Laboratory 3

Mos Daniela, 935

Miller-Rabin primality test

This algorithm checks whether a given number is likely to be prime or if it is composite.

Prerequisites

Property

$b^2 \equiv 1$ (mod n), $n \in N$, n > 2 a prime number, where the only solutions are 1 and -1. To prove it, this can be rewritten as $b^2 - 1 \equiv 0$ (mod n) $\Leftrightarrow (b-1)(b+1) \equiv$ (mod n) $\Leftrightarrow n|(b-1)$ or $n|(b+1)$, then b is congruent with 1 or -1 (mod n). In other words, there are no non-trivial square roots for 1 (mod n) when n is prime.

Fermat's little theorem

$b^{n-1} \equiv 1$ (mod n), $n \in N$ prime and gcd(b, n) = 1.

Reapeated squaring modular exponentiation

This algorithm is used for quickly computing large powers by repeatedly squaring the given number mod n. For exponents that are not powers of two, we are using the fact that $x^a * x^b = x^{a+b}$, so the given exponent will be written using the values previously obtained and the number will be the product of the corresponding powers mod n. The algorithm is presented below, as well as the functions that will be used to put together the resulting number:

```
<<Repeated squaring>>=
def repeated_sqr(nr, pow, n):
    numbers = [nr % n]
    powers = {1: numbers[-1]}
    i = 2
    while i < pow + 1:
        new_nr = numbers[-1] ** 2 % n
        numbers.append(new_nr)
        powers[i] = new_nr
        i *= 2
    return powers
```

```
def sum_powers(power, power_list):
    pws = []
    for number in power_list:
        if (power - number) >= 0:
            pws.append(number)
            power -= number
    return pws


def number_product(powers, sum_powers, n):
    result = 1
    for pw in sum_powers:
        result = (result * powers[pw]) % n

    if not -n // 2 <= result <= n // 2:
        result -= n

    return result

@
```

The Miller-Rabin primality test

The test is probabilistic with the probability of the number being prime is at least 1 - $\frac{1}{4^k}$, where k is the number of iterations or bases for which we run the test. The bases are randomly chosen between 2 and n-2.

Let $n \in N$ be an odd composite number, n $>$ 2, then it can be written as $n - 1 = 2^s t$, where t is an odd number. The code below will return a tuple with s and t.

```
<<Decompose>>=
def decompose_number(n):
    powers_two = 0
    while n > 0:
        if n % 2 == 0:
            powers_two += 1
            n //= 2
        else:
            return powers_two, n
@
```

Using the property stated above and Fermat's little theorem, we get the following equivalence:

$b^{n-1} = 1$ (mod n) $\Leftrightarrow b^{2^s t} - 1 = 0$ (mod n) and since $b^{2^s t}$ is a perfect square, we can decompose further until we get to the form

$b^{2^s t} - 1 \text{ (mod n)} = (b^{2^{s-1}t} + 1)(b^{2^{s-1}t} + 1)...(b^t + 1)(b^t - 1) = 0 \text{ (mod n)}.$

If n is a prime number, it must divide one of these factors, so we need to check if one of the situations from below holds:

$b^t = 1 \text{ (mod n) or } \exists j, 0 \leq j < s \text{ such that } b^{2^j t} = -1 \text{ (mod n)}. \ (1)$

One such number n is called a **strong pseudoprime to the base** $b \in Z$**, gcd(b, n) = 1**.

A number a for which these conditions are not holding, meaning that $a^t \neq 1$ (mod n) and $a^{2^j t} \neq -1$ (mod n) for all $j = 1, \bar{s} - 1$, is called a **witness to the compositeness of n**.

The next step is to take the sequence $b^t, b^{2t}, ..., b^{2^s t}$ all mod n, where the base b is randomly chosen. The values of these powers are computed using the repeated squaring modular exponentiation algorithm. In order to simplify the computations, I applied the algorithm only on the first number of the sequence, b^t, and then squared the rest of the elements of the sequence mod n.

After computing the sequence, we need to verify the situations stated at (1). If the sequence contains only 1s or a -1 followed by 1s, which should be true since we are only squaring the numbers, we conclude that the number is likely prime and continue iterating, choosing another base, otherwise the algorithm is stopped and the number is surely composite.

```
<<Check Primality>>=
def check_primality(base, n, power, seq):

    results = repeated_sqr(base, power, n)

    power_keys = list(results.keys())
    power_keys.reverse()

    powers = sum_powers(power, power_keys)

    first_number = number_product(results, powers, n)

    result_seq = [first_number]
    for x in seq[1:]:
        next_nr = (result_seq[-1] ** (2 ** x[0])) % n
        if not -n // 2 <= next_nr <= n // 2:
            next_nr -= n
        result_seq.append(next_nr)

    try:
        index = result_seq.index(1)
        if index == 0:
            return True
        elif index != 0 and result_seq[index - 1] == -1:
            return True
    except Exception:
        return False
    return False
```

@

There are at most 25% non-witnesses for n, so if the algorithm is run for k iterations, each having a different base and the number wasn't classified as composite, then the probability of n being prime is at least $1 - \frac{1}{4^k}$.

Testing

Since the probability of a number being prime for k = 6 iterations is quite high, above 99.975%, I will choose a list of the first 6 primes in the beginning. The tested numbers will be written in the form of Mersenne numbers, $2^a - 1, a \in N$, and will be divided into 4 parts: 5 smaller primes, 5 smaller composites (both with the number of digits below 8), 5 bigger Mersenne primes and 5 bigger composite numbers. The primes were chosen from the list of Mersenne primes, while the others where mostly chosen with a non-prime exponent. An important observation is that if a number is classified as prime, then the exponent a is also prime, but not vice versa (ex: both $2^{11} - 1$ and $2^{8191} - 1$ are composites, even if the exponents are prime numbers).

```
<<Testing>>=
def test():
    print("Running tests")
    print("Checking for small primes")
    assert miller_rabin(2 ** 5 - 1) is True
    assert miller_rabin(2 ** 7 - 1) is True
    assert miller_rabin(2 ** 13 - 1) is True
    assert miller_rabin(2 ** 17 - 1) is True
    assert miller_rabin(2 ** 19 - 1) is True

    print("\nChecking for composites")
    assert miller_rabin(2 ** 6 - 1) is False
    assert miller_rabin(2 ** 11 - 1) is False
    assert miller_rabin(2 ** 14 - 1) is False
    assert miller_rabin(2 ** 15 - 1) is False
    assert miller_rabin(2 ** 20 - 1) is False

    print("\nChecking for larger primes")
    assert miller_rabin(2 ** 1279 - 1) is True
    assert miller_rabin(2 ** 2203 - 1) is True
    assert miller_rabin(2 ** 3217 - 1) is True
    assert miller_rabin(2 ** 4253 - 1) is True
    assert miller_rabin(2 ** 11213 - 1) is True

    print("\nChecking for larger composites")
    assert miller_rabin(2 ** 345 - 1) is False
    assert miller_rabin(2 ** 451 - 1) is False
    assert miller_rabin(2 ** 8191 - 1) is False
    assert miller_rabin(2 ** 12342 - 1) is False
    assert miller_rabin(2 ** 23623 - 1) is False

@
```

This chunk can be run separately.

```
<<run_tests>>=
import math
import sys
<<Helpers>>
<<Repeated squaring>>
<<Decompose>>
<<Check Primality>>
<<Miller Rabin>>
<<Testing>>

if __name__ == "__main__":
    try:
        test()
    except Exception as e:
        print(e)

@
```

Putting everything together

The application can be run from the command line with an integer number. If the parameter is not given or it isn't a number, the tests will run instead. The command is of the form `py filename.py your_number`.

In order to make the tests available separately, the algorithm has it's own chunk. The constant set beforehand is the list of first 6 primes as bases (the k = 6 iterations), starting from 2.

```
<<Miller Rabin>>=
def miller_rabin(n):
    iterations = 6
    base = 2
    powers_two, power = decompose_number(n - 1)
    seq = [(x, power) for x in range(powers_two + 1)]

    i = 0
    for i in range(iterations):
        if not base == n:
            is_prime = check_primality(base, n, power, seq)
            if not is_prime:
                break
        base = next_prime(base + 1)

    if i < iterations - 1:
        print(f"{n} is composite")
        return False
    else:
        p = 1 - 1 / (4 ** iterations)
        print(f"{n} is likely prime, with the probability at least {p}")
        return True

@
```

The functions below are used to compute the next prime number after a given one. These are used when creating the list of basis for the Miller-Rabin test.

```
<<Helpers>>=
def check_prime(a):
    a = abs(a)
    if a == 0 or a == 1:
        return False
    for i in range(2, int(math.sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True


def next_prime(a):
    while not check_prime(a):
        a += 1
    return a

@

<<*>>=
import math
import sys
<<Helpers>>
<<Repeated squaring>>
<<Decompose>>
```

```
<<Check Primality>>
<<Miller Rabin>>
<<Testing>>

if __name__ == "__main__":
    try:
        nr = abs(int(sys.argv[1]))
        miller_rabin(nr)
    except Exception:
        test()

@
```