EDIN01 Cryptography: Project 1 Factoring Algorithms

Loïc Masure - Team 23

November 18, 2016

1 Trial division

To factorize naively one 25-digits number, we must try about 10^{12} divisions, which takes about 277 hours.

If we store all the prime numbers up to $\lfloor\sqrt{N}\rfloor$, we must pre-compute and store about

$$\frac{\lfloor\sqrt{N}\rfloor}{\ln\lfloor\sqrt{N}\rfloor}.$$

Finally this method is about 28 much faster but it costs a lot in memory: about 180 Go in RAM, that represents 16 000 SEK according to the website suggested in the topic. That represents a student budget.

2 The quadratic sieve algorithm

The algorithm described in the paper is much faster. I have been assigned to the number

$$N = 184208651242126473140033.$$

I find after an average time of 2 minutes, 45 seconds the factors

$$p = 388117391953$$

and

$$q = 474620965361.$$

To factor the number $N = 92434447339770015548544881401$, it requires 9 minutes to find the two factors: $p = 727123456789451$, and $q = 127123456789451$.

3 How my algorithm works

I decided to code in Python, as it manages the large integers on its own, and all the expensive operations (large arrays) are done with the Numpy library which runs in C.

First of all, I generate a list of r numbers which are written in a file so that we can then run the program `GaussBin.exe` which is already optimized. The solution is written in another file which is read by my Python program.

Then the program recomposes the solutions found, according to the decomposition in the Factor-Base generated in the beginning. We test if the assertion $x^2 = y^2$ is still true. Finally, we compute $\gcd(x + y, x - y)$ and test if it equals 1, N or another integer between. I usually need at max 3 solutions to find the right couple (p, q).

4 Printouts of my algorithm

You can find the main algorithm, followed by some basic functions used for the main algorithm.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov  9 14:54:02 2016
@author: loicm
Problems for now: the higher is N, the smaller the factor base has to be.
Otherwise, the computation of (x, y) such that x^2=y^2 % N is wrong...
"""
import math
import numpy as np
import naiv
import os
from time import sleep, perf_counter as pc
t0 = pc()
N = 92434447339770015548544881401        # The number to factorize
L = 1024  # The number of relations we want to have to compute x and y
L_more = 10  # The number or additional rows compare to L
list_p = naiv.generate_primes(L)

print("B= {}".format(list_p[-1]))


# We need to generate a list of numbers whose the fators are B-smooth.
m = 2
j = m
k = 0
list_r2 = []
list_all = []
while len(list_all) < L+L_more:
    r = math.floor(math.sqrt(k * N)) + j
    r2 = r**2 % N
    # We compute naïvely the prime factors up to B
    row, b = naiv.decompose_opt(r2, list_p)
#     verif = naiv.recompose(row, list_p, N)
#     print("Vérification: r={},  r2={}, verif={}".format(r, r2, verif))
    if b and r2 not in list_r2:
        test = False
        row %= 2
        for i in range(len(list_all)):
            if (row == list_all[i][2]).all():
                test = True
                break
        if not test:
            list_r2.append(r2)  # For unicity
            list_all.append((r, r2, row))
            if len(list_all) % 20 == 0:
                print(len(list_all))
#             print("Ajout de {}, ligne={}".format(r, row))
    # Increment in diagonals, is arbitrary.
    if k == m:
        m += 1
        j = m
        k = 0
    else:
        j -= 1
        k += 1
print("Fin de la boucle.")
list_all = sorted(list_all, key=lambda tutuple: tutuple[0])
list_r, list_r2, list_factor = zip(*list_all)
for r, r2 in zip(list_r, list_r2):
    assert(r2 == r**2 % N)
M = np.array(list_factor, dtype=np.int)

header = "{} {}".format(L + L_more, L)
np.savetxt("essai", M, fmt="%d")
with open("essai", 'r+') as f:
        content = f.read()
        f.seek(0, 0)
        f.write(header.rstrip('\r\n') + '\n' + content)
# We call GaussBin to compute the solutions to the binary linear system.
os.system("GaussBin.exe essai sortie")
```

```python
# We load the solution in the program.
X = np.loadtxt("sortie", dtype=np.int, skiprows=1)
for x in range(X.shape[0]):
    # We get the x such that x^2=y^2
    x1 = naiv.recompose(X[x, :], list_r, N)

    # We get the y such that x^2=y^2
    row = np.zeros(L)
    for i, c in enumerate(X[x]):
        if c == 1:
            plus, b = naiv.decompose_opt(list_r2[i], list_p)
            if b:
                row += plus
            else:
                raise Exception("Bad decomposition: {} in {} gives {}".format(list_r[i],list_p, row))
    assert((row % 2 == 0).all())
    row //= 2
    row = row.astype(np.int)
    x2 = naiv.recompose(row, list_p, N)
    print(x)
    print("Solution possible: x = {} y = {}, x2 = {}, y2 = {}".format(x1,
                                                                       x2,
                                                                       x1**2%N,
                                                                       x2**2%N))
    assert((x1 ** 2) % N == (x2 ** 2) % N)

    d = math.gcd(max(x1, x2) - min(x1, x2), N)
    if d != N and d != 1:
        p = d
        q = int(N/d)
        break

print("Final solution: N = {}, p = {}, q = {}".format(N, p, q))
assert (N == p*q)
print("Execution time: {}".format(pc()-t0))
```

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Nov  9 16:52:21 2016
@author: loicm
"""
import numpy as np
import itertools


def check_prime(n, primes):
    for p in primes:
        if not n % p:
            return False
    return True


def prime_sieve():
    primes = set()
    for n in itertools.count(2):
        if check_prime(n, primes):
            primes.add(n)
            yield n


def generate_primes(L):
    """
    Generates a list of the L first prime numbers .
    """
    # We need to compute the Bound of the factor set.
    i = 0
    list_p = []
    for p in prime_sieve():
        i += 1
        list_p.append(p)
        if i >= L:
            break
    return list_p


def decompose_opt(r, list_p):
    """
    Decompose naïvely r as a product of factors in list_p and returns an array
    with the powers corresponding to the factors.
    """
#     print("Decompose pour {}".format(r))
    res = np.zeros(len(list_p), dtype=np.int)
    i = 0
    while i < len(list_p):
        if r % list_p[i] == 0:
            res[i] += 1
            r //= list_p[i]
        else:
            i += 1
    return res, (r < list_p[-1])

def recompose(x, list_p, N):
    """
    Reconstruct the integer decomposed with the prime factor basis given in
    argument.
    """
    res = 1
    for i in zip(x, list_p):
        plus = 1
        for j in range(i[0]):
            plus *= i[1]
            plus %= N
        res *= plus
        res %= N
    return int(res % N)
```

5 Time dedicated to this project

I would say that the implementation was quite quickly (about 2h) thanks to the precise specifications. However, I faced a bug about the recomposition of the x and y which took about 10 hours to find it.