Task 1:

- **getOdds** – Uses mutual recursion with getEvens to get all elements of a list that are on an odd index. Gets a number and then appends result of getEvens to it. getEvens essentially causes a number to be skipped in this context, meaning every other index is retrieved.
- **getEvens** – Similar to above, retrieves the value skipped by getOdds by grabbing x given by getOdds.
- **sumISBN** – Performs the addition in the ISBN13 check. Takes a list, splits it into odd and even indices, multiplies the even values by 3 and then adds the sums of both together. Finally subtracts the value of the last digit, as this is the check digit itself and isn't included in the sum.
- **isValidISBN13** – Takes a list of integers and returns a Boolean indicating if it is a valid ISBN13 or not. Performs the formula indicated by the worksheet and checks if equivalent to the final (check) digit. Also checks that the list is of length 13, as sumISBN doesn't account for length.

Task 2:

- **numeralSet** – Constant list of strings. Contains the set of possible numerals, ordered by value highest to lowest. Used with subtractAppend to ensure digits are checked in order.
- **charValue** – Returns the decimal value of each character of the numeral set. Used with romanToInt to help determine values as well as priority of values been determined as negative.
- **duplicate –** Helps subtractAppend with generating numeral strings. Repeats an inputted string n times by using replicate to generate repeats and then uses concat to put them all in one string.
- **romanToInt –** Operates based on the fact that if a numeral character of lower value appears before one of higher value, then the character of lower value is a negative value. For example, IV is equivalent to -1+5=4 and XCIX is equivalent to -10+100-1+10=99. This also uses the fact that only one character of lower value can appear before a character of higher value. Therefore, it works with syntactically correct roman numerals. Works by checking if a preceding value is worth more or less than the following value in a list of chars. If it is worth less, the value of the character is subtracted from the running total, otherwise it is added. Recurses through every character until the last which just returns a positive value.
- **subtractAppend –** Performs the operation for intToRoman. Takes an integer and a list of strings (numeralSet) and returns the value of the integer as a roman numeral. Operates by recursing through the set of numerals. The running total is then checked to see how many of that numeral value can fit in it, e.g. for 3100 with numeral value M, 3 Ms fit into the value, which is the return string (MMM). This value is then subtracted from the running total and the operation is performed again with the next element of the numeral set until the running total is 0, indicating the generated string matches the value of the initially inputted number. The number of times a numeral should be repeated is determined by performing integer division on the running total by the value of the element of the set (found by romanToInt). The duplicate function is then used to generate a string which has a recursive call appended to it with the running total reduced and with the tail of numeral set.
- **intToRoman –** Shorthand for applying subtractAppend with the numeralSet constant, returns roman numerals after taking an integer input.

Task 3:

- **Tree –** New datatype to represent trees, is either an empty node (represented by Empty) or a node with a value and two subtrees.
- **endNode –** Function that quickly returns an empty tree with no subtrees.
- **insertIntoTree –** Takes an integer and a tree returning a new tree with that integer inserted into it. The function traverses the tree by comparing its value to the current node, recursing into the left subtree if the value is lower and into the right subtree if higher. This continues until an empty node is found, which then becomes an end node, fulfilling assignments up the stack.
- **flattenTree –** Performs a depth first traversal on a tree, prioritizing the left subtrees. Works by recursing leftwards until end node reached returning a blank, then goes up the stack to the parent value appending it to the list and then appending right values. This repeats until every node is visited.
- **buildTree –** Uses foldr to repeatedly apply insertTree to the result of the previous insertTree operation, recursing through each element in the inputted list of integers to build a binary tree.

- **sortWithTree –** Performs a sort on a list by building a tree and then flattening it.

Task 4 – Testing with wordSearch wordGrid wordList is convenient**:**

- **wordGrid –** Constant list of strings that stores the example word grid for ease of testing.
- **wordList –** Constant list of strings of the words to be searched for in the example.
- **trueOccurs –** Helper function. Checks if a single value of true occurs in a list of Booleans by using foldr to create a long statement with or operators between each value.
- **isIn –** Helper function. Checks if a string is within another string. Finds the length of the string to be found and then compares the string to be found against a string of equal length taken from the other string using take. It then uses a recursive application of the or operator to try this again, but with the tail of the other string. If any occurrences happen, true will be returned. The operation continues until the length of the string to be found is the same length as the string being compared against.
- **Revls –** Helper function, reverses all strings in a list of strings by mapping reverse to the list.
- **wordOccurs –** Takes a word and a list of strings, and checks if the word is a substring of any of the strings in the list. Works by mapping isIn to the list, creating a list of Booleans that trueOccurs is then applied to.
- **wordSearch –** Takes a word grid and a list of words as inputs and then returns a list of the words with messages detailing if they were found and their method of finding. Works by applying wordOccurs to a series of lists generated by other functions that represent every straight line (directions) in the word grid. If a wordOccurs instance returns positive, a message is returned with the word and direction it was found in. The recursion goes through every word in the list provided until there are none left.
- **getDowns –** Transposes a list of strings. Works by taking first elements of all lists and then appending the next elements in the tail to them recursively until the remaining tail is blank. Returns every downward string, revls of this gives every upward string.
- **getDiagonalDR –** Takes a grid and a tuple of integers (coordinates) and returns a list of characters that represents a single diagonal line going downward and rightward. Works by finding the element at those coordinates and then adding 1 to the x and y value and appending the value there until the end of a row or column.
- **getDiagonalUR –** Same as above but going upward and rightward.
- **getAllDR –** Applies getDiagonalDR to all the leftmost and topmost coordinates in a grid, returning a list of strings representing all downward rightward diagonals. Works by mapping getDiagonalDR to 2 lists of tuples (coordinates) created by using zip on 2 lazily evaluated ranges of integers. The zipped lists represent a row and column (topmost and leftmost), evaluated up to the limits of the inputted grid's width and height. The revls of this returns all upwards and leftwards diagonals.
- **getAllUR –** Same as above but applied to bottommost row and leftmost column with getDiagonalUR. The revls of this returns all downwards and leftwards columns.

Code Listing:

```
---- TASK 1 ----

--checks check digit and input length
--returns true if valid
isValidISBN13 :: [Int] -> Bool
isValidISBN13 a = (10-((sumISBN a) `mod` 10)) `mod` 10 == last a
                  && length a == 13

--adds all numbers together, multiplying evens by 3
--subtracts the last number (x13 if correct length)
sumISBN x = let e = getEvens x
                o = getOdds x
                in (sum (map (3*) e) + sum o) - last x

--gets number and then recurses into getEvens
getOdds :: [Int] -> [Int]
getOdds (x:xs) = x:(getEvens xs)
getOdds [] = []

--gets number and then recurses into getOdds
getEvens :: [Int] -> [Int]
getEvens (x:xs) = getOdds xs
getEvens [] = []

---- END TASK 1 ----

----TASK 2----

--stores ordered numerals to be appended after their
--value has been subtracted from the running total
numeralSet = ["M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"]

--determines value of numeral
charValue :: Char -> Int
charValue 'M' = 1000
charValue 'D' = 500
charValue 'C' = 100
charValue 'L' = 50
charValue 'X' = 10
charValue 'V' = 5
charValue 'I' = 1

--returns string repeated n times
duplicate :: String -> Int -> String
duplicate x n = concat (replicate n x)

--assumes syntactically correct roman numerals
--converts by determining if value of a char is negative
--and then summing them together
romanToInt :: [Char] -> Int
romanToInt (x:xs) =
  if xs == []
    then charValue x
    else if charValue x < charValue (head xs)
      then romanToInt xs - charValue x
      else charValue x + romanToInt xs

--converts to numeral by subtracting from a running total based on
--max value represented with one element from numeral set
subtractAppend :: Int -> [String] -> [Char]
subtractAppend 0 numSet= ""
subtractAppend x numSet=
  let repeats = x `div` romanToInt (head numSet) --times the numeral will repeat
```

```
    in duplicate (head numSet) (repeats) --gets numeral string
    ++ subtractAppend (x - repeats * romanToInt (head numSet)) (tail numSet) --
subtracts value of above, recurses

--Converts int to roman using subtract append
intToRoman :: Int -> [Char]
intToRoman x = subtractAppend x numeralSet

---- END TASK 2 ----

---- TASK 3 ----

--defines a tree, either empty node or node with value and 2 subtrees
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)

--returns an end node (node with no children)
endNode :: a -> Tree a
endNode x = Node x Empty Empty

--builds a tree by using foldr to recursively call insert on each new tree
buildTree :: [Int] -> Tree Int
buildTree x = foldr insertIntoTree Empty (reverse x)

--left prioritised depth first traversal
flattenTree :: Tree Int -> [Int]
flattenTree Empty = [] --base case, if empty no value stored so return no value
flattenTree (Node x left right) = flattenTree left ++ [x] ++ flattenTree right

--returns modified tree with item added to it
insertIntoTree :: Ord a => a -> Tree a -> Tree a
insertIntoTree x Empty = endNode x --if current node is empty, place x there
insertIntoTree x (Node currentVal left right) = --non-empty node
  if x < currentVal
    then Node currentVal (insertIntoTree x left) right --if lower insert left
    else Node currentVal left (insertIntoTree x right) --if higher insert right

--sorts list of nums with tree functions, depth first tree
sortWithTree :: [Int] -> [Int]
sortWithTree x = flattenTree (buildTree x)

---- END TASK 3 ----

---- TASK 4 ----

--stores the grid as list of strings
wordGrid = ["IUPGRADEEPEQ",
            "YTDZMTZVNRXS",
            "YVCECTIWALZR",
            "PCPGERSWGCRE",
            "PGLUDVDUCFNS",
            "ONTDJRRWDFOY",
            "LVRGAXAIYFKZ",
            "FAUHBGSXTELI",
            "HEGSPKHWYPOC",
            "TESZEBABIDKY",
            "NZWTUROHOIPK",
            "MXTGEADGAVLU",
            "TESSRMEMORYO",
            "DIQDTROMTKSL",
            "IRCTLAPTOPOX"]

--stores the words for convenience of testing
wordList =["LAPTOP", "KEYBOARD", "BUGS", "DISKETTE", "UPGRADE",
           "MEMORY", "HARDWARE", "FLOPPY", "HARDDRIVE", "SOFTWARE"]
```

```
----HELPER FUNCS----

--checks if true occurs in a list of bools at least once
trueOccurs :: [Bool] -> Bool
trueOccurs = foldr (||) False

--checks if a word occurs as a substring in a list of strings
wordOccurs :: String -> [String] -> Bool
wordOccurs word x = trueOccurs (map (isIn word) x)

--reverses all strings in a list of strings
revls :: [String] -> [String]
revls x = map reverse x

--checks if a substring appears in a list
isIn :: String -> String -> Bool
isIn toFind x =
  if length toFind <= length x
    then toFind == take (length toFind) x || isIn toFind (tail x)
    else False

----END HELPER FUNCS----

--takes grid x and list of words w, finds them and returns method of finding
wordSearch:: [String] -> [String] -> [String]
wordSearch x (w:ws) =
  if wordOccurs w x then (w++" right"):wordSearch x ws
  else if wordOccurs w (revls x) then (w++" left"):wordSearch x ws
  else if wordOccurs w (getDowns x) then (w++" down"):wordSearch x ws
  else if wordOccurs w (revls(getDowns x)) then (w++" up"):wordSearch x ws
  else if wordOccurs w (getAllUR x) then (w++" upright"):wordSearch x ws
  else if wordOccurs w (revls(getAllUR x)) then (w++" downleft"):wordSearch x ws
  else if wordOccurs w (getAllDR x) then (w++" downright"):wordSearch x ws
  else if wordOccurs w (revls(getAllDR x)) then (w++" upleft"):wordSearch x ws
    else (w++" notfound"):wordSearch x ws --if not found say so and carry on
wordSearch x [] = []


-- gets downs, reverse is ups
-- takes a grid and returns the transpose
getDowns :: [String] -> [String]
getDowns ([]:xs) = []
getDowns xs = map head xs : getDowns (map tail xs)

--DownRight diagonals, reverse is upleft
--get coordinates of a char and then adds chars to all below and to the right
getDiagonalDR :: [String] -> (Int,Int) -> [Char]
getDiagonalDR x (col,row) =
  if row <= length (x)-1 && col <= length (x !!0 )-1
    then ([(x !! row) !! col])  ++ getDiagonalDR x (col+1,row+1)
    else ""


--UpRight diagonals, reverse is downleft
--get coordinates of a char and then adds chars to all above and to the right
getDiagonalUR :: [String] -> (Int,Int) -> [Char]
getDiagonalUR x (col,row) =
  if row >= 0 && col <= length (x !!0 )-1
    then ([(x !! row) !! col])  ++ getDiagonalUR x (col+1,row-1)
    else ""

--applies getDiagonalUR to bottom side and left side
getAllUR :: [String] -> [String]
```

```
getAllUR x = let width = (length (x !! 0)) -1
                 height = (length x) -1
             in map (getDiagonalUR x) (zip [0,0..] [0..height]) --leftmost
             ++ map (getDiagonalUR x) (zip [1..width] [height,height..]) --bottom

--applies getDiagonalDR to top side and left side
getAllDR :: [String] -> [String]
getAllDR x = let width = (length (x !! 0)) -1
                 height = (length x) -1
             in map (getDiagonalDR x) (zip [0,0..] [0..height]) --leftmost
             ++ map (getDiagonalDR x) (zip [1..width] [0,0..]) --topmost

---- END TASK 4 ----
```