Introduction to Programming in Python

Assignment 4 (RSA Cryptosystem) Discussion

Implement the function `_reverse()` in `reverse.py` that reverses the 1D list *a* in place, ie, without creating a new list

```
>_ ~/workspace/rsa_cryptosystem

$ python3 reverse.py
b o l t o n
<ctrl-d>
n o t l o b
$ python3 reverse.py
m a d a m
<ctrl-d>
m a d a m
```

Set n to the number of elements in a

For each i in $[0, n/2]$

- Exchange $a[i]$ with $a[n-i-1]$ (use a temporary variable)

Implement the function `_distance()` in `distance.py` that returns the Euclidean distance between the vectors *x* and *y* represented as 1D lists of floats

```
>_ ~/workspace/rsa_cryptosystem

$ python3 distance.py 2
1 0 <enter>
0 1 <enter>
1.4142135623730951
$ python3 distance.py 5
-9 1 10 -1 1 <enter>
-5 9 6 7 4 <enter>
13.0
```

Set n to the number of elements in x

Set d to 0.0

For each i in $[0, n)$

- Increment d by $(x[i] - y[i])^2$

Return $\sqrt{d}$

Implement the function `_transpose()` in `transpose.py` that creates and returns a new matrix that is the transpose of the matrix represented by the argument *a*

```
>_ ~/workspace/rsa_cryptosystem

$ python3 transpose.py 2 2
1 2 <enter>
3 4 <enter>
1.0 3.0
2.0 4.0
$ python3 transpose.py 2 3
1 2 3 <enter>
4 5 6 <enter>
1.0 4.0
2.0 5.0
3.0 6.0
```

Set m to the number of rows in a and n to the number of columns in a

Set c to a 2D list of dimensions $n \times m$

For each i in $[0, n)$

- For each j in $[0, m)$
  - Set $c[i][j]$ to $a[j][i]$

Return c

Implement the function `_isPalindrome()` in `palindrome.py` that returns `True` if the argument *s* is a palindrome (ie, reads the same forwards and backwards), and `False` otherwise

```
>_ ~/workspace/rsa_cryptosystem

$ python3 palindrome.py bolton
False
$ python3 palindrome.py amanaplanacanalpanama
True
```

Set n to the number of characters in s

For each i in $[0, n/2]$

- Return `False` of $s[i]$ is different from $s[n-i-1]$

Return `True`

Implement the function `_sin()` in `sin.py` that calculates the sine of the argument x (in radians), using the formula

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
>_ ~/workspace/rsa_cryptosystem

$ python3 sin.py 60
0.8660254037844385
```

Set *total* to 0.0, *term* to 1.0, *sign* to 1, and *i* to 1

As long as *total* is different from $total + term$

- Set *term* to $term \times x/i$ to term
- If *i* is odd, increment total by $sign * term$ and toggle *sign* (ie, set it -1 if it's positive and +1 if it's negative)
- Increment *i* by 1

Return *total*

The RSA cryptosystem involves three integers n, e, and d that satisfy certain mathematical properties

The *public key* (n, e) is made public on the Internet, while the *private key* (n, d) is only known to Bob

If Alice wants to send Bob a message $x \in [0, n)$, she encrypts it using the function $E(x) = x^e \bmod n$, where $n = pq$ for two distinct large prime numbers p and q chosen at random, and e is a random prime number less than $m = (p-1)(q-1)$ such that e does not divide m

Example: suppose $p = 47$ and $q = 79$; then $n = 3713$ and $m = 3588$; further suppose $e = 7$; if Alice wants to send the message $x = 2020$ to Bob, she encrypts it as $E(2020) = 2020^7 \bmod 3713 = 516$

When Bob receives the encrypted message y, he decrypts it using the function $D(y) = y^d \bmod n$, where $d \in [1, m)$ is an integer that satisfies the equation $ed \bmod m = 1$

Continuing the example above, if $d = 2563$, then when Bob receives the encrypted message $y = 516$ from Alice, he decrypts it to recover the original message as $D(516) = 516^{2563} \bmod 3713 = 2020$

Implement a library called `rsa.py` that provides functions needed for developing the RSA cryptosystem and supports the following API

rsa	
`keygen(lo, hi)`	generates and returns the public/private keys as a tuple (n, e, d), picking prime numbers p and q needed to generate the keys from the interval `[lo, hi)`
`encrypt(x, n, e)`	encrypts `x` (int) using the public key `(n, e)` and returns the encrypted value
`decrypt(y, n, d)`	decrypts `y` (int) using the private key `(n, d)` and returns the decrypted value
`bitLength(n)`	returns the least number of bits needed to represent `n`
`dec2bin(n, width)`	returns the binary representation of `n` expressed in decimal, having the given width and padded with leading zeros
`bin2dec(n)`	returns the decimal representation of `n` expressed in binary

>_ ~/workspace/rsa_cryptosystem

```
$ python3 rsa.py S
encrypt(S) = 1743
decrypt(1743) = S
bitLength(83) = 7
dec2bin(83) = 1010011
bin2dec(1010011) = 83
```

`keygen(lo, hi)`

- Get a list of primes from the interval $[lo, hi)$
- Sample two distinct random primes p and q from that list
- Set n and m to pq and $(p-1)(q-1)$, respectively
- Get a list primes from the interval $[2, m)$
- Choose a random prime e from the list such that e does not divide m (you will need a loop for this)
- Find a $d \in [1, m)$ such that $ed \bmod m = 1$ (you will need a loop for this)
- Return the tuple[1] (n, e, d)

`encrypt(x, n, e)`

- Implement the function $E(x) = x^e \bmod n$

`decrypt(y, n, d)`

- Implement the function $D(y) = y^d \bmod n$

[1] A tuple is like a list, but is immutable. You create a tuple by enclosing comma-separated values within matched parentheses, eg, `a = (1, 2, 3)`. If `a` is a tuple, `a[i]` is the *i*th element in it

`_primes(lo, hi)`

- Create an empty list
- For each $p \in [lo, hi)$, if p is a prime, add p to the list
- Return the list

`_sample(a, k)`

- Create a list b that is a copy (not an alias) of a
- Shuffle the first k elements of b
- Return a list containing the first k elements of b

`_choice(a)`

- Get a random number $r \in [0, l)$, where l is the number of elements in a
- Return the element in a at the index r

Write a program called `keygen.py` that accepts *lo* (int) and *hi* (int) as command-line arguments, generates public/private keys (n, e, d), and writes the keys to standard output, separated by a space

```
>_ ~/workspace/rsa_cryptosystem

$ python3 keygen.py 50 100
3599 1759 2839
```

Accept *lo* (int) and *hi* (int) as command-line arguments

Get public/private keys as a tuple

Write the three values in the tuple, separated by a space

Write a program called `encrypt.py` that accepts the public-key *n* (int) and *e* (int) as command-line arguments and a message to encrypt from standard input, encrypts each character in the message, and writes its fixed-width binary representation to standard output

```
>_ ~/workspace/rsa_cryptosystem

$ python3 encrypt.py 3599 1759
CS110
<ctrl-d>
000110000000010011010100001010100011001010100011001110000110010111100100
```

Accept public-key n (int) and e (int) as command-line arguments

Get the number of bits per character (call it *width*) needed for encryption, ie, number of bits needed to encode n

Accept *message* to encrypt from standard input

For each character c in *message*

- Use the built-in function `ord()` to turn c into an integer x
- Encrypt x
- Write the encrypted value as a *width*-long binary string

Write a newline character

Write a program called `decrypt.py` that accepts the private-key n (int) and d (int) as command-line arguments and a message to decrypt (produced by `encrypt.py`) from standard input, decrypts each character (represented as a fixed-width binary sequence) in the message, and writes the decrypted character to standard output

```
>_ ~/workspace/rsa_cryptosystem

$ python3 decrypt.py 3599 2839
0001100000000100110101000010101000110010101000110011100001100101111100100
<ctrl-d>
CS110
$ python3 encrypt.py 3599 1759 | python3 decrypt.py 3599 2839
Python is the mother of all languages.
<ctrl-d>
Python is the mother of all languages.
```

Accept private-key *n* (int) and *d* (int) as command-line arguments

Get the number of bits per character (call it *width*)

Accept *message* (binary string generated by `encrypt.py`) from standard input

Assuming l is the length of *message*, for $i \in [0, l - 1)$ and in increments of *width*

- Set s to substring of message from i to $i + width$ (exclusive)
- Set y to decimal representation of the binary string s
- Decrypt y
- Write the character corresponding to the decrypted value, obtained using the built-in function `chr()`