Gideon Hale 2024-01-22 Monday

Project 1 Report: Fermat

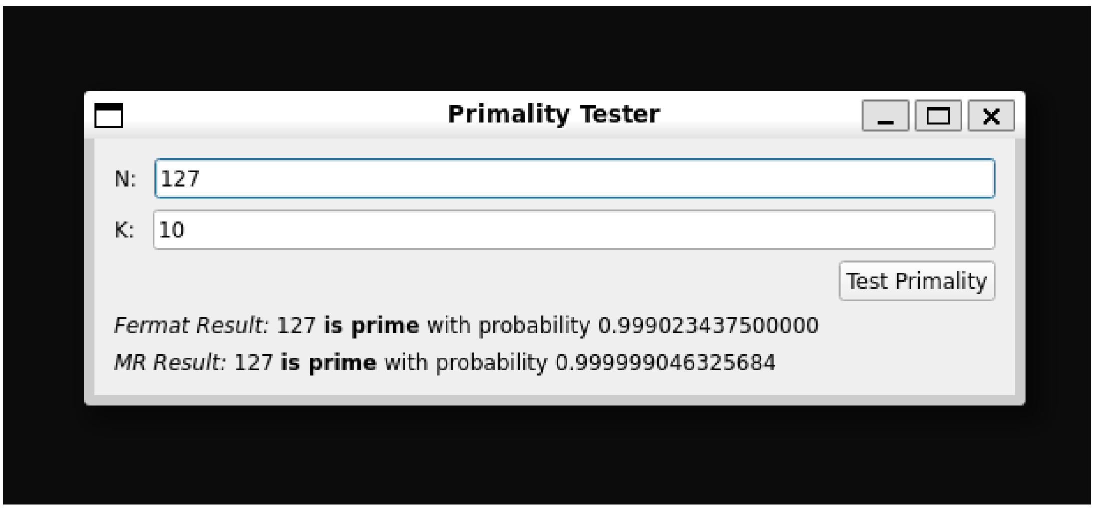
1. Code

See Appendix. :)

2. Complexity

1. mod_exp()
    a. Let n be the length of the exponent input.
    b. assume the exponent is larger than or equal to both the base and the modulus in order to evaluate the scenario with the largest complexity
    c. each time through the function:
        i. two divisions inputs in the recursive step - $O(n^2)$ time, $O(n^2)$ space
        ii. one or two multiplications in the return - $O(n^2)$ time, $O(n)$ space
        iii. one division in the return - $O(n^2)$ time, $O(n^2)$ space
    d. exponent value is halved from one call to the next, thereby in binary removing one bit, thus mod_exp() is called n times
    e. This makes for a total complexity of $O(n^3)$ time and $O(n^3)$ space
2. fermat()
    a. Let n be the length of queryNum.
    b. assume numIter is relatively small
    c. assume generating a random number takes constant time
    d. mod_exp() gets called numIter (constant) times, with queryNum as its input, giving fermat() a time and space complexity of $O(n^3)$.

3. Example



4. Experimentation

For testing purposes, I created a script that runs either algorithm fifteen or thirty times in a row to see if it got the same answer every time. When I input a Carmichael number (such as 561 or 1105) for N and a low number (<5) for K, some (or most) of the Fermat Test results would declare prime, whereas a few of them would declare composite. The lower the k-value, the more false primes.

For the Miller-Rabin Test, the results become unanimous, showing its superiority against Carmichael numbers.

5. Equations

The probability equations were easy.

The fprobability() function was derived in the book from the proportion of composite numbers less than N whose modular exponentiation would equal one, which is ≤ ½. By repeating the test with k (numIter in the code) random values of a, the ½ is multiplied by itself k times, which gets exceedingly small. Now to find the probability that the answer *is* correct, we subtract this value from 1.

The mprobability() function is exactly the same idea, except for Miller-Rabin, the probability of being accurate is ¾ each time, and by subtracting that from 1, we have ¼ probability each time of getting the answer wrong. This raised to the power k reduces even faster than fprobability().

6. Appendix

```
import random

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

# perform modular exponentiation
def mod_exp(base, exponent, modulus):

    if exponent == 0: return 1

    z = mod_exp(base, exponent // 2, modulus)

    if exponent % 2 == 0: return (z**2) % modulus
    else: return (base * z**2) % modulus
```

```python
# using Fermat's Little Theorem, find out to a certain probability if primeCandidate is prime
def fermat(queryNum, numIter):

    for i in range(numIter):
        base = random.randint(1, queryNum - 1)

        if mod_exp(base, queryNum - 1, queryNum) != 1: return "composite"

    return "prime"

# determine the probability that fermat() actually produces a correct answer
def fprobability(numIter):
    return 1.0 - (1 / 2)**numIter

# using the Miller-Rabin Test, find out to a certain probability if primeCandidate is prime
def miller_rabin(queryNum, numIter):

    for i in range(numIter):
        base = random.randint(1, queryNum - 1)\

        runningExp = queryNum - 1
        runningModExp = 1
        while runningModExp == 1:

            runningModExp = mod_exp(base, runningExp, queryNum)

            if runningModExp != 1:
                if runningModExp != queryNum - 1: return "composite"

            if runningExp % 2 != 0:
                break

            runningExp = runningExp / 2

    return "prime"

# determine the probability that miller_rabin() actually produces a correct answer
def mprobability(numIter):
    return 1.0 - (1 / 4)**numIter
```