```
*** partial sums ***
psums :: [Integer] -> [Integer]
psums [] = []
psums (x:xs) = x : next x xs

next :: Integer -> [Integer] -> [Integer]
next [] = []
next curr (x:xs) = (curr + x) : next (curr + x) xs

*** partial sums with fold ***
partialFold :: (a -> a -> a) -> [Integer] -> [Integer]
partialFold _ [] = []
partialFold f (x:xs) = x : helper f x xs

helper :: (a -> a -> a) -> Integer -> [Integer] -> [Integer]
helper _ _ [] = []
helper f prev (x:xs) = f prev x : helper f (f prev x) xs

*** binary map ***
binMap :: (a -> a -> a) -> [(a, a)] -> [a]
binMap _ [] = []
binMap f ((p1, p2):xs) = f p1 p2 : binMap(f xs)

binMap :: (a -> a -> a) -> [(a, a)] -> [a]
binMap _ [] = []
binMap f lst = map (\(x, y) -> f x y) lst

*** factorial ***
Factorial in haskell

Factorial :: Integer -> Integer
Factorial z
    | z == 0 = 0
    | otherwise = z * Factorial(z - 1)

*** merge two sorted lists ***
merge :: [a] -> [a] -> [a]
merge [] y = y
merge y [] = y
merge [] [] = []
merge (x:xs) (y:ys)
```

```
    | x < y = x : merge xs y:ys
    | otherwise = y : merge x:xs ys

*** add tuples ***
addTuple :: [ (Integer, Integer) ] -> [Integer]
addTuple [] = []
addTuple ((int1, int2):xs) = (int1 + int2) : addTuple(xs)

*** easy fibonacci ***
easyFibo :: Integer -> Integer
easyFibo 0 = 1
easyFibo 1 = 1
easyFibo n = helper 2 1 1 n

helper :: Integer -> Integer -> Integer -> Integer -> Integer
helper count x y n
    | count == n = x + y
    | otherwise = helper (count + 1) y (x + y) n

*** list of pairs to pairs of lists ***
listToPair :: [(a, b)] -> ([a], [b])
listToPair [] = []
listToPair (p1, p2):xs = ([p1], [p2]) ++ (listToPair xs)

listToPair :: [(a, b)] -> ([a], [b])
listToPair [] = []
listToPair lst = (map (\(x, y) -> x) lst, map (\(x, y) -> y) lst)

*** remove duplicates in a list ***
nub :: (Eq a) => [a] -> [a]
nub [] = []
nub (x:xs)
    | elem x xs = nub xs
    | otherwise = x : nub xs

*** check if a list is in ascending order ***
isAsc :: [Integer] -> Bool
isAsc [] = True
isAsc [x] = True
isAsc (x:y:xs)
    | x <= y = isAsc y:xs
```

```
    | otherwise = False

*** powers of two lazy evaluation ***
powerOfTwo :: [Integer]
powerOfTwo = 1 : map (*2) powerOfTwo

*** lazy sieve ***
sieve :: (Integral a) => [a] -> [a]
sieve (x:xs) = x : sieve (filter (\y -> y mod x /= 0) xs)
primes :: (Integral a) => [a]
primes = sieve (ints 2)

*** quicksort ***
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = [y | y <- xs, y < x] ++ [x] ++ [y | y <- xs, y > x]

*** datatypes ***
data Calculation = Add Int Int | Mul Int Int | Div Int Int
calc :: Calculation -> Integer
calc (Add x y) = x + y
calc (Mul x y) = x * y
calc (Div x y) = div x y

*** stack ***
newStack :: Stack a
newStack = Empty
push :: Stack a -> a -> Stack a
push s e = Stack e s
pop :: Stack a -> (Stack a,a)
pop (Stack e s) = (s,e)
isempty :: Stack a -> Bool
isempty Empty = True
isempty (Stack _ _) = False
```