Geração de números primos:
Miller-Rabin, Fermat e Lucas

Lucas João Martins

1 Códigos das implementações

mr.py

```python
# -*- coding: utf-8 -*-

import random
import sys
import utils as u


class mr:

    def __init__(self, k, bottom, up):
        self.k = k
        self.bottom = bottom
        self.up = up

    # documentar sobre seguranca que nao eh impar
    def decomposite(self, n):
        s, d = (0, 0)
        while True:
            x, y = divmod(n, 2)
            if y == 0:
                s += 1
                n = x
                continue
            else:
                d = n
                break

        return (s, d)

    def primality_test(self, n, s_d):
        s, d = s_d
        i = 0
        while i < self.k:
            i += 1
            a = random.randint(2, n-1)
            x = pow(a, d, n+1)
            if x == 1 or x == n:
                continue

            r = 0
            while r <= s-1:
                r += 1
                x = pow(x, 2, n+1)
                if x == 1:
                    return False
                if x == n:
                    break

            if x == n:
                continue

            return False

        return True

    def generate(self):
        while True:
            n = u.utils.make_number(self.bottom, self.up)
            s_d = self.decomposite(n-1)
            if not self.primality_test(n-1, s_d):
                continue
            return n

if __name__ == '__main__':
    png = mr(10, int(sys.argv[1]), int(sys.argv[2]))
    print(png.generate())
```

fermat.py

```
# -*- coding: utf-8 -*-

import random
import sys
import math
import utils as u


class fermat:

    def __init__(self, k, bottom, up):
        self.k = k
        self.bottom = bottom
        self.up = up

    def primality_test(self, n):
        i = 0
        while i < self.k:
            i += 1
            a = random.randint(1, n)
            if math.gcd(a, n) != 1 or pow(a, n-1, n) != 1:
                return False

        return True

    def generate(self):
        while True:
            n = u.utils.make_number(self.bottom, self.up)
            if not self.primality_test(n):
                continue
            return n

if __name__ == '__main__':
    png = fermat(10, int(sys.argv[1]), int(sys.argv[2]))
    print(png.generate())
```

lucas.py

```
# -*- coding: utf-8 -*-

import random
import sys
import primefac as pf
import utils as u


class lucas:

    def __init__(self, k, bottom, up):
        self.k = k
        self.bottom = bottom
        self.up = up

    def prime_factors(self, n):
        return list(pf.primefac(n))

    def primality_test(self, n):
        i = 0
        prime_factors = self.prime_factors(n-1)
        while i < self.k:
            i += 1
            a = random.randint(2, n-1)
            if pow(a, n-1, n) != 1:
                return False

            for q in prime_factors:
                if pow(a, (n-1)//q, n) != 1:
                    if q == prime_factors[-1]:
                        return True
                    else:
                        continue
                else:
                    break

        return False

    def generate(self):
        while True:
            n = u.utils.make_number(self.bottom, self.up)
            if not self.primality_test(n):
                continue
            return n

if __name__ == '__main__':
    png = lucas(10, int(sys.argv[1]), int(sys.argv[2]))
    print(png.generate())
```

2 Explicação dos algoritmos

3 Comparação entre os algoritmos

4 Complexidade dos algoritmos

5 Referências