Flolac 2008
Functional programming
Assignment 1, Due date: July 2

1. (Simple Recursive functions, 20%) Implement the well-known "*power*" function in two different new ways. The power function takes two arguments n and k and computes n^k. Your implementation only has to work for non-negative k. The following is a straightforward implementation of this function:

```
power :: Int -> Int -> Int
power n k | k < 0 = error "power: negative argument"
power n 0 = 1
power n k = n * power n (k-1)
```

You will implement two more ways in this part.

(a) Use the standard Haskell function "product", which calculates the product (multiplication) of all elements in a list. To calculate "power n k", first construct a list with k elements, all being n, and then use "product". Implement this idea as a Haskell function "`power1`".

(b) There is a different approach to calculating the power function uses less computing steps: to calculate "power n k":

- If k is even, we use $(n^2)^{k/2}$
- If k is odd, we use $n * (n^{k-1})$

Implement this idea as a Haskell function "`power2`". (Hint: Use the standard Haskell functions "even" and/or "odd")

2. (Pattern matching, 30%)

(a) Use pattern-matching with (:) and the wildcard pattern _ to define a function, `myButLast,` that find the last but one element of a list. For examples;

```
myButLast [1,2,3,4]  = 3
myButLast ['a'..'z'] = 'y'
```

(b) Use pattern-matching with (:) to define a function, `rev2,` that reverses all lists of length *2*, but leaves others unchanged. Ensure that your solution works for all lists --- that is, that the patterns you use are exhaustive. For examples:

`rev2 [1, 2] = [2, 1]`, but `rev2 [1, 2, 3] = [1, 2, 3]`.

You may use the standard Haskell function “reverse” in the body of `rev`2, but you should not use the “length” function to determine the length of the input parameter.

3. (Tail recursion, 15%) Write a tail-recursive version of the *fib* function to compute the *nth* number in the Fibonacci sequence.

```
                    fib :: Int -> Int
    fib 0 = 0,  fib 1 = 1, fib 2 = 1, fib 3 = 2,
    fib 4 = 3,  fib 5 = 5, ...
```

You need to define `fib` in terms of an auxiliary function which is tail-recursive and takes two accumulating parameters.

4. (List manipulation)

(a) (15%) A *permutation* of a list is another list with the same elements, but in a possibly different order. For example, `[1,2,1]` is a permutation of `[2,1,1]`, but not of `[1,2,2]`. Write a function

```
isPermutation :: [Int] -> [Int] -> Bool
```

that returns True if its arguments are permutations of each other. Hint: define a function, `removeOnce`, that removes the first occurrence of an element from a list, and use it to implement `isPermutation`.

(b) (20%) Let us use lists to represent sets. Write a function, **subsets**, to generate all subsets of a given set. (Note: you should not use list comprehension)

```
subsets :: [Int] -> [[Int]]
```

For examples:

```
subsets [] = [[]]
subsets [1,2,3] = [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]
```