ETH zürich

ClevAlg 2019 – Primzahltests

Clevere Algorithmen programmieren

Dennis Komm, Jakub Závodný

30. Oktober 2019

Typische Aufgabe einer Informatikerin

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

  `teiler` grösser als 1

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

  `teiler` grösser als 1

  und `teiler` kleiner als `eingabe`

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

  `teiler` grösser als 1

  und `teiler` kleiner als `eingabe`

  ob `teiler` die Zahl `eingabe` teilt

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

  `teiler` grösser als 1

  und `teiler` kleiner als `eingabe`

  ob `teiler` die Zahl `eingabe` teilt
- Finden wir so einen `teiler`, ist `eingabe` keine Primzahl

Typische Aufgabe einer Informatikerin

Testen, ob eine Zahl eine Primzahl ist

- Eingabe ist Zahl `eingabe`
- Teste für jede Zahl `teiler` mit

  `teiler` grösser als 1

  und `teiler` kleiner als `eingabe`

  ob `teiler` die Zahl `eingabe` teilt
- Finden wir so einen `teiler`, ist `eingabe` keine Primzahl
- Sonst schon

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

- $10 \% 3 = 1$, denn $9 = 3 \cdot 3$

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

- $10 \,\%\, 3 = 1$, denn $9 = 3 \cdot 3$
- $10 \,\%\, 4 = 2$, denn $8 = 4 \cdot 2$

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

- $10 \% 3 = 1$, denn $9 = 3 \cdot 3$
- $10 \% 4 = 2$, denn $8 = 4 \cdot 2$
- $11 \% 5 = 1$, denn $10 = 5 \cdot 2$

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

- $10 \% 3 = 1$, denn $9 = 3 \cdot 3$
- $10 \% 4 = 2$, denn $8 = 4 \cdot 2$
- $11 \% 5 = 1$, denn $10 = 5 \cdot 2$
- $12 \% 3 = 0$, denn $12 = 4 \cdot 3$

Teilen mit Rest (Modulo-Rechnen)

Mit „%“ kann getestet werden, ob eine Zahl eine andere teilt

- $10 \% 3 = 1$, denn $9 = 3 \cdot 3$
- $10 \% 4 = 2$, denn $8 = 4 \cdot 2$
- $11 \% 5 = 1$, denn $10 = 5 \cdot 2$
- $12 \% 3 = 0$, denn $12 = 4 \cdot 3$

⇨ Falls `eingabe % teiler` = 0,
dann wird `eingabe` von `teiler` geteilt

Primzahltest

Primzahltest

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1
print "Primzahl"
```

Primzahltest

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1
print "Primzahl"
```

- Wie lange braucht er, um Ausgabe zu erzeugen?

Primzahltest

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1
print "Primzahl"
```

- Wie lange braucht er, um Ausgabe zu erzeugen?

⇨ Was ist seine **Laufzeit**?

Primzahltest

```
eingabe = 2000003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1
print "Primzahl"
```

- Wie lange braucht er, um Ausgabe zu erzeugen?

⇨ Was ist seine **Laufzeit**?

- Hängt von Anzahl Schleifendurchläufe ab

Primzahltest

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1
print "Primzahl"
```

- Wie lange braucht er, um Ausgabe zu erzeugen?

⇨ Was ist seine **Laufzeit**?

- Hängt von Anzahl Schleifendurchläufe ab
- Schleife wird `eingabe` Mal durchlaufen

Laufzeit-Analyse

Ein Exkurs

Laufzeit

Zahlen werden im Computer binär dargestellt

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**
1	**1**

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**
1	**1**
2	**10**

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**
1	**1**
2	**10**
3	**11**

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**	4	**100**	8	**1000**	12	**1000**
1	**1**	5	**101**	9	**1001**	13	**1001**
2	**10**	6	**110**	10	**1010**	14	**1010**
3	**11**	7	**111**	11	**1011**	15	**1011**

Laufzeit

Zahlen werden im Computer binär dargestellt

0	**0**	4	**100**	8	**1000**	12	**1000**
1	**1**	5	**101**	9	**1001**	13	**1001**
2	**10**	6	**110**	10	**1010**	14	**1010**
3	**11**	7	**111**	11	**1011**	15	**1011**

Allgemein gilt

$2^{n-1} \quad \underbrace{10\ldots00}_{n}$ $\qquad 2^{n-1}+1 \quad \underbrace{10\ldots01}_{n}$ $\qquad 2^{n}-1 \quad \underbrace{11\ldots11}_{n}$

Laufzeit

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

Zurück zu den Primzahlen

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

⇨ Laufzeit „wächst“ mit Grösse des Werts von `eingabe`

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

⇨ Laufzeit „wächst“ mit Grösse des Werts von `eingabe`

- Sei `eingabe` mit n Bits dargestellt

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

⇨ Laufzeit „wächst“ mit Grösse des Werts von `eingabe`

- Sei `eingabe` mit n Bits dargestellt
- Dann ist `eingabe` also ungefähr 2^n

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

⇨ Laufzeit „wächst“ mit Grösse des Werts von `eingabe`

- Sei `eingabe` mit n Bits dargestellt
- Dann ist `eingabe` also ungefähr 2^n
- Also wächst die Laufzeit mit 2^n

Primzahltest

- Angenommen, `eingabe` ist eine Primzahl

⇨ Laufzeit „wächst“ mit Grösse des Werts von `eingabe`

- Sei `eingabe` mit n Bits dargestellt
- Dann ist `eingabe` also ungefähr 2^n
- Also wächst die Laufzeit mit 2^n
- **Exponentielle Laufzeit** bezüglich n

O-Notation

Noch ein Exkurs

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)
- Was macht der Algorithmus innerhalb der Schleife?

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)
- Was macht der Algorithmus innerhalb der Schleife?

⇨ Vier Operationen

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)
- Was macht der Algorithmus innerhalb der Schleife?

⇨ Vier Operationen

⇨ Insgesamt ca. $4 \cdot 2^n$ Operationen

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen
  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)
- Was macht der Algorithmus innerhalb der Schleife?

⇨ Vier Operationen

⇨ Insgesamt ca. $4 \cdot 2^n$ Operationen

- Uns interessiert, wie sich Laufzeit verhält, wenn n wächst

Asymptotische Laufzeit

- Die Schleife wird also ca. 2^n Mal ausgeführt
- Wir wollen **elementare Operationen** zählen

  (Addieren von zwei Zahlen, Vergleich von zwei Zahlen etc.)
- Was macht der Algorithmus innerhalb der Schleife?

⇨ Vier Operationen

⇨ Insgesamt ca. $4 \cdot 2^n$ Operationen

- Uns interessiert, wie sich Laufzeit verhält, wenn n wächst

⇨ Ignoriere Konstante 4

Asymptotische Laufzeit

O-Notation

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

Asymptotische Laufzeit

O-Notation

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

- $4 \cdot 2^n$ ist in $O(2^n)$

O-Notation

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

- $4 \cdot 2^n$ ist in $O(2^n)$
- $10 \cdot 2^n$ ist in $O(2^n)$

Asymptotische Laufzeit

O-Notation

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

- $4 \cdot 2^n$ ist in $O(2^n)$
- $10 \cdot 2^n$ ist in $O(2^n)$
- $1.6254 \cdot 2^n$ ist in $O(2^n)$

Asymptotische Laufzeit

O-Notation

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

- $4 \cdot 2^n$ ist in $O(2^n)$
- $10 \cdot 2^n$ ist in $O(2^n)$
- $1.6254 \cdot 2^n$ ist in $O(2^n)$
- $50 \cdot 2^n$ ist in $O(2^n)$

Asymptotische Laufzeit

O-Notation

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

- $4 \cdot 2^n$ ist in $O(2^n)$
- $10 \cdot 2^n$ ist in $O(2^n)$
- $1.6254 \cdot 2^n$ ist in $O(2^n)$
- $50 \cdot 2^n$ ist in $O(2^n)$
- $2\,000 \cdot 2^n$ ist in $O(2^n)$

Asymptotische Laufzeit

Und allgemein

Asymptotische Laufzeit

Und allgemein

O-Notation

Die Menge $O(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c

Und allgemein

O-Notation

Die Menge $O(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c

$$f(n) \in O(g(n))$$

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

Und allgemein

O-Notation

Die Menge $O(g(n))$ enthält alle Funktionen $f(n)$, die nicht schneller wachsen als $c \cdot g(n)$ für eine Konstante c

$$f(n) \in O(g(n))$$

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

Asymptotische Laufzeit

Und wieder konkret

- $2 \cdot n \in O(n)$

Asymptotische Laufzeit

Und wieder konkret

- $2 \cdot n \in O(n)$
- $4 \cdot n^2 \in O(n^2)$

Asymptotische Laufzeit

Und wieder konkret

- $2 \cdot n \in O(n)$
- $4 \cdot n^2 \in O(n^2)$
- $15 \cdot \log_2 n \in O(\log_2 n)$

Asymptotische Laufzeit

Und wieder konkret

- $2 \cdot n \in O(n)$
- $4 \cdot n^2 \in O(n^2)$
- $15 \cdot \log_2 n \in O(\log_2 n)$
- $10 \cdot n \in O(n^2)$

Asymptotische Laufzeit

Und wieder konkret

- $2 \cdot n \in O(n)$
- $4 \cdot n^2 \in O(n^2)$
- $15 \cdot \log_2 n \in O(\log_2 n)$
- $10 \cdot n \in O(n^2)$
- $\ldots$

Und wieder zurück zu den Primzahlen

Laufzeitverbesserung

Laufzeit

- Laufzeit also in $O(2^n)$

Laufzeit

- Laufzeit also in $O(2^n)$
- Geht es besser?

Laufzeit

- Laufzeit also in $O(2^n)$
- Geht es besser?

Beobachtung

- Wenn `eingabe` nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar.

Laufzeit

- Laufzeit also in $O(2^n)$
- Geht es besser?

Beobachtung

- Wenn `eingabe` nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar.
- Es reicht dann nur ungerade Zahlen zu testen

Laufzeit

- Laufzeit also in $O(2^n)$
- Geht es besser?

Beobachtung

- Wenn `eingabe` nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar.
- Es reicht dann nur ungerade Zahlen zu testen
- Programm muss nur noch die Hälfte der Zahlen testen

Laufzeit

- Laufzeit also in $O(2^n)$
- Geht es besser?

Beobachtung

- Wenn `eingabe` nicht durch 2 teilbar ist, dann ist sie auch nicht durch 4 oder 6 oder 8 etc. teilbar.
- Es reicht dann nur ungerade Zahlen zu testen
- Programm muss nur noch die Hälfte der Zahlen testen
- Schleife wird nur noch `eingabe`/2 Mal durchlaufen

Laufzeit

Dies führt zu einem verbesserten Primzahl-Algorithmus

Laufzeit

Dies führt zu einem verbesserten Primzahl-Algorithmus

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

Laufzeit

Dies führt zu einem verbesserten Primzahl-Algorithmus

```
eingabe = 2 000 003
teiler = 2

while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

```
eingabe = 2 000 003

rest = eingabe % 2
if rest == 0:
   print "Keine Primzahl"
   exit()

teiler = 3
while teiler < eingabe:
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 2

print "Primzahl"
```

Laufzeit

- Schleife wird ca. `eingabe`/2 Mal durchlaufen

Laufzeit

- Schleife wird ca. `eingabe`/2 Mal durchlaufen

⇨ Laufzeit verbessert sich um Faktor 2

Laufzeit

- Schleife wird ca. `eingabe`/2 Mal durchlaufen

⇨ Laufzeit verbessert sich um Faktor 2

- Sei wieder `eingabe` mit n Bits dargestellt

Laufzeit

- Schleife wird ca. `eingabe`/2 Mal durchlaufen

⇒ Laufzeit verbessert sich um Faktor 2

- Sei wieder `eingabe` mit n Bits dargestellt
- Insgesamt ca. $4 \cdot 2^n/2 = 2 \cdot 2^n$ Operationen

Laufzeit

- Schleife wird ca. `eingabe`$/2$ Mal durchlaufen

⇨ Laufzeit verbessert sich um Faktor 2

- Sei wieder `eingabe` mit n Bits dargestellt
- Insgesamt ca. $4 \cdot 2^n/2 = 2 \cdot 2^n$ Operationen

⇨ Laufzeit noch immer in $O(2^n)$

Laufzeit

- Schleife wird ca. `eingabe`/2 Mal durchlaufen

⇨ Laufzeit verbessert sich um Faktor 2

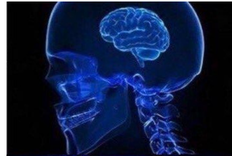
- Sei wieder `eingabe` mit n Bits dargestellt
- Insgesamt ca. $4 \cdot 2^n/2 = 2 \cdot 2^n$ Operationen

⇨ Laufzeit noch immer in $O(2^n)$

(Natürlich sollte es dennoch implementiert werden)

Laufzeit

Jede Zahl zwischen
1 und `eingabe` testen



Laufzeit

Jede Zahl zwischen
1 und `eingabe` testen

Jede zweite Zahl zwischen
1 und `eingabe` testen

Laufzeit

Jede Zahl zwischen
1 und `eingabe` testen

Jede zweite Zahl zwischen
1 und `eingabe` testen

Jede (zweite) Zahl zwischen
1 und $\sqrt{\texttt{eingabe}}$ testen

Laufzeit

Beobachtung

- Wenn `eingabe` keine Primzahl ist, dann ist `eingabe` teilbar durch eine Zahl a mit

$$1 < a < \texttt{eingabe}.$$

Laufzeit

Beobachtung

- Wenn `eingabe` keine Primzahl ist, dann ist `eingabe` teilbar durch eine Zahl a mit

$$1 < a < \texttt{eingabe}.$$

⇨ Dann ist `eingabe` auch durch Zahl b teilbar mit

$$a \cdot b = \texttt{eingabe} \quad \text{und} \quad 1 < b < \texttt{eingabe}$$

Laufzeit

Beobachtung

- Wenn `eingabe` keine Primzahl ist, dann ist `eingabe` teilbar durch eine Zahl a mit

$$1 < a < \texttt{eingabe}.$$

⇨ Dann ist `eingabe` auch durch Zahl b teilbar mit

$$a \cdot b = \texttt{eingabe} \quad \text{und} \quad 1 < b < \texttt{eingabe}$$

- Es kann nicht sein, dass

$$a > \sqrt{\texttt{eingabe}} \quad \text{und} \quad b > \sqrt{\texttt{eingabe}},$$

denn sonst wäre

$$a \cdot b > \texttt{eingabe}$$

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

```
eingabe = 2 000 003
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

```
eingabe = 2000003
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

⇨ Was ist die Laufzeit dieses Algorithmus?

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

```
eingabe = 2000003
teiler = 2

while teiler <= sqrt(eingabe):
    rest = eingabe % teiler
    if rest == 0:
        print "Keine Primzahl"
        exit()
    teiler += 1

print "Primzahl"
```

⇨ Was ist die Laufzeit dieses Algorithmus?

- Schleife wird $\sqrt{\texttt{eingabe}}$ Mal durchlaufen

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

```
eingabe = 2000003
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

⇨ Was ist die Laufzeit dieses Algorithmus?

- Schleife wird $\sqrt{\texttt{eingabe}}$ Mal durchlaufen

⇨ Laufzeit „wächst“ mit Grösse des Werts von $\sqrt{\texttt{eingabe}}$

Laufzeit

Dies führt zu einem weiter verbesserten Primzahl-Algorithmus

```
eingabe = 2000003
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

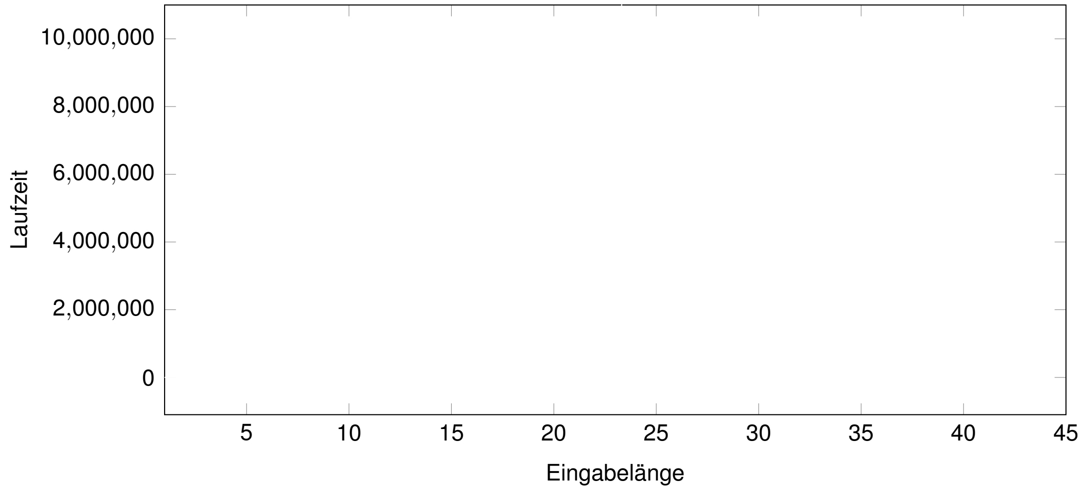
⇨ Was ist die Laufzeit dieses Algorithmus?

- Schleife wird $\sqrt{\texttt{eingabe}}$ Mal durchlaufen

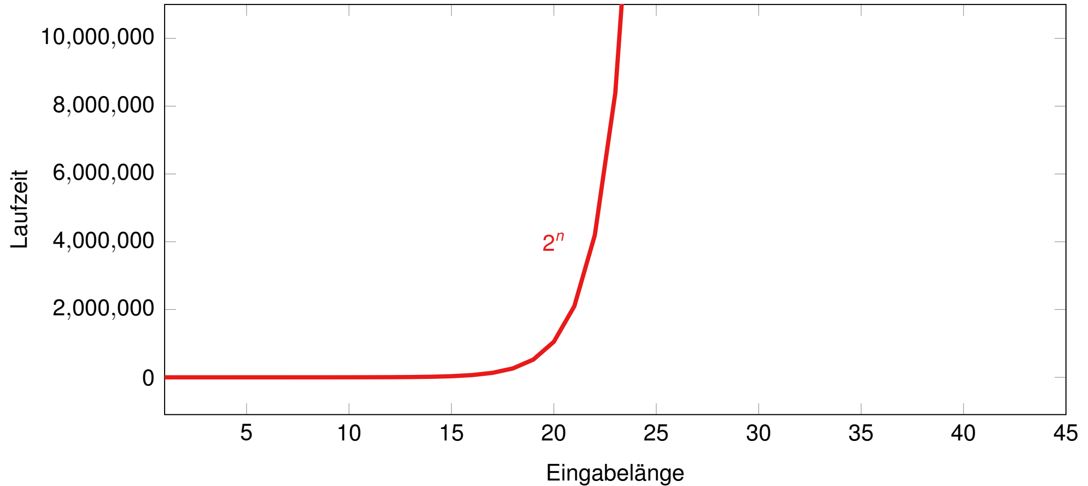
⇨ Laufzeit „wächst“ mit Grösse des Werts von $\sqrt{\texttt{eingabe}}$

⇨ Laufzeit in $O(\sqrt{2^n})$

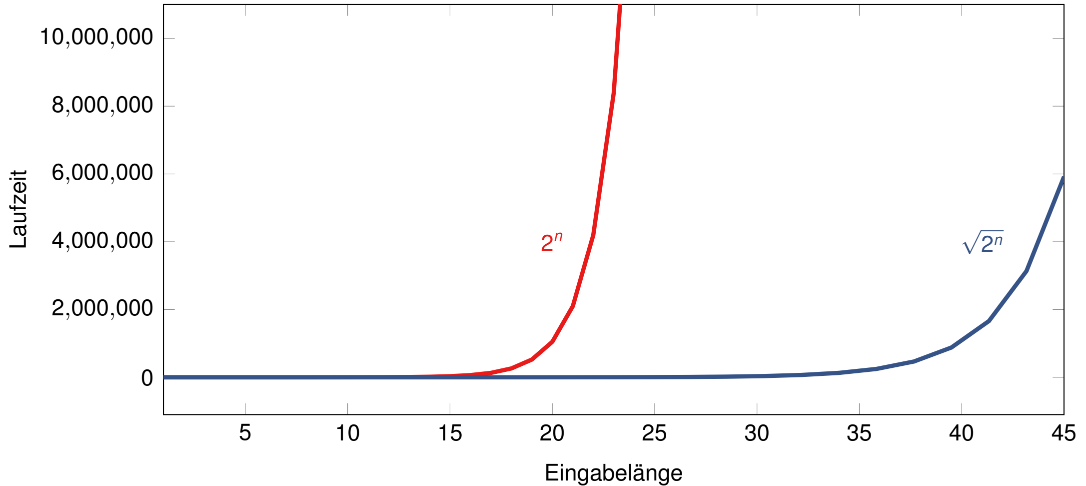
Laufzeit



Laufzeit



Laufzeit



Laufzeit

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

Laufzeit

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

Für `eingabe` = 100 000 000 000 031 bedeutet dies:

Laufzeit

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

Für `eingabe` = 100 000 000 000 031 bedeutet dies:

```
... teiler < eingabe ...
```

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

$> 100\,000\,000\,000$ Sekunden

> 3100 Jahre

Laufzeit

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

Für `eingabe` = 100 000 000 000 031 bedeutet dies:

`... teiler < eingabe ...`	`... teiler <= sqrt(eingabe) ...`

$$\frac{100\,000\,000\,000\,031 \text{ Durchl.}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} > 100\,000\,000\,000 \text{ Sekunden} > 3100 \text{ Jahre}$$

$$\frac{\sqrt{100\,000\,000\,000\,031} \text{ Durchl.}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} < \frac{10\,000\,000 \text{ Durchläufe}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} < 3 \text{ Stunden}$$

Laufzeit

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

Für `eingabe` = 100 000 000 000 031 bedeutet dies:

`... teiler < eingabe ...`	`... teiler <= sqrt(eingabe) ...`
$\frac{100\,000\,000\,000\,031 \text{ Durchl.}}{1000 \; \frac{\text{Durchläufe}}{\text{Sekunde}}}$	$\frac{\sqrt{100\,000\,000\,000\,031} \text{ Durchl.}}{1000 \; \frac{\text{Durchläufe}}{\text{Sekunde}}}$
$> 100\,000\,000\,000$ Sekunden	$< \frac{10\,000\,000 \text{ Durchläufe}}{1000 \; \frac{\text{Durchläufe}}{\text{Sekunde}}}$
> 3100 Jahre	< 3 Stunden

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

Laufzeit

Oder andersherum…

Angenommen, wir wollen maximal zehn Minuten rechnen

Laufzeit

Oder andersherum. . .

Angenommen, wir wollen maximal zehn Minuten rechnen

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

Laufzeit

Oder andersherum...

Angenommen, wir wollen maximal zehn Minuten rechnen

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

```
... teiler < eingabe ...
```

$$\frac{\texttt{eingabe}\ \text{Durchl.}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

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

Laufzeit

Oder andersherum. . .

Angenommen, wir wollen maximal zehn Minuten rechnen

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

```
... teiler < eingabe ...
```

$$\frac{\texttt{eingabe}\ \text{Durchl.}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

$$\iff \texttt{eingabe} = 600\,000$$

```
... teiler <= sqrt(eingabe) ...
```

$$\frac{\sqrt{\texttt{eingabe}}\ \text{Durchl.}}{1000\ \frac{\text{Durchläufe}}{\text{Sekunde}}} = 600\ \text{Sekunden}$$

$$\iff \texttt{eingabe} = 600\,000^2$$

$$\iff \texttt{eingabe} = 360\,000\,000\,000$$

Best- und Worst-Case-Analyse

Und noch ein Exkurs

Best- und Worst-Case-Analyse

Welcher Algorithmus ist schneller?

Best- und Worst-Case-Analyse

Welcher Algorithmus ist schneller?

```
eingabe = ...
teiler = 2

while teiler <= sqrt(eingabe):
    rest = eingabe % teiler
    if rest == 0:
        print "Keine Primzahl"
        exit()
    teiler += 1

print "Primzahl"
```

Best- und Worst-Case-Analyse

Welcher Algorithmus ist schneller?

```
eingabe = ...
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      print "Keine Primzahl"
      exit()
   teiler += 1

print "Primzahl"
```

```
eingabe = ...
ist_prim = 1
teiler = 2

while teiler <= sqrt(eingabe):
   rest = eingabe % teiler
   if rest == 0:
      ist_prim = 0
   teiler += 1

if (ist_prim == 1):
   print "Primzahl"
else:
   print "Keine Primzahl"
```

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“
- Rechter Algorithmus macht ca. $\sqrt{2^n}$ Vergleiche

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“
- Rechter Algorithmus macht ca. $\sqrt{2^n}$ Vergleiche

Angenommen, `eingabe` ist Primzahl

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“
- Rechter Algorithmus macht ca. $\sqrt{2^n}$ Vergleiche

Angenommen, `eingabe` ist Primzahl

- Dann machen beide Algorithmen ca. $\sqrt{2^n}$ Vergleiche

Best- und Worst-Case-Analyse

Angenommen, `eingabe` ist gerade Zahl

- Dann ist der linke Algorithmus sehr schnell

⇨ Schleife wird nach dem ersten Vergleich verlassen

- „Early Exit“
- Rechter Algorithmus macht ca. $\sqrt{2^n}$ Vergleiche

Angenommen, `eingabe` ist Primzahl

- Dann machen beide Algorithmen ca. $\sqrt{2^n}$ Vergleiche

(Natürlich sollte der linke implementiert werden)

Konstantes und logarithmisches Kostenmass

Der letzte Exkurs. . . versprochen

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Das ist nicht immer richtig

- Angenommen, die Zahlen sind sehr gross

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Das ist nicht immer richtig

- Angenommen, die Zahlen sind sehr gross
- Dann sollte dies in der Laufzeit berücksichtigt werden

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Das ist nicht immer richtig

- Angenommen, die Zahlen sind sehr gross
- Dann sollte dies in der Laufzeit berücksichtigt werden
- Addition von zwei n-Bit-Zahlen braucht $O(n)$ Operationen

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Das ist nicht immer richtig

- Angenommen, die Zahlen sind sehr gross
- Dann sollte dies in der Laufzeit berücksichtigt werden
- Addition von zwei n-Bit-Zahlen braucht $O(n)$ Operationen

⇨ **Logarithmisches** Kostenmass

Konstantes und logarithmisches Kostenmass

- Bislang haben wir elementare Operationen „gezählt“
- Dafür haben wir ignoriert wie gross die Zahlen sind

⇨ **Konstantes** Kostenmass

Das ist nicht immer richtig

- Angenommen, die Zahlen sind sehr gross
- Dann sollte dies in der Laufzeit berücksichtigt werden
- Addition von zwei n-Bit-Zahlen braucht $O(n)$ Operationen

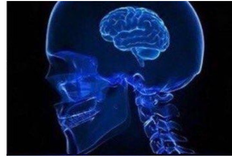
⇨ **Logarithmisches** Kostenmass

Wir werden meist das konstante Kostenmass verwenden

Was kann man sonst noch machen?

Primzahltest

Jede Zahl zwischen
1 und `eingabe` testen



Primzahltest

Jede Zahl zwischen
1 und `eingabe` testen

Jede zweite Zahl zwischen
1 und `eingabe` testen

Primzahltest

Jede Zahl zwischen
1 und `eingabe` testen

Jede zweite Zahl zwischen
1 und `eingabe` testen

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

Primzahltest

Randomisierter Monte-Carlo-Algorithmus

Primzahltest

Randomisierter Monte-Carlo-Algorithmus

Polynomieller AKS-Algorithmus

Danke für die Aufmerksamkeit