PA2

1.

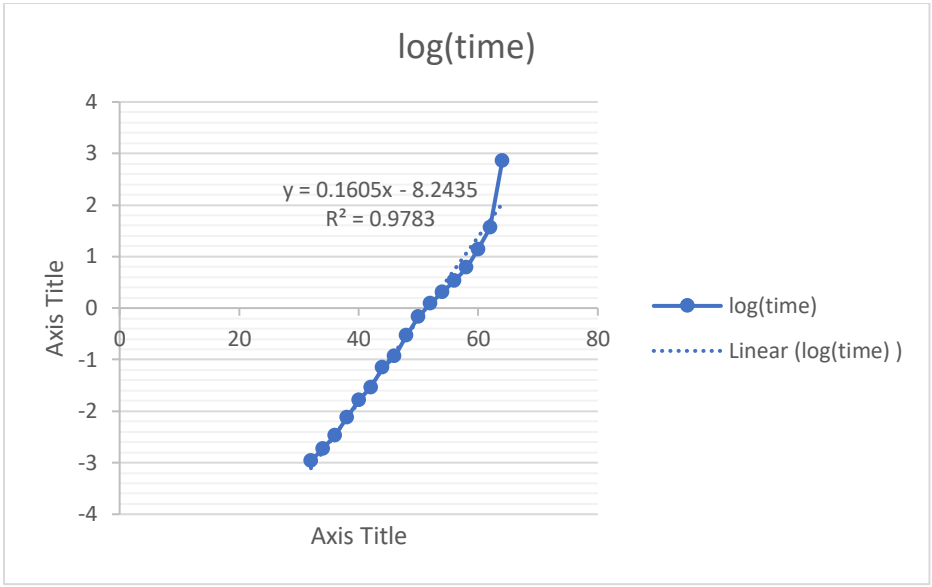
a.

```
43
44    def semiprime_factorize(n):
45        # precondition: n is a semiprime number: n = p.q
46        # returns the ordered pair of prime numbers (p,q)
47        # your code is here...
48            for i in range (2,int(math.sqrt(n))):
49                if n%i==0:
50                    return(i,n//i)
51            return(1,n)
52
53    print("Number 21. Prime Factors:   ", semiprime_factorize(21))
54
```

b.

```python
import math
import random
import time


def random_prime(n):
    # generates a random n-bit prime number
    if n < 2:
        return -1
    max = 1 << n
    while True:
        r = random.randrange(2, max + 1)
        if primality_test(r, 40):
            return r


def primality_test(n, iterations):
    if ~n & 1 or n == 3:
        if n == 2 or n == 3:
            return True
        else:
            return False
    q = n - 1
    k = 0
    while ~q & 1:
        q >>= 1
        k += 1
    for iteration in range(iterations):
        a = random.randrange(2, n - 1)
        rem = pow(a, q, n)
        if rem == n - 1 or rem == 1:
            continue
        inconclusive = False
        for j in range(1, k):
            rem = pow(rem, 2, n)
            if rem == n - 1:
                inconclusive = True
                break
        if not inconclusive:  # composite
            return False
    return True


def semiprime_factorize(n):
    # precondition: n is a semiprime number: n = p.q
    # returns the ordered pair of prime numbers (p,q)
    # your code is here...
        for i in range (2,int(math.sqrt(n))):
            if n%i==0:
                return(i,n//i)
        return(1,n)

print("Number 21. Prime Factors:   ", semiprime_factorize(21))

print('key length', '\t', 'log(time)')
for nbits in range(32, 65, 2):
    avg_time = 0
    for _ in range(20):
        p = random_prime(nbits >> 1)
        q = random_prime(nbits >> 1)
        t0 = time.time()
        (calculated_p, calculated_q) = semiprime_factorize(p * q)
        elapsed = time.time() - t0
        if (p, q) != (calculated_p, calculated_q) and (p, q) != (calculated_q, calculated_p):
            print("Error: incorrect prime factorization", (p, q), (calculated_p, calculated_q))
        avg_time += elapsed / 20
    if avg_time != 0:
        print(nbits, '\t\t\t', round(math.log10(avg_time) * 100) / 100)
```

c.



i.

ii. Using linear extrapolation, estimate the time it takes for your computer to break RSA with 2048-bit key: **log10(320.4605)**

2. List and briefly explain the four steps of operations/transformations that AES performs in each round of its encryption algorithm.
    a. Byte substitution
        i. A 16 x 16 lookup table is used to find a replace byte for a given byte in the input state array
        ii. The entries in the lookup table are created by using multiplicative inversed and bit scrambling to destroy the bit level correlations inside each byte.
    b. A Shift rows
        i. Operates on each row of the state array, each row is shifted to the right by a certain number of bytes.
    c. Mix columns
        i. Each column of 4 bytes is transformed. A function takes 4 bytes of one column as input, and outputs 4 completely new bytes which replace the original column. The result is another new matrix consisting of 16 new bytes. This is not performed in last round.
    d. Round key
        i. The 16 bytes of the matrix are now considered as 128bits and are XORed to the 128 bits of the round key. On the last round, the output is the cyphertext. Else, the resulting 128 bits are interpreted as 16 bytes and another round begins.
3.
    a. Yes. Alice can verify that the signature is bobs. Alice has Bobs public key and message he sent. In the rsa_message_sign function, the message digest is found

using SHA256. The digest is then signed by raising its power to d mod n. rsa_signature_verify then decrypts the signature by raising it to the power of e mod n. signature^(e mod n) is then compared to the message digest. If these values are equal, the signature is verified. If not equal, not verified. Alice should trust the message is from Bob if the signature has been verified by RSA.

b. After changing the dollar amount, the signature does **not** get verified.
c. 721443258457176957034558630495526614726172037471193209237173790365782395324506209521474436486843145265385337651637836473078987987616560446230415720088430124239779134054515221603113455002022895773800294509654544501929948604956458445693272330296929822429402398486595964795165602951382881380441834808965590947 66