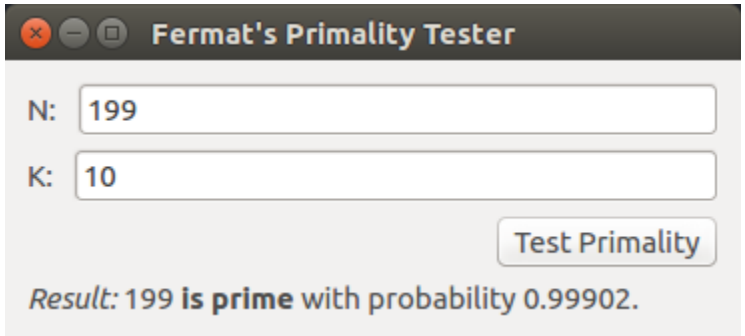
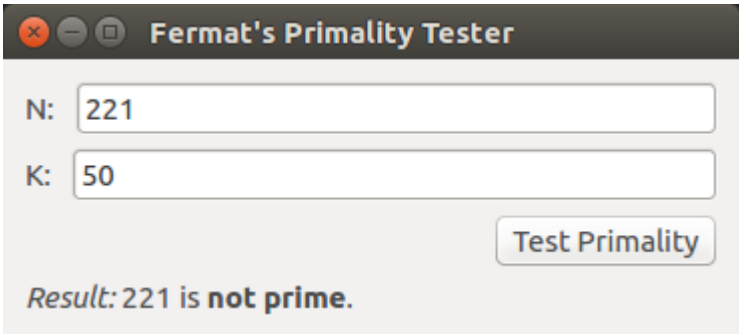
Berkeley Andrus
Lab 1 Report – Fermat and Carmichael tests.

Part 1:

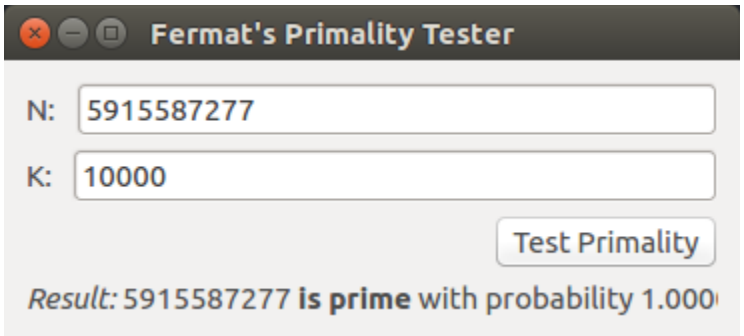
Screenshot 1: Demonstrates functionality with prime numbers.



Screenshot 2:
Demonstrates functionality with composite numbers.



Screenshot 3:
Demonstrates functionality with extremely large numbers.



Parts 2 and 3:

* I thought a screenshot of my development environment would be easier to read than code copied and pasted into a word processor. Sorry that it is small.

Working code with explanation of how it works:

Carmichael.py — ~/Documents/312Code/proj1-primality-test — Atom

fermat.py | Carmichael.py

```python
import random

# This function runs the fermat and carmichael tests on N using k unique 'a' values.
def prime_test(N, k):
    if k > N - 2:                           # Checks that inputs are within the proper range
        k = N - 2
    if k < 0:
        k = 1
    nums = []
    for _ in range(k):                      # Puts k variables (a values) in the list nums.
        n = random.randint(2, N - 1)
        while n in nums:                    # Finds a random number not already in nums.
            n = random.randint(2, N - 1)
        nums.append(n)
    for num in nums:                        # Calls mod_exp on N, using each of the a values picked above.
        if mod_exp(num, N - 1, N) != 1:     # If mod_exp ever returns a number other than 1, we know N is composite.
            return 'composite'
    for num in nums:                        # If a number passes the fermat tests, then we run carmichael test with same a values.
        if is_carmichael(N, nums[0]):
            return 'composite'              # Returning 'carmichael' here would report likely carmichael numbers.
    return 'prime'                          # Only executes if the number passed all fermat and carmichael tests.

# This function effeciently finds the value of x^y % N.
def mod_exp(x, y, N):
    if y == 0:                              # Base case
        return 1
    z = mod_exp(x, y//2, N)                 # Recursive call
    if y % 2 == 1:
        return (x * z**2) % N
    return z**2 % N


# This function finds the probability that the fermat test was correct.
# Simple formula. Each test we run is at least 50% likely to reveal a
# composite number, so the probability that a number we found to be prime
# is not actually prime is (1/2) ^ k. This means the probability it IS
# prime is 1 - 1 / (2^k).
def probability(k):
    return 1.0 - (1 / 2**k)

# This function performs the carmichael test as described in class.
def is_carmichael(N,a):
    u = N - 1
    t = 0
    while u % 2 == 0:                       # We find u and t values by dividing u by 2 as long as it is even.
        u /= 2
        t += 1
    y = N - 1                               # We set this to N - 1 so if the first x value comes out to 1, the function will return false
    for i in range(t + 1):                  # Continue calculating a ^ ((2 ^ i) * u) % N until the
                                            # result is 1. Then analyze the previous result.
        x = mod_exp(a, ((2**i) * u), N)
        if x == 1:
            if y != N - 1:
                return True
            else :
                return False
        else:
            y = x

```

Carmichael.py 3:85 LF UTF-8 Python master 33 files

Code with time complexity analysis:

Carmichael.py — ~/Documents/312Code/proj1-primality-test — Atom

fermat.py | Carmichael.py

```python
import random

# Total: O(k n^4)
def prime_test(N, k):                       # Assume parameters are n bits long.
    if k > N - 2:                           # Subtraction and comparison are O(n) time.
        k = N - 2                           # O(n) time
    if k < 0:                               # O(n) time
        k = 1
    nums = []                               # O(1) time
    for _ in range(k):                      # Loop executes k times, inner while loop happens O(n^2) times.
                                            # Total loop is O(k n^2)
        n = random.randint(2, N - 1)        # O(n) time to assign up to an n bit number
        while n in nums:                    # Executes no more than O(n^2) times
            n = random.randint(2, N - 1)    # O(n) time
        nums.append(n)                      # O(n) time
    for num in nums:                        # Runs k times and executes a O(n^3) function. Total time is O(k n^3).
        if mod_exp(num, N - 1, N) != 1:     # O(n^3) time
            return 'composite'
    for num in nums:                        # For loop runs k times and calls a O(n^4) function. Executes in O(k n^4) time.
        if is_carmichael(N, nums[0]):
            return 'composite'
    return 'prime'

# Total: O(n^3)
def mod_exp(x, y, N):                       # Function runs once in O(n^2) time. It may be called recursively up to log(N)
                                            # times, and log(N) is roughly equivalent to n. O(n^3)
    if y == 0:                              # Constant time
        return 1
    z = mod_exp(x, y//2, N)                 # The division is O(n^2) time, the function called recursively is O(n^2) time.
    if y % 2 == 1:                          # Modulo takes O(n^2) time
        return (x * z**2) % N               # This is multiple O(n^2) operations added together, which is still O(n^2) time.
    return z**2 % N                         # O(n^2) time

# Total: O(n^2)
def probability(k):
    return 1.0 - (1 / 2**k)                 # O(n^2) time

# Total: O(n^4)
def is_carmichael(N,a):                     # Assume all parameters have at most n bits. Total run time is O(n^4)
    u = N - 1                               # O(n) time
    t = 0                                   # O(1) time
    while u % 2 == 0:                       # Each loop cuts u in half, so it will execute O(log(N)) times which is
                                            # roughly equivalent to O(n). Longest step is O(n^2), so total time is O(n^3)
        u /= 2                              # O(n^2) time
        t += 1                              # O(1)
    y = N - 1                               # O(n) time.
    for i in range(t + 1):                  # Executes t + 1 times, longest step takes O(n^3) time, so we have O(t n^2
                                            # log(n)) run time where t is <= n
        x = mod_exp(a, ((2**i) * u), N)     # Function takes O(n^3) time to execute
        if x == 1:                          # Constant time
            if y != N - 1:                  # O(n) time
                return True                 # O(1) time
            else :                          # O(1) time
                return False                # O(1) time
        else:                               # O(1) time
            y = x                           # O(n) time
```

Carmichael.py 48:47 LF UTF-8 Python master 33 files

Code with Space completxity analysis:

Carmichael.py — ~/Documents/312Code/proj1-primality-test — Atom

fermat.py | Carmichael.py

```python
import random

# Total: O(n^2)
def prime_test(N, k):                       # Storing parameters is O(n) space
    if k > N - 2:
        k = N - 2                           # O(n) space.
    if k < 0:
        k = 1
    nums = []                               # Space will eventually be O(k * n) once it is populated.
    for _ in range(k):
        n = random.randint(2, N - 1)        # O(n) space
        while n in nums:
            n = random.randint(2, N - 1)    # O(n) space
        nums.append(n)                      # O(n) space
    for num in nums:                        # Space complexity is not a product of k because the stack is cleared
                                            # after each call to mod_exp.
        if mod_exp(num, N - 1, N) != 1:     # O(n^2) space to call mod_exp
            return 'composite'
    for num in nums:                        # O(n^2) for same reason as previous for loop.
        if is_carmichael(N, nums[0]):
            return 'composite'
    return 'prime'

# Total: O(n^2)
def mod_exp(x, y, N):                       # Storing parameters is O(n) space. Function calls itself up to n times,
                                            # so space is O(n^2)
    if y == 0:
        return 1
    z = mod_exp(x, y//2, N)                 # Function call is O(n^2) space
    if y % 2 == 1:
        return (x * z**2) % N
    return z**2 % N

# Total: O(n)
def probability(k):                         # Storing parameters requires O(n) space
    return 1.0 - (1 / 2**k)

# Total: O(n^2)
def is_carmichael(N,a):                     # O(n) space to store parameters
    u = N - 1                               # O(n) space
    t = 0
    while u % 2 == 0:
        u /= 2                              # u has already been assigned, no additional space.
        t += 1                              # This loop brings t up to O(n) space by the end.
    y = N - 1                               # O(n) space
    for i in range(t + 1):                  # O(n^2) space. Executing t times does not affect space because the
                                            # stack is reset.
        x = mod_exp(a, ((2**i) * u), N)     # O(n^2) space function.
        if x == 1:
            if y != N - 1:
                return True
            else :
                return False
        else:
            y = x                           # y has already been assigned, no additional space.
```

Carmichael.py 47:47 LF UTF-8 Python master 33 files

Part 4:

The probability equation was perhaps the easiest part of the project.

Each ‘a’ value that we use to test ‘N’ has at least a 50% chance of detecting a composite number. This means that if ‘k’ is the number of ‘a’ values you are using, you have a 0.5^k chance of missing a composite number, or a $1 – 0.5^k$ chance of detecting one. Thus, the probability that a number we have found is actually prime is $1 – 0.5^{k,}$, or $1 – 1/2^k$.