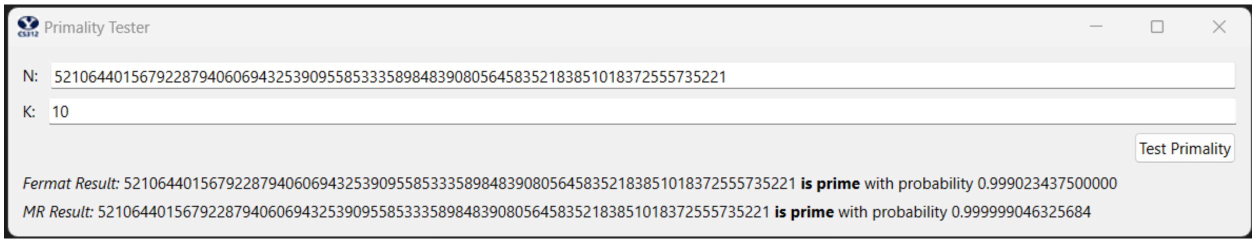
CS 312 – Fermat Project

Braden Webb

Report: Your report should consist of:

1. [10 points] At least one screenshot of your application with a working example (distinct from the one above).



N: 5210644015679228794060694325390955853335898483908056458352183851018372555735221

K: 10

Fermat Result: 5210644015679228794060694325390955853335898483908056458352183851018372555735221 **is prime** with probability 0.999023437500000

MR Result: 5210644015679228794060694325390955853335898483908056458352183851018372555735221 **is prime** with probability 0.999999046325684

2. All of the code that you wrote. If your code spans multiple files, make sure that you include all of it in your PDF appendix, and make sure that it is organized in a way that makes it easy for the reader (TA) to follow. Your code must include the following:

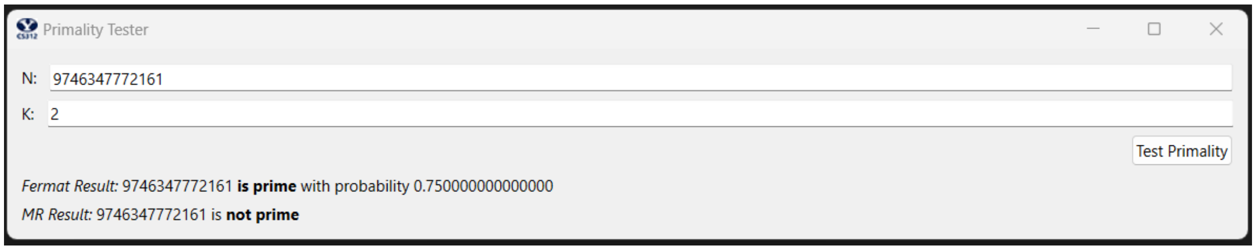
a. [20 points] A correct implementation of modular exponentiation.

b. [20 points] A correct implementation of the Fermat primality tester.

c. [10 points] A correct implementation of the Miller-Rabin algorithm.

I've attached all of the code to the end of this pdf.

3. [10 points] A brief discussion of some experimentation you did to identify inputs for which the two algorithms disagree. Give a good effort to explain why they disagree. Include a screenshot showing a case of disagreement.



N: 9746347772161

K: 2

Fermat Result: 9746347772161 **is prime** with probability 0.750000000000000

MR Result: 9746347772161 is **not prime**

This example, where the two algorithms disagree, occurs because this is a Carmichael number. As such, the probability 1 - .5^k doesn't apply for the Fermat test, and the Miller-Rabin algorithm is more reliable.

I've found that the Fermat test often works for Carmichael numbers as well; however, it is not *guaranteed* to work with the same probability as for other composite numbers. Usually, if I try composite numbers with a large enough k value (say, >= 10) both tests will almost always get the correct answer. They are far more likely to disagree with small k.

4. [10 points]. Discuss the time and space complexity of your algorithm. You must demonstrate that you really understand the complexity and which parts of your program lead to that complexity. You may do this by:

> **a. Showing and summing up the complexity of each significant subsection of your code, or**
>
> **b. Creating brief pseudocode showing the critical complexity portions, or**
>
> **c. Using another approach of your choice. For whichever approach you choose, include sufficient discussion/explanation to demonstrate your understanding of the complexity of the entire problem and any significant subparts.**

I provide small code snippets to save space here, but my actual code is much more complete and well-documented at the appendix at the end.

First, we analyze `mod_exp(x,y,N)`:

```
def mod_exp(x, y, N):
    if y == 0:
        return 1
    z = mod_exp(x, y // 2, N)
        return (z**2) % N
    else:
        return (x * z**2) % N
```

For time complexity, we observe that we recurse until y==0, each time multiplying y by ½. Thinking in binary, we see that this is equivalent to chopping off 1 digit from y for each call. As y is an n-bit integer, this means that we recurse n times. Additionally, we need to perform a multiplication of n-bit numbers each time, which is an O(n^2) operation, so our overall time complexity is O(n * n^2) = O(n^3).

For space complexity, we note that each time we recurse, we need to store the new variable z as well as a new copy of y//2. Since each of these is of size O(n), and we recurse O(n) times, our overall space complexity is O(n^2).

Next, we analyze the Fermat test:

```
def fermat(N,k):

    for _ in range(k): # Run the test k times
        a = random.randint(1, N - 1)
        if mod_exp(a, N - 1, N) != 1:
            return 'composite'
    return 'prime'
```

For time complexity, we note that we loop k times through the test. However, since k is generally a very small integer (k << n), we can think of it as a constant factor that won't affect our overall complexity. Within this loop, we select an arbitrary number 1 <= a <= N – 1, a process that we take to be O(1). We then run the modular exponentiation algorithm, which we have already determined to be O(n^3) complexity. Thus, our overall time complexity for this algorithm is O(n^3).

For space complexity, we see that looping through the k different tests doesn't affect the memory usage, and within each test, storing a only introduces an O(n) sized value. Thus, our memory usage also reduces to that of modular exponentiation, which is O(n^2).

Finally, we look to the Miller-Rabin algorithm:

```
def miller_rabin(N,k):
    for _ in range(k):
        a = random.randint(1, N - 1)
        if mod_exp(a, N - 1, N) != 1:
            return 'composite'
        e = N - 1
        while(e % 2 == 0):
            e //= 2
            x = mod_exp(a, e, N)
            if x == N - 1:
                break
            elif x != 1:
                return 'composite'
    return 'prime'
```

For time complexity, looping through the k iterations of the test only introduces a constant factor, since k << n. Generating a is an O(1) operation, and running modular exponentiation is an O(n^3) operation. Since e is likely an n-bit integer, in the case where e is a power of 2 (or near to it) we will have to loop through the while loop O(n) times. Each time, computing e //2 is an O(n) operation and computing mod_exp is O(n^3). Thus, our overall time complexity is O(k[n * (n^3 + n)]) = O(n^4).

For space complexity, storing a and e are each takes up O(n) bits, and mod_exp is O(n^2). However, since e is overwritten for each iteration of the while loop, this doesn't introduce any additional memory complexity, and our overall usage remains O(n^2).

5. [10 points] Discuss the two equations you used to compute the probabilities p of correctness for the two algorithms (Fermat and Miller-Rabin).

For both equations, I relied on the theorem from probability theory that $P(A \cap B) = P(A)P(B)$ when A and B are independent events. Additionally, we know with 100% certainty that a number is composite if it fails a test, so there is no need to call or calculate that probability when a number fails the test. We only call `fprobability()` or `mprobability()` when a number passes the test.

For the Fermat algorithm, the probability of that a^(N-1)=1 mod N when N is not prime is at most .5. Thus, the probability of k arbitrary numbers between 1 and N – 1 having this property, for N composite, is at most .5^k. Since the probability that a number N is prime given that it passed the test k times is the complement of the probability that the number N is composite given that it passed the test k times, it makes sense to calculate this probability as 1 - .5^k.

Similar logic applies to the Miller-Rabin algorithm, but we have a stronger initial probability where

$P\left(\mathrm{a}^{\mathrm{N}-1} = 1 \text{ mod N } \middle| \text{N is composite}\right) \leq$ .25, and as such
$P(N\ is\ prime\ |N\ passes\ the\ MR\ test\ k\ times) \geq 1 - .25^k$.

Appendix

My code:

```
import random

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):
    """
    Calculates the expression x^y (mod N) using recursion

    Overall time complexity: O(n^3)
    Overall space complexity: O(n^2)
    """
    if y == 0: # If we've reached the base case, return 1
        return 1
    z = mod_exp(x, y // 2, N) # Recurse O(n) times
    if y % 2 == 0:
        return (z**2) % N # Multiplication of n-bit numbers is O(n^2)
    else:
        return (x * z**2) % N # Multiplication of n-bit numbers is O(n^2)


def fprobability(k):
    """
    Calculates the probability that an integer is prime given that it
passed the Fermat
    test k times.

    Overall time complexity: O(1)
    Overall space complexity: O(1)
    (since both are with respect to n = logN, and k << n)
    """
    return 1 - .5**k


def mprobability(k):
    """
    Calculates the probability that an integer is prime given that it passed the Miller-Rabin
    test k times.

    Overall time complexity: O(1)
    Overall space complexity: O(1)
    (since both are with respect to n = logN, and k << n)
    """
    return 1 - .25**k


def fermat(N,k):
    """
    Applies Fermat's little theorem k times to test whether a given integer N is prime.

    Overall time complexity: O(n^3)
    Overall space complexity: O(n^2)
    """
    for _ in range(k): # Run the test k times
        a = random.randint(1, N - 1) # We take this to be an O(1) operation
        if mod_exp(a, N - 1, N) != 1: # If a^(N - 1) % N is not 1, we know N is composite
            return 'composite'
    return 'prime' # If it never failed for k iterations, we believe it to be prime


def miller_rabin(N,k):
    """
    Applies the Miller-Rabin test k times to determine whether a given integer N is prime.

    Overall time complexity: O(n^4)
    Overall space complexity: O(n^2)
    """
    for _ in range(k): # Run the test k times
        a = random.randint(1, N - 1) # We take this to be an O(1) operation
        if mod_exp(a, N - 1, N) != 1: # Briefly apply the Fermat test; O(n^3)
            return 'composite'
```

```python
        e = N - 1
        while(e % 2 == 0): # End once we get to an odd exponent: O(n)
            e //= 2 # Since e is even, it doesn't matter if we use /= or //= . This is O(n)
            x = mod_exp(a, e, N) # Time complexity of mod_exp is O(n^3)
            if x == N - 1: # This is what we should expect for primes
                break
            elif x != 1: # This guarantees that N is composite
                return 'composite'
    return 'prime' # If it never failed for k iterations, we believe it to be prime
```