Noah Schill -- Lab 1 Submission

Modular Exponentiation

Modular exponentiation is the process of representing numbers via an exponentiation using modulus. In this case, we are trying to represent:

$$x^y \mod N$$

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

$$z = x^{\lfloor y/2 \rfloor} \mod N$$

We can then determine two possible outcomes via seeing if y is even. This check takes constant time.

If y is even:

$$x^y \mod N = z^2 \mod n$$

If y is odd:

$$x^y \mod N = xz^2 \mod N$$

Implementation

```
def mod_exp(x, y, N):
  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
```

Complexity

Complexity can be determined through analyzing the branches of the algorithm and what happens to the exponent *y*. Representing x as a power of y (z is merely a holder for, and another multiple of x^y):

$$x^y = \begin{cases} (x^{\lfloor y/2 \rfloor})^2 \text{ if y is even} \\ x \cdot (x^{\lfloor y/2 \rfloor})^2 \text{ if y is odd} \end{cases}$$

At the very most, 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:

$$O(n^3)$$

##Fermat primality test
The Fermat primality test is a way to determine if a number is prime probabilistically. It is based on the famous *Fermat's Little Theorem* which states that if p is prime and a is not divisible by p, then:

$$a^{p-1} \equiv 1 \mod p$$

Where:

$$1 < a < p-1$$

Conveniently, this is analogous to the above modular exponentiation representation, so that function will be utilized. For a, a random integer between 1 and $p-1$ will be picked. N is a value to be tested for primality. At this point, the modular exponentiation can be invoked in the following fashion ($O(n^3)$ complexity):

$$a^{N-1} \mod N$$

After running this *k* times, if it never is equal to 1, then *N* is composite. Otherwise, *N* is *probably* prime.

Implementation

```
def fermat(N,k):
    # returns either 'prime' or 'composite'.
    for i in range(k):
        a = random.randint(1, N-1)
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
    return 'prime'
```

Complexity

All of the operations and comparisons in the algorithm run at constant time, except for the modular exponentiation which runs at $O(n^3)$. This operation is being run *k* times as a result of the loop, introducing a linear dependency. Thus, the time-complexity is:

$$O(kn^3)$$

This description could be diluted, as the exponential term will dominate the growth of the linear factor:

$$O(n^3)$$

Accuracy of Fermat test

In a best-case scenario, the Fermat test's accuracy has the following relationship with *k* iterations of the test.

$$2^k$$

This is promising -- accuracy grows incredibly with an increase in *k*, a trivial task for modern computers. However, the Fermat primality test comes with a few caveats. There are some *p* values that will satisfy this theorem, but which are composite. In this case, the *a* is called a *Fermat liar*, and the *p* is called *Fermat witness*. There are infinitely many of these Fermat pseudo-primes which will break this test. In these scenarios, the Fermat test performs no better than a random guess.

To add insult to injury, Carmichael numbers cannot be neglected. These are *n* values in this representation for which all exponents are Fermat liars. Thankfully, for this reason the Miller-Rabin test provides a better solution.

Miller-Rabin primality test

Much like the Fermat test, the Miller-Rabin test determines if a number is likely to be prime based on congruence relations. It's built off of the Fermat test, but uses more strict parameters to determine primality.

The test is defined as following. Given an odd integer n > 2, *n* can be represented as the following polynomial:

$$n = 2^s \cdot d + 1$$

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*:

$$a^d \equiv 1 \mod n$$

$$a^{2^r \cdot d} \equiv -1 \mod n \text{ (for some } 0 \leq r < s).$$

The Miller-Rabin this adds an extra case as a filter to remove many of the anomalies that passed through the Fermat test. While this is better, there are still certain n composites that can slip through the cracks (called *strong pseudoprimes*). This test will only tell the *probability* that n is prime, however accurate it may be, but is not conclusive.

When choosing a value for base *a*, a random value between 1 and n-1 will be chosen. This is done in lieu of testing every possible *a*, which would be inefficient.

Implementation

While mathematically analogous, the recursive implementation of this test is more difficult. A recursive "helper" function is used to do the majority of the work. This pseudocode will call the previous modular exponentiation implementation above.

```python
def b_helper(b, power, n):
    b_n = mod_exp(b, power, n)
    if b_n % 2 == 0:
        return "composite"
    if b_n == (n - 1):
        return "prime"
    elif b_n != 1:
        return b_helper(b_n, power // 2, n)
```

> Note how the power is being halved before each recursive call. This means that by the time `b_n` is calculated after a recursive call, this would take O(n^6) time to perform, unfortunately.

The main function serves as a driver choosing a random base `a` within stated parameters. It first checks the initial `b_0` from modular exponentiation is prime or not. It also checks if the test value is even, which would save runtime in those cases avoiding having to make recursive calls for a trivial problem. It runs the test `k` times.

```python
def miller_rabin(N,k):
    for i in range(k):
        if N % 2 == 0:
            return "composite"
        a = random.randint(2, N - 2)
        b_0 = mod_exp(a, N - 1, N)
        if b_0 == 1 or b_0 == -1:
            return "prime"
        else:
            return b_helper(b_0, N - 1, N)
    return 'composite'
```

Complexity of Miller-Rabin Test

Evaluating time complexity for Miller-Rabin test is multifaceted, but by following the steps of the pseudocode, the following steps:

$$1 + 1 + O(n^3) + 1 + O(n^6)$$
$$O(n^3) + O(n^6) = O(n^3 + n^6) = O(n^6)$$

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

$$kO(n^6) = O(kn^6)$$

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

Accuracy of Miller-Rabin Test

The Miller-Rabin has a much improved accuracy over the Fermat test. The error bound that a number determined prime by the test and is not a *strong pseudoprime* is:

$$4^k$$

Appendix : Full python file

```
import random


def prime_test(N, k):
  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)
  if y%2==0:
    return z**2 % N
  else:
    return x*(z**2) % N


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


def mprobability(k):
    return 1-(4 ** -k)


def fermat(N,k):
    for i in range(0, k):
        a = random.randint(1, N-1)
        if mod_exp(a, N-1, N) != 1:
            return 'composite'
    return 'prime'


def b_helper(b, power, n):
    b_n = mod_exp(b, power, n)
    if b_n % 2 == 0:
        return "composite"
    if b_n == (n - 1):
```

```python
            return "prime"
        elif b_n != 1:
            return b_helper(b_n, power // 2, n)


def miller_rabin(N,k):
    for i in range(k):
        if N % 2 == 0:
            return "composite"
        a = random.randint(2, N - 2)
        b_0 = mod_exp(a, N - 1, N)
        if b_0 == 1 or b_0 == -1:
            return "prime"
        else:
            return b_helper(b_0, N - 1, N)
```