Modulare Arithmetik

Modulare Arithmetik

Dieses Notebook benutzt Sage für verschiedene Rechnungen rund um das modulare Rechnen mit ganzen Zahlen. Es wird dabei zum einen auf eingebaute Funktionen von Sage zurückgegriffen und zum anderen werden auch eigene Funktionen programmiert.

Die **ganzzahlige Division** ist in Sage mit den beiden Operator // und % realisiert. Der erste liefert den ganzzahligen Quotienten, der zweite den Rest.

```
a = 13 // 3;
b = 13 % 3;
print(a);
print(b);
3*a+b
```

```
4
1
13
```

Wir können auch eine eigene Funktion für die ganzzahlige Division schreiben, die sowohl den Quotienten als auch den Rest zurückgibt. Wir geben hier gleich zwei Varianten an: eine rekursive und eine iterative Version. Man beachte, daß beide Versionen ausschließlich ganzzahlige Operationen durchführen. Eine viel schnellere Funktion würde man erhalten, wenn man erst eine reelle Division durchführt und dann aus deren Ergebnis den ganzzahligen Quotienten und Rest berechnen würde.

```
def intdiv_rek(x,y):
    """
    ganzzahlige Division mit Berechnung von Quotient und Rest
    rekursive Version
    """
    if not(isinstance(x,Integer) and isinstance(y,Integer)):
        raise TypeError
    if x<y:
        return 0,x
    else:
        q,r = intdiv_rek(x-y,y)
        return q+1,r

def intdiv_iter(x,y):
    """
    ganzzahlige Division mit Berechnung von Quotient und Rest
    iterative Version
    """
    if not(isinstance(x,Integer) and isinstance(y,Integer)):
        raise TypeError
    q = 0
    r = x
    while r>=y:
        r = r-y
        q = q+1
    return q,r
```

```
print(intdiv_rek(13,3));
print(intdiv_iter(13,3))
```

```
(4, 1)
(4, 1)
```

Um messbare Zeiten für die Ausführung dieser einfachen Funktionen zu erhalten, muß man schon mit relativ großen Zahlen arbeiten. Dabei stellt sich das Problem, das die Programmiersprache Python, die unter Sage liegt, standardmäßig keine allzu große Rekursionstiefe erlaubt. Für einen Vergleich unserer beiden Funktionen müssen wir also erst dieses Limit erhöhen. Man sieht, daß die iterative Variante viel schneller ist.

```
print(sys.getrecursionlimit());
```

```
sys.setrecursionlimit(1000000)
```
```
1000
```
```
%time
intdiv_rek(12345678,1357)
```
```
(9097, 1049)
CPU time: 0.09 s,  Wall time: 0.09 s
```
```
%time
intdiv_iter(12345678,1357)
```
```
(9097, 1049)
CPU time: 0.01 s,  Wall time: 0.01 s
```

Den **größten gemeinsamen Teiler** (ggT) zweier ganzer Zahlen kann man mit dem Euklidischen Algorithmus berechnen. In seiner erweiterten Form liefert er auch noch zugehörige **Bézout-Koeffizienten**. Sage verfügt über einen eingebauten Befehl gcd (greatest common divisor) zur Berechnung des ggTs; die erweiterte Form erhält man mit xgcd.

```
print(gcd(24,15));
d,s,t = xgcd(24,15); print(d,s,t);
24*s+15*t-d
```
```
3
(3, 2, -3)
0
```

Wieder können wir auch unsere eigene ggT-Funktion schreiben. Dazu implementieren wir eine iterative Form des erweiterten Euklidischen Algorithmus. Analog zu dem xgcd-Befehl von Sage geben wir dabei zuerst den ggT aus.

```
def erw_euklid(x,y):
    """
    erweiterter Euklidischer Algorithmus
    iterative Form
    """
    if not(isinstance(x,Integer) and isinstance(y,Integer)):
        raise TypeError
    s1 = 1; s2 = 0; t1 = 0; t2 = 1; r1 = y; r2 = x;
    while r2<> 0:
        q = r1 // r2;
        a = s1; s1 = s2; s2 = a - q*s2
        a = t1; t1 = t2; t2 = a - q*t2
        a = r1; r1 = r2; r2 = a - q*r2
    if x<=y:
        return r1,s1,t1
    else:
        return r1,t1,s1
```

```
r,s,t = erw_euklid(24,15); print(r,s,t);
s*24+t*15-r
```
```
(3, 2, -3)
0
```

Für den **Chinesischen Restesatz** stellt Sage die Funktion crt zur Verfügung. Dieser Funktion übergibt man als erstes Element eine Liste mit den Resten und als zweites Element eine Liste der zugehörigen Moduli. Als Ergebnis erhält man die eindeutige Lösung, die kleiner als das Produkt aller Moduli ist. Dabei wird automatisch überprüft, ob die Moduli teilerfremd sind.

```
x = crt([4,5,3],[7,8,13]);
print(x); print(x % 7); print(x % 8); print(x % 13)
```
```
445
4
5
3
```

Den **Restklassenring** modulo einer ganzen Zahl n erhält man mittels Integers. Im folgenden Beispiel rechnen wir modulo 7. Es wird gezeigt, wie man Elemente in einem Restklassenring erzeugen kann und wie man mit diesen rechnet. Man beachte, daß bei Eingabe einer Zahl außerhalb des Bereichs 0 bis 6 automatisch eine Reduktion auf einen solchen "kanonischen" Rest erfolgt.

```
Z7 = Integers(7);
a = Z7(13); print(a);
b = Z7(-5); print(b);
print(a+b); print(a*b)
```

```
6
2
1
5
```