

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 1:

Programmcode

```
inverse::Integer->Integer->Integer
inverse 0 _ = error"Fehler: Es gibt kein Inverses zu 0."
inverse x p = help x p p
  where help x p n | ((x*n) `mod` p) == 1   = n
                  | otherwise              = help x p (n-1)
```

Testlauf

```
Main> inverse 17 500
353
Main> inverse 17 5312
625
Main> inverse 17 1028
121
```

Beschreibung

hier haben wir die hilfsmfunktion eingefügt
wenn der Rest eins ist dann wir n aus gegeben,
sonst wird die funktion rekursiv aufgerufen bis das n gefunden wird,so dass Rest = 1

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 2:

Programmcode

```
1 type Key = (Integer, Integer)
2
3 p,q,n,phi_n::Integer
4 p = 23
5 q = 29
6 n = p*q
7 phi_n = (p-1)*(q-1)
8 pk = (17,n)
9 sk = findKey pk
10
11 findKey::Key->Key
12 findKey (e, base) = (inverse e phi_n, base)
13
14 rsa_encode, rsa_decode::Integer->Key->Integer
15 rsa_encode msg (e,base) = power msg e base
16 rsa_decode msg (d,base) = power msg d base
17
18 power, multmod::Integer->Integer->Integer->Integer
19 power x 0 _ = 1
20 power x y base
21 | odd y      = multmod x (power (multmod x x base) (y `div` 2) base) base
22 | even y     = power (multmod x x base) (y `div` 2) base
23
24 multmod x y base = ((x `mod` base)*(y `mod` base) `mod` base)
```

Testlauf

Beschreibung

Zeile 1: Typen-Festlegung von Schlüssel

Zeile 3: das ist der Implementierte RSA-Algo.

Zeile 14: Entkodierung bzw. Decodierung des Schlüssels

Zeile 19: Typen Festlegung von power und multmod

Zeile 20: diese Funktion ist die Implementierung des Algo. der in der VL. vorgestellt wurde

Zeile 21: schnelle Exponentiation

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 3a:

Programmcode

```
binadd :: (Int,Int,Int,Int,Int,Int,Int,Int) -> (Int,Int,Int,Int,Int,Int,Int,Int) ->
(Int,Int,Int,Int,Int,Int,Int,Int)
binadd (a1,a2,a3,a4,a5,a6,a7,a8) (b1,b2,b3,b4,b5,b6,b7,b8) = (x1,x2,x3,x4,x5,x6,x7,x8)
  where   x8      =(a8+b8) `mod` 2
         temp8=(a8+b8) `div` 2
         x7      =(temp8 + a7 + b7) `mod` 2
         temp7=(temp8 + a7 + b7) `div` 2
         x6      =(temp7 + a6 + b6) `mod` 2
         temp6=(temp7 + a6 + b6) `div` 2
         x5      =(temp6 + a5 + b5) `mod` 2
         temp5=(temp6 + a5 + b5) `div` 2
         x4      =(temp5 + a4 + b4) `mod` 2
         temp4=(temp5 + a4 + b4) `div` 2
         x3      =(temp4 + a3 + b3) `mod` 2
         temp3=(temp4 + a3 + b3) `div` 2
         x2      =(temp3 + a2 + b2) `mod` 2
         temp2=(temp3 + a2 + b2) `div` 2
         x1      =(temp2 + a1 + b1) `mod` 2
```

Testlauf

```
Main> binadd (0,0,0,0,0,0,1,0) (0,0,0,1,1,0,0,1)
(0,0,0,1,1,0,1,1)
Main> binadd (0,0,0,1,1,1,1,1) (0,0,1,1,1,1,1,1)
(0,1,0,1,1,1,1,0)
Main> binadd (0,1,1,1,1,1,1,1) (0,0,0,0,0,0,0,1)
(1,0,0,0,0,0,0,0)
```

Beschreibung

Die Eingabe erfolgt in 2x8Tupel vom Typ Integer, die Ausgabe in 1x8Tupel vom Typ Integer.
Das Programm addiert die eingegebenen Werte von hinten nach vorne durch Anwendung der Rechenregeln für binäre Addition.

Eventuelle Überträge werden in der Variable tempX gespeichert, wobei X für die aktuelle Position in der Addition bezeichnet. Die temp-Variable hätte man auch weglassen können, nur wäre der Quelltext dann wesentlich länger und ineffizienter geworden, da der Inhalt jeder Tempvariable für jede Rechenzeile erneut berechnet werden müsste.

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 3b:

Programmcode

```
i :: Int -> Int
i (x)
| x == 0 = 1
| x == 1 = 0

binv :: (Int, Int, Int, Int, Int, Int, Int, Int) -> (Int, Int, Int, Int, Int, Int, Int, Int)
binv (x1,x2,x3,x4,x5,x6,x7,x8) = (i(x1), i(x2), i(x3), i(x4), i(x5), i(x6), i(x7), i(x8))

binsub :: (Int,Int,Int,Int,Int,Int,Int,Int) -> (Int,Int,Int,Int,Int,Int,Int,Int) ->
(Int,Int,Int,Int,Int,Int,Int,Int)
binsub (a1,a2,a3,a4,a5,a6,a7,a8) (b1,b2,b3,b4,b5,b6,b7,b8) = binadd (a1,a2,a3,a4,a5,a6,a7,a8) (binadd
(biniv(b1,b2,b3,b4,b5,b6,b7,b8)) (0,0,0,0,0,0,0,1))
```

Testlauf

```
Main> binsub (0,0,0,0,0,0,1,0) (0,0,0,1,1,0,0,1)
(1,1,1,0,1,0,0,1)
Main> binsub (0,0,0,1,1,1,1,1) (0,0,1,1,1,1,1,1)
(1,1,1,0,0,0,0,0)
Main> binsub (0,1,1,1,1,1,1,1) (0,0,0,0,0,0,0,1)
(0,1,1,1,1,1,1,0)
```

Beschreibung

Dieses Programm nutzt 3 Hilfsprogramme: binadd, i und binv.
Binadd wurde bereits in der vorigen Aufgabe erklärt.
Die Funktion i gibt einfach nur das binäre inverse vom Eingangswert.
Die Funktion binv bekommt eine 8Tupel vom Typ Integer und wendet auf jedes Element der Tupel die Funktion i an.
Das Hauptprogramm binsub addiert zuerst 00000001 zum inversen des 2. Eingangswertes und addiert dann zu diesem Ergebnis den 1. Eingangswert.

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 4:

Programmcode

```
dec2bin :: Int -> (Int, Int, Int, Int, Int, Int, Int, Int)
dec2bin bin = (c1, c2, c3, c4, c5, c6, c7, c8)
  where
    c8 = bin `mod` 2
    c7 = (bin `div` 2) `mod` 2
    c6 = (bin `div` 4) `mod` 2
    c5 = (bin `div` 8) `mod` 2
    c4 = (bin `div` 16) `mod` 2
    c3 = (bin `div` 32) `mod` 2
    c2 = (bin `div` 64) `mod` 2
    c1 = (bin `div` 128) `mod` 2
```

Testlauf

```
Main> dec2bin 255
(1,1,1,1,1,1,1,1)
Main> dec2bin 16
(0,0,0,1,0,0,0,0)
Main> dec2bin 65
(0,1,0,0,0,0,0,1)
```

Beschreibung

Die Eingabe erfolgt in Integer, die Ausgabe ist ein 8Tupel vom Typ Integer.
Der Eingabewert wird 8 mal durch die Potenzen von 2 geteilt, die Reste werden durch `mod` ermittelt und in den Ausgabevariablen c8 bis c1 gespeichert.
Durch die 1. Potenz von 2 wird nicht dividiert, da $2^0 = 1$ und ist somit trivial.

Lösung Übungsblatt Nr. 2 / ALP 1 zur Abgabe 12.11.2003

Aufgabe 5:

Programmcode

```
myadd :: (Int, Int) -> (Int,Int) -> (Int,Int)
myadd (n1,z1) (n2,z2) = ((nok `div` (gcd nok zok)) , (zok `div` (gcd nok zok)))
  where
  nok=(z2*n1)+(z1*n2)
  zok=(z2*z1)

mysub :: (Int, Int) -> (Int,Int) -> (Int,Int)
mysub (n1,z1) (n2,z2) = ((nok `div` (gcd nok zok)) , (zok `div` (gcd nok zok)))
  where
  nok=(z2*n1)-(z1*n2)
  zok=(z2*z1)

mymul :: (Int, Int) -> (Int,Int) -> (Int,Int)
mymul (n1,z1) (n2,z2) = ((nok `div` (gcd nok zok)) , (zok `div` (gcd nok zok)))
  where
  nok=(n1*n2)
  zok=(z1*z2)

mydiv :: (Int, Int) -> (Int,Int) -> (Int,Int)
mydiv (n1,z1) (n2,z2) = ((nok `div` (gcd nok zok)) , (zok `div` (gcd nok zok)))
  where
  nok=(n1*z2)
  zok=(z1*n2)
```

Testlauf

```
Main> myadd (1,2) (1,3)
(5,6)
Main> myadd (1,2) (1,4)
(3,4)
Main> myadd (2,9) (1,4)
(17,36)

Main> mysub (1,2) (1,3)
(1,6)
Main> mysub (1,2) (1,4)
(1,4)
Main> mysub (2,9) (1,4)
(-1,36)

Main> mymul (1,2) (1,3)
(1,6)
Main> mymul (1,2) (1,4)
(1,8)
Main> mymul (2,9) (1,4)
(1,18)

Main> mydiv (1,2) (1,3)
(3,2)
Main> mydiv (1,2) (1,4)
(2,1)
Main> mydiv (2,9) (1,4)
(8,9)
```

Beschreibung

Das Grundkonzept für alle 4 Teilaufgaben ist jeweils das Gleiche:
Die Eingabe erfolgt in 2x2Tupel vom Typ Integer.
Die Ergebnisse (nok,zok) werden jeweils noch einmal durch ihren ggT dividiert.

Zur Berechnung von nok und zok werden die allgemein bekannten Rechenregeln für schriftliches Rechnen mit Brüchen genutzt. n1 steht für den 1. Nenner und z2 für den 2. Zähler, rest analog.