Pollard's Rho Algorithm

Rad Rares

This paper's subject is one solution for the integer factorization in non-trivial factors problem using Pollard's Rho Algorithm. It is based on modular arithmetic, gcd calculation, Floyd's algorthm or Tortoise and the Hare algorithm for finding a cycle and a birthday paradox type of a remark which states that t random numbers from the interval [1,n] contain a repetition with probability P>0.5 if t > 1.177nˆ1/2

The problem reduces to finding two indexes j and k (j smaller than k) such that xj = xk. When these are found, that means we got a cycle of length k-j. The Floyd's method starts with a pair(x1,x2) and computes iteratively (xi,x2i) from the previous pair(xi-1,x2(i-1)) until we find xi=x2i. Which basically means that if the x's begin at the same point and progress in a cycle and one of them is moving twice as fast, they will meet at some point.

Here we start the algorithm by reading the polynomial and the number the user want to factorize. We use the polyinomial given as the function on which we will build the sequence of xi. It starts from x0=2 sequentially to 500, x0 could've been chosen radomly as well. After we choose our inputs we start the pollard algorithm.

```
<<*>>=
<<Pollard Rho>>
<<Map Aux>>
<<Test>>
def main():
    custom = input("Do you want a custom polynomial?(y/n) \n**the default is x^2 + 1\n")
    if (custom == "y"):

        coeficients = input("Plase input the coefficients separated"
                            " by commas (i.e 1,0,1 is x^2 + 1)\n")
        coeficients = list(map(mapFunct, coeficients.split(",")))

        n = int(input("Please input the n you want to factorize\n"))

        for x0 in range(2, 500):
            result = pollardRho(x0, n, coeficients)
            if (result != "STOP AND FAILURE"):
```

```
                return result
            else:
                continue

    elif (custom == "n"):

        n = int(input("Please input the n you want to factorize\n"))

        for x0 in range(2, 500):
            if (pollardRho(x0, n) == True):
                break
            else:
                continue

    else:
        print("re-run and please answer with y/n only, thank you ^-^\n")
test()
main()
@
```

Function that I use to generate all the possible polynomials of degree 3, it's baed on the inside implementation that python uses to generate permutations

```
<<Generate Polys>>=
def generatePolys(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n - r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i + 1:] + indices[i:i + 1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
@
```

The testing is not something necessarily relate to the algorithm, but more of how fast it works for some inputs, as in I'm running the algorithm to see what happens with a negative number, a prime one and a product of two primes, these are all tested with all polynomials of degree three. The results are represented as seconds that took to complete the algorithm and the result it yielded.

```
<<Test>>=
<<Pollard Rho>>
<<Generate Polys>>
def test():
    #list of a negative int, small prime, product of two primes
    numbers = [-124,587,58829]
    allDegreeThree = set(generatePolys([1,0,1,0,1], 3))
    times = []
    import time

    for n in numbers:
        for coef in allDegreeThree:
            start_time = time.time()
            for x0 in range(2, 500):
                result = pollardRho(x0, n, coef)
                if (result != "STOP AND FAILURE"):
                    break
                else:
                    continue
            times.append((time.time() - start_time,str(n),str(coef),result))

    for item in times:
        print(" it took --- %s seconds to run pollard rho for number %s with poly %s and res
              % (item[0],item[1],item[2],item[3]))

@
```

Auxiliary function used to parse the input. Maps the received input to an int.

```
<<Map Aux>>=
def mapFunct(element):
    return int(element)
@
```

Here we compute the steps described in the algorithm. We start from the given x0 and compute the sequence x_i = f(x_i-1), f being the function provided by the user. We do this in order to be able to search in the sequence numbers of the from x_i = x_2i, because then we can be sure that we found a cycle. Further, we use the birthday paradox which tells us that since xi and x2i are in range of 1,500 xi = x2imod500 only when xi = x2i. Also, we know that one of the factors of n is found in the numbers less than sqrt(n) I did search on some resources, but what I could not get is the logic of the absolute difference deviding n. Next

we check that the the abs difference is not relatively prime with n. If it is not, this difference is our non-trivial factor. Else we continue the algorithm with a new xi and x2i.

```
<<Pollard Rho>>=
<<Gcd Division>>
<<Create Polynomial>>
def pollardRho(x0, n, coefficientList=[1, 0, 1]):
    sequenceContainer = [x0]

    lastX0 = x0
    for index in range(1, 500):
        currentX0 = createPolynomial(coefficientList, lastX0) % n
        sequenceContainer.append(currentX0)
        lastX0 = currentX0

    for index in range(1, 250):
        print("j = " + str(index))
        print("x2_j = " + str(sequenceContainer[2 * index]))
        print("x_j = " + str(sequenceContainer[index]))

        absoluteDifference = abs(sequenceContainer[2 * index] - sequenceContainer[index])
        print("|x2_j - x_j| = " + str(absoluteDifference))

        d = gcdDivision(absoluteDifference, n)
        print("d = " + str(d))
        if ((1 < d) and (d < n)):
            print(str(d) + " is a non-trivial factor of n")
            return str(d)
        elif (d == n):
            print("STOP AND FAILURE")
            return "STOP AND FAILURE"
        else:
            continue
@
```

This function receives a list of coefficients and a value x from which it creates a polynomial For example, from [1,1,0,1] and x=2 the function will return 13 (1*2^3 + 1*2^2 + 0*2^1 + 1*2^0) It just goes from 0 to the length of the coefficient list and adds to the result the computation of the product of the cofficient with x t the power of the difference between the length of the coeff and the index - 1 I did it this way, because the coefficients are assigned from left to right as the bigger power to the smallest. Also, the "-1" is used because a polynomial of order n has the max power n-1

```
<<Create Polynomial>>=
def createPolynomial(coefficientList, x):
```

```
    result = 0
    for index in range(0, len(coefficientList)):
        result = result + coefficientList[index] * pow(x, len(coefficientList) - index - 1)

    return result
@
```

This algorithm works by dividing the two numbers and switching the second one with their reminder because we know that the gcd of two numbers does not change if we replace the largest of the two by it's remainder when devided with the smaller one of them. If the smaller one is devided by the largest, they are simply switched. We do this until the b becomes 0.

```
<<Gcd Division>>=
def gcdDivision(a, b):
    while b != 0:
        a, b = b, a % b

    return a
@
```