Time and Space Complexity of the RSA Algorithm

Key Generation

1. **Miller-Rabin:**

```python
def miller_rabin(N: int, k: int) -> str:
    if N <= 1 or N % 2 == 0:
        return "composite"
    if N == 2:
        return "prime"

    for i in range(k):
        a = random.randint(2, N-1)
        y = N - 1
        while y % 2 == 0:
            y //= 2

        x = mod_exp(a, y, N)
        if x == 1 or x == N - 1:
            continue

        while y != N - 1:
            x = (x**2) % N
            y *= 2
            if x == 1:
                return "composite"
            if x == N - 1:
                break
        if y == N - 1:
            return "composite"

    return "prime"
```

- **Time Complexity**: O(n^4)

This is because we call Mod_exp which has a complexity of O(n^3) and we do this n times as we are removing one bit each time.

- **Space Complexity:** O(n^2)

This is inherited from Mod_exp as the memory is overwritten each iteration.

2. **Mod_exp**:

```
def mod_exp(x: int, y: int, N: int) -> int:
    if(y == 0):
        return 1
    z = mod_exp(x, y//2, N)
    if(y % 2 == 0):
        return (z**2) % N
    else:
        return (x*(z**2)) % N
```

   - **Time Complexity**: O(n^3)

   This is where n is the largest number of bits of the 3 input numbers. At most, we will have n recursive calls and each call will multiply 2 n-bit numbers.

   - **Space Complexity**: O(n^2)

   The inputs are each O(n) and it will have at most n recursive calls

3. **Fermat function:**

```
def fermat(N: int, k: int) -> str:
    for i in range(k):
        a = random.randint(2, N-1)
        if (mod_exp(a, N-1, N) != 1):
            return "composite"
    return "prime"
```

   - **Time Complexity**: O(n^3).

   This is because mod _exp has a complexity of O(n^3) and, the loop will be disregarded as it is k times, which in the rules for big O, O(k*n) is the same as O(n) if k is a constant.

   - **Space Complexity:** O(n^2)

   The space complexity is again O(n^2) because there aren't any significant space requirements for the algorithm itself, except for that which is inherited from Mod_exp function

4. **Ext_euclid function:**

```python
def ext_euclid(a: int, b: int) -> tuple[int, int, int]:
    if a < b:
        tmp = a
        a = b
        b = tmp

    if b == 0:
        return 1, 0, a
    x, y, d = ext_euclid(b, a % b)
    return y, x - (a // b)*y, d
```

   - **Time Complexity**: O(n^3)

   This is because the modular division is O(n^3) as each return is O(n^2) and we do this n times

   - **Space Complexity:** O(n^2)

   For each N stack of the recursion, you store a number of n size.

5. **Gen_large_primes function:**

```python
def generate_large_prime(bits=512) -> int: # W
    """
    Generate a random prime number with the sp
    Use random.getrandbits(bits) to generate a
    specified bit length.
    """
    ran_numb = random.getrandbits(bits)
    while fermat(ran_numb, 100) != "prime":
        ran_numb = random.getrandbits(bits)
    return ran_numb
```

   - **Time Complexity**: O(n^4) while using Fermat and O(n^5) for Millar-Rabin

   This is because using each of the functions have their respective time complexity and we do this at most n times

   - **Space Complexity:** O(n^2)

This is inherited from Mod_exp which is in both respective functions

6. **Gen key pair's function:**

```python
def generate_key_pairs(bits: int) -> tuple[int, int, int]:
    """
    Generate RSA public and private key pairs.
    Return N, e, d
    - N must be the product of two random prime numbers p a
    - e and d must be multiplicative inverses mod (p-1)(q-1
    """
    p = generate_large_prime(bits // 2)
    q = generate_large_prime(bits // 2)
    N = p * q
    i = (p - 1) * (q - 1)

    e = get_e(i)

    d = ext_euclid(e, i)[1]
    if d < 0:
        d += i
    return N, e, d
```

- **Time Complexity:** O(n^4)

This is because we generate 2 large primes inheriting the time complexity of that function and we call at most Ext_euclid e times which gives us 2n^4 + en^3 which gives us n^4

- **Space Complexity:** O(n^2)

This is inherited from Fermat which is used in the generating of the large prime numbers

Probability:

```
def fprobability(k: int) -> float:
    return 1 - (1/2)**k


# You will need to implement this f
def mprobability(k: int) -> float:
    return 1 - (1/4)**k
```

Fermat: The likelihood of a false positive decreases by half with each test iteration. Subtracting this value from 1 gives the probability of the result of being correct.

Miller-Rabin: Similarly, the probability of a false positive is ¼ and using the same logic we can use 1- ¼^k probability of being correct