Punkteverteilung:

1	Σ

Aufgabe (1)

(a) $n = p \cdot q = 47 \cdot 71 = 3337$

$x = (p-1) \cdot (q-1) = (47-1) \cdot (71-1) = 46 \cdot 70 = 3220$

(b) Der Rechenweg wird durch dieses Scheme-Programm beschrieben:

```
#lang scheme
;;; pseudo RSA stuff for the GBS sheet 13 (WS0910)
;;; purely functional solution without loops, mutation, IO
;;; (c) 2010 by Marek Kubica

;; some definitions from the sheet
(define p 47)
(define q 71)
(define n (* p q))
(define x (* (- p 1) (- q 1)))
(define e 79)
(define d 1019)
(define message 688232687966668003)

;; gets the magnitude of a number: 1000 = 10e3
;; magnitude returns the 3
(define magnitude
  (lambda (number)
    (if (<= number 9) 0
        (add1 (magnitude (quotient number 10))))))

;; exponentiation function
(define exp (curry expt 10))

;; split number in smaller numbers of some specific length
(define chunk
  (lambda (number chunk-size)
    (let*-values ([(exp-difference) (sub1 chunk-size)]
                  [(q r)
                   (quotient/remainder number (exp (- (
                     ➥magnitude number) exp-difference)))])
      (if (<= (magnitude r) exp-difference) (list q r)
          (cons q (chunk r chunk-size))))))

;; split the number into the biggest chunks that are smaller
   ➥ than n
(define smaller-chunks
  (lambda (message n)
    (let* ([n-size (magnitude n)]
           [candidates (chunk message (add1 n-size))])
```

```
        ; if the candidate list contains some numbers larger
           ➥than n, create a
        ; new chunked list with smaller numbers
        (if (andmap (lambda (element) (< element n))
           ➥candidates) candidates
            (chunk message n-size)))))

;; get the maximal magnitude of a list of numbers
(define maximal-magnitude
  (lambda (numbers)
    (apply max (map magnitude numbers))))

;; constructs a number by repeatedly multiplicating
;; '(12 3 4) -> 120304
(define construct-message
  (lambda (numbers)
    (let ([shift-with (exp (add1 (maximal-magnitude numbers)
       ➥))])
      (foldl (lambda (new msg) (+ (* msg shift-with) new)) 0
         ➥ numbers))))

;; do the parsing (splitting), exponentiation and modulo
(define encrypt
  (lambda (message e n)
    (construct-message
     (map (lambda (element) (remainder (expt element e) n))
          (smaller-chunks message n)))))

;; inversion of encrypt
(define decrypt
  (lambda (message d n)
    (construct-message
     (map (lambda (element) (remainder (expt element d) n))
          (smaller-chunks message n)))))

;; roundtrip works
(decrypt (encrypt message e n) d n)
```

Das Programm nutzt Erweiterungen von PLT Scheme 4 und wurde auf PLT Scheme 4.2.4 getestet. Der Fokus der Implementation lag auf der sauberen, funktionalen Problemlösung, die Performance ist außer acht gelassen wurden, so sind auch die rekursiven Funktionen bewusst nicht endrekursiv. Es wurde jedoch geachtet, keine "Tricksereien" zu verwenden, wie etwa die triviale Aufteilung einer der Nachricht durch das Konvertieren in einen String und das Aufteilen des Strings in gleich größe Blöcke. Dieser Ansatz ist zwar simpler, vermittelt jedoch den Eindruck einer unsauberen Lösung.

Um die Nachricht zu verschlüsseln ist folgender Aufruf notwendig:

```
(encrypt message e n)
```

Diese Funktion gibt die verschlüsselte Nachricht zurück, im Fall der Beispielwerte

(message = 688232687966668003 und e = 79) ist dies der Wert 157027562091227-624230158.

(c) Es ist lediglich notwendig die Umkehrfunktion zum verschlüsseln aufrufen:

```
(decrypt (encrypt message e n) d n)
```

Der Rückgabewert mit den im Blatt angegebenen Werten (d = 1019) beträgt 688232-687966668003, was genau der Eingabenachricht entspricht. Dies zeigt, dass Roundtrips zumindest mit den getesteten Daten möglich sind.