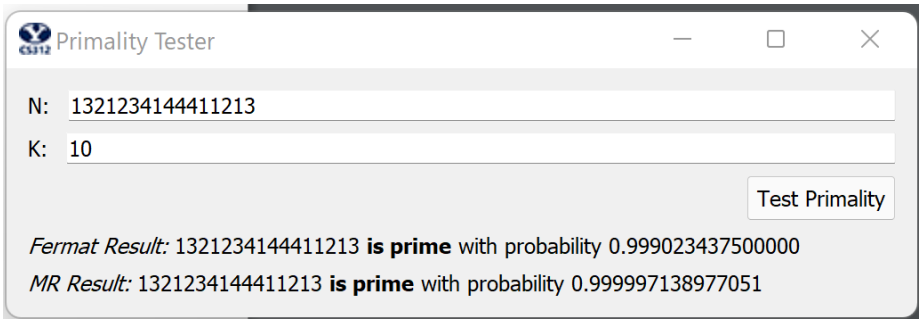
Daniel Jones – Lab 1 Submission

Part 1. Screenshot of large prime number working



Part 2a. Modular Exponentiation

```python
# Calculate x^y mod N
# Time complexity: O(n^3) for n being the chars in x and y.
#   n recursive calls with n^2 complexity per call
# Space complexity: O(n) as each recursive call creates new ints on the stack
def mod_exp(x, y, N):
    if y == 0:
        return 1                # Base case
    z = mod_exp(x, y // 2, N)   # Recursive Call with floor(y/2), n calls
    if y % 2 == 0:              # if y is even return z^2
        return (z * z) % N
    else:                       # if y is uneven return z^2 * x
        return (x * z * z) % N  # modular multiplication is n^2 for chars
```

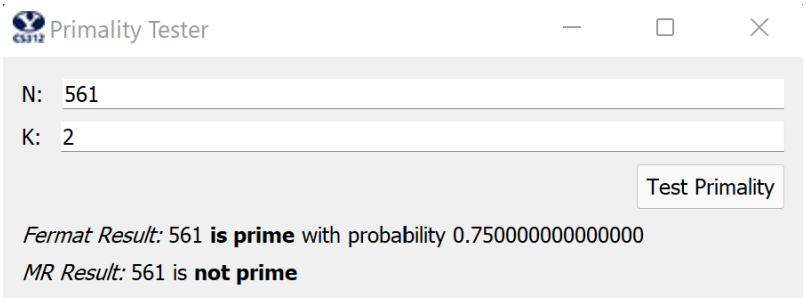
2b. Fermat Primality Tester

```python
# Fermat's primality tester for number N with k tests
# Time complexity: O(n^3) as each mod_exp call is 0(n^3) for length on N
# Space complexity: O(n) as each call to mod_exp will be O(n) for space with k calls
def fermat(N, k):
    for i in range(k):                  # k loops will do k tests of N
        a = random.randint(4, N - 2)    # choose random int in range 4 - (N-2) to avoid corner cases
        if mod_exp(a, N - 1, N) != 1:   # calc. a^(N-1) mod N which should be 1 if N is prime
            return "composite"
    return 'prime'                      # if N does not fail any of the tests, return prime
```

2c. Miller Rabin Primality Testing

```python
# Miller Rabin primality test for number N with k tests
# Time complexity: O(n^4) for n calls to mod_exp with n^3 complexity
# Space complexity: O(n) as while loop writes over variables but mod_exp uses n space
def miller_rabin(N, k):
    for i in range(k):                  # will loop k times perform k tests
        a = random.randint(4, N - 2)    # choose random int in range 4 - (N-2) to avoid corner cases
        e = N - 1                       # sets the exponent to N-1 which will be modified in the while loop
        if mod_exp(a, e, N) != 1:       # if the first test is not 1 then N is composite
            return 'composite'
        while e % 2 == 0:               # Loop while exponent can be halved (n calls)
            e = e // 2                  # Halve exponent
            if mod_exp(a, e, N) != 1:   # Once mod_exp does not equal 1 (O(n^3) complexity)
                if mod_exp(a, e, N) != N - 1:# if next mod_exp does not equal -1
                    return 'composite'  # Than N is not prime
                break
    return 'prime'                      # If N passes k tests, return prime
```

2d. I know that Fermat and Miller Rabin are only likely to disagree when a Carmichael number is used. 561 is such a number. The algorithms agreed most of the time but every once and a while the Fermat Test would get tricked into thinking that the number was prime when in fact it was not. This happens according to my research when the randomly chosen base a is relatively prime to the exponent causing the Fermat test to think the number is prime. The Miller Rabin test was less easily fooled. Neither was fooled very often as I think I avoided many of the corner cases by choosing my a from range 4 to N-2. They disagreed more for smaller values.



Part 3. Time and space complexity

Modular Exponentiation function: (n = length of arbitrarily long ints)

1)	If y is 0, return 1	base case, not important to complexity
2)	Z = x^y/2 mod N	recursive call with y/2 so n calls
3)	If y is even return z^2 mod N	O(n^2) for multiply
4)	If y is odd return x * z^2 mod N	O(n^2) for two multiplies

The recursive call n times with highest order O(n^2) for each call gives a total complexity of O(n^3). Space complexity is O(n) as only one int is stored, z

Both probability calculations:

1) 1-c^k k and c are both constants without arbitrary length so O(1)

Both time and space complexity are O(1) because c and k are fixed, non-arbitrary length integers and nothing is stored.

Fermat Test:

1) Loop k times As k is constant, this loop will not add to the complexity
2) Pick int a in range (1,N) O(1) as random is not dependent on length of input
3) If a^(N-1) mod N is not 1 return composite O(n^3) as ModExp is O(n3) for each call with arbitrary length int
4) After loop, return prime O(1) as nothing is calculated

Total time complexity is O(kn^3) but since k is always constant, this simplifies to O(n^3) based on the mod_exp calculation.
Space complexity is O(1) as nothing is stored of size n

Miller Rabin

1) Loop k times k is constant so does not affect complexity
2) Pick int a in range(1,N) O(1) as choosing int is not affected by n
3) Set e to N-1 O(n) for 1 subtraction
4) If a^e mod N is not 1 return comp. O(n^3) for call to ModExp
5) While e is even loop max of n times (where n is length of N)
6) Divide e by 2 O(n^2) for 1 divide
7) If a^e mod N is not 1 O(n^3) for call to ModExp
8) If a^e mond N is not -1 return comp. O(n^3) for call to ModExp
9) Break
10) After all loops return prime O(1) as nothing calculated

Total time complexity is O(n^4) as while loop could in worst case run n times with each call to ModExp being O(n^3) so O(n^4) combined.
Space complexity is O(n) for storing e

Part 4. Probabilities of correctness

```
# returns 1 minus the chance that the fermat tests were all false positives which is 1/2
def fprobability(k):
    return 1 - 2 ** -k


# same as fermat probability above except that each test has only a 1/4 probability of being wrong
def mprobability(k):
    return 1 - 4 ** -k
```

For the Fermat test, it can be easily proven that for any random a value chosen, there is less than a ½ chance of that value returning prime when it is in fact not prime. This means that for each test, the chance of reporting prime incorrectly is reduced by ½. This is calculated by finding the probability of each of k tests being a false positive and subtracting that value from 1. This gives us the confidence in N actualy being prime.

For the miller rabin equation this is the same logic except that for any given a there is only a ¼ chance of a false positive so the total probability of a false positive after k tests is 1/4^k. This is also subtracted from 1 to give the probability that N is in fact prime. Both of these numbers very quickly approach an asymptote with 1 as the probability of error decreases rapidly as k increases.

Appendix

```
import random


def prime_test(N, k):
    # This is main function, that is connected to the Test button. You don't
need to touch it.
    return fermat(N, k), miller_rabin(N, k)


# Calculate x^y mod N
# Time complexity: O(n^3) for n being the chars in x and y.
#   n recursive calls with n^2 complexity per call
# Space complexity: O(n) as each recursive call creates new ints on the stack
def mod_exp(x, y, N):
    if y == 0:
        return 1  # Base case
    z = mod_exp(x, y // 2, N)  # Recursive Call with floor(y/2), n calls
    if y % 2 == 0:  # if y is even return z^2
        return (z * z) % N
    else:  # if y is uneven return z^2 * x
        return (x * z * z) % N  # modular multiplication is n^2 for chars


# returns 1 minus the chance that the fermat tests were all false positives
```

```
which is 1/2
def fprobability(k):
    return 1 - 2 ** -k


# same as fermat probability above except that each test has only a 1/4
probability of being wrong
def mprobability(k):
    return 1 - 4 ** -k


# Fermat's primality tester for number N with k tests
# Time complexity: O(n^3) as each mod_exp call is 0(n^3) for length on N
# Space complexity: O(n) as each call to mod_exp will be O(n) for space with
k calls
def fermat(N, k):
    for i in range(k):  # k loops will do k tests of N
        a = random.randint(4, N - 2)  # choose random int in range 4 - (N-2)
to avoid corner cases
        if mod_exp(a, N - 1, N) != 1:  # calc. a^(N-1) mod N which should be
1 if N is prime
            return "composite"
    return 'prime'  # if N does not fail any of the tests, return prime


# Miller Rabin primality test for number N with k tests
# Time complexity: O(n^4) for n calls to mod_exp with n^3 complexity
# Space complexity: O(n) as while loop writes over variables but mod_exp uses
n space
def miller_rabin(N, k):
    for i in range(k):  # will loop k times perform k tests
        a = random.randint(4, N - 2)  # choose random int in range 4 - (N-2)
to avoid corner cases
        e = N - 1  # sets the exponent to N-1 which will be modified in the
while loop
        if mod_exp(a, e, N) != 1:  # if the first test is not 1 then N is
composite
            return 'composite'
        while e % 2 == 0:  # Loop while exponent can be halved (n calls)
            e = e // 2  # Halve exponent
            if mod_exp(a, e, N) != 1:  # Once mod_exp does not equal 1
(O(n^3) complexity)
                if mod_exp(a, e, N) != N - 1:  # if next mod_exp does not
equal -1
                    return 'composite'  # Than N is not prime
                break
    return 'prime'  # If N passes k tests, return prime
```