei `funktionen.py`

```
def wurzel_ziehen(n):
    i = 1
    while i * i < n: # Gibt Wurzel der naechstgroesseren Quadratzahl aus
        i += 1
    return i
```

Datei `anwendung.py`

```
from funktionen import wurzel_ziehen

print(wurzel_ziehen(81))
```

Module einbinden

Datei `funktionen.py`

```
def wurzel_ziehen(n):
    i = 1
    while i * i < n: # Gibt Wurzel der naechstgroesseren Quadratzahl aus
        i += 1
    return i
```

Datei `anwendung.py`

```
from funktionen import *

print(wurzel_ziehen(81))
```

Module einbinden

- Eine grosse Anzahl von Modulen existiert bereits
- Beispielsweise `math`, welches bereits eine Funktion **`sqrt()`** zum Wurzelziehen enthält

Module einbinden

- Eine grosse Anzahl von Modulen existiert bereits
- Beispielsweise `math`, welches bereits eine Funktion `sqrt()` zum Wurzelziehen enthält

```
print(sqrt(9))
```

Module einbinden

- Eine grosse Anzahl von Modulen existiert bereits
- Beispielsweise `math`, welches bereits eine Funktion `sqrt()` zum Wurzelziehen enthält

```
print(sqrt(9))
```

```
NameError: name 'sqrt' is not defined
```

Module einbinden

- Eine grosse Anzahl von Modulen existiert bereits
- Beispielsweise `math`, welches bereits eine Funktion `sqrt()` zum Wurzelziehen enthält

```
from math import sqrt

print(sqrt(9))
```

Ausgabe: 3

Ein schnellerer Primzahltest

```
def primetest3(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d < x:
        if x % d == 0:
            return False
        d += 2
    return True
```

Ein schnellerer Primzahltest

```
from math import sqrt

def primetest3(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d <= sqrt(x):
        if x % d == 0:
            return False
        d += 2
    return True
```

Ein schnellerer Primzahltest

Wie gross ist diesmal die Verbesserung?

Ein schnellerer Primzahltest

Wie gross ist diesmal die Verbesserung?

- Was ist die Laufzeit dieses Algorithmus?

Ein schnellerer Primzahltest

Wie gross ist diesmal die Verbesserung?

- Was ist die Laufzeit dieses Algorithmus?
- Schleife wird $\sqrt{\mathbf{x}}/2$ Mal durchlaufen

Ein schnellerer Primzahltest

Wie gross ist diesmal die Verbesserung?

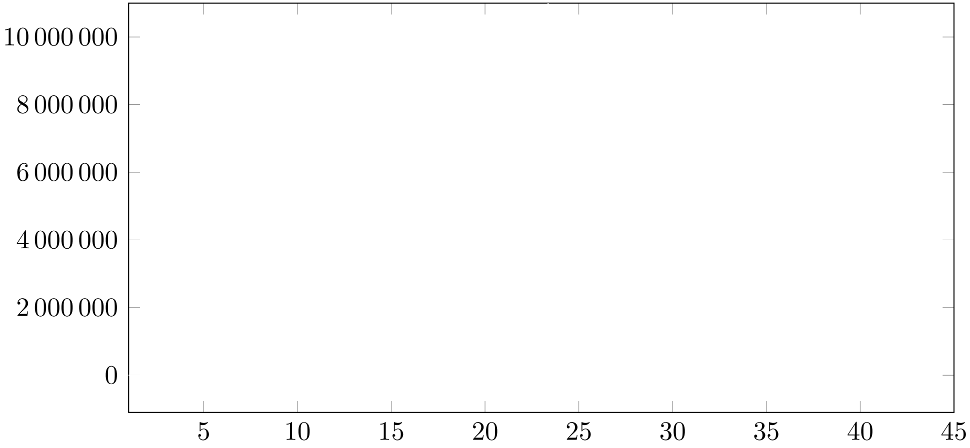
- Was ist die Laufzeit dieses Algorithmus?
- Schleife wird $\sqrt{\mathtt{x}}/2$ Mal durchlaufen
- Laufzeit „wächst“ mit Grösse des Werts von $\sqrt{\mathtt{x}}$

Ein schnellerer Primzahltest

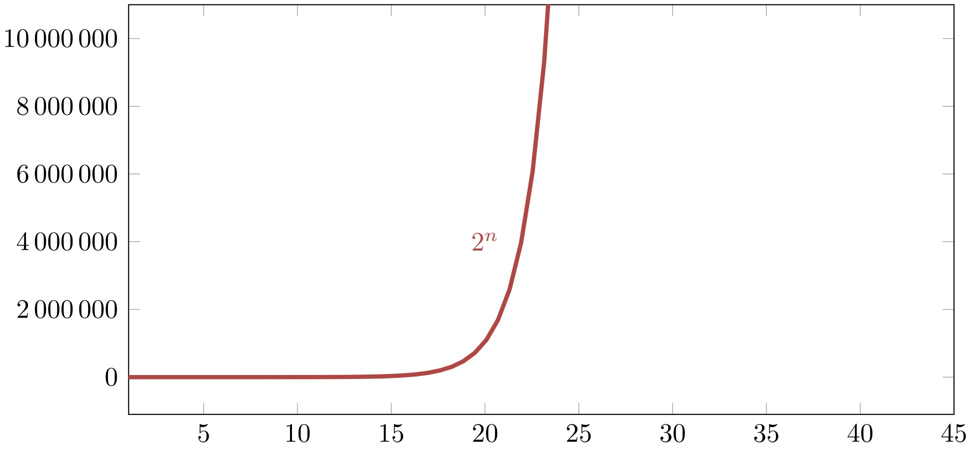
Wie gross ist diesmal die Verbesserung?

- Was ist die Laufzeit dieses Algorithmus?
- Schleife wird $\sqrt{\mathtt{x}}/2$ Mal durchlaufen
- Laufzeit „wächst“ mit Grösse des Werts von $\sqrt{\mathtt{x}}$
- Laufzeit in $\mathcal{O}(\sqrt{2^n}) = \mathcal{O}(2^{n/2}) = \mathcal{O}(1.415^n)$

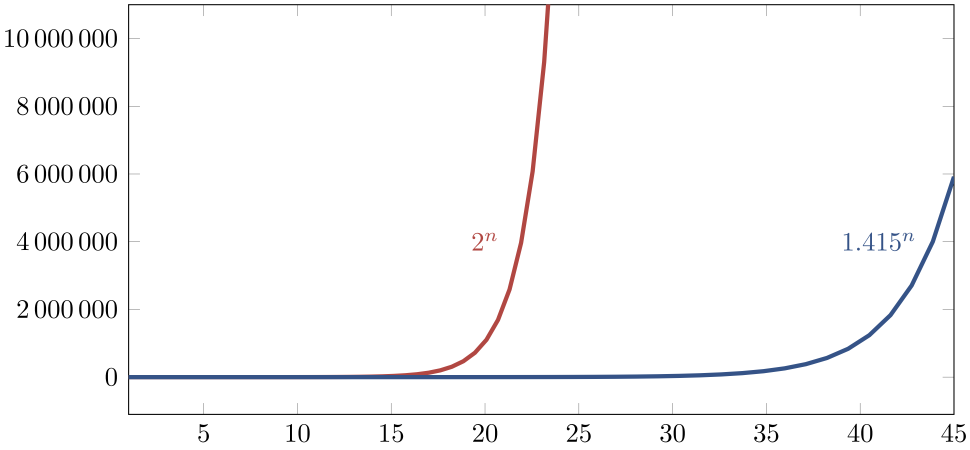
Ein schnellerer Primzahltest



Ein schnellerer Primzahltest



Ein schnellerer Primzahltest



Ein schnellerer Primzahltest

Nehmen wir vereinfacht an, unser Computer kann 1000 Durchläufe der Schleife pro Sekunde durchführen

Ein schnellerer Primzahltest

Nehmen wir vereinfacht an, unser Computer kann 1000 Durchläufe der Schleife pro Sekunde durchführen; für $\texttt{x} = 100\,000\,000\,000\,031$ bedeutet dies:

```
... d < x ...
```

$$\frac{100\,000\,000\,000\,031 \text{ Durchläufe}}{1000\,\frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$> 100\,000\,000\,000 \text{ Sekunden}$$

$$> 3100 \text{ Jahre}$$

Ein schnellerer Primzahltest

Nehmen wir vereinfacht an, unser Computer kann 1000 Durchläufe der Schleife pro Sekunde durchführen; für $\texttt{x} = 100\,000\,000\,000\,031$ bedeutet dies:

`... d < x ...`	`... d <= sqrt(x) ...`

$$\frac{100\,000\,000\,000\,031 \text{ Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$> 100\,000\,000\,000 \text{ Sekunden}$$

$$> 3100 \text{ Jahre}$$

$$\frac{\sqrt{100\,000\,000\,000\,031} \text{ Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$< \frac{10\,000\,000 \text{ Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$< 3 \text{ Stunden}$$

Ein schnellerer Primzahltest

Nehmen wir vereinfacht an, unser Computer kann 1000 Durchläufe der Schleife pro Sekunde durchführen; für $\mathtt{x} = 100\,000\,000\,000\,031$ bedeutet dies:

`... d < x ...`	`... d <= sqrt(x) ...`

$$\frac{100\,000\,000\,000\,031 \text{ Durchläufe}}{1000 \,\frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$> 100\,000\,000\,000 \text{ Sekunden}$$

$$> 3100 \text{ Jahre}$$

$$\frac{\sqrt{100\,000\,000\,000\,031} \text{ Durchläufe}}{1000 \,\frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$< \frac{10\,000\,000 \text{ Durchläufe}}{1000 \,\frac{\text{Durchläufe}}{\text{Sekunde}}}$$

$$< 3 \text{ Stunden}$$

Selbst wenn der Computer, auf dem das langsamere Programm läuft, 100 Mal schneller ist, braucht er noch 31 Jahre

Ein schnellerer Primzahltest

Oder andersherum...

Angenommen, wir wollen maximal zehn Minuten rechnen

Ein schnellerer Primzahltest

Oder andersherum...

Angenommen, wir wollen maximal zehn Minuten rechnen

Dann ergeben sich maximal „testbare“ Primzahlen in den folgenden Grössenordnungen:

Ein schnellerer Primzahltest

Oder andersherum. . .

Angenommen, wir wollen maximal zehn Minuten rechnen

Dann ergeben sich maximal „testbare“ Primzahlen in den folgenden Grössenordnungen:

```
... d < x ...
```

$$\frac{\mathtt{x}\ \text{Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

$$\Longleftrightarrow \mathtt{x} = 600\,000$$

Ein schnellerer Primzahltest

Oder andersherum...

Angenommen, wir wollen maximal zehn Minuten rechnen

Dann ergeben sich maximal „testbare“ Primzahlen in den folgenden Grössenordnungen:

```
... d < x ...
```

$$\frac{\texttt{x}\ \text{Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

$$\Longleftrightarrow \texttt{x} = 600\,000$$

```
... d <= sqrt(x) ...
```

$$\frac{\sqrt{\texttt{x}}\ \text{Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

$$\Longleftrightarrow \texttt{x} = 600\,000^2$$

$$\Longleftrightarrow \texttt{x} = 360\,000\,000\,000$$

Ein schnellerer Primzahltest

Best- und Worst-Case-Analyse

Best- und Worst-Case-Analyse

Welcher Algorithmus ist schneller?

```
def primetest3(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d <= sqrt(x):
        if x % d == 0:
            return False
        d += 2
    return True
```

```
def primetest4(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    isprime = True
    while d <= sqrt(x):
        if x % d == 0:
            isprime = False
        d += 2
    return isprime
```

Best- und Worst-Case-Analyse

Angenommen, `x` ist durch 3 teilbar

Best- und Worst-Case-Analyse

Angenommen, `x` ist durch 3 teilbar

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“
- Rechter Algorithmus macht ca. $1.415^n/2$ Vergleiche

Best- und Worst-Case-Analyse

Angenommen, `x` ist durch 3 teilbar

- Dann ist der linke Algorithmus sehr schnell
- ⇨ Schleife wird nach dem ersten Vergleich verlassen
- „Early Exit“
- Rechter Algorithmus macht ca. $1.415^n/2$ Vergleiche

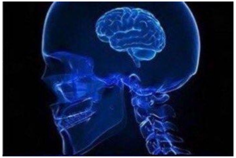
Angenommen, `x` ist Primzahl

- Dann machen beide Algorithmen ca. $1.415^n/2$ Vergleiche
- (Natürlich sollte der linke implementiert werden)

Was kann man sonst noch machen?

Primzahltest

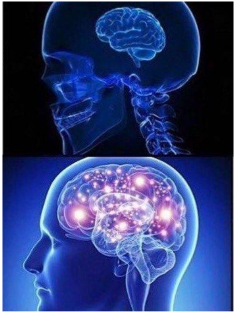
Jede Zahl zwischen
1 und $\mathtt{x}$ testen



Primzahltest

Jede Zahl zwischen
1 und x testen

Jede zweite Zahl zwischen
1 und x testen



Primzahltest

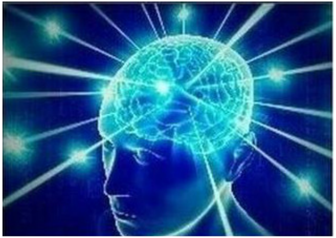
Jede Zahl zwischen
1 und x testen

Jede zweite Zahl zwischen
1 und x testen

Jede zweite Zahl zwischen
1 und $\sqrt{x}$ testen

Primzahltest

Randomisierter
Monte-Carlo-
Algorithmus



Primzahltest

Randomisierter
Monte-Carlo-
Algorithmus

Polynomieller
AKS-Algorithmus

Monte-Carlo-Algorithmus

Monte-Carlo-Algorithmus – Grundidee

Randomisierte Algorithmen machen Zufallsentscheidungen

Monte-Carlo-Algorithmus – Grundidee

Randomisierte Algorithmen machen Zufallsentscheidungen

- Eingabe x „determiniert“ Ausgabe nicht mehr
- Dasselbe x kann zu verschiedenen Ausgaben führen
- **Monte-Carlo-Algorithmus** (MC-Algorithmus) hat beschränkte Fehlerwahrscheinlichkeit
- Für `True`-/`False`-Probleme (Primzahltest etc.) gibt es MC-Algorithmen mit **einseitigem Fehler** (1MC-Algorithmus)

Monte-Carlo-Algorithmus – Grundidee

Randomisierte Algorithmen machen Zufallsentscheidungen

- Eingabe x „determiniert“ Ausgabe nicht mehr
- Dasselbe x kann zu verschiedenen Ausgaben führen
- **Monte-Carlo-Algorithmus** (MC-Algorithmus) hat beschränkte Fehlerwahrscheinlichkeit
- Für `True`-/`False`-Probleme (Primzahltest etc.) gibt es MC-Algorithmen mit **einseitigem Fehler** (1MC-Algorithmus)
- **Las-Vegas-Algorithmus** hat Fehlerwahrscheinlichkeit 0

Monte-Carlo-Algorithmus (1MC) – Beispiel

Urne mit 10^{100} Kugeln mit Farben weiss (und womöglich rot)

Monte-Carlo-Algorithmus (1MC) – Beispiel

Urne mit 10^{100} Kugeln mit Farben weiss (und womöglich rot)

- **Behauptung:** Nicht alle Kugeln in Urne sind weiss
- Wie testen?
- Zufallsstichprobe

Monte-Carlo-Algorithmus (1MC) – Beispiel

Urne mit 10^{100} Kugeln mit Farben weiss (und womöglich rot)

- **Behauptung:** Nicht alle Kugeln in Urne sind weiss
- Wie testen?
- Zufallsstichprobe

⇨ Falls eine rote Kugel in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **keine** rote Kugel in Stichprobe ⇨ Behauptung womöglich falsch

Monte-Carlo-Algorithmus (1MC) – Beispiel

Urne mit 10^{100} Kugeln mit Farben weiss (und womöglich rot)

- **Behauptung:** Nicht alle Kugeln in Urne sind weiss
- Wie testen?
- Zufallsstichprobe

⇨ Falls eine rote Kugel in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **keine** rote Kugel in Stichprobe ⇨ Behauptung womöglich falsch

- Einseitiger Fehler

Monte-Carlo-Algorithmus (1MC) – Beispiel

Urne mit 10^{100} Kugeln mit Farben weiss (und womöglich rot)

- **Behauptung:** Nicht alle Kugeln in Urne sind weiss
- Wie testen?
- Zufallsstichprobe

⇨ Falls eine rote Kugel in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **keine** rote Kugel in Stichprobe ⇨ Behauptung womöglich falsch

- Einseitiger Fehler

Rote Kugeln sind **Zeugen** für Behauptung

Einfacher Solovay-Strassen-Primzahltest

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Test, ob `x` Primzahl ist

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Test, ob `x` Primzahl ist
- **Behauptung:** `x` ist keine Primzahl
- Betrachte Menge $\{2, \ldots, \mathtt{x} - 1\}$ als Urne
- Teiler von `x` ist Zeuge für Behauptung
- Zufallsstichprobe

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Test, ob $\mathtt{x}$ Primzahl ist
- **Behauptung:** $\mathtt{x}$ ist keine Primzahl
- Betrachte Menge $\{2, \ldots, \mathtt{x}-1\}$ als Urne
- Teiler von $\mathtt{x}$ ist Zeuge für Behauptung
- Zufallsstichprobe

⇨ Falls ein Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **kein** Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung womöglich falsch

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Test, ob $\mathtt{x}$ Primzahl ist
- **Behauptung:** $\mathtt{x}$ ist keine Primzahl
- Betrachte Menge $\{2, \ldots, \mathtt{x} - 1\}$ als Urne
- Teiler von $\mathtt{x}$ ist Zeuge für Behauptung
- Zufallsstichprobe

⇨ Falls ein Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **kein** Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung womöglich falsch

- Einseitiger Fehler

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Test, ob $\mathtt{x}$ Primzahl ist
- **Behauptung:** $\mathtt{x}$ ist keine Primzahl
- Betrachte Menge $\{2, \ldots, \mathtt{x}-1\}$ als Urne
- Teiler von $\mathtt{x}$ ist Zeuge für Behauptung
- Zufallsstichprobe

⇨ Falls ein Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung bewiesen

⇨ Falls **kein** Teiler von $\mathtt{x}$ in Stichprobe ⇨ Behauptung womöglich falsch

- Einseitiger Fehler

Für $\mathtt{x} = p \cdot q$ mit Primzahlen p und q ist Wahrscheinlichkeit für Zeugen

$$\frac{2}{\mathtt{x}-2}$$

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Finde „bessere Zeugen“

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Finde „bessere Zeugen“
- (Nicht ganz triviale Zahlentheorie)

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Finde „bessere Zeugen“
- (Nicht ganz triviale Zahlentheorie)
- **Kleiner Satz von Fermat**

Falls $\mathtt{x}$ prim $\Rightarrow a^{\mathtt{x}-1} \equiv 1 \pmod{\mathtt{x}} \quad \forall a \in \{2, \ldots, \mathtt{x}-1\}$

Pierre de Fermat (1607–1665)

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Falls $\mathtt{x}$ prim $\quad \Rightarrow \quad a^{\mathtt{x}-1} \mod \mathtt{x} = 1 \quad \forall a \in \{2, \ldots, \mathtt{x}-1\}$

$$\mathtt{x} = 3: \quad 2^2 \equiv 1 \pmod 3$$
$$\mathtt{x} = 5: \quad 2^4 \equiv 3^4 \equiv 1 \pmod 5$$

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Falls $\mathtt{x}$ prim $\quad \Rightarrow \quad a^{\mathtt{x}-1} \mod \mathtt{x} = 1 \quad \forall a \in \{2, \ldots, \mathtt{x}-1\}$

$$\mathtt{x} = 3: \quad 2^2 \equiv 1 \pmod 3$$
$$\mathtt{x} = 5: \quad 2^4 \equiv 3^4 \equiv 1 \pmod 5$$

- Falls also für ein a gilt: $\quad a^{\mathtt{x}-1} \mod \mathtt{x} \neq 1$
  - $\mathtt{x}$ ist **garantiert** keine Primzahl
  - a ist Zeuge, dass $\mathtt{x}$ keine Primzahl ist
  - Man kann zeigen, dass die Anzahl Zeugen $> (\mathtt{x}-2)/2$ ist $> (\mathtt{x}-2)/2$

Einfacher Solovay-Strassen-Primzahltest (1MC)

- Falls $\mathtt{x}$ prim $\quad \Rightarrow \quad a^{\mathtt{x}-1} \mod \mathtt{x} = 1 \quad \forall a \in \{2, \dots, \mathtt{x}-1\}$

$$\mathtt{x} = 3: \quad 2^2 \equiv 1 \pmod 3$$
$$\mathtt{x} = 5: \quad 2^4 \equiv 3^4 \equiv 1 \pmod 5$$

- Falls also für ein a gilt: $\quad a^{\mathtt{x}-1} \mod \mathtt{x} \neq 1$
  - $\mathtt{x}$ ist **garantiert** keine Primzahl
  - a ist Zeuge, dass $\mathtt{x}$ keine Primzahl ist
  - Man kann zeigen, dass die Anzahl Zeugen $> (\mathtt{x}-2)/2$ ist $> (\mathtt{x}-2)/2$
- Andernfalls ist $\mathtt{x}$ womöglich prim

Einfacher Solovay-Strassen-Primzahltest (1MC)

- **Eingabe:** Zahl `x`

Einfacher Solovay-Strassen-Primzahltest (1MC)

- **Eingabe:** Zahl $\mathtt{x}$
- Wähle a zufällig aus $\in \{2, \ldots, \mathtt{x} - 1\}$

Einfacher Solovay-Strassen-Primzahltest (1MC)

- **Eingabe:** Zahl $\mathtt{x}$
- Wähle a zufällig aus $\in \{2, \ldots, \mathtt{x}-1\}$
- Berechne $z = a^{\mathtt{x}-1} \mod \mathtt{x}$

Einfacher Solovay-Strassen-Primzahltest (1MC)

- **Eingabe:** Zahl $\mathtt{x}$
- Wähle a zufällig aus $\in \{2, \ldots, \mathtt{x} - 1\}$
- Berechne $z = a^{\mathtt{x}-1} \mod \mathtt{x}$
- Falls $z \neq 1$: **Ausgabe** „$\mathtt{x}$ keine Primzahl"

Einfacher Solovay-Strassen-Primzahltest (1MC)

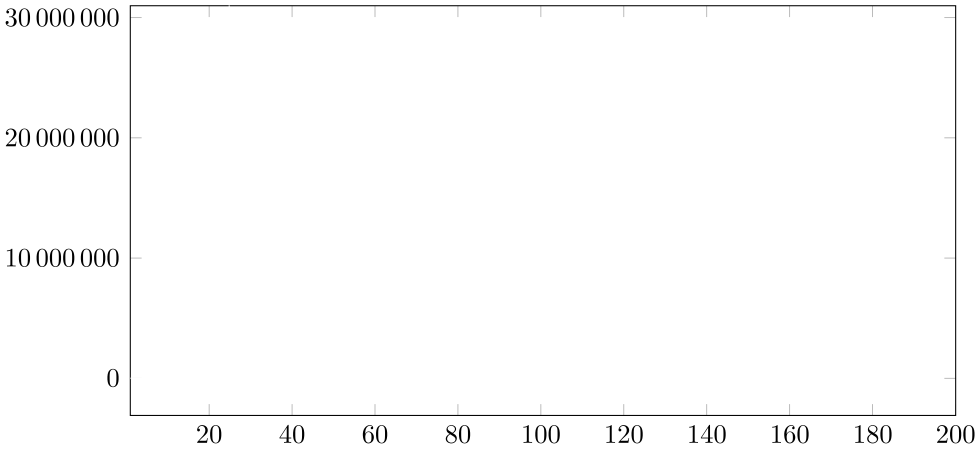
- **Eingabe:** Zahl $\mathtt{x}$
- Wähle a zufällig aus $\in \{2, \ldots, \mathtt{x} - 1\}$
- Berechne $z = a^{\mathtt{x}-1} \mod \mathtt{x}$
- Falls $z \neq 1$: **Ausgabe** „$\mathtt{x}$ keine Primzahl“
- Sonst: **Ausgabe** „$\mathtt{x}$ womöglich Primzahl“

Einfacher Solovay-Strassen-Primzahltest (1MC)

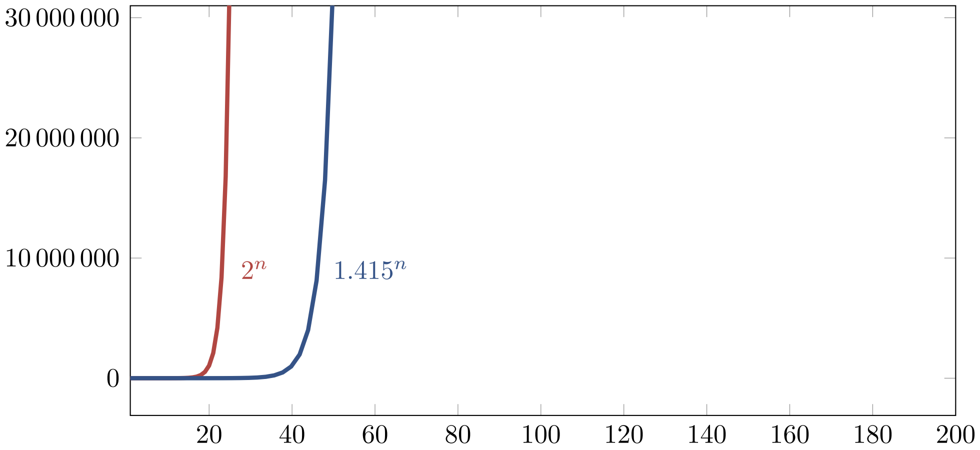
- **Eingabe:** Zahl $\mathbf{x}$
- Wähle a zufällig aus $\in \{2, \ldots, \mathbf{x} - 1\}$
- Berechne $z = a^{\mathbf{x}-1} \mod \mathbf{x}$
- Falls $z \neq 1$: **Ausgabe** „$\mathbf{x}$ keine Primzahl“
- Sonst: **Ausgabe** „$\mathbf{x}$ womöglich Primzahl“

- Kann in Polynomzeit berechnet werden
- Laufzeit $\mathcal{O}(n^3)$ statt $\mathcal{O}(1.415^n)$
- Effizienter Algorithmus

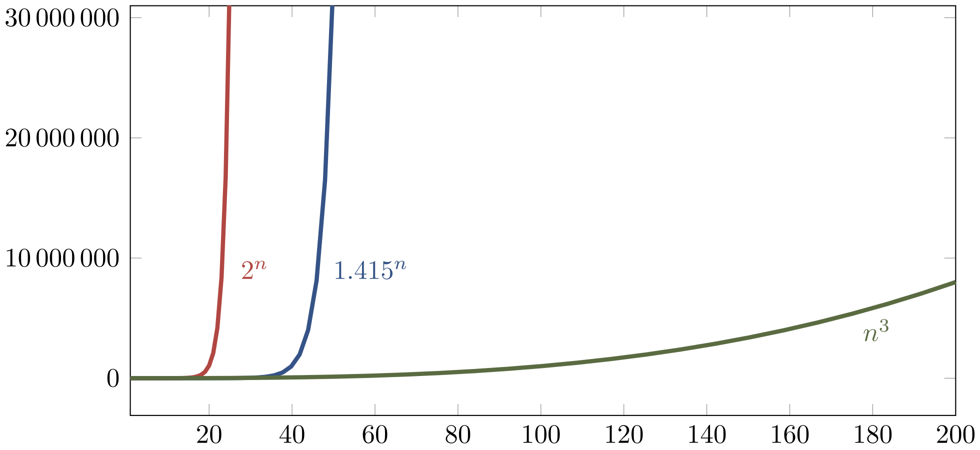
Einfacher Solovay-Strassen-Primzahltest (1MC)



Einfacher Solovay-Strassen-Primzahltest (1MC)



Einfacher Solovay-Strassen-Primzahltest (1MC)



Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei `x` Primzahl

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei $\mathtt{x}$ Primzahl
- Nach kleinem Satz von Fermat kein Zeuge in $\{2, \ldots, \mathtt{x} - 1\}$

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei $\mathtt{x}$ Primzahl
- Nach kleinem Satz von Fermat kein Zeuge in $\{2, \ldots, \mathtt{x} - 1\}$
- Richtige Antwort mit Wahrscheinlichkeit 1

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei $\mathtt{x}$ Primzahl
- Nach kleinem Satz von Fermat kein Zeuge in $\{2, \ldots, \mathtt{x} - 1\}$
- Richtige Antwort mit Wahrscheinlichkeit 1
- Sei $\mathtt{x}$ keine Primzahl

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei $\mathtt{x}$ Primzahl
- Nach kleinem Satz von Fermat kein Zeuge in $\{2, \ldots, \mathtt{x} - 1\}$
- Richtige Antwort mit Wahrscheinlichkeit 1
- Sei $\mathtt{x}$ keine Primzahl
- Mindestens Hälfte in $\{2, \ldots, \mathtt{x} - 1\}$ sind Zeugen

Einfacher Solovay-Strassen-Primzahltest (1MC)

Algorithmus besitzt einseitigen Fehler

- Sei $\mathtt{x}$ Primzahl
- Nach kleinem Satz von Fermat kein Zeuge in $\{2, \ldots, \mathtt{x} - 1\}$
- Richtige Antwort mit Wahrscheinlichkeit 1
- Sei $\mathtt{x}$ keine Primzahl
- Mindestens Hälfte in $\{2, \ldots, \mathtt{x} - 1\}$ sind Zeugen
- Richtige Antwort mit Wahrscheinlichkeit mindestens $1/2$

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben `x` laufen

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben `x` laufen
- **Falls** `x` Primzahl ist, Fehlerwahrscheinlichkeit 0

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben `x` laufen
- **Falls** `x` Primzahl ist, Fehlerwahrscheinlichkeit 0
- **Sonst** muss nur einmal Zeuge gefunden werden

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben `x` laufen
- **Falls** `x` Primzahl ist, Fehlerwahrscheinlichkeit 0
- **Sonst** muss nur einmal Zeuge gefunden werden
- W'keit $< 1/2$, dass im 1. Lauf kein Zeuge gefunden wird

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben `x` laufen
- **Falls** `x` Primzahl ist, Fehlerwahrscheinlichkeit 0
- **Sonst** muss nur einmal Zeuge gefunden werden
- W'keit $< 1/2$, dass im 1. Lauf kein Zeuge gefunden wird
- W'keit $< 1/4$, dass im 1. und 2. Lauf kein Zeuge gefunden wird

Einfacher Solovay-Strassen-Primzahltest (1MC)

Wahrscheinlichkeitsverstärkung durch mehrmaliges Ausführen und jeweils unabhängiger Wahl von a

- Lasse Algorithmus k Mal auf demselben $\mathtt{x}$ laufen
- **Falls** $\mathtt{x}$ Primzahl ist, Fehlerwahrscheinlichkeit 0
- **Sonst** muss nur einmal Zeuge gefunden werden
- W'keit $< 1/2$, dass im 1. Lauf kein Zeuge gefunden wird
- W'keit $< 1/4$, dass im 1. und 2. Lauf kein Zeuge gefunden wird
- W'keit $< 1/2^k$, dass in allen k Läufen kein Zeuge gefunden wird

Danke für die Aufmerksamkeit