Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 3

по дисциплине «Теоретико-числовые методы в криптографии»

Выполнил: студент группы
4851003/80801 С.А. Ляхова

<подпись>

Проверил: доцент, к.т.н. Е.Ю. Павленко

<подпись>

Санкт-Петербург
2020

1. Формулировка задания

Каждое из ниже указанных чисел разложить на множители:

1. ро-методом Полларда.
2. (р-1)-методом Полларда.
3. Методом непрерывных дробей.

1) 16735883892800965609
2) 54779948035654414815122917627356921709 3
3) 88651164586358827979841066245000789967131446236358463604007891 77520228258163291

В отчёте привести:

1. В случае результативного завершения работы программы: результат разложения.

2. Для ро-метода Полларда:

2.0. Параметры алгоритма (использованное отображение, начальное значение (несколько, если их пришлось менять)).

2.1. При результативном завершении работы – первые 5 и последние 5 значений a, b, НОД(a – b, n), число итераций, время работы программы, выводы (объяснение результативного завершения).

2.2. При работе программы более нескольких часов – первые 5 и последние (на момент прерывания программы) 5 значений a, b, НОД(a – b, n), число выполненных итераций, время, затраченное на их выполнение, расчетное время, оставшееся до завершения работы (через оценку сложности алгоритма), выводы (объяснение нерезультативного завершения).

3. Для (р-1)-метода Полларда:

3.1 Базу разложения (первоначальную, измененную (если потребовалось изменение, обосновать необходимость изменения)).

3.2 Значения показателей l_i.

3.3 При результативном завершении работы – основание a, при котором выполнено разложение (несколько, если потребовалось изменять основание).

3.4 При невозможности найти разложение более нескольких часов – основание a, при котором выполнено разложение (несколько, если потребовалось изменять основание), число выполненных итераций (одна итерация – прогон алгоритма с одним основанием a), время, затраченное на их выполнение, расчетное время, оставшееся до завершения работы (через оценку сложности алгоритма), выводы (объяснение нерезультативного завершения).

4. Для метода непрерывных дробей:

4.1 При результативном завершении работы:

4.1.1 Базу разложения (первоначальную, измененную (если потребовалось изменение, обосновать необходимость изменения)), при большом объеме базы – число элементов в базе и ее последний элемент, критерии исключения элементов из базы.

4.1.2 Числители P_i подходящих дробей, использованных при разложении, для D-гладких значений (P_i)^2 (mod n), при большом объеме данных – 5 значений Pi и (P_i)^2 (mod n).

4.1.3 Векторы показателей для D-гладких значений (P_i)^2 (mod n), при большом объеме данных – 5 векторов, соответствующих значениям (P_i)^2 (mod n) из предыдущего пункта.

4.1.4 Метод, использованный для поиска линейно-зависимых векторов (если была использована готовая процедура – указать, из какой библиотеки).

4.1.5 Значения s и t.

4.2 При невозможности найти разложение более нескольких часов:

4.2.1 Данные по пп. 4.1.1–4.1.3 на момент прерывания программы.

4.2.2 Расчетное необходимое число гладких чисел (P_i)^2 (mod n) и время, необходимое для их поиска.

4.3 Выводы (объяснение результатов работы).

2. Теоретические сведения

Задача разложения составного числа на множители формулируется так: для данного положительного целого числа n найти его каноническое разложение $n = p_1^{a_1} \cdot p_2^{a_2} \dots \cdot p_s^{a_s}$, где pi – попарно различные простые числа, ai ≥ 1.

На практике не обязательно находить каноническое разложение числа n. Достаточно найти его разложение на два нетривиальных сомножителя: n=pq, 1 < p ≤ q < n.

Пусть $B = \{p_1, p_2, \dots, p_s\}$ – множество различных простых чисел. Назовем множество B базой разложения. Целое число назовем B-гладким, если все его простые делители являются элементами множества B.

1. ρ-Метод Полларда

Алгоритм:

Вход. Число *n*, начальное значение *c*, функция *f*, обладающая сжимающими свойствами.

***Выход*.** Нетривиальный делитель *p* числа *n*.

1. Положить $a \leftarrow c$, $b \leftarrow c$.
2. Вычислить $a \leftarrow f(a) \pmod n$, $b \leftarrow f(b) \pmod n$, $b \leftarrow f(b) \pmod n$.
3. Найти $d \leftarrow$ НОД $(a - b, n)$.
4. Если $1 < d < n$, то положить $p \leftarrow d$ и результат: *p*. При $d = n$ результат: «Делитель не найден»; при $d = 1$ вернуться на шаг 2.

Сложность алгоритма: $O(\sqrt[4]{n})$

2. (p-1)-Метод Полларда

Алгоритм:

***Вход*.** Составное число *n*.

***Выход*.** Нетривиальный делитель *p* числа *n*.

1. Выбрать базу разложения $B = \{p_1, p_2, \dots, p_s\}$.
2. Выбрать случайное целое *a*, $2 \le a \le n - 2$, и вычислить $d \leftarrow$ НОД(a, n). При $d \ge 2$ положить $p \leftarrow d$ и результат: *p*.
3. Для $i = 1, 2, \dots, s$ выполнить следующие действия.
   3.1 Вычислить $l \leftarrow \left[\frac{\ln n}{\ln p_i}\right]$.
   3.2 Положить $a \leftarrow a^{p_i^l} \pmod n$.
4. Вычислить $d \leftarrow$ НОД $(a - 1, n)$.
5. При $d = 1$ или $d = n$ результат: «Делитель не найден». В противном случае положить $p \leftarrow d$ и результат: *p*.

Сложность алгоритма: $O(B * logB * (log\, n)^2)$

3. Метод непрерывных дробей

Данный метод является вариацией алгоритма Диксона*, в котором в качестве чисел s_i выбираются числители подходящих дробей для непрерывной дроби числа $\sqrt{n}$.

Был применен метод разложения квадратного корня (который является квадратичной иррациональностью) в бесконечную дробь, не требующего высокой точности вычисления корня. Для получения наборов линейно зависимых векторов e был реализован метод исключения Гаусса.

Алгоритм

Вход. Составное число m.
Выход. Нетривиальный делитель p числа m.

1. Построить базу разложения $B = \{p_0, p_1, \dots, p_h\}$, где $p_0 = -1$ и $p_1, \dots, p_h$ — попарно различные простые числа, по модулю которых m является квадратичным вычетом.
2. Берутся целые числа u_i, являющиеся числителями подходящих дробей к обыкновенной непрерывной дроби, выражающей число $\sqrt{m}$. Из этих числителей выбираются $h + 2$ чисел, для которых абсолютно наименьшие вычеты u_i^2 по модулю m являются B-гладкими:
$$u_i^2 mod\ m = \prod_{j=0}^{h} p_j^{\alpha_{ij}} = v_i,$$
где $\alpha_{ij} \geq 0$. Также каждому числу u_i сопоставляется вектор показателей $(\alpha_{i0}, \alpha_{i1}, \dots, \alpha_{ih})$.
3. Найти методом Гаусса такое непустое множество $K \subseteq \{1,2, \dots, h+1\}$, что $\bigoplus_{i \in K} e_i = 0$, где $\bigoplus$ - операция исключающее ИЛИ, $e_i = (e_{i1}, e_{i2}, \dots, e_{ih})$, $e_{ij} \equiv \alpha_{ij}\ (mod\ 2), 0 \leq j \leq h$.
4. Положить $x \leftarrow \prod_{i \in K} u_i\ mod\ m$, $y \leftarrow \prod_{j=1}^{h} p_j^{\frac{1}{2}\sum_{i \in K} \alpha_{ij}}\ mod\ m$.Тогда $x^2 \equiv y^2\ (mod\ m)$.
5. Если $x \not\equiv y\ (mod\ m)$, то положить $p \leftarrow \gcd(x - y, m)$ и выдать результат: p. В противном случае вернуться на шаг 3 и поменять множество K.

Временная сложность алгоритма: $O(\exp(2 * \sqrt{\ln(n)\ (\ln(\ln(n)))}))$
Объем базы разложения: $h \approx O(\exp(\frac{2}{3} * \sqrt{\ln(n)\ (\ln(\ln(n)))}))$

3. Результаты работы

В результате выполнения лабораторной работы были реализованы три алгоритма разложения чисел на множители.

Результаты разложения приведены в таблицах ниже:

- **ρ-метод Полларда**

Результаты работы для первого числа n = 16735883892800965609:

n	p	q	f(x)	c	Число итераций
16735883892800965609	2978456267	5618979227	$x^2+1(mod.n)$	2	22068

i	a	b	НОД(a-b,n)
1	26	698	1
2	698	237389175098	1
3	487226	956508730157756382	1
4	237389175098	15813493505184678849	1
5	3899386648052104123	6062757303659634892	1
…	…	…	…
22064	3857531528175848832	3999941129058439638	1
22065	5956269886358653816	15531430238931673527	1
22066	12451997804181098917	6649237267778822551	1
22067	5810495192901313296	12805804528760661792	1
22068	9523442906851112025	11990343567530179249	2978456267

Время прошедшее с момента запуска: 42.8194 secs

Результат: $16735883892800965609 = 5618979227 * 2978456267$

Результаты работы для второго числа n = 54779948035654414815122917627356921709 3:

Для второго числа результативного завершения после 8 часов работы и не было, что, видимо, означает, что были подобраны неудачные начальное значение и функция f(x). Программа была запущена с несколькими другими начальными значениями, но по прошествии более часа работа завершена также не была. Скорее всего, все попытки зацикливались, так и не получив верное a-b.

Результаты работы для входных данных $f(x)= x^2+22(mod.n)$, c=2:

i	a	b	НОД (a-b,n)
1	5	26	1
2	26	458330	1
3	677	44127887745906175987802	1
4	458330	131218766814689339124291034 36157960687	1
5	210066388901	576700323401902286206253484 64705074578	1
…	…	…	…
77558556	4979221547186671688587940483 80265620355	19967948279531511731487610540 3566128386	1
77558557	21789387385941173555852843735 4658559392	52694520947858625723763904899 939675767	1
77558558	17698108636378547864775847770 471186730	34321522395728810187997583221 286203578	1
77558559	53442909104726870538789719811 2735747775	31143094388704875349009789759 0647432840	1
77558560	25009768713485665691636618504 7643035445	41906052984700195609662879407 8558796364	1

Теоретическая сложность

$O(\sqrt[4]{n}) => O(\sqrt[4]{5477994803565441481512291762735692170 93}) = 88378834775$

Время прошедшее с момента запуска: ≈ 8 часов.

Количество операций на момент завершения: 77558560

Время на одну итерацию: $\frac{28800}{77558560} \approx 0.37$ мс

Теоретическое время работы алгоритма: $8{,}838 \cdot 10^9 \cdot 0.37$ мс ≈ 378 дней

Результаты работы для третьего числа n = 886511645863588279798410662450007899671314462363584636040078917752 0228258163291:

Для третьего числа программа также не нашла нетривиальные множители по истечении 3 часов, что, по-видимому, также говорит о неудачно взятых начальных данных c=2, $f(x)=x^2+22$.

i	a	b	НОД
1	26	698	1
2	698	237389175098	1
3	487226	3175730538240678461306580834663554696424259898	1
4	237389175098	882512012099317232508256755790817633398382454388034707271785138247016944682697 8	1
5	56353620453708903309626	5936687544325307894094771159962175534072222254464509360305871935264086507450500	1
…	…	…	…
12260345	2376129958755241384832676151291831328428000731847049302832142124165156015933874	1929443368548394397377576831653119646470253995364986165358730154727711282831820	1
12260346	8731171724599472986664511877234502344497531263294891633220295144033645464075 31	1069063723050478856742101273689363079074868849230683636715224661697481313577 02	1
12260347	6302667054086642975227223803183349684695465471661256380626533595336733021852572	4064235791570375416662297386309243281518724570064728539430764555401093482845 07	1

122603 48	469385616265693170231431732875 850250680917141080586367078874 5194568616020459066	859777721949223208851427488599 070430099180897474747132547890 6503768118903574310	1
122603 49	688892934869913103813162158901 361636132334977726112347663684 405293610755666962	884432553553759681546936811903 900963533730912829667066015374 0236077097837724651	1

Теоретическая сложность

$$O\left(\sqrt[4]{n}\right) = 54565873429571684545$$

Время прошедшее с момента запуска: ≈ 3 часов.

Количество операций на момент завершения: 12260349

Время на одну итерацию: $\frac{10800}{12260349} \approx 0.88$ мс

Теоретическое время работы алгоритма: $54565873429571684545 * 0.88$ мс $\approx 1{,}523 * 10^9$ лет

- **(ρ-1)-метод Полларда**

Случайное целое a = 2, база разложения увеличивается на каждой итерации алгоритма на одно простое число.

Результаты работы для первого числа n = 16735883892800965609:

i	p_i	l_i
1	2	63
2	3	40
3	4	31
4	5	27
5	7	22
…	…	…
33820	398471	3
33821	398473	3

33822	398477	3
33823	398491	3
33824	398509	3

Время прошедшее с момента запуска: 8.00912 secs

Результат: $16735883892800965609 = 5618979227 * 2978456267$

Результаты работы для второго числа n = 54779948035654414815122917627356921709 3:

i	p_i	l_i
1	2	128
2	3	81
3	5	55
4	7	45
5	11	37
…	…	…
2430	21379	8
2431	21383	8
2432	21391	8
2433	21397	8
2434	21401	8

Время прошедшее с момента запуска: 0.745076 secs

Результат: $54779948035654414815122917627356921709 3 = 32813290267639542253 * 16694439231434948281$

Результаты работы для третьего числа n = 88651164586358827979841066245000789967131446236358463604007891775202282581632 91:

i	p_i	l_i
1	2	
2	3	
3	5	
4	7	
5	11	
…	…	…
10814950	194934197	9
10814951	194934203	9
10814952	194934217	9
10814953	194934221	9
10814954	194934253	9

Время, прошедшее с момента запуска – 4 часа.

В связи со слишком большой разрядностью числа, выяснить результат его факторизации не удалось. Исходя из этого, можно сказать, что сложность алгоритма не получится высчитать, т.к. для её расчёта нужно знать наибольшее число базы разложения, которое нет возможности узнать.

Сравнивая данное число с прошлым, имеющим в 2 раза меньше разрядов, можно сделать вывод, что алгоритм будет работать на несколько порядков дольше, чем с прошлым числом.

Сложность алгоритма: $B * \ln B * (\ln N)^2 = 99991 \times 11.51 \times 32630.52 \approx 122959669953151$

Время одной итерации: $\frac{2268072 \text{ мс}}{1097} \approx 2067.52 \text{ мс}$

Расчетное время до завершения:

$122959669953151 \times 2067.52 \text{ мс} \approx 2.54 * 10^{17} \text{часов} \approx 2.9 * 10^{13} \text{лет}$

- **Метод непрерывных дробей**

Для всех предложенных чисел не был получен верный ответ или время вычисления было достаточно велико. Возможно, данное явление связано с особенностями языка Python, а именно реализацией словаря, потому что база формируется достаточно долго даже для первых 200 простых чисел.

Т.к. для заданных чисел получить верный ответ не удалось, алгоритм был проверен на Примере 6.12 из учебного пособия.

Разложим на множители методом непрерывных дробей число $n = 21299881$. Результаты работы алгоритма:

B = {-1, 2, 3, 5, 7, 11, 19}

P_i $\{27691, 50767, 1389169, 12814433311, 2764443209657, 20276855015255, 127498600693396, 2390521616955537\}$

ei $\{[1,0,0,1,1,0,0], [0,1,1,0,1,0,0], [1,0,0,1,0,1,0], [0,0,0,1,1,1,0], [1,1,0,0,0,0,1], [1,0,0,0,0,1,0], [0,0,1,1,0,0,0], [1,0,0,0,1,0,0]\}$

$$s = 21196679$$

$$t = 18480$$

$$21299881 = 5531 * 3851$$

Результат верный.

Результаты работы для первого числа n = 16735883892800965609:

Объем базы разложения, рассчитанной по формуле $h \approx 5625$:

B = {-1, 2, 3, 5, 17, …, 121013, 121039, 121063, 121081, 121139}

P_i = {53182369051, 61364271982, 789553632835, 2307296626523, 4850473423593322, 9304942745329505, 14155416168922827, 23460358914252332, 186663333789835123, 1882666171109809355961…}

s = 63762425652672

t = 63762425652672

По истечению 3 часов работы программа выдала результат «Делитель не найден». Возможно, это связано с объемом базы разложения.

Результаты работы для второго числа n = 547799480356544148151229176273569217093:

Объем базы разложения, рассчитанной по формуле $h \approx 623442$:

B = [-1, 3, 17, 29, 31, 41, 47, 53, 67, 79, 83, 89, 97, 107, 113, 131, 157, 167, …}

Сложность алгоритма: $O\left(\exp\left(2 * \sqrt{\ln(n)\,(\ln(\ln(n)))}\right)\right) \approx 2{,}42 * 10^{17}$

Время одной итерации: $\frac{10800}{5625}$мс ≈ 1.92 мс

Расчетное время до завершения:

$$2{,}42 * 10^{17} * 1{,}92 \text{ мс} \approx 4{,}65 * 10^{17}\text{часов} \approx 1{,}47 * 10^{7} \text{ лет}$$

Результаты работы для третьего числа n = 886511645863588279798410662450007899671314462363584636040078917752022825816329 1:

Объем базы разложения, рассчитанной по формуле $h \approx 801858025$:

B = [-1, 5, 11, 19, 23, 31, 41, 43, 71, 73, 89, 107, 113, 137, 139, 149, 167, …}

Сложность алгоритма: $O\left(\exp\left(2 * \sqrt{\ln(n)\,(\ln(\ln(n)))}\right)\right) \approx 5{,}16 * 10^{26}$

Расчетное время до завершения:

$$5{,}16 * 10^{26} * 1{,}92 \text{ мс} \approx 9{,}91 * 10^{26}\text{часов} \approx 3{,}14 * 10^{16} \text{ лет}$$

4. Вывод

В результате выполнения данной лабораторной работы было реализовано три алгоритма разложения чисел на простые множители. Из вычислений видно, что задача факторизации оказалась невыполнимой для чисел большого порядка. Реализованным алгоритмам удалось найти верный ответ лишь для первых двух чисел. Третье число оказалось большим для всех алгоритмов.

Также не удалось получить ответ для чисел варианта за конечное время методом непрерывных дробей.

Данная работа показывает, что вычислительные мощности оказались малы для решения задачи факторизации. Однако, используя компьютер с

большой вычислительной мощностью, и возможно более разумно реализованный алгоритм, можно получить результаты. Именно на этом принципе основаны многие алгоритмы современной криптографии.

Приложение 1

```
from alg_evklida import gcd_us
from timeit import default_timer as timer


def rohP(n, c):
    a = c
    b = c
    i = 1
    while 1:
        a = func(a, n)
        b = func(b, n)
        b = func(b, n)
        d = gcd_us(a - b, n)
        print(i, ". a: ", a, "; b: ", b, "; НОД: ", d)
        if d == n:
            return -1
        if d != 1:
            break
        i += 1
        if i == 6:
            input()
    return d


def func(x, n):
    return ((x * x) + 22) % n


num = int(input())
start = int(input())
t = timer()
while 1:
    res = rohP(num, start)
    if res != -1:
        break
    start += 1
print("Total elapsed: {:g} secs\n".format(timer() - t))
```

Приложение 2

```
from math import log as ln
from alg_evklida import gcd_us, simple
from timeit import default_timer as timer

def roh1P(num):
    i = 1
    p = 2
    a = 2
    d = gcd_us(a, num)
    if d > 1:
        return d
    else:
        while 1:
            l = int(ln(num) / ln(p))
            print(i, ". База:", p, "; l_i:", l)
            a = pow(a, pow(p, l), num)
            d = gcd_us(a - 1, num)
            if d != 1 and d != num:
                return d
            p += 1
            if p > 5:
                while simple(p) != 0:
                    p += 1
                if p > 10000:
                    print(p)
            i += 1


num = int(input())
t = timer()
delit = roh1P(num)
print("Total elapsed: {:g} secs\n".format(timer() - t))
print(num, "=", delit, "*", num / delit)
```

Приложение 3

```
import mpmath

#  проверить s на гладкость
def check_smoothness(s, B):
    list_alph = []
    list_alph.append(0)

    while (s & 1) == 0:
        s >>= 1
        list_alph[0] += 1

    for i in range(2, len(B)):
        temp = B[i]
        list_alph.append(0)
        while (s % temp == 0):
            s /= temp
            list_alph[len(list_alph) - 1] += 1

    if (s == 1):
        return list_alph
    return False

#  сделать бинарный вектор
def make_binvector(vector):
    bin_vector = 0
    for i in range(len(vector)):
        bin_vector |= ((vector[i] % 2) << (len(vector) - 1 - i))
    return bin_vector

# найти B-гладкие числители подходящих дробей
def make_base_B(n, B, h_2):
    mpmath.mp.dps = 100000
    q = mpmath.mp.sqrt(n)

    P_min_1 = 1
    P_0 = int(q)
    list_P = []
    cur_a = P_0

    q = 1 / (q - cur_a)
    cur_a = int(q)

    matrix = []
    bin_matrix = {}
    list_alph = check_smoothness((P_0 * P_0) % n, B)
    if (list_alph != False):
        matrix.append(list_alph)
        list_P.append(P_0)

    j = 0
    P_min_2 = P_min_1
    P_min_1 = P_0
    while (len(list_P) < h_2):
        P_i = P_min_1 * cur_a + P_min_2

        list_alph = check_smoothness((P_i * P_i) % n, B)
        if (list_alph != False):
            vector = [0]
            vector.extend(list_alph)
            bin_vector = make_binvector(vector)
            matrix.append(vector)
```

```
            bin_matrix.update({j: bin_vector})
            j += 1
            list_P.append(P_i)
        else:
            list_alph = check_smoothness(abs(((P_i * P_i) % n) - n), B)
            if (list_alph != False):
                vector = [1]
                vector.extend(list_alph)
                bin_vector = make_binvector(vector)
                matrix.append(vector)
                bin_matrix.update({j: bin_vector})
                j += 1
                list_P.append(P_i)

        P_min_2 = P_min_1
        P_min_1 = P_i

        q = 1 / (q - cur_a)
        cur_a = int(q)
    return list_P, matrix, bin_matrix

# исключить одинаковые элементы из списка
def exclude_same_elements(list_elem):
    i = 0
    while (i < len(list_elem)):
        val = list_elem[i]
        count = list_elem.count(val)
        if (count & 1) == 1:
            count -= 1

        came_in = False
        for j in range(count):
            list_elem.remove(val)
            came_in = True

        if (came_in == True):
            i -= 1
        i += 1
    return list_elem

# найти нулевую сумму векторов
# метод Гаусса
def gauss_method(bin_matrix):
    dict_vec_sum = {}

    for i in range(len(bin_matrix)):
        dict_vec_sum.update({i: [i]})

    key = max(bin_matrix, key=bin_matrix.get)
    max_val = bin_matrix[key]

    bit = 1
    while (bit < max_val):
        bit <<= 1
    if (bit > max_val):
        bit >>= 1

    while (len(bin_matrix) > 1):
        key = min(bin_matrix, key=bin_matrix.get)

        if (bin_matrix[key] == 0):
            return exclude_same_elements(dict_vec_sum[key])
```

```
        key = max(bin_matrix, key=bin_matrix.get)
        max_val = bin_matrix.pop(key)

        while (bit & max_val) == 0:
            bit >>= 1

        for i in bin_matrix.keys():
            if (bin_matrix[i] & bit) == 0:
                continue
            bin_matrix[i] ^= max_val
            dict_vec_sum[i].extend(dict_vec_sum[key])
            dict_vec_sum[i] = exclude_same_elements(dict_vec_sum[i])

        dict_vec_sum.pop(key)
    return False

# символ Якоби
def JacobiSymbol(a, n):
    g = 1
    while a != 0:
        k = 0
        while a & 1 == 0:
            k += 1
            a = a >> 1
        if k & 1 != 0 and (n % 8 == 3 or n % 8 == 5):
            g *= -1
        if a % 4 == 3 and n % 4 == 3:
            g *= -1
        c = a
        a = n % c
        n = c
    return g

# прочитать базу простых чисел
def read_primes(primes, h, num):
    with open(primes) as file:
        text = file.read()

    from_file = text.split(',')
    num_list = []

    for n in range(1000000):
        if (len(num_list) < h):
            temp = int(from_file[n])
            if (JacobiSymbol(num, temp) == 1):
                num_list.append(temp)
        else:
            return num_list

# НОД
def gcd(a, b):
    return a if b == 0 else gcd(b, a % b)

# метод непрерывных дробей
def fraction_method(n, h):
    B = [-1]
    B.extend(read_primes("primes.txt", h, n))

    h = len(B) - 1
    print("h:", h)
    print("base:", B)
    ret = make_base_B(n, B, h + 2)
```

```
    list_P = ret[0]
    print("P_i:", list_P)
    matrix = ret[1]
    print("Bin vectors:", matrix)
    bin_matrix = ret[2].copy()
    print("Bin matrix:", bin_matrix)

    sum_vectors_idx = gauss_method(bin_matrix)
    print("sum_vectors_idx", sum_vectors_idx)

    # считаем s
    s = 1
    for i in sum_vectors_idx:
        s *= list_P[i]
    s %= n
    print("s = ", s)

    # считаем t
    t = 1
    for i in range(1, len(B)):
        deg_sum = 0
        for j in sum_vectors_idx:
            deg_sum += matrix[j][i]
        deg_sum >>= 1
        t *= pow(B[i], deg_sum)
    t %= n
    print("t = ", t)

    if (s % n == t % n) or (s % n == (t % n) - n):
        print("Error!")
        return

    result = gcd(abs(s - t), n)
    print("Result: {} = {} * {}".format(n, result, n // result))

# 164023
# 850441
# 16735883892800965609
num = int(input())
h = int(input())
fraction_method(num, h)
```