Project 1 - complexity report
KAJ Jacobs

	mod exp	fermat	miller rabin	ext euclid	large prime	key pairs
time complexity	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)
space complexity	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)	O(log N)

```python
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
```

Here we know that the base case or best case is O(1) when y equals 0 but typically this function will involve the other statements. The recursive calls keep happening and halving y, the exponent, meaning that they will loop y/2 times and the return statements of multiplication happen in O(1) time. Therefore the total time complexity will be O(log y) or O(log N) because the number of recursive calls is proportional to log y. The system adds a new frame each recursion and that is determined by y/2 so the space complexity is also O(log N).

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

This tests the primality of N by testing a random a, k times; such that this will at most be ran k times so we can treat k as a constant. It then runs mod_exp which we know has the time complexity of O(log N) so it would be O(k log N) or reduced to just O(log N) since we know k is a constant with a large N. Since this will just loop and write over a then use mod_exp the space complexity doesn't change from mod_exp's. The space complexity is O(log N).

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

This tests the primality of N by testing a random a, k times; such that this will at most be ran k times so we can treat the loops as O(1) since k is a constant. This then runs mod_exp with a slight difference of the second number passing in getting higher per each run on the second loop t. Both loops are treated as constants because we assume k being a relatively small number so the meat of this function comes down to the mod_exp recursion. So our time complexity of O(k^2 log N) simplifies to O(log N). With this using recursion like previously stated in mod_exp it runs y/2 which means that the space complexity is also O(log N).

```python
def ext_euclid(a: int, b: int) -> tuple[int, int, int]:
    if b == 0:
        return a, 0, 1
    else:
        x, y, d = ext_euclid(b, a % b)
    return y, (x - ((a//b)*y)), d
```

This function recursively calls itself but each time after the first will do b, a% b since we know that a always has to be bigger. Since the function's recursion depends on the smaller number or the b of the function call and it gets smaller and smaller each recursion similar to the above y/2 we can say that its time complexity is O(log b) or O(log N). Similarly because this is based on the function's recursion which is determined by b that keeps getting smaller each call the space complexity is O(log N).

```python
def generate_large_prime(bits=512) -> int:
    x = random.getrandbits(bits)
    while miller_rabin(x, 20) != "prime":
        x = random.getrandbits(bits)
    return x
```

This function is used to calculate p and q for the key pairs function and uses miller rabin as the main portion. With that in mind and k being 20 which is relatively small, this function will follow miller rabin's recursion and have the same time and space complexity of O(log N).

```python
def generate_key_pairs(bits: int) -> tuple[int, int, int]:
    p = generate_large_prime(bits)
    q = generate_large_prime(bits)
    N = p*q
    relativeN = (p-1)*(q-1)

    for testprime in primes:
        if  relativeN % testprime != 0:
            e = testprime
            break

    d = ext_euclid(relativeN, e)[1]
    if d < 0:
        d = d + relativeN

    return N, e, d
```

This function uses a combination of other functions that we have already identified before which helps us look jump forward in determining its complexity. Because we call generate large prime twice we can take that as O(2 log N) everything leading up to and after the ext euclid call is done in O(1) time which means we focus on that plus that O(2 log N). Since both are essentially O(log N) there is no dominant term or a term that is higher so the time complexity is O(log N). Because both use O(log N) as their space complexity, the same rule applies and this function's space complexity is O(log N).

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


# You will need to implement this f
def mprobability(k: int) -> float:
    return fprobability * (3/4)
```

Determining the probability of correctness for fermat and miller rabin is fairly simple as there is a given equation: 1 / (2^k). Miller rabin only changes this slightly by multiplying ¾ to that same equation making it extremely effective at lower intervals of k. As k increases, the amount of tests run on either equation rises such that the ¾ from miller rabin is treated as a constant and not significant.