Discussion Worksheet - Week 1

Topics: Python Refresher, Introduction to Haskell

1. Write a **Haskell** expression called `pythagorean` whose value is a list comprehension that contains tuples of the form `(Int, Int, Int)` where for each tuple `(x, y, z)`, `x^2 + y^2 == z^2`. Values in the tuple must be in increasing order (i.e. The list should include `(3, 4, 5)` as opposed to `(4, 3, 5)`). Include a type definition for `pythagorean`.

Example:
`take 3 pythagorean` should return `[(3,4,5),(6,8,10),(5,12,13)]`

---

2. Write a **Python** function called flattenList to flatten a nested list using recursion.

For Example:

```
Input: [[8, 9], [10, 11, ‘geeks’], [13]]
Output: [8, 9, 10, 11, ‘geeks’, 13]

Input: [[‘A’, [ ‘B’, ‘C’]], [‘D’, ‘E’, ‘F’]]
Output: [‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’]
```

---

3. Given a list of integers, write a **Python** program to find its mode. Assume the list has a unique mode.

For Example:

```
Input: [1, 0, 1, 2, 3, -1, 5, 1, 2, 1]
Output: 1
```

---

4. **Trees**

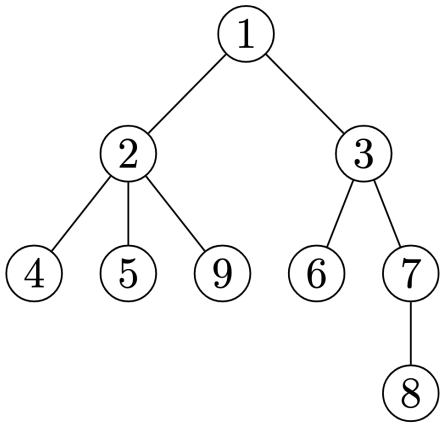
Consider the following **Python** class:

```
class Node:
    def __init__(self, value: int, children=[]):
        self.value = value
        self.children = children
```

We can use this class to create a tree structure as follows:

```
# create an example tree
root = Node(1, [
    Node(2, [
        Node(4),
        Node(5),
        Node(9)
    ]),
    Node(3, [
        Node(6),
        Node(7, [
            Node(8)
        ])
    ])
])
```

Here's a visualization:



(a) Write a function that takes a tree as input and returns its largest value.

For Example:

```
Input: root (as defined above)
Output: 9
```

(b) Let the height of a tree be defined as the number of nodes in the longest path between the root node and any leaf node in the tree (the height of the empty tree = 0). Write a function that takes a tree as input and returns the height of the tree.

For Example:

```
Input: root (as defined above)
Output: 4
```

(the example has height 4 because the longest path between root and leaf is [1], [3], [7], [8], which is a path containing 4 nodes)

---

5. For a given integer, the digit distance is the sum of the absolute differences between consecutive digits.

For example:

```
Input: 6
Output: 0

Input: 61
Output: 5
Reason: |6-1| = 5

Input : 71253
Output: 12
Reason: |7-1|+|1-2|+|2-5|+|5-3| = 6+1+3+2 = 12
```

Write a **Python** function `digit_distance` that determines the digit distance of a given positive integer, using recursion.

---

6. Suppose we have a **Haskell** function `digitToInt` that takes in a character and returns the integer value of the character.

Write a **Haskell** function `numberToInt` that uses `digitToInt`, and recursion to return the integer value of the string. (Assume all inputs are valid integers.)

Hint: Depending on your implementation, the reverse function might also come in handy, which takes in a list and returns a list with the same elements but in reverse order. To test your implementation, add `import Data.Char(digitToInt)` to the top of the file.

For example:

```
Input: “123”
Output: 123

Input: “0”
Output: 0

Input: “-27”
Output: -27
```

---

7. **Haskell Quirks**

In class a lot of people like to ask Carey, "Can you do *X* in Haskell?" Typically, his answer is "Go try it!" So let's do just that. If you haven't yet, install `ghci` and open it up. Try to predict what each of the following bits of code will do, and then run them. Remember that you can load a source file into `ghci` using `:load`.

Note: If your interpreter ever hangs, you can manually escape using Ctrl+C.

(a)

```
infinity = [1..]
two_infinities = infinity ++ infinity
```

(b)

```
hmm = (1, "2", 3, "4")
hmmmmmm = [1, "2", 3, "4"]
```

(c)

```
hrm = ((1,), (2, 3), (5, 6))
hrmmmmm = [(1,), (2, 3), (5, 6)]
```

(d)

```
hectillion :: Integer = 10 ^ (3 * (10 ^ 300) + 3)
hectillion = hectillion + 1
```

(e)

```
yo = fst ("doh", "rey", "mi")
```

(f)

```
likes :: Int = 100
comments :: Int = 2
ratioo = likes / comments
```

---

8. Describe what the following Haskell list comprehension does and determine what list it generates

```
[(x,y) | x <- [1..15], y <- [1..15], x * y == 15]
```

---

9. Implement the following function using Python in a recursive fashion:

```
def isSolvable(a, b, c):
```

This function should return true if there exists nonnegative integers *x* and *y* such that the equation $ax + by = c$ holds true. It should return false otherwise.

Ex: `isSolvable(7, 5, 45) == true //x == 5 and y == 2`
Ex: `isSolvable(1, 3, 40) == true //x == 40 and y == 0`
Ex: `isSolvable(9, 23, 112) == false`

---

10. Are the following Haskell expressions/lists definitions valid? If so, what do they evaluate to? If not, what is the problem?

(a) `numbers = [1..3] ++ [1,9..50] ++ []`

(b) `names = "Alice" ++ ["B", "o", "b"]`

(c) `words = 'F' : ('u' : 'n' : "ction") ++ ['A', 'D'..'T']`

(d) `classnumber = "CS" ++ [1, 3..] ++ [1..]`

(e) `take 3 (2 : 3 : 4 : 5 : 6)`

Challenge. **Side Effects**

In class we learned that the basis for all functional programming languages, Lambda Calculus, is computationally equivalent to a Turing Machine. That is, everything that can be computed by a Turing Machine can also be computed via Lambda Calculus. We also learned that in Lambda Calculus, all functions are *pure*, meaning they have no side effects.

One student brought up an interesting point: if functional programming languages such as Haskell do not have side effects, how can we translate a program such as the following (written in C++) to Haskell?

```
int global = 0;

int func1(int arg) {
 return arg + global++; // SIDE EFFECT!
}

int func2(int arg) {
  return arg * global;
}
```

```
int func(int arg) {
  int y = func1(arg);
  int z = func2(arg); // ORDER OF EXECUTION MATTERS!

  if (z == 7)
    return 0;
  else return y;
}
```

Hint: Is there a way to encapsulate the state of the program within a function?