Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2016

Dr. Stefanie Demirci

Computer Aided Medical Procedures
Technische Universität München

Administratives

- Zentralübung (Mittwoch, 09:45 – 11:15)
  - Besprechung des Aufgabenblattes der Vorwoche
  - Fragen von allgemeinem Interesse
  - Probeklausur
- Tutorfragestunden
  - Freitag, 09:00 - 11:00, Raum 0938
  - Freitag, 13:00 - 15:00, Raum 0938
  - Detailliertere und individuelle Fragen
- Kontakt-E-Mail: algods@mailnavab.in.tum.de
- Sprechstunde (Termin per E-Mail)
- Moodle
  - `https://www.moodle.tum.de/course/view.php?id=25832`
  - Diskussionsforum & Bekanntmachungen

`http://campar.in.tum.de/Chair/TeachingSs16AuD`

onlineTED

- `https://www.onlineted.de`
- Webbasiertes Abstimmungssystem
- Live-Umfragen in Lehrveranstaltungen

Euklidischer Algorithmus

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

aus *Euklid: Die Elemente, Buch VII (Clemens Thaer)*

- Vorschrift zur **Berechnung** des *größten gemeinsamen Teilers*
- Keine **Definition** des ggT im mathematischen Sinn
- Verschiedene Algorithmen mit gleichem Ziel ($\rightsquigarrow$ Sortierung!)

Größter gemeinsamer Teiler

Wie lautet der größte gemeinsame Teiler von 11 und 4,
$ggT(11, 4)$?

Euklidischer Algorithmus - Originalformulierung

Input: natürliche Zahlen AB, CD
Output: größten gemeinsamer Teiler von AB, CD

1. Setze $m := AB$ und $n := CD$
2. Falls $m < n$, vertausche m und n
3. Berechne $r := m - n$
4. Setze $m := n$ und $n := r$
5. Falls $r > 0$, gehe zu Schritt 2.
6. Der Output ist m

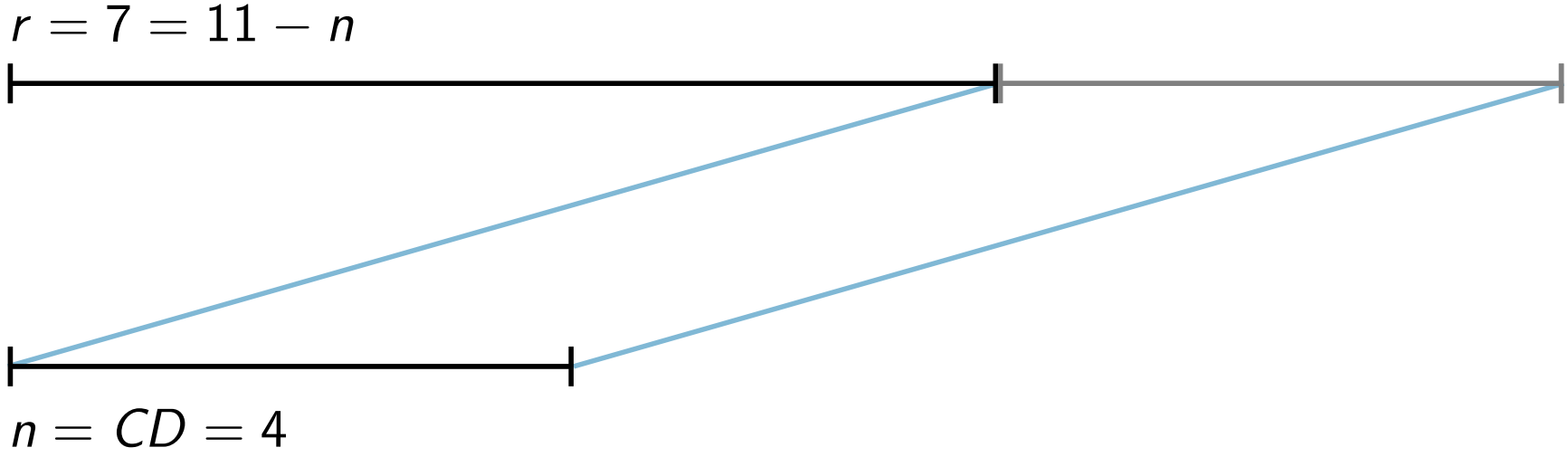
Ablauf nach Originalformulierung

$m = AB = 11$

$n = CD = 4$

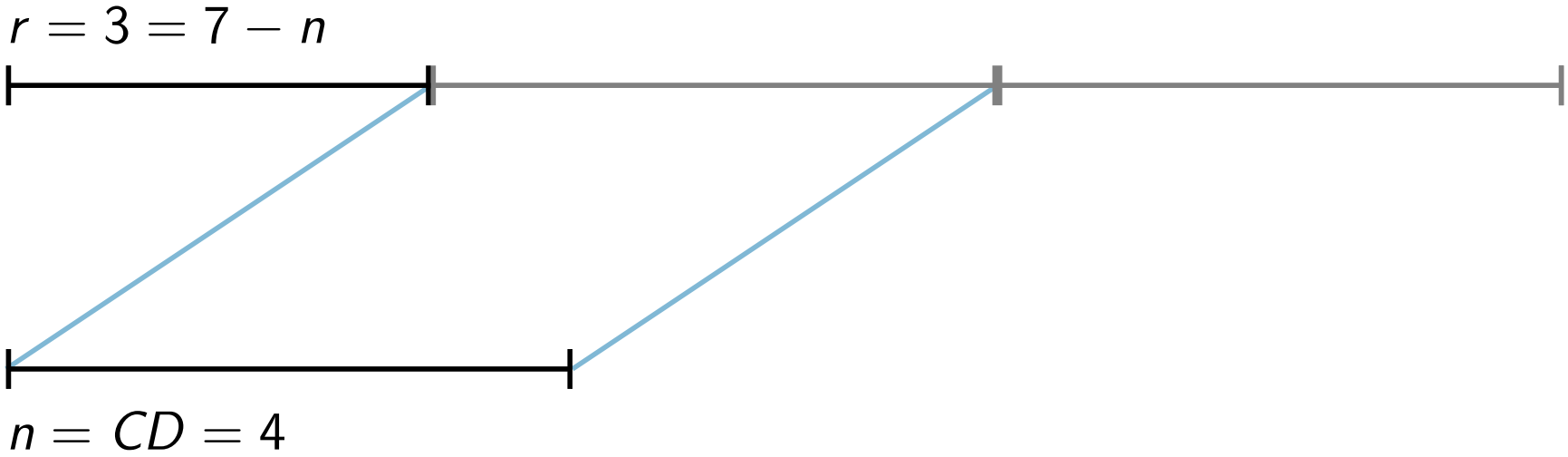
0 Rechenschritte

Ablauf nach Originalformulierung



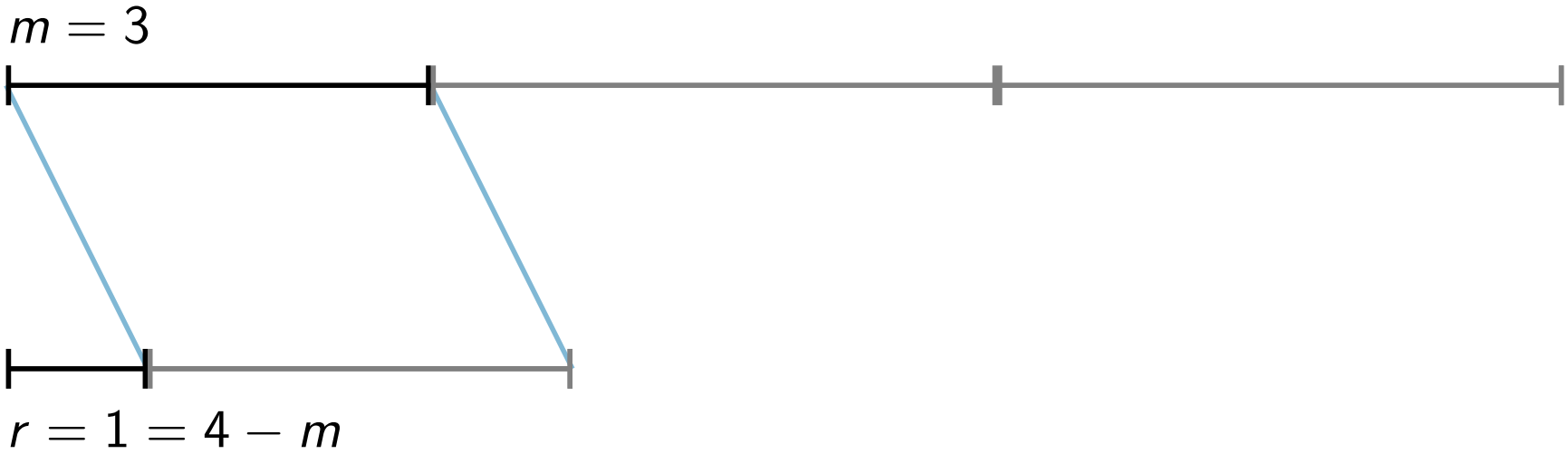
1 Rechenschritt

Ablauf nach Originalformulierung



2 Rechenschritte

Ablauf nach Originalformulierung



3 Rechenschritte

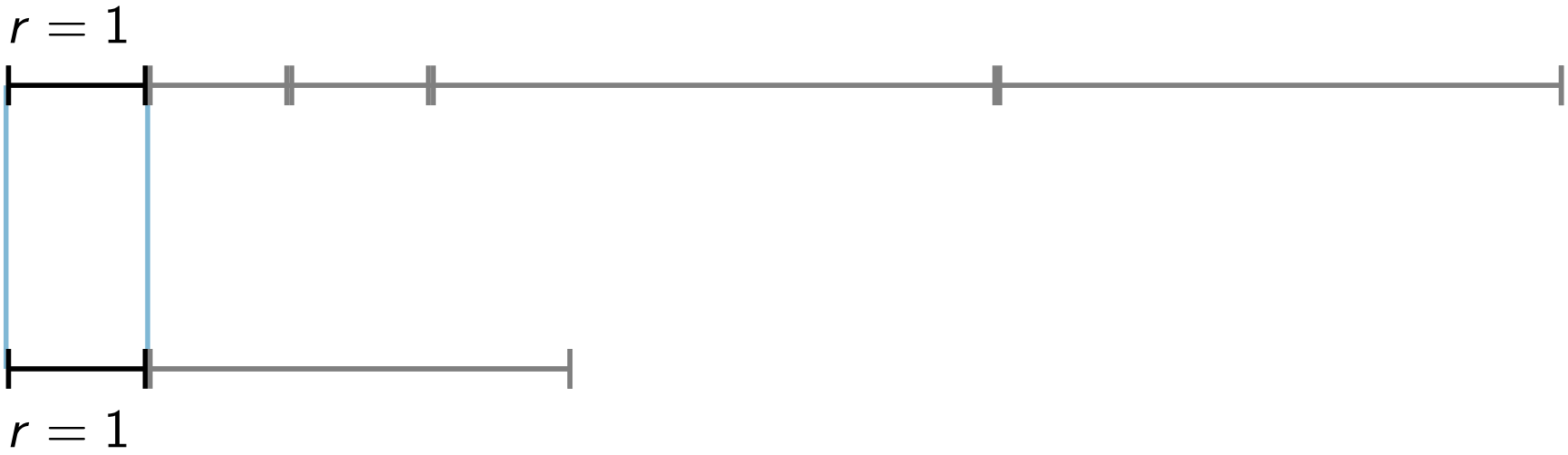
Ablauf nach Originalformulierung

4 Rechenschritte

Ablauf nach Originalformulierung

5 Rechenschritte

Ablauf nach Originalformulierung



$\Rightarrow ggT(11, 4) = 1$ 11 ist eine Primzahl!

5 Rechenschritte

Algorithmus

- präzise/eindeutige, endliche Beschreibung
- Bausteine:
  - Elementarer Verarbeitungsschritt
  - Sequenz (Abfolge elementarer Schritte)
  - Bedingter Verarbeitungsschritt
  - Wiederholung

Variablen

- Speicherung von Werten mittels Typ-behafteter, eindeutig benannter Variablen
  - `int` für ganze Zahlen (Integer)
  - `double` für Fließkommazahlen (Double Precision Floating Point)
  - `char` für Zeichen (Character)
  - ...
- Zuweisung über =
- Kurze Schreibweise für Updates
  - `i *= 4;` statt `i = i * 4;` (ebenso +=, -=, *=, /=, ...)
  - `i++;` statt `i = i + 1;` (ebenso --)

```
int i = 23;
int j = 2 * (i-2);              // Klammerung!
char s = 'a';
```

Elementarer Verarbeitungsschritt

- Mathematische Operatoren +, −, ∗, /, . . .
- Zuweisungen =, +=, −=, ∗=, /=, . . .
- . . .

Sequenz

```
int i = 10;
i = i+2; // Wert von i ist 12
```

Bedingter Verarbeitungsschritt

- `if (b) {...}`
- `if (b) {...} else {...}`
- `if (b) {...} else if (...) {...} else {...}`
- Klammern `{...}` zur Sicherheit immer schreiben
- b steht für einen logischen Ausdruck oder eine logische (boolsche) Variable:

Logische Ausdrücke

- Logischer Typ (`bool`) mit Werten `true` (wahr, 1) und `false` (unwahr, 0)
- Vergleichs-Operatoren: == (gleich), != (ungleich), <, <=, ...
- Verknüpfungen: && (AND; beide wahr), || (OR; mindestens eines von beiden wahr), ...
- Negation: `!a` (NOT)

Wiederholung

Schleifen für Wiederholungen

- `while (b) {...}` (Test vor jeder Iteration)
- `do {...} while (b);` (Test nach jeder Iteration)
- `for i = 1...100 {...}` (Gewisse Anzahl an Iterationen)

Pseudocode

- Informelle Veranschaulichung eines Algorithmus
- Identifizierung der elementaren Bausteine nicht offensichtlich

Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid(a,b):

- Falls $a = 0$, liefere b zurück
- Solange $b > 0$ wiederhole
  - Falls $a > b$ setze $a = a - b$
  - sonst setze $b = b - a$
- Liefere a zurück

Beispiel Programmiersprache: C

- Kompilierte Sprache, sehr effizient

```
int euklid(int a, int b)
{
  if (a==0)
    return b;
  while (b>0)
  {
    if (a>b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

Konvention der Vorlesung

- Pseudocode + C Syntax

```
Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid(a,b):
{
  if (a==0)
    return b;
  while (b>0)
  {
    if (a>b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

Konvention der Vorlesung – Funktionsdefinition

```
Input: Natürliche Zahlen a,b  //Einsprungspunkt
Output: ggT(a,b)  //Aus-/Rückgabe
euklid(a,b):  //Funktionsname
{
  if (a==0)
    return b;
  while (b>0)
  {
    if (a>b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

- Aufruf mittels `euklid(a,b)` (z.B. `euklid(6,3)`)

Konvention der Vorlesung – Identifikation der Bausteine

```
Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid(a,b):
{ //Beginn einer Sequenz
  if (a==0) //Bedingter Verarbeitungsschritt
    return b; //Elementarer Verarbeitungsschritt
  while (b>0) //Wiederholung
  { //Beginn einer Sequenz
    if (a>b) //Bedingter Verarbeitungsschritt
      a=a-b; //Elementarer Verarbeitungsschritt
    else //Bedingter Verarbeitungsschritt
      b=b-a; //Elementarer Verarbeitungsschritt
  } //Ende einer Sequenz
  return a; //Elementarer Verarbeitungsschritt
} //Ende einer Sequenz
```

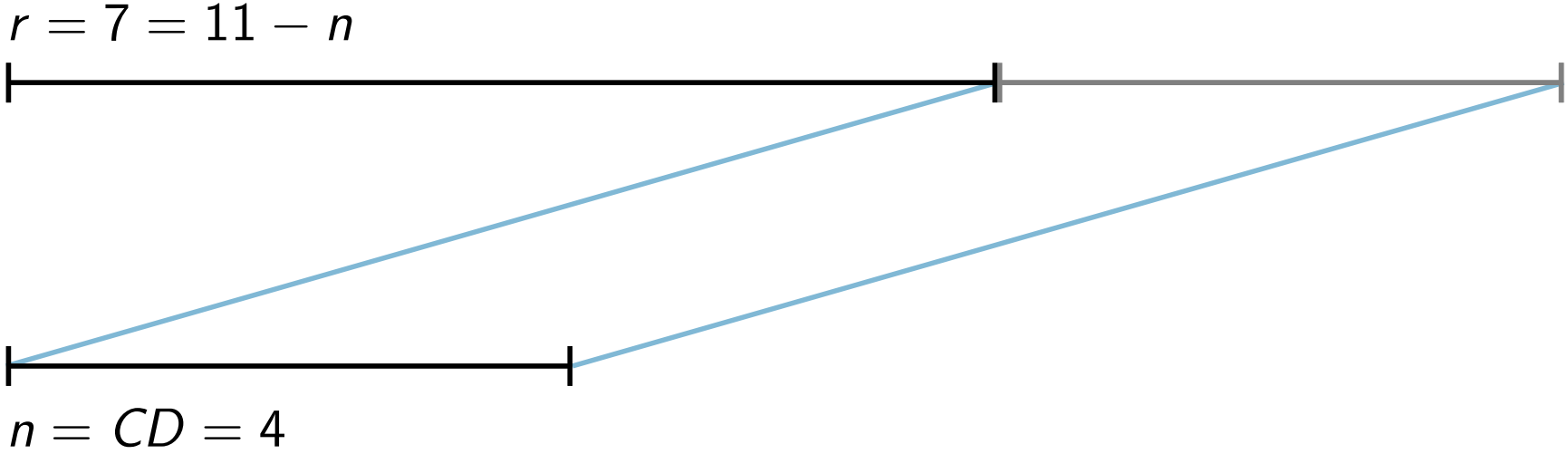
Alternativer Ablauf nach Originalformulierung

$m = AB = 11$

$n = CD = 4$

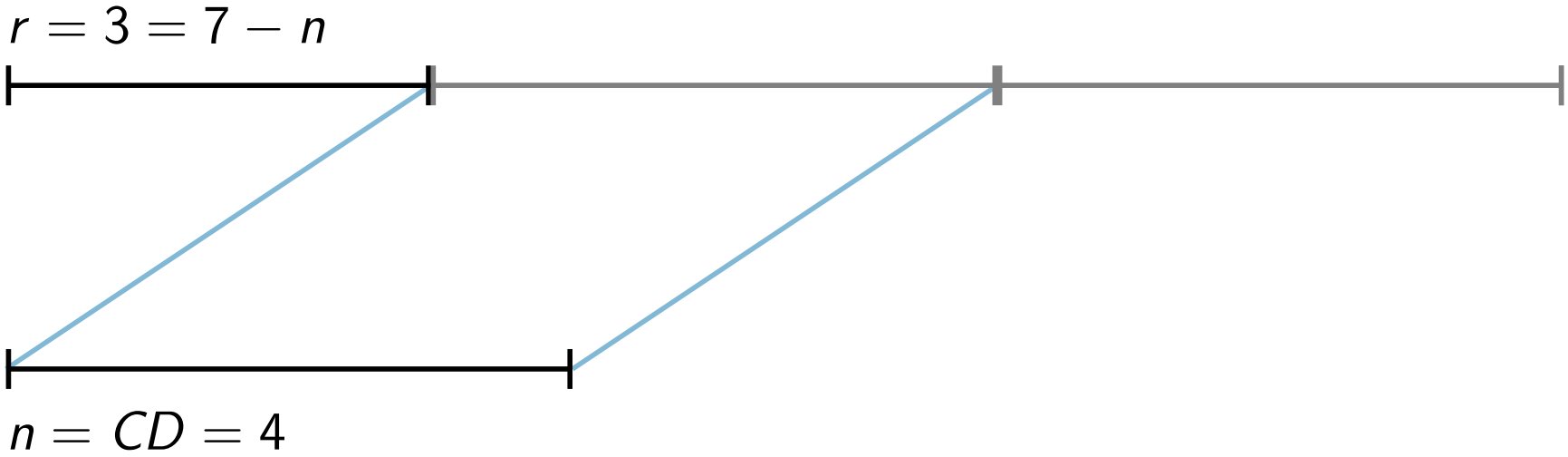
0 Rechenschritte

Alternativer Ablauf nach Originalformulierung



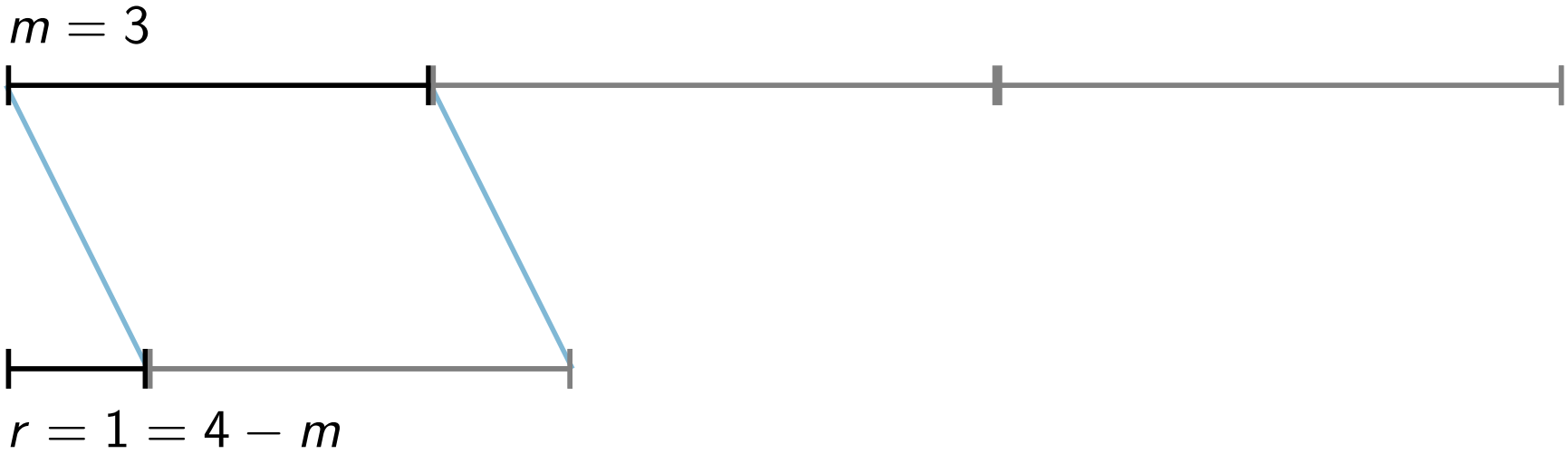
1 Rechenschritt

Alternativer Ablauf nach Originalformulierung



2 Rechenschritte

Alternativer Ablauf nach Originalformulierung



3 Rechenschritte

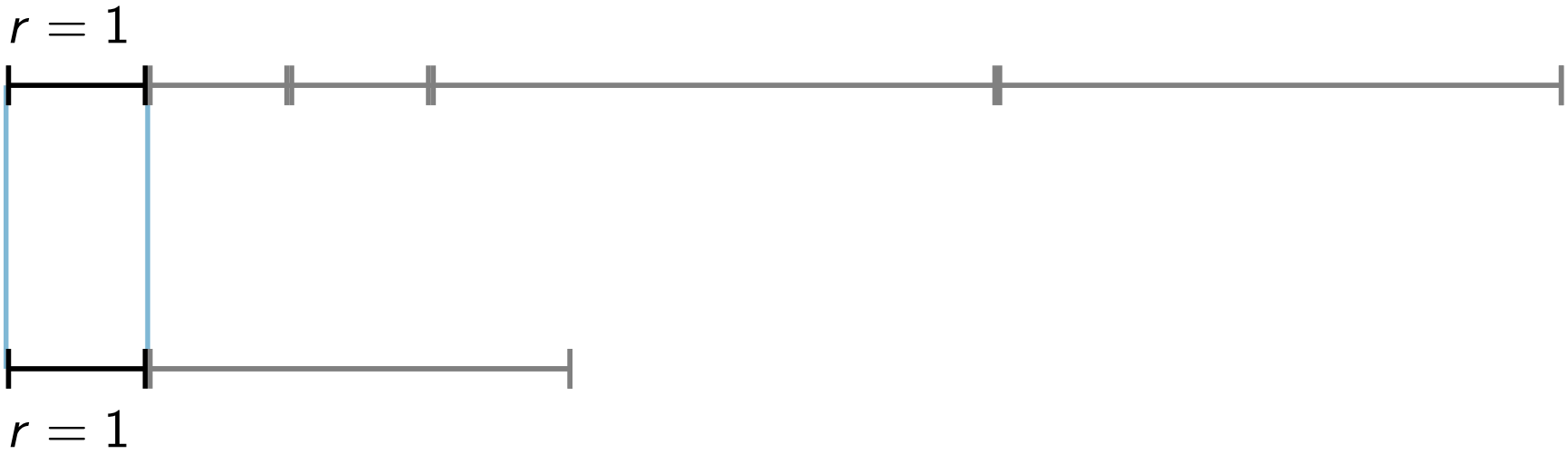
Alternativer Ablauf nach Originalformulierung

4 Rechenschritte

Alternativer Ablauf nach Originalformulierung

5 Rechenschritte

Alternativer Ablauf nach Originalformulierung

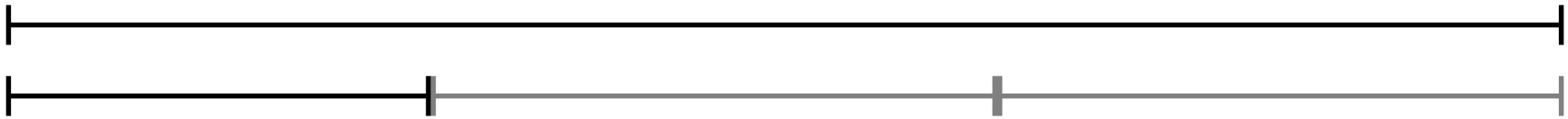


$\Rightarrow ggT(11, 4) = 1$ 11 ist eine Primzahl!

5 Rechenschritte

Beobachtung

$$AB = 11 = 2 \cdot CD + 3$$



$$\begin{array}{rcrcrcrcr} AB & : & CD & = & & 11 & : & 4 & = & 2 \\ & & & & - & 8 & & & & \\ \hline & & & & & 3 & & & & \\ \hline\hline \end{array}$$

- Die *Modulo-Berechnung* liefert den Rest nach Teilung.
- Wir schreiben $11 \bmod 4 = 3$ ...
- ... und programmieren `int rest = 11 % 4;`

Ablauf mit Modulo

$m = AB = 11$

$n = CD = 4$

0 Rechenschritte

Ablauf mit Modulo

$r = 3 = 11 \bmod n$

$n = CD = 4$

1 Rechenschritt

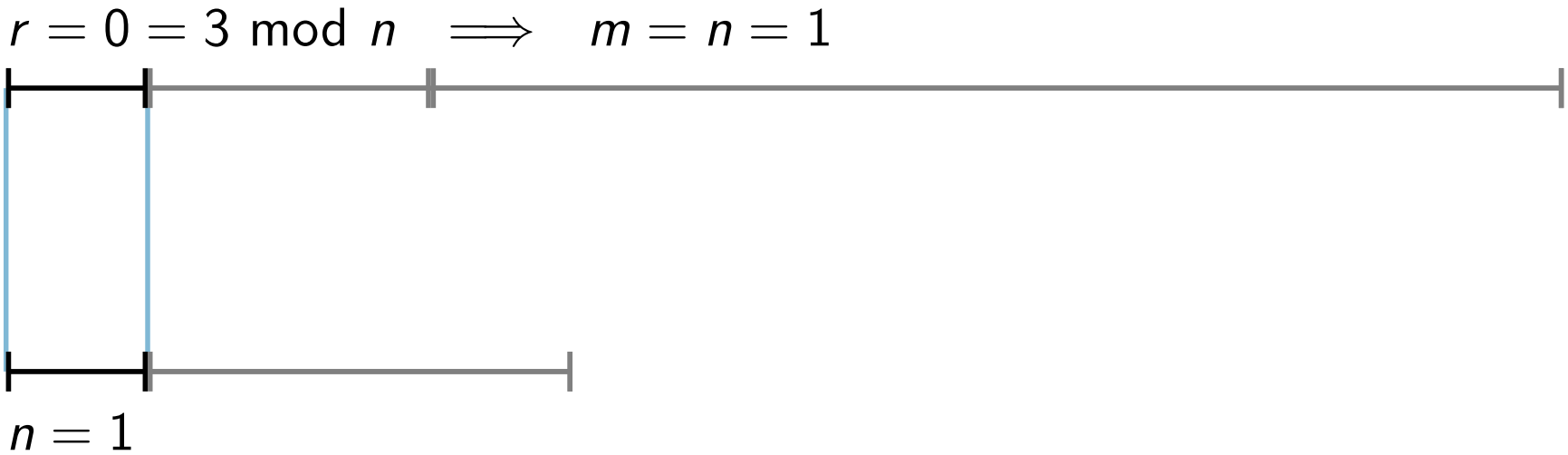
Ablauf mit Modulo

$m = 3$

$r = 1 = 4 \bmod m$

2 Rechenschritte

Ablauf mit Modulo



$\Rightarrow ggT(11, 4) = 1$ Abbruch bei Rest 0!

3 Rechenschritte

Caveat

- Die Zählung von Rechenschritten ist stark vereinfacht!
- Tatsächlich dauert die Berechnung einer Teilung (bzw. einer Modulo-Operation) erheblich länger als eine Differenz!

Implementierung via Modulo

```
Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid_modulo(a,b):
{
   while (b != 0)
   {
      int rest = a%b;
      a = b;
      b = rest;
   }
   return a;
}
```

Rekursion

- Funktionen rufen sich selbst wieder auf, nachdem die Parameter geändert wurden.
- Abbruchbedingungen beenden diese Kette von Aufrufen.

- Beispiel: Addition der Zahlen *m* **bis** *n*

Iterative Variante

```
Input: Natürliche Zahlen m,n
Output: Summe der Zahlen m bis n
summe_iterativ(m,n):
{
   int wert = 0;
   for (int i = m; i <= n; ++i)
      wert += i;
   return wert;
}
```

Rekursive Variante

```
Input: Natürliche Zahlen m,n
Output: Summe der Zahlen m bis n
summe_rekursiv(m,n):
{
   if (m > n)
      return 0;
   else
      return m + summe_rekursiv(m+1, n);
}
```

Aufrufkette

- `summe_rekursiv(2, 4)`
- $2 +$ `summe_rekursiv(3, 4)`
- $2 + 3 +$ `summe_rekursiv(4, 4)`
- $2 + 3 + 4 +$ `summe_rekursiv(5, 4)`
- $2 + 3 + 4 + 0$
- $2 + 3 + 4$
- $2 + 7$
- 9

ggT mit Rekursion

- Mittels dreier Eigenschaften läßt sich der ggT rekursiv berechnen:

$$\begin{aligned} \mathrm{ggT}(ab, cd) &= \mathrm{ggT}(cd, ab) && (1)\\ \mathrm{ggT}(ab, cd) &= \mathrm{ggT}(ab, cd \bmod ab) && (2)\\ \mathrm{ggT}(ab, cd) &= \mathrm{ggT}(ab, cd + \lambda \cdot ab) && (3) \end{aligned}$$

Implementierung via Rekursion

```
Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid_rekursiv(a,b):
{
  if (b==0)
    return a;
  else if (a==0)
    return b;
  else if (a>b)
    return euklid_rekursiv(a-b, b);
  else
    return euklid_rekursiv(a, b-a);
}
```

Implementierung via Rekursion und Modulo

```
Input: Natürliche Zahlen a,b
Output: ggT(a,b)
euklid_rekursiv_modulo(a,b):
{
   if (b==0)
      return a;
   else
      return euklid_rekursiv_modulo(b, a%b);
}
```

Apropos...

- ggT via Primfaktorzerlegung?

Zusammenfassung

- Feststellung: Unterschiedliche Algorithmen für die selbe Aufgabe
- Angenommen eine Aufgabe sei gegeben (z.B.: Automatisierte Sortierung von Paketen)
  - Welche Algorithmen eignen sich hierzu?
  - Welche Unterschiede gibt es
  - Welcher Algorithmus eigenen sich am besten?