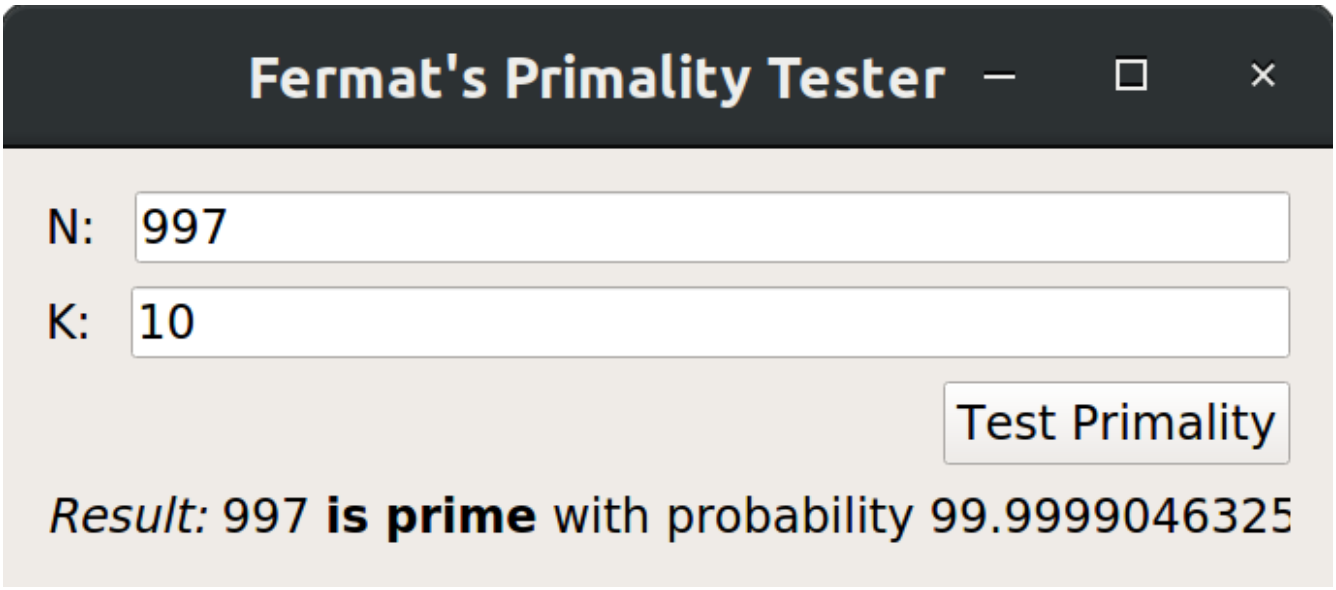
Lab 1 Write Up



Code That I Wrote For This Lab (fermat.py)

```
import random

##This function = O(n^3) + O(n^2) = O(n^3)
def prime_test(N, k):

	triedNumbers = set()

	# Create loop which generates k random numbers between 1 and N-1 and feeds them
	to our mod_exp function along with N-1 and N as number, power, and mod
	respectively.
	##This section of code = O(n) * O(n) = O(n^2)
	for i in range(k):
		rand = random.randint(1, N-1)
		##This section of code = O(n)
		shouldBe1 = mod_exp(rand, N-1, N)
		if shouldBe1 == 1:
			# During this loop, if the test returns 1, we will push it into an array of
			tried numbers to use again later.
			triedNumbers.add(rand)
		else:
			# If one of them returns something other than 1, then we know that the
			number isn't prime and return 'composite.'
			return 'composite'
```

```
        # If the loop completes successfully, we will then test if the number is a Carmichael
        number by iterating over the tested numbers and performing the Miller-Rabin tests
        (this will be done in the is_carmichael function).
        ##This section of code = O(n) * O(n^2) = O(n^3)
        for number in triedNumbers:
                ##This section of code = O(n^2)
                if is_carmichael(N, number):
                        return 'carmichael'
        # If all Carmichael tests return false, then we will return 'prime.' Otherwise we will
        return 'carmichael.'
        return 'prime'


##As y doubles in magnitude, each additional bit requires another recursive round to remove
it. This makes this function = O(n)
def mod_exp(x, y, N):
        # This function implements an algorithm to solve modular exponentiation problems
        more efficiently than simply evaluating the entire exponetial statement and then
        modding it, since this method does not scale well. Instead, we will first check if power
        == 0. If yes, we return 1.
        if y == 0:
                return 1
        # Otherwise, we recursively call this function but pass in half the value of power
        (integer division aka floor division). This results in a stack of recursive calls until power
        reaches zero, at which point the function returns 1 and begins to unwind.
        z = mod_exp(x, y//2, N)

        # Each function call then takes the value returned from the recursive layer under it and
        puts it into a new variable Z. The function then checks if power in the current context is
        even.
        if y % 2 == 0:
                # If it is, the function returns Z^2 % mod.
                return z**2 % N
        else:
                # If it isn't, then the function returns number * z^2 % mod.
                return (x * z**2) % N

        #The final layer will return the completed answer, having kept the size of the numbers
        involved down to a reasonable level by modding the results of every step, rather than
        just at the end.
```

```
##This code = O(1) because the number of steps performed doesn't vary based on the representative magnitude of k.
def probability(k):
    # Because roughly half all tests performed in the fermat test will fail on composite numbers, doing 1 test gives us a 1/2 chance of erroneously declaring a composite number as prime. Combining this test with the Miller-Rabin test catches at least 3/4 of false primes. In general, the probability of error is 1/(4^k), where k is the number of tests performed. This means that our function needs to return (1-(1/(4^k))) * 100
    return (1-(1/(4**k))) * 100


##This function = O(n) * O(n) = O(n^2)
def is_carmichael(N,a):

    power = N-1

    # This test will consist of iteratively calling the mod_exp function, but halving the exponent each time before passing in the arguments.
    while power % 2 == 0:
        power = power//2
        ## This section of code = O(n)
        result = mod_exp(a, power, N)
        # We continue until either the exponent cannot be divided by 2 again, or we obtain a number other than 1.
        if result != 1:
            # If the result is N-1, then we still don't know whether or not the number is prime. However, if it is any other number, we know that this number is a Carmichael number, and in either case we return the appropriate boolean value.
            isCarmichael = False if result == (N-1) else True
            return isCarmichael

    return False
```

On Time Complexity

While I noted the complexity of code snippets as well as overall functions in the code with double pound signs, I shall summarize them again here:

- mod_exp: O(n)
  - This is because as y doubles in magnitude, each additional bit takes another recursive round to remove it.
- is_carmichael: $O(n^2)$
  - This is because mod_exp, which is O(n), is called iteratively in a loop based on the magnitude of the original number passed in, which means the loop itself is O(n).
- prime_test: $O(n^3)$
  - This is because the is_carmichael function, which is $O(n^2)$, is called inside a loop based on the number of tests performed in the previous part of the function, which makes the loop itself O(n)
  - The Fermat test section of the code is $O(n^2)$, but since it is smaller in complexity than the carmichael test (mentioned in the previous bullet point), it is discounted.

    - Fermat test section is a loop based on the number of tests (O(n)), with a call to `mod_exp` inside of it (O(n)).
- `probability`: O(1), because the number of steps that are taken to calculate the chance of being correct based on the number of tests does not change depending on how many tests are performed.

On Determining Accuracy Of The Test

According to the book, performing Fermat's test once will identify about ½ of composite numbers. Performing this test more than once results in a $\frac{1}{2}^k$ chance of failure, with k being the number of tests performed. However, the book also states that performing fermat's test and checking for Carmichael numbers once identifies about ¾ of composite numbers. This means that performing the test more than once results in a $\frac{1}{4}^k$ chance of failure, with k being the number of tests performed. This means that the overall accuracy returned to the GUI is determined by $(1 - \frac{1}{4}^k) \times 100$.