# All from Chapter 3 of "CS for All" at Harvey Mudd College.

import sys
sys.setrecursionlimit(20000)  # allow functions to recurse 20,000 times.

# Cryptography and Prime Numbers

# We want to encode messages (e.g. numbers - text can be represented as
# numbers) in such a way that we (or our receiver) can decode them. Eg
# we might use the function
# codedMessage = f( message ),
# where f(x) = 2 * x;
# Then our receiver can read our message, using the inverse function,
# g(x) = x / 2
# so, message = g(codedMessage), bringing back the original message.

# Most encryption techniques work like this, only it's difficult for
# an interloper to determine the secret inverse function. These techniques
# use a combination of public and private keys.

# We want two functions: encrypt() and decrypt(). If x is our message
# (e.g. a number) we will encrypt it it using the RSA function
# f(x) = x ** e mod n
#
# where e and n are carefully chosen numbers. If we choose e and n
# appropriately, a receiver can decrypt the encrypted message even.
# Others cannot, even if they know e and n.

# How do we choose e and n?
#
# (1) We choose two different large prime numbers p and q at random.
# (2) n = p * q.
# (3) Compute e in two steps.
#   (a) Let m = (p - 1) * (q - 1)
#   (b) e = a random prime < m and not a divisor of m.

# Eg from the text, let's work with small primes for our example
# p = 3
# q = 5
# n = 3 * 5 = 15
# m = (3 - 1) * (5 - 1) = 8
# Now choose
# e = 3, since it's < n (i.e. < 15).
# Now we can encrypt any number < n; let's encrypt 13.
# codedMessage = f(x) = f(13) = 13 ** 3 mod 15 = 7
#
# Now, how does our receiver decrypt this 7 to get the original 13?
# Well, when we computed e, we should have also computed a decryption
# exponent d, which has two properties:
# (1) 1 < d < m - 1 (i.e. d lies between 1 and m - 1).
# (2) (e * d) mod m = 1
We call $d$ the "multiplicative inverse of $e$; and there is just one $d$ with these properties.) Then we can retrieve our original message with,

$$g(y) = y \mod d \mod n \quad (y \text{ is our coded message})$$

In our example, $e = 3$ and $m = 8$, so $d = 3$ (only coincidentally same as $e$).
So, $g(7) = 7 \mod 3 \mod 15 = 13$ (and 13 was indeed our original message!)

Notice, that

1. $e$ and $n$ are "public keys", anyone who knows them can encode a message that only one with $d$ can read; but no one else can (even if they have the public keys and a coded message).
2. $d$ is a "private key. One must have this $d$ to decode a coded message.

Hence the name public/private key encryption.

Notice that, if we know $e$ and $n$, we "can" compute the factors $p$ and $q$ and so eventually find $d$. But this factorization of $n$ to find the large primes $p$ and $q$ (as opposed to our small example $p$ and $q$) is computationally hard, meaning it would take a long time even with powerful computers. (One wonders how long it takes the NSA; they are trying!)

We'll do this later. First we have to generate some primes.

```
# Generating Primes #

Recall, a prime number is a number that has no factors (divisors) other than 1 and itself.

eg 2, 3, 5, 7, 11, 13, 17, 19, 23 are all primes

but, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22 are not.

It's easy to determine if $n$ is prime: we simply see if any of the numbers $i$ between 2 and $(n-1)$ divide into it evenly; if there is in $i$ such that $n \mod i = 0$ then $n$ is NOT prime.

We can write a more general predicate (a function returning True or False)

```python
def divisors(n, low, high):
    """ Returns True if $n$ has a divisor in the range from low to high; otherwise returns False."

    if low > high:
        return False
    elif n \mod low == 0:
        return True
    else:
        return divisors(n, low + 1, high)
```

```
# >>> divisors(35, 2, 10)
# True
# >>> divisors(17, 2, 16)
# False
# >>>
```

Now we can define a predicate that says a number $n$ is prime if it has no divisors from 2 to $(n-1)$.

```python
def isPrime(n):
```
For n > 2, returns True if n is prime, False otherwise.

```python
if divisors(n, 2, n-1):
    return False
else:
    return True
```

# Or, better

def isPrime(n):
    """For n > 2, returns True if n is prime, False otherwise."
    return not divisors(n, 2, n-1)

# Now we can write a function that computes a list of primes from n to n some limit.

def listPrimes(n, limit):
    """Returns a list of prime numbers between n and limit."
    if n == limit:
        return []
    elif isPrime(n):
        # Use it
        return [n] + listPrimes(n+1, limit)
    else:
        # Lose it
        return listPrimes(n+1, limit)

# range() for lists

```
Sieve of Eratosthenes (repeated sifting)

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...]

Pick the first number 2, and cross out all divisible by 2:


Pick the next uncrossed number 3, and cross out all divisible by 3


Pick the next uncrossed number 5, and cross out all divisible by 5:

(25, 35, and so on will be crossed out).

We can continue like this as far as we like. (Ad infinitum, really!)

# Assuming we have a function sift

def sift(divisor, List):
    """Returns the list of number List, but where all numbers in it
    that are evenly divided by divisor are sifted out.
    ""
    return "..." # We must still define this.

# we can define our function for computing some primes,

def primeSieve(numbers):
    """Returns a list of all primes in numbers, using a prime sieve.
    ""
    if numbers == []:
        return []
    else:
        prime = numbers[0] # the first number is prime.
        return [prime] + primeSieve(sift(prime, numbers[1:]))

# To do the sifting, we can use the function filter(). Eg if we have

def isNotDivisibleBy2(n):
    """Predicate testing if n is not even, ie not divisible by 2.
    ""
    return n % 2 != 0

# we can use filter to sift, keeping only those numbers satisfying
# our predicate with the built-in filter():

>>> filter(isNotDivisibleBy2, range(3, 21))
[3, 5, 7, 9, 11, 13, 15, 17, 19]

# Notice filter() takes two arguments, a function p and a list L, and
# returns a list containing just the elements in L for which the predicate
# p holds (returns True).

# Although filter is built in to Python, we define it ourselves

def filter(p, L):
    if L == []:
        return []
    elif p(L[0]): # Does the predicate hold for the first element?
        # If so, use it.
        return [L[0]] + filter(p, L[1:])
    else:
        # If not, lose it
        return filter(p, L[1:])

# >>> def isFourLetters(word):
```python
>>> word = "gosh"
>>> len(word) == 4
True

>>> filter(isFourLetters, ['gosh', 'darn', 'foo', 'yuk', 'me', 'heck'])
['gosh', 'darn', 'heck']

# Lambda functions

# We'd like to define a more general,

def isNotDivisibleBy(m):
    """Predicate testing if n not divisible by m."
    return n % m != 0

# We COULD define a two-argument predicate

def isNotDivisibleBy(n, m):
    """Predicate testing if n is divisible by m."
    return n % m != 0

# but our built-in filter() works only with predicates having one
# argument.

# Lambda functions (functions without names) help us here.

# For example, we can use a lambda function in place of
# isNotDivisibleBy2 here.

>>> filter(lambda n: n%2 != 0, range(0,20))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

# That is,
# lambda n: n%2 != 0
# is a function that, when applied to a number n, returns True is it's odd
# and False if it's even.

# Similarly,
# lambda x: x + 1
# is a function that adds one to things.

>>> (lambda x: x + 1)(1)
2
>>> (lambda x: x + 1)(3)
4

# We can use a lambda function for defining sift()

def sift(divisor, List):
    """Returns the list of number List, but where all numbers in it
    that are evenly divided by divisor are sifted out."""
    return filter(lambda x: x % divisor != 0, List)

# So,
#
>>> range(5,25,2)
[5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```
def primeSieve(numbers):
    """Returns a list of all primes in numbers, using a prime sieve."
    if numbers == []:
        return []
    else:
        prime = numbers[0]  # the first number is prime.
        return [prime] + primeSieve(sift(prime, numbers[1:])),

# >>> primeSieve(range(2,200))