

SWE3 - Übung 3

WS 2025/26

Aufwand in h: 4

Tim Peko

Inhaltsverzeichnis

1. Aufgabe: Rationale Zahl als Datentyp	2
1.1. Lösungsidee	2
1.1.1. Designentscheidungen	2
1.1.2. Korrektheit	2
1.1.3. Euklidischer Algorithmus	3
1.2. Teststrategie	7
1.3. Ergebnisse	7
1.3.1. Demo Output	7
1.3.2. Testfälle	8

1. Aufgabe: Rationale Zahl als Datentyp

1.1. Lösungsidee

Die Klasse `rational_t` repräsentiert rationale Zahlen als gekürzte Brüche $\frac{\text{Zähler} = Z = \text{nominator} = n}{\text{Nenner} = N = \text{denominator} = d}$ mit `int` als `value_type`.

Invarianten:

- Nenner ist nie 0, ist immer positiv.
- 0 wird als $\frac{0}{1}$ repräsentiert.

Konstruktion und alle Operationen rufen `normalize()` auf, welche mittels euklidischem Algorithmus (beschrieben in Abschnitt 1.1.3) kürzt und das Vorzeichen in den Zähler schiebt.

Fehlerbehandlung:

- Ungültige Eingaben (Nenner 0) lösen `invalid_rational_error` aus.
- Division durch 0 in `/` löst `division_by_zero_error` aus.

Vergleiche nutzen Kreuzmultiplikation, um Rundungsfehler zu vermeiden. Streams werden als „<n/d>“ ausgegeben und „n“ oder „n/d“ eingelesen.

1.1.1. Designentscheidungen

Datentyp

`int` als `value_type` gemäß Aufgabenstellung; API könnte später auf `long long` / `std::int64_t` erweitert werden. Um Überläufe zu vermeiden, werden Zwischenrechnungen in `long long` durchgeführt und anschließend normalisiert/gekürzt.

- Invariante Darstellung: Der Nenner ist stets positiv; das Vorzeichen liegt ausschließlich im Zähler. Die Null wird kanonisch als $0/1$ gespeichert. Die Methode `normalize()` erzwingt diese Regeln und kürzt mit Euklidischem Algorithmus (`gcd`).

Operatoren

Die zusammengesetzten Operatoren (`+=`, `-=`, `*=`, `/=`) bilden die zentrale Implementierung; die binären Operatoren (`+`, `-`, `*`, `/`) delegieren darauf, um Code-Duplikation zu vermeiden. Vergleichsoperatoren nutzen Kreuzmultiplikation ($a/b < c/d \iff ad < cb$) in `long long`, wodurch Rundung vermieden wird.

Interoperabilität mit `int`

Für Ausdrücke mit linkem `int`-Operand (z. B. `3 + rational_t(2,3)`) sind freie Operatoren definiert, um symmetrisches Verhalten zu gewährleisten.

Streams

operator `<<` gibt in der geforderten Form „<n/d>“ bzw. „<n>“ für ganze Zahlen aus. Operator `>>` akzeptiert „n“ oder „n/d“ und setzt `failbit` bei ungültigem Format, wirft aber auch eine Ausnahme bei „n/0“.

1.1.2. Korrektheit

Vergleich

Kreuzmultiplikation ist korrekt, solange das Produkt im `long long`-Bereich

bleibt. Die anschließende Normalisierung begrenzt die Größe \rightarrow realistisch innerhalb typischer Übungsdaten unkritisch.

Robustheit

Jede Methode, die potenziell die Invariante verletzen kann, ruft `normalize()` auf oder prüft mit `is_consistent()`.

Ausnahmen/Exceptions

- `invalid_rational_error` (\leftarrow `std::invalid_argument` \leftarrow `std::logic_error`):
Tritt bei ungültiger Konstruktion auf (z. B. Nenner 0). Das ist ein Verstoß gegen die API-Vertragsbedingungen und daher eine Logik- bzw. Argumentfehler-Kategorie.
- `division_by_zero_error` (\leftarrow `std::domain_error` \leftarrow `std::logic_error`):
Tritt zur Laufzeit beim \neq mit einer rationalen 0 auf. Es entspricht einem Problem mit unserer Eingabedomäne und ist daher ein Logik-/Argumentfehler.

...

The exceptions listed in the `[stdexcept.syn]` section of ISO C++20 standard (the iteration used in this answer) are:

```
1 namespace std {
2     class logic_error;
3     class domain_error;
4     class invalid_argument;
5     class length_error;
6     class out_of_range;
7     class runtime_error;
8     class range_error;
9     class overflow_error;
10    class underflow_error;
11 }
```

cpp

Now you *could* argue quite cogently that either `overflow_error` (the infinity generated by IEEE754 floating point could be considered overflow) or `domain_error` (it is, after all, a problem with the input value) would be ideal for indicating a divide by zero.

...

— paxdiablo auf [Stack Overflow](#)

1.1.3. Euklidischer Algorithmus

Der Euklidische Algorithmus ist ein Algorithmus, der den größten gemeinsamen Teiler (ggT) zweier Zahlen berechnet. Er terminiert in $O(\log(\min(|Z|, |N|)))$.

```
1 int gcd(int a, int b) {
2     while (b != 0) {
3         int t = b;
4         b = a % b;
5         a = t;
6     }
7     return a;
8 }
```

cpp

Der Algorithmus basiert auf der Eigenschaft, dass $\gcd(a, b) = \gcd(b, a \bmod b)$. Durch wiederholte Anwendung dieser Regel wird das Problem auf kleinere Zahlen reduziert, bis eine der Zahlen 0 wird. Der ggT ist dann die andere Zahl.

Beispiel für $\gcd(48, 18)$:

1. $\gcd(48, 18) \Rightarrow \gcd(18, 48 \bmod 18)$
2. $\gcd(18, 12) \Rightarrow \gcd(12, 18 \bmod 12)$
3. $\gcd(12, 6) \Rightarrow \gcd(6, 12 \bmod 6)$
4. $\gcd(6, 0) \Rightarrow 6$

Das Verhalten des Euklidischen Algorithmus ist in Abbildung 1 dargestellt. Dabei wird die Divergenz des Algorithmus für verschiedene Eingaben visualisiert. In Abbildung 2 und Abbildung 3 wird die Anzahl der benötigten Schritte und die Ergebnisse des Euklidischen Algorithmus für $a, b \in [0, 128]$ visualisiert. Die Zeitkomplexität wird in Abbildung 4 veranschaulicht.

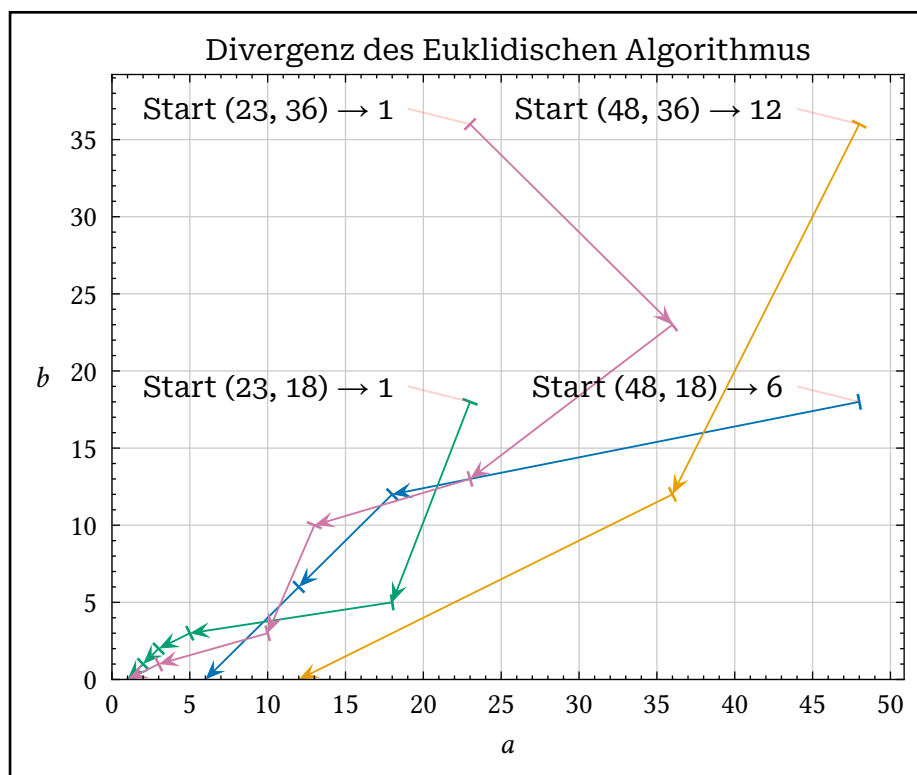


Abbildung 1: Verschiedene Beispiele für die Divergenz des Euklidischen Algorithmus

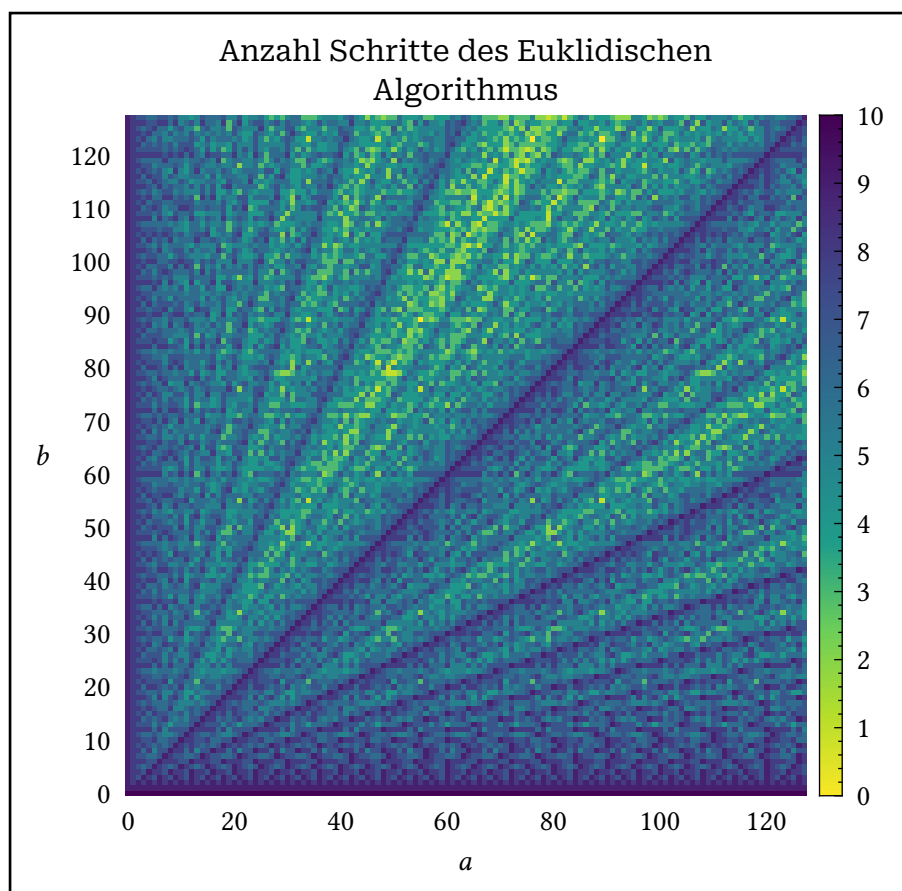
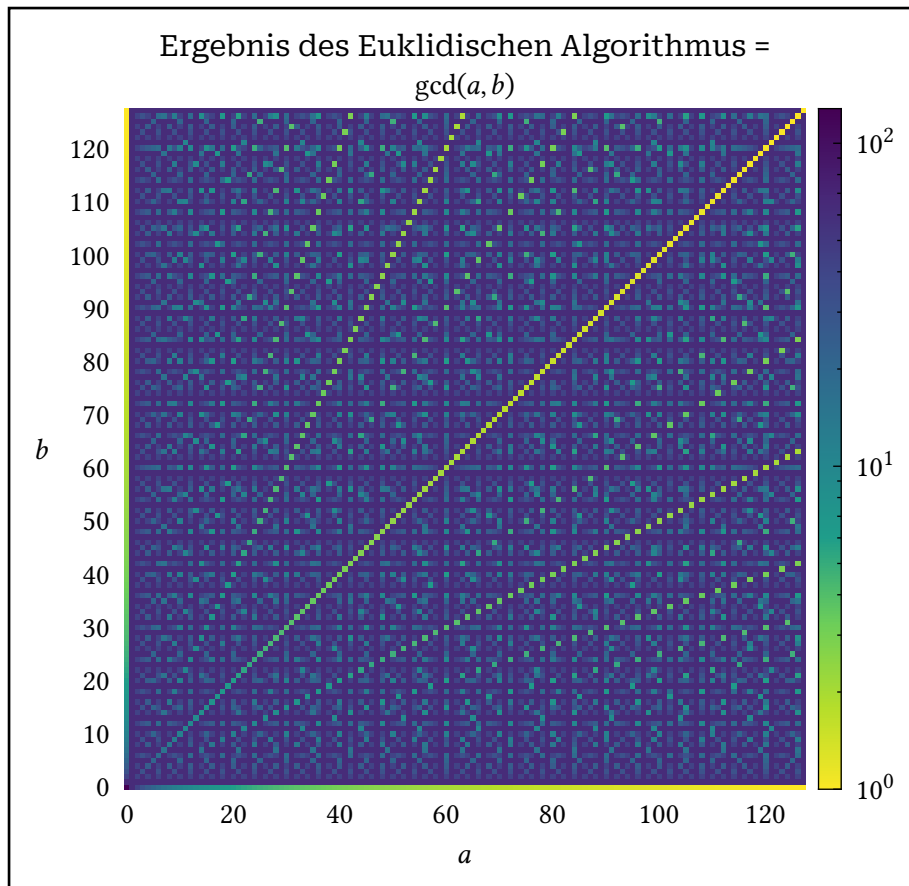
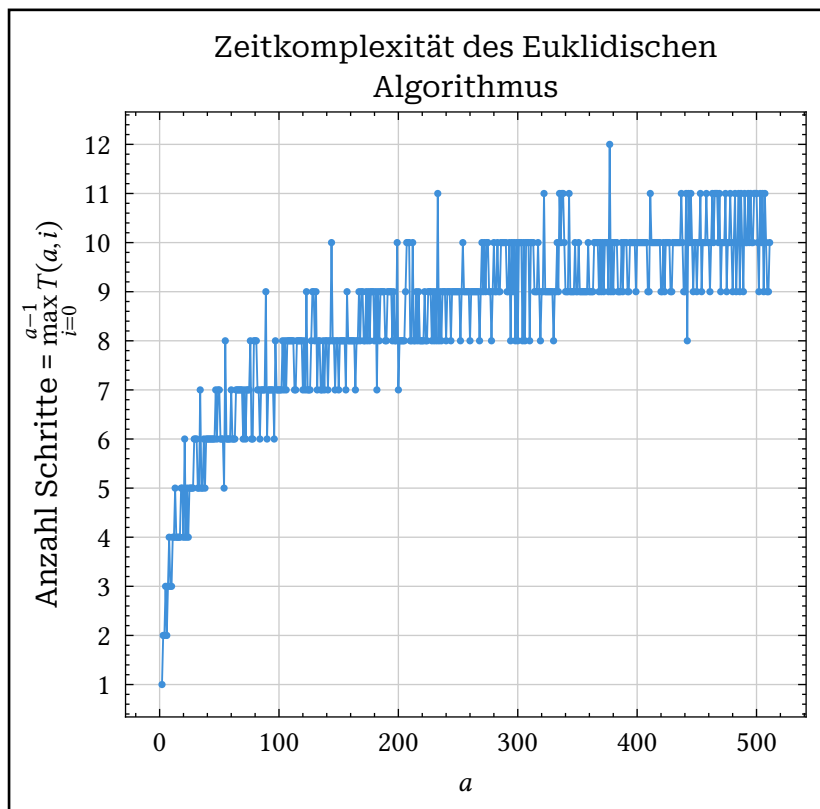


Abbildung 2: Anzahl Schritte des Euklidischen Algorithmus für $a, b \in [0, 128]$

Abbildung 3: Ergebnisse des Euklidischen Algorithmus für $a, b \in [0, 128]$ Abbildung 4: Zeitkomplexität des Euklidischen Algorithmus $\max()$

Die Implementierung des Euklidischen Algorithmus im Projekt verwendet Absolutwerte, um negative Eingaben zu handhaben, und gibt im Grenzfall beider Eingaben gleich 0 den Wert 1 zurück, damit die Normalisierung definiert bleibt.

1.2. Teststrategie

1.3. Ergebnisse

1.3.1. Demo Output

Die `main`-Funktion in der `main.cpp` Datei entspricht genau des in der Aufgabenstellung beschriebenen Beispiels. Die Musterausgabe in der Konsole ist wie folgt:

- Beispiel

```
int main() {
    rational_t r{1, 2};
    std::cout << r * -10 << '\n'
               << r * rational_t(20, 2) << '\n';
    r = 7;
    std::cout << r + rational_t(2, 3) << '\n'
               << 10 / r / 2 + rational_t(6, 5) << '\n';
}
```

- ergibt

```
<5>
<5>
<23/3>
<67/35>
```

Abbildung 5: Musterausgabe der `main`-Funktion in der Aufgabenstellung

Die Ausgabe der eigenen Implementierung ist einsehbar in Abbildung 6 und deckt sich prinzipiell mit der Musterausgabe. Beachte, dass die Ausgabe von $\frac{1}{2} * -10$ faktisch nicht 5 (wie in Abbildung 5 dargestellt) sondern -5 ist, was in meiner Konsole mit Ligatures ($\leftarrow 5$) angezeigt wird.

```
 $\leftarrow 5$ 
<5>
<23/3>
<67/35>

C:\Users\Timer\Documents\Coding\Exercises\
36) exited with code 0 (0x0).
Press any key to close this window . . .|
```

Abbildung 6: Ausgabe der `main`-Funktion unserer Übungsimplementierung

1.3.2. Testfälle