1 Mergesort

Fill in the blanks.

```haskell
mergesort ______ = ______
mergesort ______ = ______
mergesort xs = ______________________________
    where (left, right) = _____________
mergesorted __________ = xs
mergesorted __________ = ys
mergesorted (x:xs) (y:ys)
    | x<=y    = _____________________________
    | otherwise = _____________________________
split [] = ________________
split [x] = ________________
split (x:y:xs) = ________________
    where (left, right) = ________________
```

Questions:

1. What’s the type of these three functions? I’ve answered the first for you as an example.

   (a) :t mergesort
       Answer: mergesort :: Ord a => [a] -> [a]
   (b) :t mergesorted
       Answer: mergesorted :: ________________________________
   (c) :t split
       Answer: split :: ________________________________

2. What’s the result of split [1..7]?
   Answer: ____________________________________________

3. What about mergesorted (split [1..7])?
   Answer: ____________________________________________

4. Is this merge sort faster or slower than quicksort (no random choice of pivot) in the **worst case**?

5. Is this sort stable?
2 If ... Then ... Else

In last week's lecture we have seen the syntax and semantics of simple boolean expressions:

<table>
<thead>
<tr>
<th>syntax</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::=</td>
<td>if true then t1 else t2 → t1</td>
</tr>
<tr>
<td>true</td>
<td>if false then t1 else t2 → t2</td>
</tr>
<tr>
<td>false</td>
<td>t → t'</td>
</tr>
<tr>
<td>if t then t else t</td>
<td>if t then t1 else t2 → if t' then t1 else t2</td>
</tr>
</tbody>
</table>

Below we will use Haskell's recursive datastructure (recall Ast from HW2) to implement the boolean expression defined above, so that

\[ \text{IFTHENELSE FALSE TRUE (IFTHENELSE TRUE FALSE TRUE)} \]

means “if false then true else (if true then false else true)”. The eval function below implements the one-step evaluation relation “→”, so that eval t returns IFTHENELSE TRUE FALSE TRUE. Fill in the blanks.

```haskell
data Ast = TRUE
    | FALSE
    | IFTHENELSE _________________
deriving (Show)

eval (IFTHENELSE ________________) = t1

eval (IFTHENELSE ________________) = t2

eval (IFTHENELSE t t1 t2) = _________________
```

The evalstar function implements the multi-step evaluation “→*” relation, which is the reflexive, transitive closure of one-step evaluation → (i.e., keep evaluating until you can’t evaluate any more). E.g., calling evalstar t for the t defined above returns FALSE. Fill in the blanks.

```haskell
evalstar TRUE = _________________

evalstar FALSE = _________________

evalstar t = _________________
```

Questions:

1. let t : IFTHENELSE

   Answer: IFTHENELSE :: _________________

2. let t' = IFTHENELSE (IFTHENELSE TRUE FALSE TRUE) TRUE (IFTHENELSE FALSE FALSE TRUE)

   eval t'
   Answer: _________________

   eval (eval t')
   Answer: _________________

   evalstar t'
   Answer: _________________