## Camy Ngo – Lab 1 report

**Modular Exponentiation:**

****

Modular exponentiation is **exponentiation performed over a modulus**. In this case, we are trying to represent:



If *y* is 0, then x^y mod N must be 1, as anything to the power of 0 is 1. Let z:



We can then determine two possible outcomes via seeing if y is even. This check runs O(n) operations

If y is even:



If y is odd:

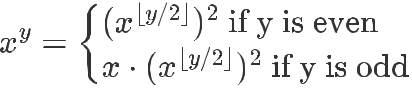


**Implementation:**

def mod\_exp(x, y, N):  
 if y == 0: return 1;  
 z = mod\_exp(x, y // 2, N);  
 # y is an even number -> N = z^2  
 if y % 2 == 0:  
 return z \*\* 2 % N  
 else:  
 return x \* (z \*\* 2) % N

**Complexity:**

Representing x as a power of y (z is merely a holder for, and another multiple of x^y):



This algorithm will terminate after n calls, assuming n bits is the maximum size of x, y or

N. The secondary case (if y is odd) in a worst-case scenario would yield a total running time of:



**Fermat primality tester:**

Fermat’s little theorem states that if p is a prime number, then for any integer a, the number a p – a is an integer multiple of p.



Where:



For a higher probability, a random integer between 1 and p-1 will be picked. N is a value to be tested for primality. With this implementation, the modular exponentiation will be running with O(n^3) complexity:



**Implementation:**

def fermat(N, k):  
 # if N is an even number return composite  
 if (N % 2) == 0:  
 return 'composite'  
  
 # Run through the number of tests  
 for i in range(0, k):  
 ran = random.randint(1, N - 1)  
 # If the result of the modular exponentiation is not 1 we know the number is composite  
 if mod\_exp(ran, N - 1, N) != 1:  
 return 'composite'  
 # reduce the value of k for every loop  
 k -= 1  
 return 'prime'

**Complexity:**

All of the operations and comparisons in the algorithm run at constant time, except for the modular exponentiation which runs at O(n^3). Thus, the time-complexity is:



According to the rule of dominate growth of the linear factor, this k can be factor out, leading to run time of:



**Miller-Rabin primality test**

****

Like the Fermat test, the Miller-Rabin test determines if a number is likely to be prime based on corresponding relations.

Given an odd integer n > 2, *n* can be represented as the following polynomial:



Where *s, d* are both positive integers, and *d* is odd. Choosing some base *a* in **one of the following congruences**, *n* is **strongly probable** relative prime to *a*:



**Implementation:**

When choosing a value for base *a*, a random value between 1 and n-1 will be chosen.

# Time complexity of k\*n^2 because we call mod\_exp (which is order n^3) k times.  
def miller\_rabin(N, k):  
 for i in range(0, k):  
 # If the exponent is odd we want to move on to test the next base  
 if N % 2 == 0:  
 return "composite"  
 ran = random.randint(2, N - 2)  
 b = mod\_exp(ran, N - 1, N)  
 if b == 1 or b == -1:  
 return "prime"  
 # if the result of mod\_exp is not one and also not N - 1 -> test as N-1  
 else:  
 return miller\_helper(b, N - 1, N)  
 # reduce the value of k for every loop  
 k -= 1

def miller\_helper(base, power, N):  
 result = mod\_exp(base, power, N)  
 if result % 2 == 0:  
 return "composite"  
 if result == (N - 1):  
 return "prime"  
 elif result != 1:  
 return miller\_helper(result, power // 2, N)

**Complexity:**

By following the steps of the pseudocode, we having the following complexity:



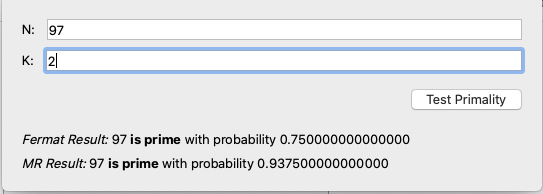
where n^6 dominates. This is run *k* times:



Again, n^6 dominates, yielding an overall time complexity of O(n^6) complexity.

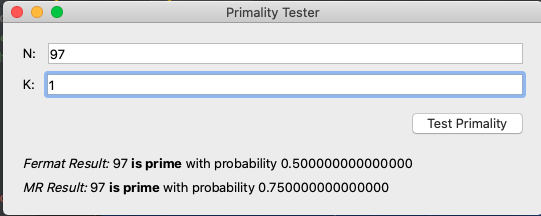
**Fermat Error Probability:**

The formula for calculating the probability that the fermat algorithm is correct is written as: 1 - 1/2^k, where k represents the number of tests performed. If you are testing a number N, a^N-1 = 1mod N for at most half of the values between a and N. This means that you have a 50% chance of revealing a composite N if you choose one value for a. If you perform k = 2 tests your probability of being correct is 1 - (1/2 \* 1/2). This would be true for any value of k > 0 that you choose



**Miller Rabin Probability:**

The Miller Rabin test combines the Fermat algorithm with a square root test on the exponent to give us a more in depth test. In this case, if you are testing a number N, 3/4 of the values between 1 and N - 1 will be able to reveal a composite N. This means that we have a 75% chance of revealing a composite if we undergo one test of N. If we increase the number of tests to 2, our probability of revealing a composite N is 1 - (1/4 \* 1/4). The probability formula can be modeled as 1 - 3/4^k for k > 0



**Probability Functions**

Function fprobability(k): Fermat probability

* Return 1 - (1/2^k)
* explanation: Time complexity of O(kn^2) because we are multiplying 1/2, which is order O(n^2), k times.

Function mprobability(k): Miller Rabin probability

* Return 1 - (1/4^k)
* mprobability explanation: Time complexity of O(kn^2) because we are multiplying 1/4, which is order O(n^2), k times

ALL final code:

import random  
  
  
# If n is prime, then always returns true,  
# If n is composite than returns false with  
# high probability Higher value of k increases  
# probability of correct result  
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)  
  
  
def mod\_exp(x, y, N):  
 if y == 0: return 1;  
 z = mod\_exp(x, y // 2, N);  
 # y is an even number -> N = z^2  
 if y % 2 == 0:  
 return z \*\* 2 % N  
 else:  
 return x \* (z \*\* 2) % N  
  
  
# Time complexity of n^2 with integer k which is size n bits  
def fprobability(k):  
 # You will need to implement this function and change the return value.   
 return 1 - (1 / 2 \*\* k)  
  
  
# Time complexity of n^2 with integer k which is size n bits  
def mprobability(k):  
 # You will need to implement this function and change the return value.   
 return 1 - (4 \*\* -k)  
  
  
# Time complexity of k\*n^3 because we call mod\_exp (which is order n^3) k times  
def fermat(N, k):  
 # if N is an even number return composite  
 if (N % 2) == 0:  
 return 'composite'  
  
 # Run through the number of tests  
 for i in range(0, k):  
 ran = random.randint(1, N - 1)  
 # If the result of the modular exponentiation is not 1 we know the number is composite  
 if mod\_exp(ran, N - 1, N) != 1:  
 return 'composite'  
 # reduce the value of k for every loop  
 k -= 1  
 return 'prime'  
  
  
def miller\_helper(base, power, N):  
 result = mod\_exp(base, power, N)  
 if result % 2 == 0:  
 return "composite"  
 if result == (N - 1):  
 return "prime"  
 elif result != 1:  
 return miller\_helper(result, power // 2, N)  
  
  
# Time complexity of k\*n^2 because we call mod\_exp (which is order n^3) k times.  
def miller\_rabin(N, k):  
 for i in range(0, k):  
 # If the exponent is odd we want to move on to test the next base  
 if N % 2 == 0:  
 return "composite"  
 ran = random.randint(2, N - 2)  
 b = mod\_exp(ran, N - 1, N)  
 if b == 1 or b == -1:  
 return "prime"  
 # if the result of mod\_exp is not one and also not N - 1 -> test as N-1  
 else:  
 return miller\_helper(b, N - 1, N)  
 # reduce the value of k for every loop  
 k -= 1