Мультиполиномиальное квадратичное решето

Гвасалиа М. Дородный Д.А.

25 июня 2021 г.

Аннотация

Реализация алгоритма MPQS с использованием системы компьютерной алгебры Sage на Python 2.7.

1 Введение...

Алгоритм основывается на том, что при выполнении соотношения

$$A^2 \equiv B^2 \mod N \tag{1}$$

Где N - число, которое необходимо факторизовать, и $A \neq \pm B \mod N$, $(A \pm B, N)$ будут являться факторами числа N. Алгоритм предлагает эффективный способ построения подобных отношений, что позволяет быстро факторизовать большие N.

2 Описание алгоритма

2.1 Алгоритм

Algorithm 1: MPQS

Result: 2 делителя числа N
Выбор множителя $k : k * N \equiv 1 \mod 8$;
Выбор параметров M, F, T;
Подсчет тестового значения;
Генерация факторной базы $fBase$;
while *Недостаточно факторизаций* **do**
  Подсчет коэффициентов многочлена $Q(x)$;
  Решение $Q(x) = 0 \mod p_i \forall p_i \in fBase$;
  Просеивание полученных решений;
  for *Значения массива, большие тестового* **do**
    Вычислить $Q(x)$;
    Перебрать делители по факторной базе;
    if *Полностью факторизуется* **then**
      сохранить степени разложения и $\sqrt{Q(x)} \mod kn$;
    else
      выполнить LPP;
    end
  end
end

LPP (Large Prime Procedure) - если после неполной факторизации остается множитель L, то можно найти еще одно значение $Q(x)$ с таким же множителем, тогда при перемножении получим множитель L^2

2.2 Подсчет коэффициента k

Для того, чтобы из многочлена $Q(x)$ могли получаться квадратичные вычеты необходимо, чтобы дискриминант многочлена был равен N

$$B^2 - 4AC = N \tag{2}$$

Т.е. N должно быть равно 0 или 1 по модулю 4. Следовательно, если $N \mod 4 \equiv 3$, необходимо подобрать $k, k * N \mod 8 \equiv 1$. Так же k должно максимизировать значение функции Кнута-Шреппеля. Предпочтительнее выполнять проверку по модулю 8, т.к. в этом случае 2 будет входить в факторную базу.

2.3 Выбор параметров

С помощью квадратичной регрессии по данным из [1] были получены следующие формулы для параметров:

$$\begin{aligned}
F &= 2.93 * x^2 * -164.4 * x + 2455 \\
M &= 386 * x^2 - 23209.3 + 352768 \\
T &= 0.0268849 * x + 0.783929 \\
x &= \log_{10} N
\end{aligned}$$

2.4 Потсчет тестового значения

Тестовое значение используется для отсеивания значений, полученных на этапе просеивания, которые не будут факторизоваться по нашей факторной базе.

$$\left\lfloor \log \frac{M * \sqrt{\frac{kN}{2}}}{p_{max}^T} \right\rfloor \tag{3}$$

2.5 Генерация факторной базы

В качестве факторной базы берем -1, 2 а так же первые F простых числел, по модулю которых наше число является квадратичным вычетом.

```
def genFBase(N, fSize):
  fBase = []
  i = 1
  while len(fBase) < fSize:
    fBase = [-1] + filter(lambda p: kronecker(N, p) == 1, primes_first_n(fSize * 3 * i))
    i += 1
  fBase = fBase[0:fSize]
  return fBase
```

Так же подсчитываем и сохраняем список логарифмов факторной базы и список корней из kN по всей факторной базе.

```
for p in fBase[1:]:
    pField = IntegerModRing(p)
    sns.append((pField(kn).nth_root(2)).lift()) #roots of N modulo p_i

logp = [numerical_approx(log(p)) for p in fBase[1:]] #logarythms of factor base
```

2.6 Основной цикл алгоритма

2.6.1 Подсчет коэффициентов

Из 2 следует следующее соотношение

$$B^2 \equiv kN \mod 4A \tag{4}$$

Обозначим $\sqrt{A}$ как некое вероятно простое $d : (\frac{d}{kN}) = 1$. Тогда коэффициенты многочлена можно посчитать следующим образом:

$$\begin{aligned}
A &= d^2 \\
B_0 &= \sqrt{kN} \mod d \\
B &= \frac{B_0 - (B_0^2 - kN)}{2B_0} \mod A \\
C &= \frac{B^2 - kN}{A}
\end{aligned}$$

```
d = next_probable_prime(d)
while not (kronecker(d, kn) == 1 and mod(d, 4) == 3):
  d = next_probable_prime(d)

a = d ^ 2
dRing = IntegerModRing(d)

b = dRing(kn).nth_root(2).lift()
b = mod(b - (b^2 - kn) * dRing((2 * b) ^ (-1)).lift(), a).lift()
b = b - a if b % 2 == 0 else b
c = floor((b ^ 2 - kn) / a)
```

2.6.2 Подсчет решений

Находим решения уравнения $Q(x) \equiv 0 \mod p_i \forall p_i \in FBase$

$$\begin{aligned}
x_1 &= \frac{-\sqrt{kN} - B}{A} \\
x_2 &= \frac{\sqrt{kN} - B}{A}
\end{aligned}$$

2.6.3 Просеивание

Просеиваем массив из $2 * M$ элементов следующим образом: Для каждого элемента массива, индекс которого соответствует числу из отрезка от $-M$ до M, и который равен x_1 или $x_2 \mod p_i$, прибавляем $\log p_i$

```
qs = [0 for i in range(0, 2 * M + 1)]

termflag = 0
for i in range(len(fBase) - 1):
  p = fBase[i + 1]
  pRing = IntegerModRing(p)
  if pRing(a) == 0:
    termflag = 1
    break
  root1 = pRing((-b + sns[i]) / a).lift()
  root2 = pRing((-b - sns[i]) / a).lift()
```

```
  l1 = root1 + ceil((-M - root1) / p) * p
  l2 = root2 + ceil((-M - root2) / p) * p
  k = 0
  while l1 + k <= M or l2 + k <= M:
    if l1 + k <= M:
      qs[int(l1 + k + M)] += logp[i]
    if l2 + k <= M:
      qs[int(l2 + k + M)] += logp[i]
    k += p
```

2.6.4 Внутренний цикл алгоритма

Для всех значений массива, которые превышают тестовое значение выполнить:

Т.к. $(Ax + B)^2 = A * Q(x)$

Вычисляем значение многочлена $Q(x) = A * x^2 + 2 * B * x + C$ и выполняем перебор делителей. Если в результате перебора получилось полное разложение на множители, записываем вектор степеней разложения в матрицу решения и $Ax + B, d \mod kN$ в массив корней.

Если факторизация неполная, то находим 2 числа с одинаковым множителем l и записываем в матрицу решения сумму их векторов степеней разложения, а в массив корней $Ax + B, d * l \mod kN$

```
qAndX = enumerate(qs)
ourPick = filter(lambda x: x[1] > testValue, qAndX) #apply test value

kysField = IntegerModRing(kn)

preFac = map(lambda x: (divFac(q(x[0] - M), fBase), (h(x[0] - M), d)), ourPick) # ( (multiplier, vec), (H(x), d) )

for parvec, sqroot in preFac:
  if parvec[0] == 1: #check if factorisation was full on our fBase
    fullFac.append((parvec[1], sqroot))
  else:
    if parvec[0] in lsAndFacs: #check if there is another entry with the same non-fBase multiplier
      found = lsAndFacs[parvec[0]] # (VEC, ( H(X), d ))
      fullFac.append(([x + y for (x, y) in zip(parvec[1], found[0])], (sqroot[0] * found[1][0], found[1][1] * sqroot[1] * par
      lsAndFacs.pop(parvec[0])
    else: #if not - add it to the partial factorisation list
      lsAndFacs[parvec[0]] = (parvec[1], sqroot)
```

2.7 Получение решения

Приводим матрицу решения по модулю 2 и находим левое ядро. Базисные векторы левого ядра являются нужными нам решениями. Тогда для каждого базисного вектора считаем X - произведение соответствующих решению элементов массива корней и складываем соответствующие степени векторов разложения и делим итоговый вектор на 2 и согласно полученному вектору степеней разложения получаем число Y.

Тогда если $X \neq \pm Y \mod N$ и $1 < (X \pm Y, N) < N$ полученный НОД является нетривиальным фактором N

```
GF = IntegerModRing(2)
vecs = map(lambda x: x[0], fullFac) #fixed
A = matrix(GF, vecs)
allSol = A.left_kernel().basis()
allFacs = set()
for sol in allSol:
    whatever = filter(lambda x: x[0] == 1, zip(sol, fullFac)) # (1, (VEC, (H(X), AL)))
```

```

    sumVecs = reduce(lambda xs, ys: [x + y for (x, y) in zip(xs, ys)], map(lambda x: x[1][0], whatever))
    squareVec = map(lambda x: x / 2, sumVecs) #exp vec/2 <=> square root

    soll = mod(product(map(lambda x: x[1][1][0], whatever)), kn) # product H(x)
    alsoR = product(map(lambda x: x[1][1][1], whatever)) #product AL
    solr = mod(alsoR * product(map(lambda x, y: y ** x, squareVec, fBase)), kn) # number from factor base and respective powe
    if soll != solr and soll != -solr:
      allFacs.add(gcd(soll + solr, N).lift())
      allFacs.add(gcd(soll - solr, N).lift())
print(filter(lambda x: 1 < x < N, allFacs))
```

2.8 Эффективность алгоритма

Число знаков	Время работы (сек)
20	65
30	17.5
40	234
45	300
50	3615
55	23455

Таблица 1: Производительность реализации

Список литературы

[1] Robert D. Silverman, The Multiple Polynomial Quadratic Sieve

[2] Michael Østergaard Pedersen, A Parallel Implementation of the Quadratic Sieve Algorithm