Pollard's ρ Algorithm

Lab Assignment 3 - Public Key Cryptography, UBB-CS Year 3

Popa Alex Ovidiu, group 936

Problem Statement: Pollard's ρ algorithm. The implicit function will be

$$f(x) = x^2 + 1$$

but it will also allow the use of a function f given by the user.

Proposed Solution:

1. implement a runner function for Pollard's ρ algorithm for an arbitrary number and function.
2. implement a function to test the function result against known results for a set of numbers
3. implement a function to plot the size of the result of Pollard vs the number of iterations needed to achieve it

Pollard's ρ method aims to find a non-trivial factor of a given composite number (n=p*q), and bases itself around two important principles:

1. Floyd's Tortoise and Hare cycle detection algorithm
2. The Birthday Problem

To put it shortly, **Floyd's Tortoise and Hare principle** is inspired from the children story, in the sense that we have two pointers, one for each "animal", and one progresses slower than the other one, and they will eventually meet if there is a cycle to be found.
I think the best example to understand it in the field of CS, or at least that is how I managed to grasp the concept, is to think about how you would find a cycle in a Linked List: you would take a fast pointer, which always skips an element, and a slow pointer, which takes them one by one. The "hare" will catch up with the "tortoise" at a certain lap, as the "hare" loops repeatedly, and thus prove that there is a cycle and its starting point is where they met.

The **Birthday Problem** says that in a set of n randomly chosen people, some pair of them will have the same birthday with a certain probability P. Naively, P reaches 100% when n=367, because the number of days covers more than a year, however P reaches 99.9% when n=70.
This is closely related to the notion of mutually exclusive events in probability and statistics, which says that P(A) = 1 - P(nonA), meaning that if event A happens with a probability P(A), then its complementary event nonA happens with probability 1 - P(A).
Some remarks about the implementation:

1. The algorithm will run indefinitely for prime numbers, so the number is checked beforehand for primality.

2. The algorithm may not find the factors and return a failure for composite n. In that case, the constant number from the polynomial is increased until a result is found.

Euler's algorithm for gcd implementation, needed for Pollard's algorithm below.

```
<<gcd>>=
def gcd(x, y):
    while y:
        r = x % y
        x = y
        y = r
    return x

@
```

Basic implementation of primality testing- this is needed because one must test that a number is prime, because if Pollard's runs on a prime numbers, it enters an infinite loop due to the gcd computation.

```
<<Primality>>=

from math import sqrt
def prime(x):
    if x<2 or (x>2 and x%2==0):
        return False
    for d in range(3, int(sqrt(x)+1),2):
        if x%d==0:
            return False
    return True
@
```

Method which calls the function f for value val, the trick being that f is a string of the form '[1,0,2]', which represents the function

$$f(x) = 2x^2 + 1$$

```
<<funcall>>=
def funcall(f, val):
    coeff = f.replace('[','').replace(']','').split(',')
    s = 0
    for i in range(len(coeff)):
        s+=int(coeff[i])*(val**i)
    return s

@
```

Pollard's ρ Algorithm implementation, following the algorithm from the lecture. Now, where do the two principles presented above come into play in the implementation?
Well, if we take a look at xj = f(xj-1) and xj+1 = f(xj) = f(f(xj-1)), we can easily associate the first number to the "tortoise" and the second one to the "hare", as the latter is always one step ahead. This decreases the cost of computing numerous GCD calculations, and ensures the cycle can be found.
The idea is that when we use a polynomial, say xˆ2+1, we generate a sequence of numbers which seems random, but it's in fact pseudorandom, as it can be replicated.
Generating the sequence numbers modulo n makes use of the birthday paradox, i.e. the numbers will be repeated more often than they would if modulo n wouldn't be used, due to the fact that if we try finding two numbers x and y in the interval [1,n-1] which have the same result mod n, it is clear that x=y.
As far as I've tested, using f=xˆ2-2 requires the algorithm to do about 10x more iterations until it finds the solutions, and I believe that it's exactly because of these two principles, i.e. the cycles are longer.

```
<<PollardImpl>>=
<<gcd>>
<<funcall>>
def pollard(n, x0, f):
    x = [x0]
    j = 1
    while True:
        xj = funcall(f, x[-1]) % n
        x.append(xj)
        xj = funcall(f, x[-1]) % n
        x.append(xj)
        d = gcd(abs(x[2 * j] - x[j]), n)
        if 1 < d < n:
            return d
        elif d == n:
            return None
        j += 1

@
```

Pollard Function runner, with x0 going from 2 until a solution is found. Another posibility which wasn't implemented is to randomly generate a new free element of the coefficient.

```
<<PollardRunner>>=
<<PollardImpl>>
<<Primality>>
def pollardRunner(n, f):
    if prime(n):
        return None, None
    x0 = 2
    it = 1
    result = pollard(n, x0, f)
    while result is None:
        it+=1
        x0+=1
        result = pollard(n, x0, f)

    return result, it
@
```

Plot the number of iterations for large numbers (iterations vs. size of the largest factor), for numbers between two bounds defaulted to 1000 and 100000.

```
<<Plotter>>=
import matplotlib.pylab as plt
from math import log
<<PollardRunner>>

def plot(x=1000, y=100000, f='[1,0,1]'):
    data = {}
    for i in range(x,y):
        factor, it = pollardRunner(i, f)
        if factor is not None:
            data[it] = int(log(factor, 2))
    result = data.items()
    xAxis = [pair[0] for pair in result]
    yAxis = [pair[1] for pair in result]
    plt.title('iterations v factor size (bits)')
    plt.xlabel('iterations')
    plt.ylabel('sizeof factor (bits)')
    plt.scatter(xAxis,yAxis, color='red')
    plt.show()
@
```

Required tests for Pollard's Rho Algorithm.
First testing a big number to make sure it also works for larger sizes.

```
<<BigNumberTest>>=
<<PollardRunner>>

def testBigNumber():
    n=10967535067
    f='[1,0,1]'
    factors = [104723, 104729]
    return pollardRunner(n, f)[0] in factors

@
```

Each test calls the function for a specific number and tests the result against the already known divisors of the number.
Notice that it is divisors instead of factors due to the fact that Pollard's Rho algorithm might result in a number which isn't necessarily a prime factor, but can be a composition of smaller ones.
Source for the test data: http://www.positiveintegers.org/IntegerTables/50000-51001

```
<<SmallNumberTest>>=
<<PollardRunner>>
TESTS = {
        50262:[2, 3, 6, 8377, 16754, 25131],
        50294:[2, 25147],
        50303: [11, 17, 187, 269, 2959, 4573],
        50589: [3, 7, 9, 11, 21, 33, 63, 73, 77, 99, 219, 231, 511, 657, 693, 803, 1533, 240
        50595: [3, 5, 15, 3373, 10119, 16865],
        50638: [2, 7, 14, 3617, 7234, 25319],
        50645: [5, 7, 35, 1447, 7235, 10129],
        50807: [23, 47, 1081, 2209],
        50900: [2, 4, 5, 10, 20, 25, 50, 100, 509, 1018, 2036, 2545, 5090, 10180, 12725, 254
        50967: [3, 7, 9, 21, 63, 809, 2427, 5663, 7281, 16989]
        }
def testSmallNumbers():
    f='[1,0,1]'
    result = True
    for k in TESTS.keys():
        result = result and (pollardRunner(k,f)[0] in TESTS[k])
    return result

@
```

Test suite runner.

```
<<TestsRunner>>=
<<BigNumberTest>>
<<SmallNumberTest>>

def runTests():
    assert testBigNumber() and testSmallNumbers()
    print("TESTS PASSED")
@
```

Main function, Pollard runs for the given arguments, after which the tests and plotter function are called.

```
<<*>>=
<<TestsRunner>>
<<Plotter>>
import sys

def main():
    n = 7031
    f = '[1,0,1]'
    params = sys.argv[1:]
    if len(params) == 4 and params[0] == "-n" and params[2] == "-f":
        n = int(params[1])
        f = params[3]
    elif len(params) == 2 and params[0] == "-n":
        n = int(params[1])
    print("Running Pollard with n={} and f={}".format(n, f))
    result, iters = pollardRunner(n, f)
    print("Result is {}, reached in {} iterations".format(result, iters))


main()
runTests()
plot()
@
```

Runner command example:

```
notangle pollard.md >pollard.py && python pollard.py -n 10967535067 -f [1,0,1]
```

Use -n to change the number and -f to change the function. They default to n=7031 and f=x^2+1 if they are both omitted. The function defaults to f=x^2+1 if the -f flag is omitted.