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.

Example solution:

```
def trionacci(n):
    if n<3:
        return 1
    else:
        return trionacci(n-1) + trionacci(n-2) + trionacci(n-3)
```

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.

Example solution:

```
# using a while loop
def factorial(n):
    ans = 1
    x = 1
    while x <= n:
        ans *= x
        x += 1
    return ans

# using recursion
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

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.

Example solution:

```
def isPalindrome(s):
    if len(s) < 2:
        return True
    return s[0]==s[len(s)-1] and isPalindrome(s[1:len(s)-1])
```

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?

Example solution:

```
def slowFlooredSquareRoot(n):
    x = 0
    while x*x <= n:
        x += 1
    return x-1

# we assume the floored square root is in the interval [a,b]
def flooredSquareRootHelper(a, b, n):
    mid = (a+b+1)/2
    if a==b:
        return a
    elif mid*mid == n:
        return mid
    elif mid*mid < n:
        return flooredSquareRootHelper(mid, b, n)
    else:
        return flooredSquareRootHelper(a, mid-1, n)

def flooredSquareRoot(n):
    return flooredSquareRootHelper(0, n, n)
```

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.

Example solution: First, here are some helper functions which will be useful.

```
# compute the index of the first time x appears in the union of intervals
def firstTime(x, intervals):
    answer = 0
    for L in intervals:
        if x > L[1]:
            answer += L[1] - L[0] + 1
        elif x > L[0]:
            answer += x - L[0]
    return answer

# compute the index of the last time x appears in the union of intervals
def lastTime(x, intervals):
    answer = 0
    for L in intervals:
        if x >= L[1]:
            answer += L[1] - L[0] + 1
        elif x >= L[0]:
            answer += x - L[0] + 1
    return answer-1
```

Now, here is a slow implementation of `calcNthSmallest`(n, intervals) (at least, it is slow when the intervals can be very long).

```
def calcNthSmallest(n, intervals):
    for L in intervals:
        for x in xrange(L[0], L[1]+1):
            first = firstTime(x, intervals)
            last = lastTime(x, intervals)
            if first<=n and n<=last:
                return x
```

The reason it is slow for long intervals is that we loop over the entire range from L[0] to L[1]+1. To make this faster, we can use a binary search over the interval [L[0], L[1]].

```
# Binary searches for the nth smallest number being in the interval
# [a,b]. If no such number in [a,b] is found, [False, ‘’] is returned.
# Otherwise, [True, x] is returned, where x is the nth smallest
# number.
def binarySearch(a, b, n, intervals):
    if a>b:
        return [False, ‘’]
    mid = (a+b)/2
    first = firstTime(mid, intervals)
    last = lastTime(mid, intervals)
```

```
    if first<=n and n<=last:
        return [True, mid]
    elif first>n:
        return binarySearch(a, mid-1, n, intervals)
    else:
        return binarySearch(mid+1, b, n, intervals)

def calcNthSmallest(n, intervals):
    # The answer has to be in one of the intervals, so try them all in
      a for loop.
    for L in intervals:
        answer = binarySearch(L[0], L[1], n, intervals)
        if answer[0]:
            return answer[1]
```