1) The following Turing machine starts by going left end marker to right of the word. If the word is already sorted (zeros followed by ones), it only once switches from the starting state s denoting zeros to the ones state o. However, if at the state of looking for ones a zero is encountered, it switches to the swapping state w, and instead of using the state s we use the state c to denote that the word has changed and keep doing the same changes. Reaching the end of input, if the word has not changed (state s or o) we accept, otherwise we return to go back to the start of the word using the state l and start over.

$$M = (\{s, t, r\}, \{0, 1\}, \{\vdash, 0, 1\sqcup\}, \vdash, \sqcup, \delta, s, t, r)$$

	$\vdash$	0	1	$\sqcup$
s	(s,$\vdash$,R)	(s,0,R)	(o,1,R)	(t,$\sqcup$,L)
o		(w,1,L)	(o,1,R)	(t,$\sqcup$,L)
w			(c,0,R)	
c		(w,1,L)	(c,1,R)	(l,$\sqcup$,L)
l	(s,$\vdash$,R)	(l,0,L)	(l,1,L)	

```
bubble_once :: Ord a =>  [a] -> [a]
bubble_once (x:y:xs) =
  if x > y then y:bubble_once (x:xs)
  else x:bubble_once (y:xs)
bubble_once l = l

bubble :: Ord a => [a] -> [a]
bubble l = if l == s then l else bubble s
  where s = bubble_once l
```

```
def bubble(list):
    change = True
    while (change):
        change = False
        for i in range(len(list) - 1):
            if list[i] > list[i + 1]:
                temp = list[i]
                list[i] = list[i + 1]
                list[i + 1] = temp
                change = True
    return list
```

All three programs halt on all inputs and have the same complexity $O(n^2)$. Imperative arrays allow random access, but for the current task this functionality is not useful, so for the task the three data structures are equivalent. If accessing the n-th element would be important, the complexities would differ.

2) a) Informally we can approach the Problem the following way: Similar as it is done using the diagonalization argument we assume, that the web developer is a freelancer and derive

a contradiction: If he develops his own website, he should not develop his own website, which is contradicting. However, if he does not develop his own website, he should develop his own website, which is again contradicting. Thus, our assumption must be false. Hence, the web developer is no freelancer.

A more formal proof would be as follows: Assume $f_1, f_2, f_3, \dots$ be freelancers. Let $d_i(f_j) \in \{\times, \checkmark\}$, indicating if f_1 develops the website for f_j. Clearly, $d_i(f_i)$ indicates whether a freelancer develops its own website or not. We get a table with, for example, the following shape:

	f_1	f_2	f_3	$\dots$		f_w		$\dots$
d_1	$\times$							
d_2		$\checkmark$						
d_3			$\checkmark$					
$\vdots$				$\ddots$				
d_w						$d_w(f_w)$		
$\vdots$								$\ddots$

Assuming the web developer is a freelancer as well we would have a f_w in the sequence above representing the web developer. For the web developer only the diagonal is of importance. Namely, the function d_w would invert the diagonal, i.e., $d_w(f_i) = \overline{d_i(f_i)}$, where

$$\overline{d_i(f_i)} = \begin{cases} \checkmark & \text{, if } d_i(f_i) = \times \\ \times & \text{, if } d_i(f_i) = \checkmark \end{cases}$$

But then $d_w(f_w) = \overline{d_w(f_w)}$, which is a contradiction. Thus, our assumption is false and, hence, the web developer is not a freelancer.

b) First note that there are uncountably many choices for g since there are uncountably many functions $\mathbb{N} \to \mathbb{N}$ and the family $(f_i)_{i \in \mathbb{N}}$ is countable. We define g to be distinct from the f_i on the diagonal by adding one, i.e. defining $g(n) = n^2 + 1$. Then for all n, $g(n) = n^2 + 1 \neq n^2 = f_n(n)$, as desired.

3a) We define $g_0 = g$ as in 2b) and keep on diagonalising away, making g_j distinct from each of $g_0, \dots, g_{j-1}, f_0, f_1, \dots$ by defining $g_j(n) = g_n(n) + 1$ if $n < j$ and otherwise $f_{n-j}(n) + 1$.