```haskell
-- RECURSION --
-- Function to determine the maximum of a list.
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "Maximum of empty list."
maximum' [x] = x
maximum' (x:xs)
        | x > maxTail = x
        | otherwise = maxTail
        where maxTail = maximum' xs
        -- Basically takes the head of a list and looks to see if it is bigger than the maximum' of the tail. If it is, then that value is the biggest.
        -- [5,2,6] -> [5], [2,6]. 5 is bigger. -> [2], [6]. 6 is bigger. 6 > 5 > 2 => 6.

-- Maybe a little clearer function to determine the maximum of a list.
maximum'' :: (Ord a) => [a] -> a
maximum'' [] = error "Maximum of an empty list."
maximum'' [x] = x
maximum'' (x:xs) = max x (maximum'' xs)

-- Replicating the replication function. Takes two numbers and returns the second one [n] times.
replicate' :: (Ord i, Num i) => i -> a -> [a] -- Num is not a subclass of Ord, so we have to specify both.
replicate' n x
        | n <= 0 = []
        | otherwise = x:replicate' (n-1) x -- Takes n and subtracts 1.
        -- replicate 3 5 -> 5:replicate 2 5 -> 5:5:replicate 1 5 -> 5:5:5:replicate 0 5 -> 5:5:5:[] -> [5,5,5]

-- Take function replicated in Haskell.
take' :: (Ord i, Num i) => i -> [a] -> [a]
take' n _
        | n <= 0 = []
take' _ [] = []
take' n (x:xs) = x:take' (n-1) xs

-- Reverse function. Take the last element of a lit and put it in the front, second to last then put it in second, etc.
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]

-- Repeat function. Generates an infinite list, so don't call it unless you have something like "take" being used.
repeat' :: a -> [a]
repeat' x = x:repeat' x

-- Zip function to zip up two lists into a tuple.
zip' :: [a] -> [b] -> [(a,b)]
zip' [] _ = []
zip' _ [] = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys

-- Elem function to see if a given element is part of a list.
elem' :: (Eq a) => a -> [a] -> Bool
elem' _ [] = False
elem' a (x:xs)
        | a == x = True
        | otherwise = elem' a xs

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
        let
                smallSort = quicksort [a | a <- xs, a<=x]
                largeSort = quicksort [a | a <- xs, a > x]
        in
                smallSort ++ [x] ++ largeSort


pigLatin :: String -> String
pigLatin [] = ""
pigLatin (x:xs) = reverse (tail (reverse (string ++ [x] ++ "ay " ++ pigLatin (takeFrom ((length string)+1) xs))))
        where string = (takeWhile (/=' ') xs)

takeFrom :: Int -> String -> String
takeFrom _ "" = ""
takeFrom 0 string = string
takeFrom n (x:string) = takeFrom (n-1) string

pal :: String -> String
pal [] = ""
pal string = reverse single ++ " " ++ pal (takeFrom ((length single)+1) string)
        where single = takeWhile (/=' ') string
```