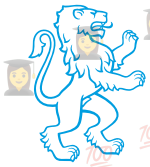

Kantonsschule Im Lee

Informatik

Skript

Cyril Wendl

Inhaltsverzeichnis

Kapitel 1

Randomisiertes Kommunikationsprotokoll

Wir wollen den Begriff des ***deterministischen Algorithmus*** einführen. Wikipedia beschreibt diesen Begriff (treffend) wie folgt:

> **Definition 1.1** (deterministischer Algorithmus)**:**
> Ein deterministischer Algorithmus ist ein Algorithmus, bei dem nur definierte und reproduzierbare Zustände auftreten. Für die gleiche Eingabe folgt auch immer die gleiche Ausgabe und zusätzlich wird die gleiche Folge an Zuständen durchlaufen. Zu jedem Zeitpunkt ist der nachfolgende Abarbeitungsschritt des Algorithmus eindeutig festgelegt. Das bedeutet auch, dass alle Zwischenergebnisse innerhalb des Algorithmus immer gleich sind.

Unsere bisherigen Algorithmen waren alle deterministisch. Sie haben auf gleichen Eingaben immer genau gleich gearbeitet und dabei stets die exakt gleiche Folge von Zuständen durchlaufen. Der nächste Schritt des Programms war dabei immer (von Anfang an) eindeutig festgelegt.

Ein randomisierter Algorithmus wollen wir uns vorstellen, wie ein deterministischer Algorithmus, der mindestens eine Zufällig gewählte Zahl erhält, welche dann das Resultat der Berechnung beeinflusst. Man kann sich auch vorstellen, dass der randomisierte Algorithmus an mindestens einer Stelle eine Münze wirft und seine Berechnung / Entscheidung (zum Beispiel bei einem `if`) abhängig vom Ausgang dieses zufälligen Münzwurfs macht.

Ziel dieses Kapitels ist zu zeigen, dass randomisierte Algorithmen deutlich effizienter als bestmögliche deterministische Algorithmen sein können.

Dazu werden wir uns im Detail mit einem Beispiel eines randomisierten Algorithmus (randomisiertes Kommunikationsprotokoll) beschäftigen.

Das Beispiel und die Notation wurden dem exzellenten Buch [**TheoretischeInformatik**] entnommen.

1.1 Beschreibung des Problems

- Sei R_1 ein Rechner, der sich an der ETH in Zürich befindet und sei R_2 ein Rechner am MIT in Boston, USA.
- Der Datensatz in Zürich ist das binäre Wort $x = x_1x_2 \dots x_n$ der Länge n. Der Datensatz in Boston ist das binäre Wort $y = y_1y_2 \dots y_n$ derselben Länge wie x.

- Wir stellen uns im Folgenden einen grossen Datensatz mit $n \approx 10^{16}$ vor.
- Wir wollen prüfen, ob die beiden Datensätze x und y identisch sind, ob also entweder $x = y$ gilt oder $x \neq y$.

Unser Ziel ist es also, einen Kommunikationsalgorithmus (ein Protokoll) zu entwerfen, der feststellt, ob sich die Inhalte der Datenbanken von R_1 und R_2 unterscheiden.

Bemerkung 1.1:
Das offensichtliche Vorgehen, um diese Frage zu beantworten, ist (zum Beispiel) den Datensatz x von Zürich Bit für Bit nach Boston zu senden, damit die Leute am MIT ihren Datensatz y Bit für Bit mit dem Datensatz von Zürich vergleichen können. Dieses Vorgehen benötigt einen Austausch von genau n Bits.
Man kann beweisen[a], dass es keinen deterministischen Algorithmus gibt, der mit einem Austausch von weniger als n Bit auskommt (bereits $n-1$ Bits reichen nicht aus). Offensichtlich ist diese Situation nicht sehr befriedigend, da es nicht praktisch ist, solch grosse Datenmengen zu senden. Des Weiteren ist die Wahrscheinlichkeit, dass bei der Übertragung solch vieler Bits mindestens ein Bit falsch übertragen werden könnte, nicht zu vernachlässigen.

[a] siehe $D(EQ) = n$ in https://en.wikipedia.org/wiki/Communication_complexity

Im Folgenden wollen wir ein randomisiertes Kommunikationsprotokoll entwerfen, welches dieses Problem mit dem Austausch von wesentlich weniger Bits lösen kann. Dazu benötigen wir allerdings ein paar mathematische Werkzeuge, welche wir hier vorstellen wollen.

1.2 Mathematische Werkzeuge

In diesem Unterabschnitt stellen wir die nötigen mathematischen Werkzeuge vor, welche benötigt werden um das randomisierte Protokoll vollständig verstehen zu können.

Es ist wichtig, diese Werkzeuge gut zu verstehen.

1.2.1 Fundamentalsatz der Arithmetik / Primfaktorzerlegung

Jede natürliche Zahl lässt sich als Produkt von endlich vielen Primzahlen darstellen. Ordnet man diese Primzahlen der Grösse nach, so ist diese Darstellung sogar eindeutig.

Insbesondere kann man jede natürliche Zahl $n \geq 2$ eindeutig darstellen als

$$n = p_1^{i_1} p_2^{i_2} \cdot \ldots \cdot p_k^{i_k}, \tag{1.1}$$

wobei $p_1 < p_2 < \ldots < p_k$ Primzahlen und $i_1, i_2, \ldots, i_k$ positive ganze Zahlen sind.

Beispiel 1.1:

$$36 = 2^2 \cdot 3^2, \quad (p_1 = 2, p_2 = 3, i_1 = 2, i_2 = 2)$$
$$99825 = 3^1 \cdot 5^2 \cdot 11^3, \quad (p_1 = 3, p_2 = 5, p_3 = 11, i_1 = 1, i_2 = 2, i_3 = 3)$$

Aufgabe 1.1

Finde die Primfaktorzerlegung der Zahl 1960 und stelle sie in obiger Form dar.

Lösungsvorschlag zu Aufgabe 1.1

$$1960 = 2^3 \cdot 5^1 \cdot 7^2$$

1.2.2 Division mit Rest

Es seien a und b positive ganze Zahlen, dann existieren eindeutige ganze Zahlen q (Quotient) und r (Rest), sodass

$$a = bq + r \tag{1.2}$$

und $0 \leq r < b$.

Dabei ist q das Resultat der ganzzahligen Division von a/b und r ist der Rest der Division.

Beispiel 1.2:
Seien $a = 17$ und $b = 5$. Dann ist die ganzzahlige Division von $q = a/b = 17/5 = 3$ mit einem Rest $r = a - bq = 2$.

Daraus folgt direkt, dass wenn

$$a \bmod b = r, \tag{1.3}$$

dann existiert eine ganze Zahl q, sodass

$$a = bq + r. \tag{1.4}$$

Dabei kann q durch die Berechnung $q = (a - r)/b$ ermittelt werden.

Beispiel 1.3:
Seien $a = 79$ und $b = 100$. Dann ist $r = a \bmod b = 79 \bmod 100 = 79$ und $q = (a - r)/b = (79 - 79)/100 = 0$.

Aufgabe 1.2

(a) Bestimme die oben beschriebene Darstellung $a = bq + r$ für $a = 121$ und $b = 9$.
(b) Schreibe eine Python-Funktion `def division_mit_rest(a, b)`, welche für gegebene positive ganze Zahlen a und b, die Zahlen q und r berechnet und in einer Liste der Form `[q, r]` zurückgibt.

✔ Lösungsvorschlag zu Aufgabe 1.2

(a)

$$121 = 9 \cdot 13 + 4,$$

wobei $a = 121$, $b = 9$, $q = 13$ und $r = 4$.

(b)

```
def division_mit_rest(a, b):
    r = a % b
    q = a // b
    return [q, r]
```

Programm 1.1: Division mit Rest

1.2.3 Nummer

Sei $x = x_1x_2 \dots x_n$, $x_i \in \{0, 1\}$ für $i = 1, 2, \dots, n$ eine endliche Folge von binären Ziffern (ein binäres Wort). Dann bezeichnet

$$\text{Nummer}(x) = \sum_{i=1}^{n} x_i \cdot 2^{n-i} \tag{1.5}$$

die (kürzeste) Darstellung von x im Dezimalsystem.

Beispiel 1.4:

$$x = 00101 \Rightarrow \text{Nummer}(x) = 2^2 + 2^0 = 4 + 1 = 5$$
$$x = 01011 \Rightarrow \text{Nummer}(x) = 2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11$$

Aufgabe 1.3

(a) Berechne Nummer(1011).
(b) Berechne Nummer(110101).

✔ Lösungsvorschlag zu Aufgabe 1.3

(a)

$$\text{Nummer}(1011) = 2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11$$

(b)

$$\text{Nummer}(110101) = 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$$

Aufgabe (Challenge) 1.4

Schreibe eine Python-Funktion `def Nummer(x)`, welche Nummer(x) für einen gegebenen binären String (binäres Wort) x berechnet.

Lösungsvorschlag zu Aufgabe 1.4

```
def Nummer(x):
    dec = 0
    power = len(x) - 1

    for ziffer in x:
        dec += int(ziffer) * 2**power
        power -= 1

    return dec
```

Programm 1.2: Nummer

1.2.4 Die Funktion Prim und der Primzahlsatz

Euklid bewies bereits in der Antike, dass es unendlich viele Primzahlen gibt. Dies ist äquivalent zur Aussage, dass es keine grösste Primzahl gibt.

Wir wollen einmal die ersten 25 natürlichen Zahlen auflisten und die Primzahlen unter ihnen unterstreichen

$$0, 1, \underline{2}, \underline{3}, 4, \underline{5}, 6, \underline{7}, 8, 9, 10, \underline{11}, 12, \underline{13}, 14, 15, 16, \underline{17}, 18, \underline{19}, 20, 21, 22, \underline{23}, 24, \ldots$$

Dank der Arbeit von Euklid wissen wir, dass in dieser Auflistung von natürlichen Zahlen immer wieder eine Primzahl auftauchen wird, egal wie gross die Zahlen in der Liste werden.

Definition 1.2 (Prim)**:**
Im Folgenden wollen wir mit Prim(n) die Anzahl der Primzahlen kleiner oder gleich n bezeichnen.

Beispiel 1.5:
Es existieren genau die vier Primzahlen $2, 3, 5$ und 7, die kleiner oder gleich 9 sind. Damit ist $\text{Prim}(9) = 4$ ebenso ist $\text{Prim}(10) = 4$.
Es gilt auch

- $\text{Prim}(10) = \text{Prim}(7) = \text{Prim}(9) = \text{Prim}(8) = 4$
- $\text{Prim}(2) = 1$
- $\text{Prim}(16) = 6$.

Aufgabe 1.5

Bestimme die Zahl Prim(33).

Lösungsvorschlag zu Aufgabe 1.5

Die Primzahlen ≤ 33 sind: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31$. Dies sind 11 Zahlen. Damit ist $\text{Prim}(33) = 11$.

Eine natürliche Zahl $n \geq 2$ heisst Primzahl, wenn sie keinen echten Teiler hat. Deshalb könnte man intuitiv annehmen, dass grosse Primzahlen äusserst selten sind, da sehr viele Zahlen „potenzielle“ echte Teiler der Zahl zu sein scheinen.

Die Frage nach der Häufigkeit der Primzahlen ist eine alte Frage der Mathematik. Es hat sich herausgestellt, dass Primzahlen in einem gewissen Sinne überhaupt nicht selten sind, sondern sogar häufig auftauchen. Ein zentraler Begriff im Zusammenhang mit der Verteilung von Primzahlen ist die ***Primzahllücke*** (prime gap)[1].

Ein starkes Resultat im Zusammenhang mit der Frage nach der Häufigkeit von Primzahlen ist der ***Primzahlsatz***. Diesen werden wir hier präsentieren, aber nicht beweisen. Der Beweis dieses Satzes ist sehr schwierig und wir werden ihn deshalb hier nicht führen.

Theorem 1.1:
Primzahlsatzytheorem:primzahlsatz Der Primzahlsatz besagt

$$\lim_{n\to\infty} \frac{\text{Prim}(n)}{n/\ln(n)} = 1. \tag{1.6}$$

Der Primzahlsatz sagt, dass die Anzahl der Primzahlen ungefähr so schnell wächst wie die Funktion $n/\ln(n)$.

Der Satz sagt aber nichts über den Grenzwert der Differenz der beiden Funktion $n/\ln(n)$ und $\text{Prim}(n)$ aus! Diese Thematik wird in Aufgabe 1.6 aufgegriffen.

Aufgabe 1.6

Gegeben sind zwei Funktionen f und g, welche auf ganz $\mathbb{N}$ definiert sind und $g(n) \neq 0$ für alle $n \in \mathbb{N}$. Es gelte die Gleichung

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Ist es korrekt, dass Gleichung (1.6) die Gleichung

$$\lim_{n\to\infty} |f(n) - g(n)| = 0$$

impliziert? Begründen Sie die Behauptung oder finden Sie ein Gegenbeispiel.

[1] https://en.wikipedia.org/wiki/Prime_gap

✔ Lösungsvorschlag zu Aufgabe 1.6

Wir definieren $f(n) := n$ und $g(n) := n + 2025$. Dan gilt zwar

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{n+2025} = 1$$

aber $\lim_{n\to\infty} |f(n) - g(n)| = 2025 \neq 0$.

Daraus folgt offensichtlich sofort, dass

$$\lim_{n\to\infty} \frac{\text{Prim}(n^2)}{n^2/\ln(n^2)} = 1. \tag{1.7}$$

In der Form 1.7 werden wir den Primzahlsatz später verwenden.

Exkurs 1.1:
Der Primzahlsatz in **??** macht eine Aussage über einen Grenzwert, sagt jedoch nichts für endliches n aus. Wir wollen aber die Zahl $n/\ln(n)$ als Approximation für $\text{Prim}(n)$ für endliches n verwenden. Es gibt verschiedene mathematische Resultate, welche dieses Vorgehen rechtfertigen. So bewies im Februar 2010[**EstimatePrimes**] Pierre Dusart das folgende präzisere Resultat zu $\text{Prim}(n)$:

$$\text{Prim}(n) \geq \frac{n}{\ln(n) - 1}, \quad \text{für } n \geq 5393, \tag{1.8}$$

$$\text{Prim}(n) \leq \frac{n}{\ln(n) - 1.1}, \quad \text{für } n \geq 60184. \tag{1.9}$$

1.2.5 Berechnen von Prim in Python

Der Algorithmus ***Sieb des Eratosthenes*** (englisch: Eratosthene's Sieve) ist bereits seit der Antike bekannt. Der Algorithmus bestimmt für eine gegebene natürliche Zahl n alle Primzahlen, die kleiner oder gleich n. Man schreibt alle natürlichen Zahlen grösser gleich 2 bis und mit n auf:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, \ldots, n.$$

Danach betrachtet man die erste Zahl in der Auflistung, also die Zahl 2. Diese Zahl wurde noch nicht durchgestrichen, also ist sie eine Primzahl und das Sieb des Eratosthenes gibt diese Zahl aus. Danach wird jedes ganzzahlige Vielfache von 2, dass kleiner oder gleich n ist in der Liste gestrichen:

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, 21, \cancel{22}, \ldots$$

Danach betrachtet der Algorithmus die nächste Zahl, also die Zahl 3. Da diese noch nicht gestrichen wurde, ist 3 eine Primzahl und wird ausgegeben. Nun werden alle ganzzahligen Vielfache von 3 ebenfalls gestrichen (falls sie nicht sowieso bereits gestrichen wurden):

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, \cancel{21}, \cancel{22}, \ldots$$

Nun betrachtet der Algorithmus wieder die nächste Zahl in der Liste, also die Zahl 4. Diese wurde aber bereits gestrichen und deshalb schaut sich der Algorithmus die nächste Zahl an (die Zahl 5). Dieses Vorgehen wird wiederholt, bis der Algorithmus bei der Zahl n ankommt.

Aufgabe 1.7

(a) Erkläre, warum dieser Algorithmus genau alle Primzahlen kleiner oder gleich n ausgibt.
(b) Wie kann das Sieb des Eratosthenes verwendet werden, um $\text{Prim}(n)$ zu bestimmen?

Lösungsvorschlag zu Aufgabe 1.7

(a) Verlangt war nur eine informelle Erklärung. Wir zeigen hier etwas genauer:

$$\text{eine natürliche Zahl } m \text{ ist eine Primzahl}$$
$$\Longleftrightarrow$$
$$m \text{ wird vom Sieb des Eratosthenes für jede Eingabe } n \geq m \text{ ausgegeben.}$$

Wir zeigen die beiden Richtungen.

$\Rightarrow$:
Da m eine Primzahl ist, besitzt sie keine echten Teiler. Damit ist sie nicht das Vielfache einer Zahl aus der Menge $\{2, 3, 4, \ldots, m-1\}$. Deshalb hat das Sieb die Zahl m bis zum Erreichen von m nicht durchgestrichen und somit wird m vom Sieb ausgegeben.

$\Leftarrow$:
Sei $n \geq m$ die Eingabe für den Algorithmus *Sieb des Eratosthenes*. Nach Konstruktion des Algorithmus, wird m genau dann ausgegeben, wenn m kein Vielfaches einer Zahl aus der Menge $\{2, 3, 4, \ldots, m-1\}$ ist. Damit besitzt m aber keinen echten Teiler und somit ist m eine Primzahl.

(b) Für die Eingabe n liefert das Sieb alle Primzahlen $\leq n$. Um $\text{Prim}(n)$ zu bestimmen, müssen diese Primzahlen nur noch gezählt werden.

Aufgabe 1.8

Schreiben Sie eine Python-Funktion `def sieb(n)`, welche den Algorithmus von Eratosthenes implementiert und alle Primzahlen $\leq n$ ausgibt. Vervollständigen Sie dazu die folgende Vorlage:

```
def sieb(n):
    # starte mit einer leeren Liste von
    # Primzahlen (wir haben noch keine gefunden)
    primzahlen = []

    # starte mit einer Liste der Länge n + 1
    # nicht_gestrichen[i] ist genau dann True, wenn
    # die Zahl i (i = 0, 1, ..., n) noch
    # nicht gestrichen wurde
    # zu Beginn ist noch keine Zahl durchgestrichen
    nicht_gestrichen = (n + 1) * [True]

    i = 2
    # prüfe, ob die Zahl i (i = 2, ..., n) schon gestrichen ist
    while ...:
        if ...
            ...
            ...
```

```
            while j <= n:
                nicht_gestrichen[j] = False
                ...
                ...
        i += 1
    print(primzahlen)
```

Programm 1.3: Sieb des Eratosthenes

✔ Lösungsvorschlag zu Aufgabe 1.8

```
def sieb(n):
    # starte mit einer leeren Liste von
    # Primzahlen (wir haben noch keine gefunden)
    primzahlen = []

    # starte mit einer Liste der Länge n + 1
    # nicht_gestrichen[i] ist genau dann True, wenn
    # die Zahl i (i = 0, 1, ..., n) noch
    # nicht gestrichen wurde
    # zu Beginn ist noch keine Zahl durchgestrichen
    nicht_gestrichen = (n + 1) * [True]

    i = 2
    # prüfe, ob die Zahl i (i = 2, ..., n) schon gestrichen ist
    while i <= n:
        if nicht_gestrichen[i]:
            primzahlen.append(i)
            j = 2 * i
            while j <= n:
                nicht_gestrichen[j] = False
                j += i
        i += 1
    print(primzahlen)
```

Programm 1.4: Sieb des Eratosthenes

Mithilfe des Siebs von Eratosthenes, ist es nun ganz einfach, eine Funktion zu definieren, welche Prim(n) berechnet. Dazu ändern wir den Befehl `print(primzahlen)` in unserer Funktion `sieb` ab zu `return primzahlen` und können dann schreiben:

```
def prim(n):
    return len(sieve(n))
```

Programm 1.5: Prim

Die Laufzeit des *Siebs des Eratosthenes* kann an zwei Stellen deutlich verbessert werden. Die folgende Aufgabe hilft Dir dabei, dieses Stellen zu erkennen.

Aufgabe 1.9 Verbesserung 1

Sei $n \geq 2$ eine natürliche Zahl. Zeige, dass folgende zwei Aussagen äquivalent sind:

1. Die Zahl n ist eine Primzahl.
2. Die Zahl n besitzt keinen echten Teiler $\leq \lfloor \sqrt{n} \rfloor$.

Dabei bezeichnet für $x \in \mathbb{R}$ der Ausdruck $\lfloor x \rfloor$ (sprich: x abgerundet) die grösste ganze Zahl $\leq x$ (Abrundeklammer).

Lösungsvorschlag zu Aufgabe 1.9

1. $\Rightarrow$ 2.:

Da n eine Primzahl ist, besitzt sie keinen echten Teiler und somit insbesondere auch keinen echten Teiler $\leq \lfloor \sqrt{n} \rfloor$.

2. $\Rightarrow$ 1.:

Sei n keine Primzahl. Damit besitzt n eine Faktorisierung $n = ab$ mit $a, b \in \mathbb{N}$ und $a, b \geq 2$. Wir zeigen, dass mindestens einer der Faktoren $\leq \lfloor \sqrt{n} \rfloor$ ist und n somit einen echten Teiler $\leq \lfloor \sqrt{n} \rfloor$ hat. Angenommen $a, b > \lfloor \sqrt{n} \rfloor$ (beide Faktoren sind grösser). Dann gilt offensichtlich $a, b \geq \lfloor \sqrt{n} \rfloor + 1$, da a und b natürliche Zahlen sind. Beachten Sie, dass $\lfloor \sqrt{n} \rfloor > \sqrt{n} - 1$ gilt. Nun folgt:

$$ab \geq \left(\lfloor \sqrt{n} \rfloor + 1\right)^2 > \left(\sqrt{n} - 1 + 1\right)^2 = n$$

und somit der Widerspruch $ab > n$, obwohl $ab = n$ gilt.

Aufgabe 1.10 Implementation von Verbesserung 1

Implementieren Sie Verbesserung 1 aus Aufgabe 1.9 in Python.

✔ Lösungsvorschlag zu Aufgabe 1.10

```
def sieb_verbesserung1(n):
    primzahlen = []
    nicht_gestrichen = (n + 1) * [True]

    i = 2
    while i * i <= n:
        if nicht_gestrichen[i]:
            j = 2 * i
            while j <= n:
                nicht_gestrichen[j] = False
                j += i
        i += 1

    # sammle alle nicht gestrichenen Zahlen
    # dies sind genau die Primzahlen <= n
    i = 2
    while i <= n:
        if nicht_gestrichen[i]:
            primzahlen.append(i)
        i += 1

    print(primzahlen)
```

Programm 1.6: Verbesserung 1

🏆 Aufgabe (Challenge) 1.11 Verbesserung 2

Wir wenden das Sieb des Eratosthenes auf die Eingabe $n > 2$ an.

Angenommen, das Sieb des Eratosthenes hat soeben eine Primzahl $p > 2$ gefunden. Zeige, dass die Vielfachen hp von p mit $h \in \{2, 3, \ldots, p-1\}$ bereits vom Algorithmus gestrichen wurden. Damit genügt es, die Vielfachen $\geq p^2$ zu streichen (solange sie kleiner oder gleich n sind).

✔ Lösungsvorschlag zu Aufgabe 1.11

Sei hp für ein $h \in \{2, 3, \ldots, p-1\}$ ein Vielfaches von p. Angenommen h ist eine Primzahl. Da $h < p$, wurde h bereits als Primzahl vom Sieb (vor p) gefunden und die Vielfachen $\leq n$ von h wurden gestrichen. Insbesondere also auch das Vielfache hp von h. Falls h keine Primzahl ist, dann besitzt h eine Zerlegung $h = ab$ für eine Primzahl a mit $a < h < n$. Damit wurde aber $hp = abp$ als bp-faches Vielfache von a bereits gestrichen.

Aufgabe (Challenge) 1.12 Implementation von Verbesserung 2

Ergänzen Sie Ihr Programm aus Aufgabe 1.10 um Verbesserung 2.

Lösungsvorschlag zu Aufgabe 1.12

```
def sieb_verbesserungen_1_und_2(n):
    primzahlen = []
    nicht_gestrichen = (n + 1) * [True]

    i = 2
    while i * i <= n:
        if nicht_gestrichen[i]:
            j = i * i
            while j <= n:
                nicht_gestrichen[j] = False
                j += i
        i += 1

    # sammle alle nicht gestrichenen Zahlen
    # dies sind genau die Primzahlen <= n
    i = 2
    while i <= n:
        if nicht_gestrichen[i]:
            primzahlen.append(i)
        i += 1

    print(primzahlen)
```

Programm 1.7: Verbesserung 2

Aufgabe 1.13

(a) Bestimme $\mathrm{Prim}(n)$ für $n = 10'000'000$ (zehn Millionen) mithilfe der Funktion `prim` (aus Aufgabe 1.10 oder Challenge 1.12) und vergleiche den Wert mit der Schätzung $n/\ln n$. Wie gross ist der relative Fehler der Schätzung?

(b) Lade das Python-Programm `prim_plots.py` von Moodle herunter und führe es aus. Betrachte den generierten Plot. Was sagt der Plot aus? Wie sind die Achsen skaliert?

(c) Schreibe eine Python-Funktion `def prim_approx(n)`, welche $\mathrm{Prim}(n)$ mithilfe der Abschätzung, welche von dem Primzahlsatz 1.6 geliefert wird, approximiert.

(d) Verwende die sehr schnelle Funktion `prim_approx` anstelle von `prim`, um den Wertebereich des Plots von $8 \cdot 10^7$ auf 10^{10} zu erhöhen.

✔ Lösungsvorschlag zu Aufgabe 1.13

(a) Das Python-Programm liefert $\text{Prim}(10000000) = 664'579$. Die Schätzung liefert $10^7/\ln(10^7) \approx 62'0421$. Der relative Fehler ist etwa 6.645 Prozent.

(b) Beide Achsen sind logarithmiert. Der Plot zeigt, wie stark $f(n) = \text{Prim}(n)$ verglichen mit der linearen Funktion $g(n) = n$ wächst.

(c) Der folgende Code berechnet die Approximation:

```
import numpy as np

def prim_approx(n):
    return n / np.log(n)
```

(d) Folgende Anpassungen müssen in `prim_plots.py` durchgeführt werden:

```
# Plots
N = [ 10**1, 10**2, 10**3, 10**4, 10**5, 10**6, 5*10**6, 10**7,
     4*10**7, 8*10**7 , 10**10]

results = []
for n in N:
    results.append(prim_approx(n))
```

Aufgabe 1.14

Erstelle einen zweiten Plot, welcher das Verhältnis $\text{Prim}(n)/(n/\ln(n))$ plottet. Was stellst Du fest? Was würde man erwarten?

✔ Lösungsvorschlag zu Aufgabe 1.14

Der entsprechende Code ist:

```
# quotient Prim(n)/n
ratio = []
for k in range(len(N)):
    ratio.append(results[k] / N[k])
fig, ax = plt.subplots()
ax.loglog(N, ratio, 'go-', label ='f(n) / g(n) = Prim(n) / n')
ax.set(xlabel='n', ylabel='count', title='Prim(n) / n')
ax.grid()
legend = ax.legend(loc='upper right', shadow=True, fontsize='x-large
    ')
plt.show()
```

1.2.6 Binäre Darstellung einer Zahl

Dies ist das letzte mathematische Werkzeug, das wir benötigen.

Sei $n \geq 1$ eine natürliche Zahl. Wir bezeichnen mit $\text{Bin}(n)$ die kürzeste binäre Darstellung von n, also $\text{Nummer}(\text{Bin}(n)) = n$. Die binäre Darstellung ist offenbar genau dann am kürzesten, wenn die erste Ziffer der Darstellung eine 1 und keine 0 ist. Beispielsweise sind sowohl 100 als auch 00100 binäre Darstellungen der Zahl 4, doch nur die erste ist die kürzeste.

Aufgabe 1.15

Bestimme $\text{Bin}(11)$ und $\text{Bin}(52)$.

Lösungsvorschlag zu Aufgabe 1.15

$\text{Bin}(11) = 1011$, $\text{Bin}(52) = 110100$

Aufgabe (Challenge) 1.16

Schreibe eine Python-Funktion `def Bin(n)`, welche $\text{Bin}(n)$ für eine natürliche Zahl n berechnet.

Lösungsvorschlag zu Aufgabe 1.16

```
def Bin(n):
    if n == 0:
        return "0"
    binary_str = ""
    while n > 0:
        binary_str = str(n % 2) + binary_str
        n = n // 2
        return binary_str
```

Um die Länge der binären Darstellung einer Zahl zu bestimmen, haben wir immer zuerst die binäre Darstellung der Zahl berechnet und anschliessend die Anzahl Stellen der Darstellung durch Zählen bestimmt. Im Folgenden wollen wir eine sehr wichtige Formel finden, welche uns die Darstellungslänge einer Zahl in einer beliebigen Basis b sofort gibt.

Theorem 1.2 (Anzahl Ziffern in Zahlendarstellung)**:**
ytheorem:AnzahlZiffern Es seien $n > 0$ und $b > 1$ natürliche Zahlen. Die kürzeste b-adische Darstellung von n (Darstellung ohne führende Nullen) hat genau $\lceil \log_b(n+1) \rceil$ Stellen.

Beweis 1.1:
Es sei s die Anzahl der Stellen der kürzesten b-adischen Darstellung von n. Die grösste s-stellige Zahl in Basis b (kürzeste Darstellung) ist

$$\sum_{k=0}^{s-1}(b-1)b^k = (b-1)\sum_{k=0}^{s-1}b^k \stackrel{??}{=} (b-1)\frac{b^s-1}{b-1} = b^s - 1,$$

wobei wir **??** verwendet haben. Die kleinste s-stellige Zahl Basis b (kürzeste Darstellung) ist b^{s-1}. Somit gilt

$$\begin{aligned} b^{s-1} - 1 < n \leq b^s - 1 &\iff \\ b^{s-1} < n + 1 \leq b^s &\iff \\ s - 1 < \log_b(n+1) \leq s, & \end{aligned}$$

wobei wir verwendet haben, dass $\log_b$ eine streng monoton wachsende Funktion ist. Dann folgt aber $\lceil \log_b(n+1) \rceil = s$.

1.3 Ein randomisiertes Kommunikationsprotokoll

1.3.1 Formulierung des Kommunikationsprotokolls

Nun sind wir in der Lage das randomisierte Kommunikationsprotokoll zu beschreiben. Zur Erinnerung: R_1 ist ein Rechner, der sich an der ETH in Zürich befindet (mit Datensatz x) und R_2 ist ein Rechner am MIT in Boston, USA (mit Datensatz y). Wir wollen prüfen, ob $x = y$ gilt (ob die Datensätze identisch sind).

Das Protokoll $R = (R_1, R_2)$ für die beiden Rechner arbeitet wie folgt.

Vorgehen 1.1 (Kommunikationsprotokoll)**:**
yprocedure:protokoll **Ausgangssituation**:
R_1 hat n Bits $x = x_1 \dots x_n$, R_2 hat n Bits $y = y_1 \dots y_n$.
Phase 1 (Zürich):
R_1 wählt zufällig mit einer uniformen Wahrscheinlichkeitsverteilung (alle gleiche Wahrscheinlichkeit gewählt zu werden) eine Primzahl $p \leq n^2$ aus.
Phase 2 (Zürich):
R_1 berechnet die Zahl $s = \text{Nummer}(x) \bmod p$ und schickt die binären Darstellungen von s und p an R_2
Phase 3 (Boston):
Nach Empfang von s und p berechnet R_2 die Zahl $q = \text{Nummer}(y) \bmod p$.

- Falls $q \neq s$, dann gibt R_2 die Ausgabe „ungleich“ aus.
- Falls $q = s$, dann gibt R_2 die Ausgabe „gleich“ aus.

Phase 4 (optional):
Sende ein einziges Bit von Boston nach Zürich, wobei 0 := „ungleich“ und 1 := „gleich“. Damit ist das Resultat des Vergleichs auch in Zürich bekannt.

Aufgabe 1.17

Erkläre, warum das Protokoll garantiert richtig entscheidet, falls die Ausgabe „ungleich“ ist.

Lösungsvorschlag zu Aufgabe 1.17

Da $s = \text{Nummer}(x) \bmod p$ und $q = \text{Nummer}(y) \bmod p$ die Reste der Modulo-Operation mit p sind, ist es nicht möglich, dass s und q unterschiedlich sind, wenn x und y gleich sind.

1.3.2 Analyse des Kommunikationsprotokolls

Jetzt analysieren wir die Arbeit von $R = (R_1, R_2)$. Zuerst bestimmen wir die Komplexität gemessen als die Anzahl der Kommunikationsbits. Anschliessend analysieren wir die Zuverlässigkeit (Fehlerwahrscheinlichkeit) von $R = (R_1, R_2)$.

1.3.2.1 Kommunikationsaufwand

Die einzige Kommunikation (ohne die optionale Phase 4, sonst kommt noch 1 Bit hinzu) besteht darin, dass R_1 die binären Darstellungen der Zahlen s und p an R_2 schickt.

Die Primzahl p wurde definitionsgemäss als $p \leq n^2$ gewählt und da n^2 keine Primzahl ist, gilt

$p < n^2$. Die Zahl s erfüllt als Rest bei der Division durch p die Ungleichung $s < p$. Damit gilt also $s < p < n^2$. Der gesamte Kommunikationsaufwand, wir nennen ihn A, ist die Summe der Längen der kürzesten binären Darstellungen von p und s:

$$\begin{aligned} A := \lceil \log_2 (p+1) \rceil + \lceil \log_2 (s+1) \rceil \leq \\ \left\lceil \log_2 \left(n^2 + 1 \right) \right\rceil + \left\lceil \log_2 \left(n^2 + 1 \right) \right\rceil = \\ 2 \left\lceil \log_2 \left(n^2 + 1 \right) \right\rceil \overset{??}{\leq} \\ 2 \left\lceil \log_2 \left(n^2 \right) \right\rceil + 2 = 4 \lceil \log_2 (n) \rceil + 2, \end{aligned}$$

wobei wir **??** verwendet haben. Insgesamt haben wir also

$$A \leq 4 \lceil \log_2 (n) \rceil + 2 \tag{1.10}$$

gefunden.

Theorem 1.3:
ytheorem:Estimate Für jede natürliche Zahl $n > 0$ gilt

$$\lceil \log_2(n+1) \rceil \leq \lceil \log 2(n) \rceil + 1.$$

Beweis 1.2:
Wir beweisen zunächst die Ungleichung $\log_2(n+1) \leq \log_2(n) + 1$. Dazu berechnen wir zunächst

$$\log 2(n) + 1 = \log 2(n) + \log 2(2) = \log 2(2n).$$

Damit gilt also

$$\begin{aligned} \log_2(n+1) \leq \log_2(n) + 1 \iff \\ \log_2(n+1) \leq \log_2(2n) \iff \\ n + 1 \leq 2n \iff \\ 1 \leq n, \end{aligned}$$

wobei wir verwendet haben, dass $\log 2$ (streng) monoton wachsend ist. Die Ungleichung ist offensichtlich für alle natürlichen Zahlen $n > 0$ korrekt. Wir berechnen nun

$$\begin{aligned} \log_2(n+1) \leq \log_2(n) + 1 \Rightarrow \\ \lceil \log_2(n+1) \rceil \leq \lceil \log_2(n) + 1 \rceil = \lceil \log_2(n) \rceil + 1. \end{aligned}$$

Aufgabe 1.18

Bestimmen Sie eine möglichst gute obere Schranke für den Kommunikationsaufwand A, falls $n := 10^{16}$.

Lösungsvorschlag zu Aufgabe 1.18

Mit Gleichung (1.10) finden wir

$$\begin{aligned} A \leq \\ 4\lceil \log_2(n) \rceil + 2 = \\ 4\left\lceil \log_2\left(10^{16}\right) \right\rceil + 2 = \\ 4 \cdot 16 \cdot \lceil \log_2(10) \rceil + 2 = 4 \cdot 16 \cdot 4 + 2 = 258. \end{aligned}$$

1.3.2.2 Fehlerwahrscheinlichkeit

Bei der Analyse der Fehlerwahrscheinlichkeit unterscheiden wir zwei Möglichkeiten bezüglich der tatsächlichen Relation zwischen den Datensätzen x und y.

(a) Sei $x = y$. Dann gilt

$$s = \text{Nummer}(x) \bmod p = \text{Nummer}(y) \bmod p = q$$

für alle Primzahlen p. Also gibt R_2 die korrekte Antwort „gleich“. In diesem Fall ist die Fehlerwahrscheinlichkeit 0.

(b) Sei $x \neq y$. Wir bekommen die falsche Antwort „gleich“ genau dann, wenn Nummer(x) und Nummer(y) denselben Rest z beim Teilen durch p haben:

$$z := \text{Nummer}(x) \bmod p = \text{Nummer}(y) \bmod p.$$

Mithilfe von Gleichung (1.2) (Division mit Rest), können wir auf eindeutige Weise schreiben:

$$\begin{aligned} \text{Nummer}(x) &= x' \cdot p + z, \\ \text{Nummer}(y) &= y' \cdot p + z, \end{aligned}$$

für eindeutige natürlichen Zahlen x' und y'. Kombinieren dieser beiden Gleichungen gibt uns

$$\text{Nummer}(x) - \text{Nummer}(y) = x' \cdot p - y' \cdot p = (x' - y') \cdot p.$$

Das Protokoll entscheidet also genau dann falsch, wenn die Primzahl p die Zahl

$$|\text{Nummer}(x) - \text{Nummer}(y)|$$

teilt. Wir wollen uns nun überlegen, mit welcher Wahrscheinlichkeit eine solche „schlechte“ Wahl von p ist.

Fehlerwahrscheinlichkeit bei $x \neq y$

Wir wissen, dass p mit uniformer Wahrscheinlichkeitsverteilung als eine von Prim(n^2) vielen Primzahlen gewählt wurde. Es ist also hilfreich festzustellen, wie viele der Prim(n^2) vielen Primzahlen die Zahl $|\text{Nummer}(x) - \text{Nummer}(y)|$ teilen. Da die Länge von x und y gleich n sind, gilt

$$w := |\text{Nummer}(x) - \text{Nummer}(y)| < 2^n. \tag{1.11}$$

Gemäss dem Fundamentalsatz der Arithmetik (Primfaktorzerlegung), können wir w auf eindeutige Weise[2] als $w = p_1^{i_1} p_2^{i_2} \dots p_k^{i_k}$ darstellen, wobei $p_1 < p_2 < \dots < p_k$ Primzahlen und $i_1, i_2, \dots, i_k$ positive ganze Zahlen sind. Unser Ziel ist zu beweisen, dass $k \leq n-1$ ist. Dass w also höchstens $n-1$ verschiedene Primfaktoren besitzt.

Beweis 1.3:
Wir beweisen diese Aussage indirekt (durch Widerspruch).
Angenommen die Aussage wäre falsch, also $k \geq n$. Dann ist für genügend grosse n

$$w = p_1^{i_1} p_2^{i_2} \dots p_k^{i_k} \geq p_1 p_2 \dots p_n > 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = n! > 2^n.$$

Dies ist aber ein Widerspruch zur Tatsache, dass $w < 2^n$. Damit ist unsere Annahme falsch und w kann nicht mehr als $n-1$ unterschiedliche Primfaktoren haben.

Da jede Primzahl aus $\{2, 3, \dots, n^2\}$ die gleiche Wahrscheinlichkeit hat, (zufällig) gewählt zu werden, ist die Wahrscheinlichkeit, ein p zu wählen, das w teilt, (für genügend grosses n) höchstens

$$\frac{n-1}{\text{Prim}(n^2)} \approx \frac{n-1}{n^2/\ln(n^2)} < \frac{\ln(n^2)}{n},$$

wobei wir die Approximation $\text{Prim}(n) \approx n/\ln(n)$ verwendet haben[3]. Die Fehlerwahrscheinlichkeit des Kommunikationsprotokolls ist also höchstens $\ln(n^2)/n$, was für $n = 10^{16}$ höchstens $0.736827 \cdot 10^{-14}$ ist.

Aufgabe 1.19

Erklären Sie, warum die Zahl w in Gleichung (1.11) grösser als 1 ist.

Lösungsvorschlag zu Aufgabe 1.19

Die Zahl w ist definiert unter der Annahme $x \neq y$ und der Annahme, dass Nummer(x) und Nummer(y) denselben Rest z beim Teilen durch eine Primzahl haben. Damit muss die Differenz w aber ein Vielfaches einer Primzahl sein. Da 2 die kleinste Primzahl ist, folgt $w \geq 2$.

1.3.3 Praktische Durchführung

Sollte uns eine Fehlerwahrscheinlichkeit von $0.736827 \cdot 10^{-14}$ noch zu hoch sein, kann das Protokoll zum Beispiel 10-mal wiederholt werden (mit 10 zufälligen Primzahlen) und man erhält dann eine Fehlerwahrscheinlichkeit von höchstens

$$\left(\frac{n-1}{\text{Prim}(n^2)}\right)^{10} \approx \left(\frac{\ln(n^2)}{n}\right)^{10} = \frac{2^{10} \cdot (\ln(n))^{10}}{n^{10}}$$

für genügend grosse n. Für $n = 10^{16}$ ist dies höchstens $0.472 \cdot 10^{-141}$. Eine Fehlerwahrscheinlichkeit, die man gerne in Kauf nimmt.

[2] In Aufgabe 1.19 erklären Sie, warum $w \geq 2$ gilt.
[3] siehe **??**, Exkurs 1.1

Aufgabe 1.20

Verwende Deine Python-Funktionen `sieb`, `Bin` und `Nummer` um das Kommunikationsprotokoll in Python zu implementieren.

Lösungsvorschlag zu Aufgabe 1.20

siehe Abschnitt 2.2

Kapitel 2

Details

2.1 Werkzeuge

Theorem 2.1 (geometrische Summe)**:**
ytheorem:geometrischeSumme Es sei $q \neq 0$ eine reelle Zahl und n eine natürliche Zahl. Dann gilt

$$\sum_{k=0}^{n} q^k = q^0 + q^1 + \ldots + q^n = \frac{q^{n+1} - 1}{q - 1}.$$

Beweis 2.1:
Wir beweisen die Aussage durch vollständige Induktion.

- Für $n = 0$ gilt die Aussage, da

$$\sum_{k=0}^{0} q^k = q^0 = 1 = \frac{q^{0+1} - 1}{q - 1}.$$

- Die Aussage gelte nun für eine natürliche Zahl n. Wir zeigen, dass sie auch für $n+1$ gilt.

$$\sum_{k=0}^{n+1} q^k = q^{n+1} + \sum_{k=0}^{n} q^k = q^{n+1} + \frac{q^{n+1} - 1}{q - 1} = \frac{q^{n+2} - 1}{q - 1}.$$

Theorem 2.2 (verallgemeinerte geometrische Summe)**:**
ytheorem:verallgemeinertegeometrischeSumme Es sei $q \neq 0$ eine reelle Zahl und n und $m \leq n$ natürliche Zahlen. Dann gilt

$$\sum_{k=m}^{n} q^k = q^m + q^{m+1} + \ldots + q^n = \frac{q^{n+1} - q^m}{q - 1}.$$

Beweis 2.2:
Unter Verwendung von **??** finden wir

$$\sum_{k=m}^{n} q^k = \sum_{k=0}^{n} q^k - \sum_{k=0}^{m-1} q^k = \frac{q^{n+1} - 1}{q - 1} - \frac{q^m - 1}{q - 1} = \frac{q^{n+1} - q^m}{q - 1}.$$

2.2 Protokoll

Die Funktionen `sieb`, `Nummer` und `Bin` haben wir schon geschrieben. Diese können wir nun hier verwenden. Die Funktion `is_equal` wird nicht benötigt, kann aber verwendet werden um (deterministisch) zu prüfen, ob zwei Datensätze identisch sind oder nicht.

```
import math

import numpy as np


def sieb_verbesserung_1_und_2(n):
    primzahlen = []
    nicht_gestrichen = (n + 1) * [True]

    i = 2
    # abgerundete Quadratwurzel von n
    wurzel_n = int(math.sqrt(n))
    while i <= wurzel_n:
        if nicht_gestrichen[i]:
            primzahlen.append(i)
            j = i * i
            while j <= n:
                nicht_gestrichen[j] = False
                j += i
        i += 1

    # sammle alle nicht gestrichenen Zahlen
    i = 2
    while i <= n:
        if nicht_gestrichen[i]:
            primzahlen.append(i)
        i += 1
    return primzahlen


def Nummer(x):
    dec = 0
    power = len(x) - 1

    for ziffer in x:
        dec += int(ziffer) * 2**power
        power -= 1

    return dec


def Bin(n):
    if n == 0:
        return "0"
    binary_str = ""
```

```
    while n > 0:
        binary_str = str(n % 2) + binary_str
        n = n // 2
    return binary_str


def is_equal(x, y):
    nx = len(x)
    ny = len(y)

    if nx != ny:
        return False
    for i in range(nx):
        if x[i] != y[i]:
            return False
    return True


def random_primzahl(n):
    primes = sieb_verbesserung_1_und_2(n)
    p = np.random.choice(primes, 1)[0]
    return p


def R1(x):
    n = len(str(x))
    p = random_primzahl(n**2)
    s = Nummer(x) % p
    return [s, p]


def protokoll(x, y):
    sp = R1(x)
    q = Nummer(y) % sp[1]
    if sp[0] == q:
        return True
    else:
        return False


x = "100110101"
y = "100110101"

print(protokoll(x, y))
print(protokoll(x, y))
```

Programm 2.1: `protokoll.py`