Q1.

EX 1

(Following RAS in the lecture)

x e (mod p*q)

- Step 1: **p = 3 and q = 7**
- Step 2: N = 3 * 7 = 21
- Step 3: (N) = (p-1)(q-1) = 2*6 = 12 coprime
- Step 4: choose number e { 1 < e < φ(N) = 12 and coprime with φ(N) }
  2 3 4 5 6 7 8 9 10 11

  choose e = 5
- Step 5: choose d such as de (mod φ(N)) = 1.
  e = 5: 5d (mod 12) = 1 → d = 5
  public key (5, 21)
  private key (5, 21)
- m = 3: c = 3^5 mod 21 = 12 (ciphertext)
- c = 12 → m = 12 5 mod 21 = 3 (plaintext)

EX 2

- Step 1: **p = 3 and q = 5**
- Step 2: N = 3 * 5 = 15
- Step 3: Φ(N) = (p-1)(q-1) = 2*4= 8
  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 coprime
- Step 4: choose number e { 1 < e < Φ(N) = 8 and coprime with N and Φ(N)}
  1 2 3 4 5 6 7
  e = 7
- Step 5: choose d such as de (mod Φ(N) = 12) = 1.
  7d (mod 8) = 1
  7, 14, 21, 28, 35, 42, 49, 56,
  d = 15
  public key (7, 15)
  private key (15, 15)
- x e (mod p*q) = x 7 (mod 15)
  x =2 → 2 7 (mod 15) = 8
  8 15 (mod 15) = 2

Q2.

```
import cryptography.exceptions
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

# Load the public key.
with open('public_key', 'rb') as f:
     public_key = serialization.load_pem_public_key(f.read(),
default_backend())
     f.close()

# Load the document.
with open('document.txt', 'rb') as f:
     document = f.read()
     f.close()
```

```
# Load the signature.
with open('signature', 'rb') as f:
     signature = f.read()
     f.close()

# Perform the verification.
try:
    public_key.verify(
        signature,
        document,
        padding.PSS(
            mgf = padding.MGF1(hashes.SHA256()),
            salt_length = padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )
    print "The document and signature files are successed verification!"
except cryptography.exceptions.InvalidSignature as e:
    print('ERROR: document and/or signature files are failed verification!')
```

Q3.

- **Sign**

```
import base64
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding

# Load the private key.
with open('private_key', 'rb') as f:
     private_key = serialization.load_pem_private_key(
        f.read(),
        password = 'noway',
        backend = default_backend(),
     )
     f.close()

# Load the document.
with open('message.txt', 'rb') as f:
     document = f.read()
     f.close()

# Sign the message.txt file.
signature = private_key.sign(
        document,
        padding.PSS(
            mgf = padding.MGF1(hashes.SHA256()),
            salt_length = padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )
```

```
# Save the signature.
with open('signature', 'wb') as f:
     f.write(signature)
     f.close()
```

- **Verify**

```
import cryptography.exceptions
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

# Load the public key.
with open('public_key', 'rb') as f:
                public_key   =   serialization.load_pem_public_key(f.read(),
default_backend())
     f.close()

# Load the document.
with open('document.txt', 'rb') as f:
     document = f.read()
     f.close()

# Load the signature.
with open('signature', 'rb') as f:
     signature = f.read()
     f.close()

# Perform the verification.
try:
    public_key.verify(
        signature,
        document,
        padding.PSS(
            mgf = padding.MGF1(hashes.SHA256()),
            salt_length = padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )
    print "The document and signature files are successed verification!"
except cryptography.exceptions.InvalidSignature as e:
    print('ERROR: document and/or signature files are failed verification!')
```

Q4.

- **encrypt**

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes


# Generate the private/public key pair.
private_key = rsa.generate_private_key(
    public_exponent = 65537,
```

```
    key_size = 2048,
    backend = default_backend(),
    )

message = 'The document is successfully decrypted. Congratulation!'

ciphertext = private_key.public_key().encrypt(
        message, None
        )

# Save the message hash.
with open('message_hash', 'wb') as f:
     f.write(ciphertext)
     f.close()
```

- **decrypt**

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

# Load the private key.
file_public_key = open('private_key', 'rb')
private_key  =  serialization.  load_pem_private_key(file_public_key.read(),
None, default_backend())
file_public_key.close()

# Load the  cipthertext document.
file_document = open('message_hash', 'rb')
ciphertext = file_document.read()
file_document.close()


plaintext = private_key.decrypt(
      ciphertext,
      padding.OAEP(
      mgf=padding.MGF1(algorithm=hashes.SHA256()),
      algorithm=hashes.SHA256(),
      label=None
      )
    )
print (plaintext)
```