НАО «Восточно-Казахстанский технический университет имени Д. Серикбаева»

Дисциплина «Теоретико-числовые методы в криптографии»

САМОСТОЯТЕЛЬНЫЙ ПРАКТИКУМ К ЛЕКЦИИ 14

Факторизация, проверка простоты и дискретный логарифм

Методические указания и задания для самостоятельной работы

Статус: дополнительный самостоятельный практикум к теме 14. Аудиторные практические часы для этой темы рабочей программой не выделены. Сначала изучите методические указания и разберите примеры, затем выполните свой вариант.

1 Цель и результаты работы

Освоить единую цепочку вычислений: факторизация заданного числа, проверка кандидата на простоту, построение простого модуля, проверка образующего элемента и решение задачи дискретного логарифмирования. Научиться сопровождать результат проверяемыми промежуточными данными.

Средства: Python 3.8 или новее, стандартная библиотека, текстовый редактор или среда разработки. Внешние пакеты не требуются. Все входные значения в данной работе — целые числа; диапазоны допустимых значений указаны в функциях.

Перед началом: повторите алгоритм Евклида, точное вычисление целого квадратного корня, теорему Ферма и понятие порядка элемента. Заведите один файл practice14.py и поместите в него блоки Python из раздела 4 в указанном порядке.

2 Математическая памятка

Метод	Основное вычисление	Критерий результата
Ферма	$D = a^2 - N$	Полный квадрат D даёт пару a − b, a + b
Ро Полларда	$d = \gcd(\lvert x - y \rvert, N)$	Нетривиальный делитель: 1 < d < N
Миллер — Рабин	$w - 1 = 2^s d$	Первый остаток 1 или некоторый остаток −1
Сертификат	$R = g^{(p-1)/r} \bmod p$	Полная факторизация; все НОД(R − 1, p) = 1
Образующий	$g^{(p-1)/r} \bmod p$	Для простого p все эти остатки отличны от 1
BSGS	$g^j = h(g^{-m})^i$	Совпадение даёт x = im + j

В таблице r пробегает различные простые делители $p - 1$. Проверка сертификата включает также условие $a^{p-1} \equiv 1 \pmod{p}$. При проверке предполагаемой простоты нельзя заранее пользоваться свойствами поля по этому модулю.

В методе Ферма считаем **проверки разности** $D = a^2 - N$, включая начальное значение. В ро-методе одна итерация означает один шаг переменной x, два шага переменной y и одно вычисление НОД. Эти счётчики измеряют разную работу; сравнивать их как время выполнения напрямую нельзя.

3 Разобранный пример

3.1 Факторизация методом Ферма

Разложим $N = 5959$. Начальное значение равно $a_0 = 78$.

a	$D = a^2 - N$	b	**Полный квадрат?**
78	125	11	нет
79	282	16	нет
80	441	21	да

Последняя строка даёт $a = 80, b = 21$, поэтому множители равны 59 и 101. Проверяем их произведение и простоту. Для 59 достаточно простых делителей 2, 3, 5, 7; для 101 — также 2, 3, 5, 7. Число проверок разности равно 3.

3.2 Ро-метод Полларда

Для наглядной короткой трассы возьмём $N = 8051$, $f(x) = x^2 + 1 \pmod N$, $x_0 = y_0 = 2$.

Итерация	x	y	d
1	5	26	1
2	26	7474	1
3	677	871	97

Нетривиальный НОД 97 найден на третьей итерации. Второй множитель равен $8051/97 = 83$. Если получилось $d = N$, меняем параметр c и повторяем поиск с начального значения 2. В индивидуальных вариантах последовательно используются $c = 1,2,\dots,8$ с ограничением 1000 итераций на запуск.

3.3 Проверка кандидата по двум основаниям

Пусть $w = 2047$. Разложение $w - 1 = 2 \cdot 1023$ даёт $s = 1, d = 1023$. Оба основания 2 и 3 взаимно просты с w. По основанию 2 цепочка состоит из одного остатка 1: раунд пройден. По основанию 3 остаток равен 1565: раунд не пройден, число составное. Точное разложение $2047 = 23 \cdot 89$ подтверждает результат.

При положительном результате пишите «раунд пройден», а не «число простое». При двух положительных результатах проверьте учебное число пробными делениями. Для $w = 9, a = 3$ сначала находится $\gcd(3,9) = 3$; цепочка степеней в нашей реализации остаётся пустой.

3.4 Построение модуля и сертификат

Найдём наименьшее простое $p \geq 90$. Чётные числа больше 2 сразу исключаются. Нечётные кандидаты 91, 93, 95 составные; их наименьшие простые делители равны соответственно 7, 3, 5. Следующий кандидат 97 прост.

Разложим $p - 1 = 96 = 2^5 \cdot 3$. Наименьший образующий равен 5: число 2 не подходит, поскольку $2^{48} \equiv 1$; для 3 также $3^{48} \equiv 1$; для 4 — $4^{48} \equiv 1 \pmod{97}$.

При $g = 5$ проверяем $5^{96} \equiv 1$, а для простых делителей 2 и 3 получаем остатки 96 и 35. НОД чисел 95 и 34 с 97 равны 1. Таким образом, одна таблица степеней подтверждает и сертификат простоты, и порядок образующего элемента.

В программе поиск следующего простого первоначально использует точную проверку пробными делениями. Затем отдельно строится и проверяется сертификат. Это два способа обоснования, которые в учебной работе можно сопоставить.

3.5 Дискретный логарифм

Решим $5^x \equiv 56 \pmod{97}$ при $0 \leq x < 96$. Выбираем $m = 10$ и вычисляем $v = 5^{-10} \equiv 11 \pmod{97}$.

j	Малый шаг	i	Большой шаг
0	1	0	56
1	5	1	34
2	25	2	83
3	28	3	40
4	43	—	—
5	21	—	—
6	8	—	—
7	40	—	—
8	6	—	—
9	30	—	—

Совпадение 40 найдено при $i = 3$, $j = 7$. Получаем $x = 3 \cdot 10 + 7 = 37$. Исходное сравнение выполняется. В массиве больших шагов значения после первого совпадения уже не нужны.

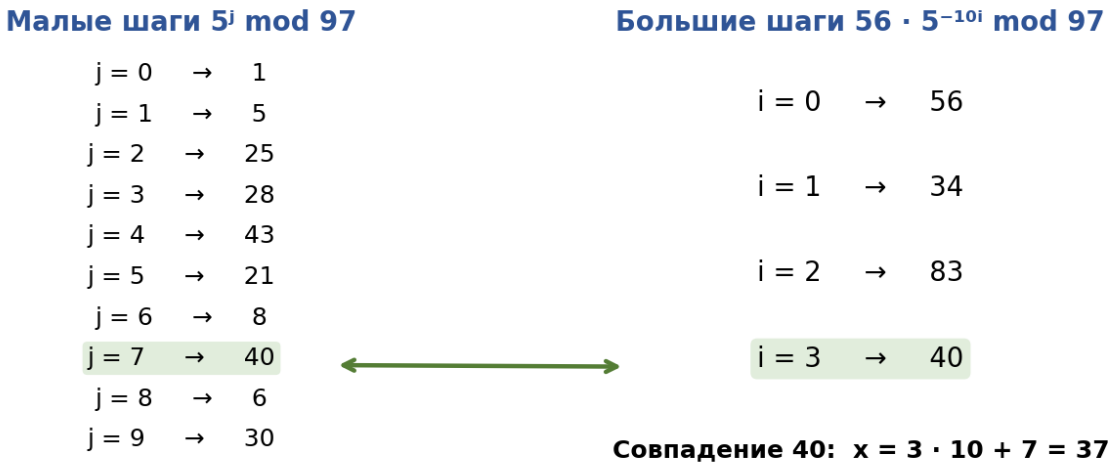


Рисунок 1. Поиск совпадения при вычислении дискретного логарифма.

4 Программные примеры

4.1 Точная факторизация и проверка малых чисел

Функция `factor_trial` возвращает простые множители с повторениями. Например, для 96 результат равен `[2, 2, 2, 2, 2, 3]`. Функция `is_prime_trial` даёт точный ответ; её скорость достаточна для чисел из вариантов. `isqrt` вычисляет целую часть квадратного корня без ошибки округления.

```
from math import gcd, isqrt, prod
from secrets import randbits
def factor_trial(n):
    if n < 2:
        raise ValueError('Требуется целое n >= 2')
    factors, d = [], 2
    while d * d <= n:
        while n % d == 0:
            factors.append(d)
            n //= d
        d = 3 if d == 2 else d + 2
    if n > 1:
        factors.append(n)
    return factors
def is_prime_trial(n):
    if n < 2:
        return False
    if n % 2 == 0:
        return n == 2
    return all(n % d for d in range(3, isqrt(n) + 1, 2))
```

4.2 Метод Ферма

Возвращаются пара множителей и список строк `(a, delta, b)`, где `b` — целая часть квадратного корня из `delta`. Квадрат найден только при `b*b == delta`. Значение `None` вместо пары означает, что нетривиальная пара не получена: закончился лимит либо достигнута тривиальная пара. Эти причины различаются по последней строке трассы.

```
def fermat(n, max_checks=10000):
    if n < 3 or n % 2 == 0 or max_checks < 1:
        raise ValueError('Нужно нечётное n >= 3 и max_checks >= 1')
    a = isqrt(n)
    a += (a * a < n)
    trace = []
    for _ in range(max_checks):
        delta = a * a - n
        b = isqrt(delta)
        trace.append((a, delta, b))
        if b * b == delta:
            pair = (a - b, a + b)
            return (pair if pair[0] > 1 else None), trace
        a += 1
    return None, trace
```

4.3 Ро-метод и повторные запуски

Один запуск возвращает делитель или `None` и строки `(i, x, y, d)`. Если последняя строка содержит `d == n`, траектория не дала нетривиального делителя. Если все НОД равны 1, исчерпан лимит. Обе ситуации требуют изменения параметров или другого метода, а не вывода о простоте.

```
def rho(n, c=1, x0=2, max_steps=1000):
    if n < 3 or n % 2 == 0 or max_steps < 1:
        raise ValueError('Нужно нечётное n >= 3 и max_steps >= 1')
    x = y = x0 % n
    trace = []
    for i in range(1, max_steps + 1):
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        d = gcd(abs(x - y), n)
        trace.append((i, x, y, d))
        if d != 1:
            return (d if d < n else None), trace
    return None, trace
```

Функция повторных запусков возвращает отсортированную пару множителей и список `(c, trace)` для всех использованных значений параметра. Найденная пара в общем случае ещё может требовать дальнейшей факторизации.

```
def rho_restarts(n):
    attempts = []
    for c in range(1, 9):
        d, trace = rho(n, c=c)
        attempts.append((c, trace))
        if d is not None:
            return tuple(sorted((d, n // d))), attempts
    return None, attempts
```

4.4 Один раунд Миллера — Рабина

Формат результата: `(passed, s, d, chain, common)`, где `passed` означает прохождение раунда, `chain` содержит фактически вычисленные остатки, `common` — НОД основания и модуля. При составности, найденной по НОД, цепочка пуста. Функция рассчитана на нечётные числа не меньше 5; специальные случаи 2 и 3 обрабатываются вне неё.

```
def mr_round(n, a):
    if n < 5 or n % 2 == 0 or not 2 <= a <= n - 2:
        raise ValueError('Нужно нечётное n >= 5 и 2 <= a <= n-2')
    s, d = 0, n - 1
    while d % 2 == 0:
        s, d = s + 1, d // 2
    common = gcd(a, n)
    if common > 1:
        return False, s, d, [], common
    x = pow(a, d, n)
    chain = [x]
    if x in (1, n - 1):
        return True, s, d, chain, common
    for _ in range(s - 1):
        x = x * x % n
        chain.append(x)
        if x == n - 1:
            return True, s, d, chain, common
        if x == 1:
            break
    return False, s, d, chain, common
```

4.5 Два способа построить учебное простое число

`next_prime` возвращает наименьшее простое не меньше `start` и список отброшенных **нечётных** кандидатов с их наименьшими простыми делителями. Чётные числа больше 2 пропускаются без записи в этот список.

```
def next_prime(start):
    if start <= 2:
        return 2, []
    candidate = start if start % 2 else start + 1
    rejected = []
    while not is_prime_trial(candidate):
        rejected.append((candidate, factor_trial(candidate)[0]))
        candidate += 2
    return candidate, rejected
```

`random_small_prime` генерирует равномерные нечётные кандидаты заданной битовой длины и точно отбрасывает составные. Диапазон от 3 до 16 битов выбран для учебного эксперимента с пробными делениями. Результат разных запусков может различаться. Если установленный лимит исчерпан, функция сообщает об этом исключением и не возвращает непроверенный кандидат.

```
def random_small_prime(bits):
    if not 3 <= bits <= 16:
        raise ValueError('Учебный диапазон: от 3 до 16 битов')
    for _ in range(10000):
        candidate = randbits(bits) | (1 << (bits - 1)) | 1
        if is_prime_trial(candidate):
            return candidate
    raise RuntimeError('Лимит кандидатов исчерпан')
```

Битовая операция `|` устанавливает старший и младший разряды в 1. Остальные разряды остаются равномерными; каждый допустимый нечётный кандидат имеет одинаковое число исходных битовых строк. Ограничение числа попыток сохраняет равномерность результата при условии успешного завершения.

4.6 Сертификат простоты

Передайте полную факторизацию `n-1` как список с повторениями. Функция проверит произведение, простоту каждого множителя, условие для полной степени и НОД для каждого различного простого делителя. Результат — логическое значение и строки `(q, residue, gcd_value)`. Неверная структура факторизации вызывает `ValueError`; корректно заданные данные с неудачным свидетелем дают `False`.

```
def lucas_certificate(n, a, factors):
    if n <= 2 or not 1 < a < n:
        raise ValueError('Требуется n > 2 и 1 < a < n')
    if prod(factors) != n - 1:
        raise ValueError('Нужна полная факторизация n-1')
    if not all(is_prime_trial(q) for q in factors):
        raise ValueError('Все переданные множители должны быть простыми')
    checks = []
    for q in sorted(set(factors)):
        residue = pow(a, (n - 1) // q, n)
        checks.append((q, residue, gcd(residue - 1, n)))
    valid = pow(a, n - 1, n) == 1
    valid = valid and all(t[2] == 1 for t in checks)
    return valid, checks
```

Логическое `False` здесь означает, что данный свидетель не подтвердил простоту. Например, простой модуль может иметь выбранный элемент малого порядка. Из одного неудачного свидетеля нельзя автоматически заключать, что модуль составной.

4.7 Поиск наименьшего образующего

Модуль предварительно проверяется точным учебным методом. В цикле испытываются кандидаты по возрастанию. Для модуля 2 единственный ненулевой элемент равен 1; основные задания используют модули не меньше 5.

```
def primitive_root(p):
    if not is_prime_trial(p):
        raise ValueError('Модуль должен быть простым')
    if p == 2:
        return 1
    divisors = set(factor_trial(p - 1))
    for g in range(2, p):
        if all(pow(g, (p - 1) // q, p) != 1 for q in divisors):
            return g
    raise RuntimeError('Образующий не найден')
```

4.8 Большие и малые шаги

Функция рассчитана на образующий всей группы ненулевых остатков. Она сначала проверяет эти условия. Обратный элемент для большого шага находится по теореме Ферма, поскольку простота модуля уже установлена. Возвращаются `(x, m, baby_values, giant_values, (i, j))`; индексы массивов начинаются с нуля.

```
def bsgs(p, g, h):
    if p < 5 or not is_prime_trial(p):
        raise ValueError('Требуется простое p >= 5')
    if not 1 < g < p or not 1 <= h < p:
        raise ValueError('Требуется 1 < g < p и 1 <= h < p')
    if any(pow(g, (p - 1) // q, p) == 1
           for q in set(factor_trial(p - 1))):
        raise ValueError('g должен порождать всю группу')
    order = p - 1
    m = isqrt(order)
    m += (m * m < order)
    baby, value = {}, 1
    for j in range(m):
        baby.setdefault(value, j)
        value = value * g % p
    step = pow(pow(g, m, p), p - 2, p)
    giant, value = [], h
    for i in range(m):
        giant.append(value)
        if value in baby:
            x = i * m + baby[value]
            if x < order and pow(g, x, p) == h:
                return x, m, list(baby), giant, (i, baby[value])
        value = value * step % p
    raise RuntimeError('Логарифм не найден')
```

Порядок вставки ключей словаря соответствует порядку малых шагов. В рассматриваемой группе эти значения различны. Оценка порядка $\sqrt{p-1}$ групповых операций относится к ядру BSGS, а не к предварительной проверке простоты и факторизации $p-1$. Запись трасс также добавляет память к реализациям алгоритмов.

4.9 Полный запуск разобранного примера

Добавьте следующий блок после всех определений. Аргумент 56 в вызове `bsgs` — правая часть сравнения для дискретного логарифма.

```
pair, trace = fermat(5959)
print("Ферма:", pair, "проверок:", len(trace))
pair, attempts = rho_restarts(8051)
print("Po:", pair, "последняя строка:", attempts[-1][1][-1])
for a in (2, 3):
    print("MR 2047, a =", a, ":", mr_round(2047, a))
p, rejected = next_prime(90)
g = primitive_root(p)
print("Простое и образующий:", p, g)
print("Отброшены:", rejected)
print("Сертификат:", lucas_certificate(p, g, factor_trial(p - 1)))
result = bsgs(p, g, 56)
print("BSGS: x, m, (i, j):", result[0], result[1], result[4])
print("Малые шаги:", result[2])
print("Большие шаги:", result[3])
```

Ожидаемый вывод

```
Ферма: (59, 101) проверок: 3
Po: (83, 97) последняя строка: (3, 677, 871, 97)
MR 2047, a = 2 : (True, 1, 1023, [1], 1)
MR 2047, a = 3 : (False, 1, 1023, [1565], 1)
Простое и образующий: 97 5
Отброшены: [(91, 7), (93, 3), (95, 5)]
Сертификат: (True, [(2, 96, 1), (3, 35, 1)])
BSGS: x, m, (i, j): 37 10 (3, 7)
Малые шаги: [1, 5, 25, 28, 43, 21, 8, 40, 6, 30]
Большие шаги: [56, 34, 83, 40]
```

Если вывод отличается, сначала проверьте порядок блоков, исходные значения и отступы. Не изменяйте индивидуальные данные так, чтобы они совпали с разобранным примером.

5 Порядок выполнения индивидуального варианта

Задание 1. Факторизация методом Ферма — 15 баллов. Для своего N вычислите a_0, выпишите все проверенные значения D, найдите успешные a_* и b. Укажите множители и число проверок. Подтвердите произведение и простоту обоих множителей. Сопоставьте число проверок с формулой из лекции.

Задание 2. Po-метод Полларда — 15 баллов. Разложите то же N с начальным значением 2. Начните с $c = 1$ и при необходимости увеличивайте c до 8. Для каждого запуска укажите число итераций и причину остановки. Для успешного запуска выпишите первые три строки и последнюю строку `(i, x, y, d)`. Объясните различие с числом проверок метода Ферма.

Задание 3. Проверка кандидата — 20 баллов. Для w выполните раунды по основаниям 2 и 3. Укажите s, d, НОД с основанием, цепочку фактически вычисленных остатков и результат каждого раунда. Найдите точную факторизацию w пробными делениями. Сформулируйте итоговый вывод о простоте и отдельно вывод, который был бы возможен только по двум раундам.

Задание 4. Построение простого и образующего — 20 баллов. Найдите наименьшее простое $p \geq A$, перечислите отброшенные нечётные кандидаты с делителями. Разложите $p - 1$ и найдите наименьший образующий g. Для каждого меньшего кандидата $2 \leq a < g$ предъявите хотя бы один простой делитель r числа $p - 1$, для которого $a^{(p-1)/r} \equiv 1 \pmod{p}$. Постройте сертификат простоты со свидетелем g: приведите все остатки и НОД. Дополнительно получите одно случайное 8-битовое простое функцией `random_small_prime(8)` и проверьте его битовую длину и простоту.

Задание 5. Дискретный логарифм — 20 баллов. С найденными p и g решите $g^x \equiv h \pmod{p}$. Выпишите m, множитель большого шага, все малые шаги и большие шаги до первого совпадения,

индексы i, j, показатель x. Проверьте диапазон и исходное сравнение. Укажите число сохранённых малых шагов и вычисленных больших шагов.

Задание 6. Контрольные случаи и отчёт — 10 баллов. Проверьте `fermat(49)`, `fermat(5959, max_checks=2)`, `bsgs(97, 5, 1)` и попытку `lucas_certificate(97, 5, [2, 3])`. Объясните каждый результат. В отчёте свяжите промежуточные вычисления с математическими условиями корректности.

6 Индивидуальные варианты

В таблице N — число для двух методов факторизации; w — кандидат для тестов; A — нижняя граница поиска простого; h — правая часть сравнения. Модуль p и образующий g нужно вычислить по заданию 4. Все значения приведены в десятичной записи.

№	N	w	A	h
1	5959	97	100	78
2	8051	341	123	106
3	6893	101	146	121
4	8509	2047	169	171
5	9301	109	192	7
6	10147	1373653	215	46
7	11771	113	238	5
8	13439	91	261	215
9	15811	127	284	149
10	17473	25326001	307	193
11	18643	131	330	141
12	20437	15	353	125
13	21473	137	376	316
14	23843	25	399	231
15	28321	139	422	400
16	30523	121	445	232
17	32743	149	468	365
18	34889	703	491	96
19	39187	157	514	361
20	43489	9	537	476

7 Требования к отчёту и оценивание

Отчёт включает номер варианта и исходные данные, краткое описание каждого метода, требуемые трассы и таблицы, исходный код, результаты запуска и проверку ответов. Приложите объяснение хотя бы одного случая, когда алгоритм не позволяет сделать вывод о простоте. Программа без математической интерпретации не заменяет отчёт.

Максимум — 100 баллов; распределение указано в заданиях. Для полного балла необходимы правильный результат, требуемые промежуточные данные и обоснование. Частичный балл

выставляется за корректные этапы даже при последующей арифметической ошибке. Допускается иная реализация, если она сохраняет условия алгоритма; для сопоставления трасс выполните также запуск с указанными параметрами.

В заключении ответьте: какие результаты дают доказательство, какие — вероятностное свидетельство, а какие зависят от лимита поиска? Почему таблица степеней используется и для проверки порядка элемента, и при решении обратной задачи?

8 Контрольные вопросы

1. Почему для проверки полного квадрата используется `isqrt`, а не округление вещественного корня?
2. Что означает возвращённое значение `None` в каждом из двух методов факторизации?
3. Почему при $d = N$ в ро-методе меняется траектория?
4. Почему `True` из `mr_round` нельзя безусловно переименовать в «простое»?
5. Чем различаются результаты проверки структуры сертификата и проверки его свидетеля?
6. Почему в BSGS используется порядок группы, а не числовое значение модуля?
7. Когда дискретный логарифм равен нулю?
8. Почему результат случайной генерации не обязан совпасть с результатом другого студента?

9 Литература

Основной материал — лекция 14 данного комплекта. Для дополнительного чтения: Boneh D., Shoup V., *A Graduate Course in Applied Cryptography*, версия 0.6, январь 2023, § 16.1.1; Katz J., Lindell Y., *Introduction to Modern Cryptography*, 2-е изд., 2015, § 9.1.2; Trappe W., Washington L. C., *Introduction to Cryptography with Coding Theory*, 2-е изд., 2006, § 6.3, 6.4, 7.2.2. Используются предоставленные пользователем издания.