Cryptography Lab3 Homework - Pollard's p - 1

Bardas Alexandru-Cristian, Group 321

Pollard's p - 1 Method

The $p - 1$ algorithm was developed by J.M. Pollard in the 1970's. The basic idea of the algorithm is to use some information about the order of an element of the group $\mathbb{Z}_p$ to find a factor p of n. It is used to efficiently find any prime factor p of an odd composite number n for which $p - 1$ has only small prime divisors, then we are able to find a multiple k of $p - 1$ without knowing $p - 1$, as a product of powers of small primes. The algorithm is based on the following theorem:

Fermat's Little Theorem :

Let p be a prime and $a \in \mathbb{Z}$ such that $p \nmid a$. Then $a^{p-1} \equiv 1$ (mod p).

Proof :

Consider the following two sets of equivalence classes:
$A = \{\hat{a}, 2 * \hat{a}, ..., (p-1) * \hat{a}\}$,
$B = \{\hat{1}, \hat{2}, ..., \widehat{p-1}\} = \mathbb{Z}_p$ - $\{\hat{0}\}$.
First, we want to show that $A = B$.
Clearly, $A \subseteq B$, since $p \nmid a$ and p divides none of $1, 2, 3, ..., p - 1$.
Now, suppose that $\exists r, s \in \mathbb{N}$ such that $1 \leq r \leq s \leq p - 1$ and $\hat{ra} = \hat{sa}$.
Then, $r \hat{*} a - s \hat{*} a = 0$
$r * \hat{a} - s * \hat{a} = 0$
$(r - s) * \hat{a} = 0$
$r - s = 0$, since $\hat{a} \neq 0$ mod p(since $p \nmid a$).
But since $r \leq p$ and $s \leq p$, $r - s \equiv 0$ (mod p) $\Rightarrow r = s$. So, all elements of A are different mod p, which means that the cardinality of A is $p - 1$. And, since $|A| = p - 1$, $|B| = p - 1$ and $A \subseteq B$, we can conclude that $A = B$, that is the equivalence classes of A are congruent to the equivalence classes of B under a certain rearrangement. Hence, by multiplying all the elements in A and B we get that
$a * 2a * 3a * \cdots * (p - 1) * a \equiv 1 * 2 * 3 * \cdots * (p - 1)$ (mod p)

$a^{p-1} * (p-1)! \equiv (p-1)! \pmod{p}$
$a^{p-1} \equiv 1 \pmod{p}$, since $(p-1)! \not\equiv 0 \pmod{p}$.

The idea for Pollard's p - 1 Method

The idea is that if $p|n$ and p is prime, then $a^{p-1} \equiv 1 \pmod{p}$ or $d = a^{p-1} - 1 \equiv 0 \pmod{p}$, for any a relatively prime to p, so computing $gcd(d, n) = p$ gives us the factor of n we were looking for. Evidently, we cannot directly compute d because we do not know p at first. We could just compute a^m with an exponent $m = 1, 2, 3, ...,$ until $gcd(d, n) = p$, that is until $m = p - 1$. However, that would not be more efficient than doing trial division. There is however a clever way to choose m. The idea is to notice that we do not need to exponentiate a to exactly the power of $m = p - 1$ since, if m is such that $p - 1|m$, that is $m = c * (p-1)$, then $a^m - 1 = a^{c*(p-1)} - 1 = a^{(p-1)^c} - 1 = 1^c - 1 \equiv 0 \pmod{p}$, so that $gcd(a^m, n) = p$. So, we need to choose an integer m and we will get a factor of n if $p-1|m$. By Choosing m as a product of many small prime factors, the chances that this condition holds will increase.

Pollard's p - 1 Algorithm

Step 1. Generating the exponent k as follows:

$k = \Pi\{q^i \mid q$ prime, $i \in \mathbb{N}$ - $\{0\}$, $q^i \leq B\}$, where B is a bound.

```
<<generateK>>=
def generateK(B) :
    # I use the Sieve of Eratosthenes to generate a list of
    # all prime numbers less than the bound B
    prime = [True for i in range(B + 1)]
    p = 2
    while p * p <= B:
        # If prime[p] is not changed, then it is a prime
        if prime[p] is True:
            # Update all multiples of p
            for i in range(p * p, B + 1, p):
                prime[i] = False
        p += 1
    k = 1
    # Compute k as the product of all the smaller primes
    # to their max power s.t. they are less than or equal to B
    for i in range(2, B + 1):
        if prime[i] is True:
            k *= (i ** findBiggestExponent(i, B))
    return k
@
```

Here, I've used to auxiliary function "findBiggestExponent", which I've created to return the biggest exponent that when we take the given i to that power, let's say K, $i^K \leq B$ and $i^{K+1} > B$.

```
<<findBiggestExponent>>=
def findBiggestExponent(a, B):
    # Returns the maximum exponent i s.t. a^i <= B
    i = 1
    aux = a
    while a <= B:
        a *= aux
        i += 1
    return i - 1
@
```

Step 2. Randomly choose an a, such that $1 < a < n - 1$.

Step 3. Compute $a = a^k$ mod n.

Step 4. Compute $d = gcd(a - 1, n)$.

If $d = 1$, go to Step 1 and increase B. If $d = n$, then go to Step 2 and change a. Else, return d, as it is a non-trivial factor of n.

In order to compute the gcd of two numbers, I've implemented Euclid's recursive gcd algorithm.

```
<<euclidGCD>>=
def euclidGCD(a, b):
    # Calculates gcd(a,b) using the euclidean algorithm
    if b == 0:
        return a
    return euclidGCD(b, a % b)
@
```

To implement the rest of the logic of this algorithm, I've designed a function that puts everything side by side and does the required computations.

```
<<pollardP>>=
def pollardP(n, B):
    # Function that handles the computations
    # for pollard's p-1 algorithm
    k = generateK(B)
    d = 1
    a = 1
    done = False
    while a < n - 1:
        b = pow(a, k, n) # Python has implemented fast modular exponentiation
```

```
        # faster than repeated squaring modular exponentiation
        d = euclidGCD(b - 1, n)
        if d == 1:
            B = B + 1
            k = generateK(B)
        elif d == n:
            a = a + 1
        else:
            done = True
            break
    if done is False:
        print("Failure")
    else:
        print("Non-trivial factor of n = " + str(n) + " is d = " + str(d))
    return d
@
```

I've also implemented some test cases for the given created algorithm.

```
<<tests>>=
def pollardTest():
    d = 1
    d = pollardP(257, 13)
    # 257 is the 3rd Fermat Prime( 2^2^n + 1)
    assert(d == 1 or d == 257) # prints Failure
    # construct a big number
    p = 100711409 * 100711423
    d = pollardP(p, 49)
    assert(d == 100711409 or d == 100711423) # prints a divisor
    # Construct another number
    p = pow(2, 11) * (pow(2, 10) - 1)
    d = pollardP(p, 83)
    assert (d == pow(2, 11) or d == pow(2, 10) - 1) # prints a divisor
    d = pollardP(5039, 101) # Factorial Prime ( 7! - 1)
    assert(d == 1 or d == 5039) # prints Failure
    #Construct another big number
    p = pow(3, 6) * pow(4, 7) * pow(5, 2)
    d = pollardP(p, 109)
    assert (p % d == 0 and d != 1 and d != p) # assert d is a non-trivial factor
    return

@
```

Finally, the main driver of the program.

```
<<main>>=
if __name__ == "__main__":
    #Driver part of the program
```

```
    print("---Tests---")
    pollardTest()# Prints out testing results
    print("Tests were successful!")
    print("\n---Program---")
    n = int(input("Give odd composite n = "))
    opt = input("Want custom bound?(implicit bound = 13) y/n : ")
    if opt == 'y':
        B = int(input("Give bound B = "))
    else:
        B = 13 # implicit bound
    pollardP(n, B)
@

<<*>>=
<<findBiggestExponent>>
<<generateK>>
<<euclidGCD>>
<<pollardP>>
<<tests>>
<<main>>

@
```