Homework 2

This homework covers the following topics from class:

- Functional Programming, part 1
    - Functional Programming Intro, Functions, Operators, Precedence, Control Flow, Bindings (Let and Where), Tuples, Lists (part 1)
- Functional Programming, part 2
    - Lists (part 2), List Comprehensions, Pattern Matching

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

For every homework, you must turn in a **PDF file** with your answers via Gradescope. You may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work. As a reminder, homework is graded on effort, not correctness – please try every question!

Also, for all Haskell functions, please make sure to include a type signature at the top of the function!

For homework #2, you must complete the following **required** problems:

- HASK1: Simple Functions (2 min)
- HASK2: Recursion, Precedence, Guards (4 min)
- HASK3: Recursion, Guards (4 min)
- HASK4: Where, Tuples (5 min)
- HASK5: Recursion, Where, Helper Functions (5 min)
- HASK6: Simple List Processing, Recursion (5 min)
- HASK7: List Comprehensions (5 min)
- HASK8: Recursion, Pattern Matching (10 min)
- HASK9: Recursion, Lists (20 min)

For homework #2, you may optionally complete the following additional problems:

N/A

HASK1: Simple Functions (2 min)

(2 min.) Write a Haskell function named `largest` that takes in 2 `String` arguments and returns the longer of the two. If they are the same length, return the first argument.

Example:

`largest "cat" "banana"` should return `"banana"`.

`largest "Carey" "rocks"` should return `"Carey"`.

Your answer:

HASK2: Recursion, Precedence, Guards (4 min)

(4 min.) Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named `reflect` that takes in an `Integer` and returns that same `Integer`, but he wrote it in a very funny way:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect num+1
  | num > 0 = 1 + reflect num-1
```

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry's code (i.e. can you fix it so that it works properly)?

Your answer:

HASK3: Recursion, Guards (4 min)

(4 min.) Write a pair of Haskell functions named `is_odd` and `is_even` that each take in 1 `Integer` argument and return a `Bool` indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.

You may not use any builtin arithmetic, bitwise or comparison operators (including `mod`, `rem` and `div`). You may only use the addition and subtraction operators (+ and -) and the equality operator (==).

You must implement THREE versions of these functions: (1) with regular if statements, (2) using guards, and (3) using pattern matching.

Example:
`is_even 8` should return `True`.
`is_odd 8` should return `False`.

Hint: The functions can call one another in their implementations. (This is called mutual recursion).

Your answer:

HASK4: Where, Tuples (5 min)

(5 min.) Write a Haskell function called *quad* that finds the roots of a quadratic equation with coefficients a, b and c. Your solution must take three Double parameters and return a tuple containing the positive and negative roots. If the coefficient a is zero (resulting in division by zero) or the roots would be imaginary then the function must return the tuple (0, 0). **Your solution must have a type signature and use the *where* clause.**

Here's how it would be used:

```
*Main> quad 1 (-5) 6
(3.0,2.0)
*Main> quad 10 1 2
(0.0, 0.0)
*Main> quad 1 10 0
(0.0, -10.0)
```

Your answer:

HASK5: Recursion, Where, Helper Functions (5 min)

(5 min.) In this problem you're going to write a function called *sum_is_divisible* that determines whether the sum of integers in the range [a,b] inclusive is evenly divisible by a third integer c, and returns True if so and False otherwise. Your function may assume that a <= b. You must use recursion in a useful way. Here's how it might be used:

```
Main*> sum_is_divisible 3 5 6
True
Main*> sum_is_divisible 1 3 5
False
```

Your top-level function must have a type signature.

Hints:

1. You will need to use a nested function within a let or where clause to solve this problem!
2. Haskell's modulo function is called `mod`. The infix version looks like this: *6 `mod` 3*. The prefix version look like this: *mod 6 3*

Your answer:

HASK6: Simple List Processing, Recursion (5 min)

(5 min.) In this problem we'll learn how to process items in a list, using Haskell's length, head and tail functions, and a little bit of recursion. Don't worry if we haven't covered Haskell lists in class yet… they're just like python's lists!

Here's how we define a simple list of integers in Haskell:

```
nums = [3, -2, 1, 4]
```

Given this, you are to write a function, called *find_min,* that finds the minimum value in a list of integers. Your function must have the following type signature:

```
find_min :: [Int] -> Int
```

which indicates that this function's first parameter is a *list of integers*, and it returns an *integer* result.

You will find the following Haskell functions useful:
length x: Returns the number of values in list x, so *length nums* would return 4
head x: Returns the first value in the list x, so *head nums* would return 3
tail x: returns a new list containing all but the first value in a list x, so *tail nums* would return [-2, 1, 4]

Write the *find_min* function.

Your answer:

HASK7: List Comprehensions (5 min)

Part A: (2 min.) Write a Haskell function named `all_factors` that takes in an `Integer` argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function's implementation should be a single, one-line list comprehension.**

Example:

`all_factors 1` should return `[1]`.
`all_factors 42` should return `[1, 2, 3, 6, 7, 14, 21, 42]`.

Your answer:

Part B: (3 min.) A perfect number is defined as a positive integer that is equal to the sum of its proper divisors (where "proper divisors" refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.

Using the `all_factors` function, write a Haskell expression named `perfect_numbers` whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:
`take 4 perfect_numbers` should return `[6, 28, 496, 8128]`.

Hint: You may find the init and sum functions useful.

Your answer:

HASK8: Recursion, Pattern Matching (10 min)

(10 min.) Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length `n` in 1 way, even if `n` is 0. Make sure to use pattern matching in your solution.

Examples:

`count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.

`count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.

`count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return 0.

`count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.

`count_occurrences [] [10, 50, 40, 20, 50, 40, 30]`
should return 1.

`count_occurrences [] []` should return 1.

`count_occurrences [5] []` should return 0.

Your answer:

HASK9: Recursion, Lists (20 min)

Part A: (10 min.) Write a Haskell function named `fibonacci` that takes in an `Int` argument `n`. It should return the first `n` numbers of the Fibonacci sequence (for this problem, we'll say that the first two numbers of the sequence are 1, 1).

Examples:

`fibonacci 10` should return `[1,1,2,3,5,8,13,21,34,55]`.

`fibonacci -1` should return `[]`.

Hint: You may find it easier to build the list in reverse in a right-to-left manner, then use the reverse function.

Your answer:

Part B: (10 min.) In this problem, you'll be writing a function in Haskell that run-length-encodes a List of integers. Run-length encoding (RLE) is a data compression technique that encodes consecutive repeated values in a sequence by replacing them with a single value and a count of its repetitions. For example, given the sequence `1, 1, 1, 5, 5, 3, 1, 1`, RLE would compress it by generating a list of tuples of the form `(value, count)`. The first three `1`s are encoded as `(1, 3)`, followed by the two `5`s as `(5, 2)`, and so on. This results in a more compact representation of the original data.

For example:

rle [1, 1, 1, 5, 5, 3, 1, 1] should return [(1, 3), (5, 2), (3, 1), (1, 2)].
rle [4, 4, 4, 4, 4] should return [(4, 5)].
rle [7, 8, 9]` should return [(7, 1), (8, 1), (9, 1)].
rle [] should return [].

Requirements:
1. The function must be implemented recursively and avoid using imported functions like `group` or `span`.
2. The function must handle cases where the input list is empty and return an empty list in such scenarios.
3. The function must correctly count and group consecutive occurrences only.

Your answer: