Lab 3

Exercise 1: Consider the *Trionacci* sequence defined as follows.

$$T_i = \begin{cases} 1 & \text{if } i = 0 \text{ or } i = 1 \text{ or } i = 2 \\ T_{i-1} + T_{i-2} + T_{i-3} & \text{otherwise} \end{cases}$$

Implement a function `trionacci`(n) which returns the nth Trionacci number.

Exercise 2: The *factorial* of n is $n! = 1 \cdot 2 \cdot \ldots \cdot n$ (we define $0! = 1$). Implement `factorial`(n) in two ways: one using a `while` loop, and the other using recursion.

Exercise 3: Last lab we had the following exercise:

An integer is said to be a *palindrome* if its digits are the same forward and backwards (not including leading zeroes). For example, 12321 is a palindrome, as is 5. 1231 on the other hand is not a palindrome, and neither is 50 (remember we are not including leading zeroes). Write a function `isPalindrome`(n) which returns `True` if n is a palindrome and `False` otherwise.

In today's lab, implement `isPalindrome` using recursion. Specifically, check if the first and last characters are equal, and recurse on the middle substring if required.

Exercise 4: Define a function `flooredSquareRoot`(n) which takes a positive `int` or `long` n and computes its square root, rounded down to the nearest integer. Python has a buit-in `sqrt` function which could be helpful here, but don't use it.

Do two implementations. In the first, use a while loop starting from 0 and going upward. Call that function `slowFlooredSquareRoot`(n). Next, implement `flooredSquareRoot`(n) using binary search. Experiment by evaluating these functions on various inputs. Try n being a billion — notice a difference in the time it takes to compute the answer?

Exercise 5: Implement a function `calcNthSmallest`(n, intervals) which takes as input a non-negative `inT` n, and a list of intervals $[[a_1, b_1], \ldots, [a_m, b_m]]$ and calculates the nth smallest number (0-indexed) when taking the union of all the intervals with repetition. For example, if the intervals were $[1, 5], [2, 4], [7, 9]$, their union with repetition would be $\{1, 2, 2, 3, 3, 4, 4, 5, 7, 8, 9\}$ (note $2, 3, 4$ each appear twice since they're in both the intervals $[1, 5]$ and $[2, 4]$). For this list of intervals, the 0th smallest number would be 1, and the 3rd and 4th smallest would both be 3.

Your implementation should run quickly even when the a_i, b_i can be very large (like, one trillion), and there are several intervals.