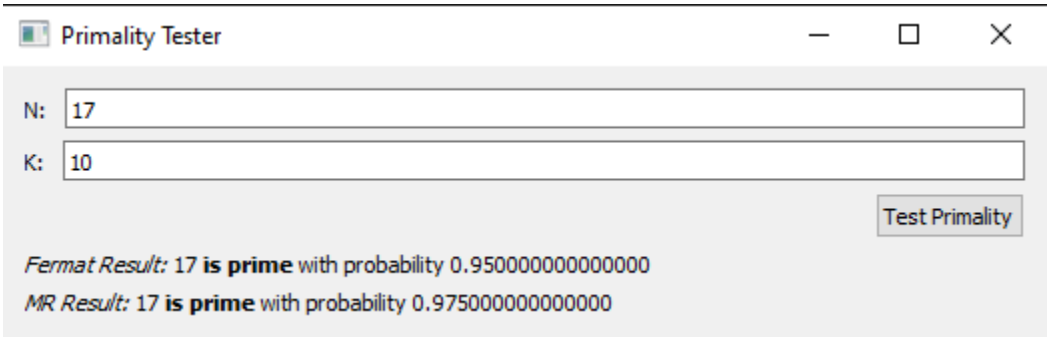
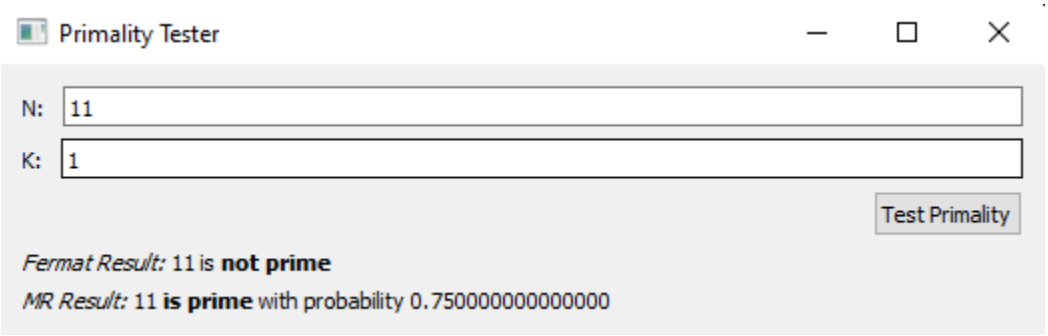
Thomas Neyman
CS 4412 - Advanced Algorithms
January 21 2022
Project 1

Fermat and Miller Rabin Primality Testing

Program screenshots:



Here we can see that 17 if very likely prime thanks to the number of iterations k we performed on the number



Here, 11 got past the test for the Fermat equation because not enough iterations of the algorithm were tried, thus the probability for determining if it was prime was lower.

Code Written:

(All code can be seen at https://github.com/NeymanThomas/4412-Projects/tree/main/Project-1 as well)

```
def mod_exp(x, y, N):
    # This function handles calculating (a^N-1) % N
    #
    # <PSEUDO CODE>
    # function modexp(x, y, N)
    # Input: Two n-bit integers x and N, an integer exponent y
```

```
    # Output: x^y % N
    #
    # if y = 0: return 1
    # z = modexp(x, |y/2|, N)
    # if y is even:
    #   return z^2 % N
    # else:
    #   return x * z^2 % 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):
    # Finding the probability for Fermat Algorithm is extremely simple.
    # Because the function has a one-to-one relation with elements that
    # fail the test and elements that pass, this implies there is a 1/2
    # chance the test will fail. Increasing the amount of tests done
    # exponentially decreases this chance by changing the probability to
    # 1 / 2 * k where k is the number of tests run.
    return 1 - ( 1 / (2 * k))


def mprobability(k):
    # For this test, the more bases of a that are tried, the more accurate
    # the test will be. If N is composite, then at most 1/4 of the bases are
    # strong liars for N. So compounding this probability with k gives you
    # a probability of 1 / 4 * k error. This is superior to the Fermat test.
    return 1 - (1 / (4 * k))

def fermat(N,k):
    # This function uses the Fermat algorithm to determine if a number is prime
or composite.
```

```
    # k represents the amount of iterations when testing if the input 'N' is
    # prime or not. The larger k, the more precise the algorithm. the 'a' value
is chosen
    # at random with the random.randint() method. this is done k number of
iterations.
    # If the algorithm finds a case where a^(N-1)%1 == 1 it will return prime
    # early, otherwise it will return composite.
    #
    # <PSEUDO CODE>
    # function primality(N)
    # Input: Positive integer N
    # Output: yes/no
    #
    # Pick a positive integer a < N at random
    # if a^(N-1) % N = 1:
    #   return yes
    # else:
    #   return no

    # the algorithm doesn't work for numbers 1 - 4, so we have to just check
    # those values manually
    if N == 1 or N == 4:
        return 'composite'
    elif N == 2 or N == 3:
        return 'prime'
    else:
        for i in range(k):
            # it doesn't make sense to mod by 1, so start at 2
            a = random.randint(2, N - 2)

            if mod_exp(a, N - 1, N) == 1:
                return 'prime'
    return 'composite'



def miller_rabin(N,k):
    # this function did not have any pseudo code provided in the text but it also
    # functions quite similarly to the Fermat test.
```

```
    # once again we need to check manually for values 4 or less
    if N == 1 or N == 4:
        return 'composite'
    elif N == 2 or N == 3:
        return 'prime'
    else:
        # integer 'a' will need to be 1 < a < N - 1
        # we are trying to achieve
        # - a^d = 1 (mod N)     or
        # - a^(2^r)d = -1 (mod N) for some 0 <= r <= s
        # when N is an odd prime it will pass because of Fermat's little theorem
where
        # - a^(n-1) = 1 (mod N)
        # the only only square roots of 1 mod N are 1 and -1

        # We need to factor out powers of 2 from N - 1 so that we end up with 2^r
* d + 1
        d = N - 1
        while (d % 2 == 0):
            d //= 2

        # now we loop for k iterations
        for i in range(k):
            # a is once again our random value
            a = random.randint(2, N - 2)
            # x holds the value for a^d mod N
            x = mod_exp(a, d, N)
            # if x returned as 1 or N - 1, we can exit early and know it's prime
            if (x == 1 or x == N - 1):
                return 'prime'

            # loop an r number of times
            while (d != N - 1):
                # x = x^2 mod N
                x = (x * x) % N
                d *= 2

                if (x == 1):
```

```
                return 'composite'
            if (x == N - 1):
                return 'prime'

        return 'composite'
```

Experimentation:

In order to test the code, various inputs with various k iteration values were entered over and over. First I tested composite numbers over and over to see if the test somehow slipped up and said one of them was prime. However the test was able to identify a composite number 100% of the time without variance from the k value.

Next, I needed to test prime numbers, so I entered prime numbers into the program starting off with large k values like 20 - 10. This showed that the program very reliably correctly identified the number was prime with very few cases of it misidentifying usually from the Fermat Test.

Then when lowering the amount of iterations drastically to 1 or 2, the program became much less consistent with identifying prime numbers correctly. Of course the Fermat algorithm failed more often, but not by a very large margin. This showed to me the algorithms were implemented correctly and were operating as expected.

Time and Space Complexity:

Modular exponentiation:

```
if y == 0:                          # This is a single check so O(1)
    return 1

 z = mod_exp(x, y // 2, N)          # Here, recursion is occurring at most log N times
                                    # so we get O(log n)
 if y % 2 == 0:                     # modulus / multiplication occur as many times as
                                    # many times as recursion calls, so O(log n)
    return (z ** 2) % N
 else:
    return x * (z ** 2) % N
```

This shows a time complexity of O(log n) since the function recursively calculates log N steps. The space complexity is O(n log n) because it needs to hold memory for the size of the input N plus each recursive call.

Fermat's Theorem:

```
    if N == 1 or N == 4:                        # this is a simple O(1) call
        return 'composite'
    elif N == 2 or N == 3:
        return 'prime'
    Else:                                       # The loop is called k times, so
                                                # O(k) or O(n)
        for i in range(k):
            a = random.randint(2, N - 2)        # simple O(1)

            if mod_exp(a, N - 1, N) == 1:       # We know the function will take
                                                # O(log N) time to finish
                return 'prime'
    return 'composite'
```

Because we have a function that requires O(log n) time inside a loop of k iterations, the time complexity total will be O(k log n), or O(n log n). a will be created k number of times and we know that the function requires O(n log n) memory, so compounded the space complexity is O(n log n).

Miller Rabin Algorithm:

```
d = N - 1
while (d % 2 == 0):                     # O(log n)
     d //= 2
     for i in range(k):                 # O(k) or O(n)
          a = random.randint(2, N - 2)
          x = mod_exp(a, d, N)          # O(log n)
          if (x == 1 or x == N - 1):    # O(1)
              return 'prime'

          while (d != N - 1):           # O(log n)
              x = (x * x) % N
              d *= 2
              if (x == 1):
                  return 'composite'
              if (x == N - 1):
                  return 'prime'
          return 'composite'
```

The miller rabin algorithm has more lines of code, but when compounded together to find the time complexity, we are still left with O(n log n). This is due to having an O(log n) algorithm nested inside a loop of O(n) iterations. The space complexity is similar to the Fermat test and won't exceed O(n log n) space as well. The loop creating a base 'a' each time compounded with the function gives us that value.

Algorithms for Probability:

Fermat: for *n*, if *n* = a composite integer then at least of all gcd(a, n) are fermat witnesses. We can show this by

$(a * b) \wedge N - 1 \equiv a \wedge N - 1 * b \wedge N - 1 \equiv a \wedge N - 1 \not\equiv 1 \bmod N$

So basically there is a one-to-one mapping of values that pass and fail the test in the set of N -1. Therefore, the probability of a test passing with the Fermat algorithm is 1 / (2 * k). The probability gets exponentially smaller with extra iterations.