1) a) Calling `invmod numbere ((primep-1)*(primeq-1))` in ghci after loading the supplied program yields `391800632124473`. To verify it is correct, one may then proceed with computing `it * numbere `mod` ((primep-1)*(primeq-1))` which indeed yields `1`.

b) We first supplement the Haskell program with the two lines corresponding to the slides:

```
publickey = (numbere,primep*primeq)
privatekey = invmod numbere ((primep-1)*(primeq-1))
```

Calling `encrypt "cat"` then yields `189056892230455`, upon which `decrypt it` yields `"cat"` again, as desired. This makes use of that `encode "cat"` yields `18947`, which is verified to be correct by computing `decode it`, which yields `"cat"` again.

Calling `encrypt "mouse"` yields `3899450432642`, upon which `decrypt it` yields `"mouse"` as desired. This makes uses of that `encode "mouse"` yields `17625144`, which is verified to be correct by computing `decode it`, yielding `"mouse"` again.

Note that since the word `"mouse"` is longer than the word `"cat"`, the code of the former is longer than the code of the latter, but the same does not hold for their encryptions, due to the modular arithmetic.

c) Encryption gives unique numbers modulo $p \cdot q$. Since the latter is a 15-digit number here, codes should be no longer than 15 digits, corresponding roughly to 10 letters ($\approx \frac{15}{\log_{10} 26}$).[1] For longer codes, i.e. on a larger domain, encrypting is not *injective* so different strings may be encrypted as the same number and are (incorrectly) decrypted as the same string.

d∗) The idea of using the Chinese remainder theorem is to replace computations modulo (the larger number) $p \cdot q$ by computations modulo (the much smaller) p and q separately. We implement this in its basic form, as explained on wikipedia (accessed 9–12–2019). In particular, we supplement the Haskell program with:

```
numberdp = invmod numbere (primep-1)
numberdq = invmod numbere (primeq-1)
invqp = invmod primeq primep
decryptcrt :: Integer -> String
decryptcrt c = decode (mq + h * primeq) where
  mp = expmod c numberdp primep
  mq = expmod c numberdq primeq
  h = invqp * (mp - mq) `mod` primep
```

This results in the same decryptions as in the second item.

2) By definition $\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty}(\sup\{\frac{f(n)}{g(n)} \mid m \geq n\})$. We first show that if n tends to ∞, the latter suprema tend to 0. By high school math, we have $\frac{f(n)}{g(n)} \leq \frac{\log n}{n} \leq \frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}}$. Using this and that for any $\epsilon > 0$ there is an n such that $\frac{1}{n} < \epsilon$, we have for all $m \geq n^2$ that

[1] Because we switch from a base-10 (the digits) to a base-26 (the alphabet) representation. This analogous to asking how many bits one needs to store arbitrary numbers $< 10^{100}$, where the answer is $100 \cdot (\log_2 10)$ with the factor $\log_2 10 \approx 3.322$ reflecting the increase in length incurred by switching from base-10 to base-2.

$\frac{f(m)}{g(m)} \leq \frac{1}{\sqrt{m}} \leq \frac{1}{\sqrt{n^2}} = \frac{1}{n} < \epsilon$, from which we conclude. From this we see $f \in o(g)$ and, using the theorem on slide 10 of week 10, also $f \in O(g)$ and therefore $g \in \Omega(f)$. Note that for any f, g it holds $f \in O(g)$ iff $g \in \Omega(f)$.

The converse is analogous, rewriting the first inequality above into $\sqrt{n} \leq \frac{g(n)}{f(n)}$ to see that the suprema tend to ∞, concluding that $\limsup_{n\to\infty} \frac{g(n)}{f(n)} = \infty$. We have $g \notin o(f)$ and, using the theorem on slide 10 of week 10 again, also $g \notin O(f)$, and therefore $f \notin \Omega(g)$.

Combining both we have $f \notin \Theta(g)$ which is equivalent to $g \notin \Theta(f)$.

3) Unfolding the definition of O in the assumption $f, g \in O(h)$, for f, g, h functions from $\mathbb{N}$ to $[0, \infty)$, gives positive real numbers c, c' and natural numbers m, m' such that for all $n \geq m$, $f(n) \leq c \cdot h(n)$ and for all $n' \geq m'$, $f(n') \leq c' \cdot h(n')$. means

   a) Intuitively, since comparison of functions using O is up to some factor, choosing the factor large enough can compensate for adding the function values. Formally, we claim $f + g \in O(h)$. To see this, we set m'' to $\max(m, m')$ and c'' to $c + c'$, and verify that for all $n'' \geq m''$, $(f + g)(n'') \leq c'' \cdot h(n'')$ by computation:

   $$(f+g)(n'') =_{\text{def}} f(n'')+g(n'') \leq_{\text{ass (2x)}} (c\cdot h(n''))+(c'\cdot h(n'')) =_{\text{distr}} (c+c')\cdot h(n'') =_{\text{def}} c''\cdot h(n'')$$

   where both assumptions may be applied because $n'' \geq m'' = \max(m, m') \geq m, m'$.

   b) Intuitively, taking a product of function values should yield a greater function; e.g. it allows to go from linear to quadractic. Formally, we claim there are functions f, g, h from $\mathbb{N}$ to $[0, \infty)$, such that $f \cdot g \notin O(h)$. To see this, set each of f, g, h to the identity function (all linear functions) mapping $n \in N$ to $n \in [0, \infty)$. We indeed have $f, g \in O(h)$, since, more generally $f \in O(f)$ for any function function f (set m to 0 and c to 1). To see that $(f \cdot g) \notin O(h)$, suppose there were a positive real number c'' and natural number m'' such that for all $n'' \geq m''$, it holds $n'' \cdot n'' = (f \cdot g)(n'') \leq c'' \cdot h(n'') = n''$. This is impossible: taking n'' large enough, in particular setting $n'' = \max(m'', c'') + 1$, we would then have:

   $$(f \cdot g)(n'') =_{\text{def}} f(n'') \cdot g(n'') =_{\text{def}} (\max(m'', c'') + 1) \cdot (\max(m'', c'') + 1) > c'' \cdot (\max(m'', c'') + 1) =_{\text{def}} c'' \cdot h(n'')$$

   Contradiction.

4∗) Take $f(n) = 1$ if n even, and 0 otherwise, and set $g(n) = 1 - f(n)$. If $f \in O(g)$ were the case, there would be a real number c and natural number m such that $f(n) \leq c \cdot g(n)$ for all $n \geq m$, then choosing $n \geq m$ even we would have $f(n) = 1 \not\leq 0 = c \cdot 0 = c \cdot (1 - f(n) = c \cdot g(n)$, contradicting $f \in O(g)$. Symmetrically $g \notin O(f)$. In general, any pair of functions f, g such that always eventually the one 'by far exceeds' the other and vice versa, will do.

5∗)
- Alice and Bob compute n by modular exponentiation as $(r^a)^b \bmod p$ and $(r^b)^a \bmod p$ respectively, using that $(r^a)^b = r^{a \cdot b} = r^{b \cdot a} = (r^b)^a$.
- Given r, p and $r^e \bmod p$ there is no method known to find the exponent e, that is essentially better than enumerating all possibilities. That is, knowing each of $r, p, r^a \bmod p$ and $r^b \bmod p$ does not help to find a and b. (Note we *can* easily compute $r^a \cdot r^b \bmod p$, i.e. the number $r^{a+b} \bmod p$ having the *sum* of the exponents a and b, but for n we would need their *product*.)