Lab 13

Exercise 1: Implement Karatsuba's algorithm in Python.

Example solution: It's a bit easier to implement if you work with a string of digits or list of digits, so that you can easily append the necessary amount of zeroes at various points, and also easily take the first and second halves of digits.

```
# a and b here are the numbers to be multiplied, given as strings
def recurse(a, b):
    # pad a and b with 0s in the beginning so they are the same length
    if len(a) > len(b):
        b = '0'*(len(a)-len(b)) + b
    if len(b) > len(a):
        a = '0'*(len(b)-len(a)) + a
    n = len(a)

    if n == 1:
        return str(int(a)*int(b))

    # the code is easier if n is even
    if n%2 == 1:
        n += 1
        a = '0' + a
        b = '0' + b

    # ah is the first half of digits of a, and al is the second half,
    # and similarly for bh and bl
    ah = a[:n/2]
    al = a[n/2:]
    bh = b[:n/2]
    bl = b[n/2:]

    # do the three recursions of karatsuba's algorithm
    x = recurse(ah, bh)
    y = recurse(al, bl)
    z = recurse(str(int(ah)+int(al)), str(int(bh)+int(bl)))

    # substract off the extra stuff we don't need from z
    z = str(int(z) - int(x) - int(y))

    x += '0'*n
    z += '0'*(n/2)
    return str(int(x)+int(y)+int(z))
```

```
def karatsuba(a, b):
    return int(recurse(str(a), str(b)))
```

Exercise 2: Implement a function `fibonacci`(n) for computing the nth Fibonacci number, using repeated squaring.

Example solution:

```
def matrixMultiply(A, B):
    n = len(A)
    m = len(A[0])
    p = len(B[0])
    C = []
    for i in xrange(n):
        C += [[0]*p]
    for i in xrange(n):
        for j in xrange(p):
            for k in xrange(m):
                C[i][j] += A[i][k]*B[k][j]
    return C

def matrixPower(A, n):
    if n==0:
        # return the identity matrix I, which has all 1s on the diagonal
        # and 0s everywhere else.  I*T = T for any matrix T
        I = []
        for i in xrange(len(A)):
            I += [[0]*len(A)]
        for i in xrange(len(A)):
            I[i][i] = 1
        return I
    B = matrixPower(A, n/2)
    B = matrixMultiply(B, B)
    if n%2 == 1:
        B = matrixMultiply(B, A)
    return B

def fibonacci(n):
    A = [[1,1],[1,0]]
    B = matrixMultiply(matrixPower(A, n), [[1],[1]])
    return B[1][0]
```

Exercise 3: Recall the *Trionacci* sequence defined in lab 3:

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

Implement a function `trionacci`(n) which returns the nth Trionacci number. Your function should only require $O(\log_2 n)$ integer multiplications.

Example solution: The solution is essentially the same as for Fibonacci, except that our matrix is different. We use the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Then

$$A \cdot \begin{bmatrix} T_i \\ T_{i-1} \\ T_{i-2} \end{bmatrix} = \begin{bmatrix} T_i + T_{i-1} + T_{i-2} \\ T_i \\ T_{i-1} \end{bmatrix} = \begin{bmatrix} T_{i+1} \\ T_i \\ T_{i-1} \end{bmatrix},$$

so what we want is the last entry in the product

$$A^n \cdot \begin{bmatrix} T_2 \\ T_1 \\ T_0 \end{bmatrix} = A^n \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

```
def trionacci(n):
    A = [[1,1,1],[1,0,0],[0,1,0]]
    B = matrixMultiply(matrixPower(A, n), [[1],[1],[1]])
    return B[2][0]
```