P.HARINI
SEC 7
2300033404

NPS LAB EXPERIMENT 13

```
import random
from sympy import isprime, mod_inverse

def generate_prime_candidate(length):
    """Generate an odd prime candidate of specified bit length."""
    p = random.getrandbits(length)
    # Ensure p is odd and has the desired bit length
    p |= (1 << length - 1) | 1
    return p

def generate_prime_number(length):
    """Generate a prime number of specified bit length."""
    p = 4  # Start with a non-prime dummy value
    while not isprime(p):
        p = generate_prime_candidate(length)
    return p

def generate_keypair(bits):
    """Generate RSA keypair of specified bit length."""
    # Step 1: Generate two distinct prime numbers
    p = generate_prime_number(bits)
    q = generate_prime_number(bits)
    while p == q:
        q = generate_prime_number(bits)

    # Step 2: Compute n = p * q
    n = p * q
```

P.HARINI
SEC 7
2300033404

```
    # Step 3: Compute the totient (φ(n))
    phi = (p - 1) * (q - 1)

    # Step 4: Choose e (typically 65537)
    e = 65537

    # Step 5: Compute d, the modular inverse of e
    d = mod_inverse(e, phi)

    return ((e, n), (d, n))  # Return public and private keys

def encrypt(public_key, plaintext):
    """Encrypt the plaintext using the public key."""
    e, n = public_key
    # Convert plaintext to integer for encryption
    plaintext_int = int.from_bytes(plaintext.encode('utf-8'), 'big')
    # Encrypt: ciphertext = (plaintext^e) mod n
    ciphertext = pow(plaintext_int, e, n)
    return ciphertext

def decrypt(private_key, ciphertext):
    """Decrypt the ciphertext using the private key."""
    d, n = private_key
    # Decrypt: plaintext = (ciphertext^d) mod n
    plaintext_int = pow(ciphertext, d, n)
    # Convert integer back to bytes and decode to string
    plaintext = plaintext_int.to_bytes((plaintext_int.bit_length() + 7) // 8, 'big').decode('utf-8',
errors='ignore')
    return plaintext

# Example usage for experimentation
```

P.HARINI
SEC 7
2300033404

```
if __name__ == "__main__":
    bits = 32  # Increase this for better security; e.g., 512 or 1024 for real applications
    public_key, private_key = generate_keypair(bits)

    print("Public Key:", public_key)
    print("Private Key:", private_key)

    message = "Experiment RSA"
    print("Original Message:", message)

    # Encrypt the message
    ciphertext = encrypt(public_key, message)
    print("Encrypted Message (ciphertext):", ciphertext)

    # Decrypt the message
    decrypted_message = decrypt(private_key, ciphertext)
    print("Decrypted Message:", decrypted_message)
```

- **generate_prime_candidate**: Generates a random odd number with the specified bit length.
- **generate_prime_number**: Uses `generate_prime_candidate` to repeatedly find a prime.
- **generate_keypair**: Produces the RSA keypair, ensuring the public and private keys are compatible with the bit length and properties required by RSA.
- **encrypt**: Encrypts the message by converting it to an integer, applying RSA encryption, and returning the ciphertext.
- **decrypt**: Converts the ciphertext back to plaintext by applying the decryption formula and converting it back to a readable string.