1. Median [35%]. The median of a group of values is the “middle” value. That is, when the group is sorted in some way, the median is the value for which there are the same number of values before it as after it.

For example, the values \{4, 5, 6, 7, 8, 9, 10\} are sorted in increasing order. In this order, the median is 7, since there are 3 values before 7 and 3 values after.

Note: with even length, there is no “middle value”. In \{3, 4, 5, 6\}, the “middle” is between 4 and 5.

For the purpose of this question, we define **median** of a list to be, when the list is sorted:

- If there are an odd number of values, the median is the value that has the same number of values before it as after it.
- If there are an even number of values, the median is the value that has one more value before it as after it.

So the median of (\textbf{list 4 5 6 3}) is 5, since when sorted the list is (\textbf{list 3 4 5 6}), and here 5 has \{3,4\} before it and \{6\} after it.

Submit your solution for this question in the file **median.rkt**
Ordering function. What “before” and “after” mean depends on how the items are sorted.
We say that a list is sorted by \textit{pred} if every value is “less than” the next, as defined by the predicate \textit{pred}.

For example:

- \((\text{list } 2 \ 3 \ 7 \ 8)\) is sorted by \(\leq\), since \((\leq \ 2 \ 3), (\leq \ 3 \ 7), \text{ and } (\leq \ 7 \ 8)\).
- \((\text{list } 8 \ 7 \ 3 \ 2)\) is sorted by \(\geq\), since \((\geq \ 8 \ 7), (\geq \ 7 \ 3), \text{ and } (\geq \ 3 \ 2)\).
- \((\text{list } #\c \ #\e \ #\g)\) is sorted by \(\text{char}\leq\), since \((\text{char}\leq \ #\c \ #\e)\) and \((\text{char}\leq \ #\e \ #\g)\).

The median will depend on what predicate is used to sort the items.

For example, if the letters in “banana” are sorted by \(\text{char}\leq\), the result is \((\text{list } #\a \ #\a \ #\a \ #\b \ #\n \ #\n)\).
And the median of this list is \#\b.

On the other hand, if these letters are sorted by \(\text{char}\geq\), the result is \((\text{list } #\n \ #\n \ #\b \ #\a \ #\a \ #\a)\).
And the median of this list is \#\a.

Pivoting. In order to find the value we want, we can use an approach similar to \textit{quicksort}, using a \textit{pivot}.
But we don’t need to sort the whole list; we only need to look at a part of it. \textit{We can find the median faster than sorting the whole list.}

Algorithm Outline: At each step, like quicksort, use the first item in the list as a pivot, and create lists containing the values that come before, and the values that come after. Either the value you seek \textit{is} the pivot, or it is in one of these lists. Unlike quicksort, recurse on only one of the lists.

You must recurse on only one of the lists.

For example, consider \((\text{list } 16 \ 9 \ 4 \ 27 \ 1 \ 15 \ 25 \ 2 \ 12 \ 20)\). There are 10 values, so the median will be the value with 5 values before it.

- We are seeking the item in \((\text{list } 16 \ 9 \ 4 \ 27 \ 1 \ 15 \ 25 \ 2 \ 12 \ 20)\) that has 5 items before it. Using 16 as the pivot, the lesser values are \(\{9, 4, 1, 15, 2, 12\}\), and the greater values are \(\{20, 25, 27\}\). So our target is the item in the lesser values that has 5 values before it.

- We are seeking the item in \((\text{list } 9 \ 4 \ 1 \ 15 \ 2 \ 12)\) that has 5 items before it. Using 9 as a pivot, the lesser values are \(\{4, 1, 2\}\), and the greater values are \(\{15, 12\}\). So our target will have both the pivot and these 3 lesser values before it. So our target is the item with 1 value before it in \(\{15, 12\}\).

- We are seeking the item in \((\text{list } 15 \ 12)\) that has 1 item before it. Using 15 as the pivot, the lesser values are \(\{12\}\), and the greater values are \(\{}\). Since our target is item with 1 value before it, and the pivot has 1 value before it, the pivot is our target. So we produce 15.
Write a function `median`. It consumes a `(listof X)`, which is not guaranteed to be sorted, and a predicate to determine if a pair of values is in order: a `(X X -> Bool)`. It produces the **median** of the list, as defined above.

For example,

- `(check-expect (median (list 16 9 4 27 1 15 25 2 12 20) <=) 15)`
- `(check-expect (median (list 15 19 12 21 17) <=) 17)`
- `(check-expect (median (string->list "banana") char<=?) \\
  #\b)`
- `(check-expect (median (string->list "banana") char=>?) #\a)`
- `(check-expect (median (list -7 -4 -2 5 6) (lambda (x y) (<= (abs x) (abs y)))) 5)`

**!** Do not sort the list! Examine the quicksort code, and use the pivot idea described above.

---

### 2. Map Madness [30%]
Submit your solutions for all parts of this question in the file `map-madness.rkt`

We say a **generic binary tree** is anything that has all the following properties:

- It is possible to determine if it is a leaf.
- Unless is is a leaf, it is defined in terms of two children, and we can determine the value of these children.
- It is possible to construct it.

In addition: a generic binary tree **may** have a label to store additional information; if so, the value of this label can be determined.

For example, a `BT`, defined on slide 9, is a generic binary tree: we can tell if it is a leaf using `empty?`; a non-leaf `Node` is a defined in terms of 2 children, and we can determine these values using `node-left` and `node-right`; this value can be constructed using `make-node`. In addition, it has a label, and we can determine the value of this label using `node-key`.

For another example, I define:

```racket
;; a XmasTree is one of:
;; 'needle
;; 'ornament
;; (list (anyof 'red 'green 'white) XmasTree XmasTree)
```

There are 2 kinds of leaves, but we can tell if it a leaf using `symbol?`; a non-empty `XmasTree` is defined in terms of 2 children, and we can determine these values using `second` and `third`; this value can be constructed, using `list`. In addition, it has a label, and we can determine the value of this label using `first`.

2.1. **Mapping a tree.** Recall that `map` transforms each value in a list, while keeping the length of the list unchanged. We can transform the keys and leaves of a tree, while keeping the shape the same. For example, for a `BT`, we could add 1 to each key.
Another example, with a XmasTree, we could transform each leaf by turning every "needle" into an "ornament", and also transform each key by turn "white" into "red" and vice-versa, leaving "green" unchanged.

<table>
<thead>
<tr>
<th>Exercise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write a function <strong>bt-map</strong>. Its parameters are as follows:</td>
</tr>
<tr>
<td>1. <strong>tree</strong>, a generic binary tree that we here call <strong>T</strong>.</td>
</tr>
<tr>
<td>2. <strong>key-xform</strong>, a function to transform a key to another key: a <strong>(K -&gt; K)</strong>.</td>
</tr>
<tr>
<td>3. <strong>leaf-xform</strong>, a function to transform a leaf to another leaf: a <strong>(L -&gt; L)</strong>.</td>
</tr>
<tr>
<td>4. <strong>make-s</strong>, a function to construct a generic binary tree, given a key and 2 children: a <strong>(K T T -&gt; T)</strong>.</td>
</tr>
<tr>
<td>5. <strong>t-key</strong>, a function to extract the key from a <strong>T</strong> that is not a leaf: a <strong>(T -&gt; K)</strong>.</td>
</tr>
<tr>
<td>6. <strong>t-left</strong>, a function to extract the left child: a <strong>(T -&gt; T)</strong>.</td>
</tr>
<tr>
<td>7. <strong>t-right</strong>, a function to extract the right child: a <strong>(T -&gt; T)</strong>.</td>
</tr>
<tr>
<td>8. <strong>leaf?</strong>, a function to determine if a value is a leaf: a <strong>(T -&gt; Bool)</strong>.</td>
</tr>
</tbody>
</table>

It shall consume any such generic binary tree, transform each key with **key-xform**, transform each leaf with **leaf-xform**, and produce a generic binary tree of the same shape.

For example, here is a function that consumes a **BT**, transforming each key with **add1**, leaving each leaf unchanged:

```scheme
;; bt-increment: BT -> BT
(define (bt-increment tree)
  (bt-map tree
    add1
    (lambda (x) x)
    make-node
    node-key
    node-left
    node-right
    empty?))
```

And here is an example of how it behaves:

```scheme```
(check-expect
 (bt-increment
  (make-node 3 (make-node 7 empty (make-node 8 empty empty)) (make-node 2 empty empty)))
 (make-node 4 (make-node 8 empty (make-node 9 empty empty)) (make-node 3 empty empty)))
```

That is:

```
     3
   /  \
  7   4
 /   /  \
2   8   3
 /       /  \
8       9
```

CS 135 Winter 2021 Assignment 10
And here is a function that consumes a XmasTree, swaps 'red with 'white, and turns every leaf into 'ornament:

;; tree-redecorate: XmasTree -> XmasTree
(define (tree-redecorate tree)
  (bt-map tree
    (lambda (s) (cond [(symbol=? s 'red) 'white][[(symbol=? s 'white) 'red][else s]])
    (lambda (l) 'ornament)
    list
    first
    second
    third
    symbol?))

And here is an example of how it behaves:

(check-expect
  (tree-redecorate
    (list 'red
      (list 'white 'needle 'needle)
      (list 'green 'ornament 'needle)))
  (list 'white
    (list 'red 'ornament 'ornament)
    (list 'green 'ornament 'ornament)))

2.2. Folding a tree. Recall that foldr and foldl takes a list, transform it into a single value. We can do the same with binary trees.
Write a function \texttt{bt-fold}. Its parameters are as follows:

1. \texttt{tree}, a generic binary tree that we here call \(T\).
2. \texttt{node-f}, a function that consumes the key of a non-leaf and the results produced by calls on the two children, and produces a new result: a \((K R R \rightarrow R)\).
3. \texttt{leaf-f}, a function that consumes a leaf, and produces a new result: a \((L \rightarrow R)\).
4. \texttt{t-key}, a function to extract the key from a \(T\) that is not a leaf: a \((T \rightarrow K)\).
5. \texttt{t-left}, a function to extract the left child: a \((T \rightarrow T)\).
6. \texttt{t-right}, a function to extract the right child: a \((T \rightarrow T)\).
7. \texttt{leaf?}, a function to determine if a value is a leaf: a \((T \rightarrow \text{Bool})\).

It combines the values in the tree into a single value.

For example, here is a function that counts how many times ‘empty’ appears in a \(\text{BT}\). (That is, it counts the leaves.)

\[
;\ (\text{count-empties} \ tree) \ \text{Count the leaves in tree.}
;\ \text{count-empties}: \ \text{BT} \rightarrow \ \text{Nat}
\]

\[
\text{define} \ (\text{count-empties} \ tree) \\
(\text{bt-fold} \ tree \\
 (\lambda (k l r) (+ l r)) \\
 (\lambda (x) 1) \\
 \text{node-key} \\
 \text{node-left} \\
 \text{node-right} \\
 (\lambda (x) (\text{not} \ (\text{node?} \ x)))))
\]

And here are some examples of how it works:

\[
(\text{check-expect} \ (\text{count-empties} \ \text{empty}) \ 1) \\
(\text{check-expect} \ (\text{count-empties} \ (\text{make-node} \ 3 \ \text{empty} \ \text{empty})) \ 2) \\
(\text{check-expect} \ \\
 (\text{count-empties} \\
 (\text{make-node} \ 3 \ (\text{make-node} \ 7 \ \text{empty} \ \text{empty}) \ (\text{make-node} \ 2 \ \text{empty} \ \text{empty}))) \\
4)
\]

---

3. Mastermind [35%]. Recall the game Mastermind, described in assignment 07. Previously you wrote code to \textit{score} guesses. Now you will write a function to \textit{make} guesses.

Submit your solutions for all parts of this question in the file \texttt{mastermind.rkt}

To help you build intuition on the puzzle, here is an open-source online implementation of the game:

https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/guess.html

\[
\ldots\ldots
\]

\textit{Automatic Guessing}. Recall that you wrote the function \((\text{mastermind-peg} \ \text{guess} \ \text{correct})\), that scores \textit{guess}, given that \textit{correct} is the correct answer.
We will “wrap up” `mastermind-peg` so a puzzle is a function that consumes only a guess, and produces the scoring for that guess.

```racket
;; A MastermindPuzzle is a ((list Nat Nat Nat Nat) -> (list Nat Nat))
;; Where the function grades the guesses,
;; producing a list counting the black and white pegs.
```

This can be accomplished easily using `lambda`. We create a puzzle where the solution is `list 2 4 6 1`:

```racket
(define example-puzzle
 (lambda (guess) (mastermind-peg guess (list 2 4 6 1))))
```

Now `example-puzzle` is a one-parameter function similar to `mastermind-peg`, but it consumes only a guess:

```racket
(check-expect (example-puzzle (list 1 4 3 2)) (list 1 2))
(check-expect (example-puzzle (list 5 5 5 5)) (list 0 0))
```

In order to make this easier, you are provided with the file `mastermind-maker.rkt`. Download this file, and put it in the same directory as your solution. (Do not submit `mastermind-maker.rkt`.)

At the top of the `mastermind.rkt` file that you created, write the following:

```racket
(require "mastermind-maker.rkt")
```

This tells DrRacket to read the module. Now you have a new function: `make-puzzle`. It behaves as above:

```racket
(define puzzle-a (make-puzzle (list 2 4 6 1)))
(check-expect (puzzle-a (list 1 4 3 2)) (list 1 2))
(check-expect (puzzle-a (list 5 5 5 5)) (list 0 0))
```

Write a function `solve-puzzle`. It consumes a `MasterMindPuzzle`, and produces the `list Nat Nat Nat Nat` that was used to create the `MasterMindPuzzle`.

For example,

```racket
(define puzzle-a (make-puzzle (list 2 4 6 1)))
(check-expect (solve-puzzle puzzle-a) (list 2 4 6 1))

(define puzzle-b (make-puzzle (list 5 5 6 6)))
(check-expect (solve-puzzle puzzle-b) (list 5 5 6 6))
```

You will need to have the computer make guesses, then check them by calling the `MasterMindPuzzle`.

- For full marks, your solution must make no more than 10 guesses.
- A solution that uses over 50 guesses will not receive any correctness marks.

There are many strategies that you could use to solve this problem. You could experiment using the online version mentioned above. Many strategies can solve the puzzle in at most 10 guesses.

Here is one approach:

- Start with a list of all solutions, that is, a list which contains all the possible permutations of size 4 made out of the numbers from 1 to 6.

- Make a guess. Then remove from your list all solutions that are incompatible with the result you get from the puzzle. Repeat until there is only one possible solution in your list.

For example, if you make the guess `(list 1 1 2 2)`, and get back one black peg and zero white pegs, you
know that the solution must contain exactly one 1, or exactly one 2. So can be sure that neither of (list 3 4 5 6) and (list 1 1 1 1) are the solution. (If (list 3 4 5 6) was the solution, (list 1 1 2 2) would score zero black. If (list 1 1 1 1) was the solution, (list 1 1 2 2) would score two black.)

The `make-puzzle` function that we provided is special. It is written so it counts how many times you call it. That is, it counts how many guesses your program makes.

To see how many times a puzzle has been called, call it with the argument 'get-count. Like so:

```scheme
(define puzzle-a (make-puzzle (list 2 4 6 1)))
(check-expect (puzzle-a 'get-count) 0) ; (Just created, so count is zero.)
(check-expect (puzzle-a (list 1 4 3 2)) (list 1 2))
(check-expect (puzzle-a (list 5 5 5 5)) (list 0 0))
(check-expect (puzzle-a 'get-count) 2) ; (We made 2 guesses already.)
```

Here is a test that solves the puzzle, and also checks that you didn’t use too many guesses:

```scheme
(define puzzle-b (make-puzzle (list 5 5 6 6)))
(check-expect (solve-puzzle puzzle-b) (list 5 5 6 6))
(check-expect (<= (puzzle-b 'get-count) 10) true)
```

If you’re curious, you may take a look at how `make-puzzle` counts how many times it is called. You are not expected to understand or work with such code. It is not written in the functional paradigm that we use in this course.

**Bonus [2%].** This is a bonus. Completion is completely optional. It may not be closely related to the material we have been studying recently.

We will not answer *any* questions about the bonus.

Write `solve-puzzle` in such a way that it can solve any puzzle using at most 5 guesses.

*There is nothing separate to submit for the bonus. We will test if your solution to `solve-puzzle` meets this criterion.*