Поиск дискретного логарифма

Сергей Николенко

Computer Science Club, 2015

Outline

Задача

- На прошлой лекции мы узнали, как раскладывать числа на множители.
- Теперь попробуем решать другую базовую задачу криптографии: дискретный логарифм.

Задача

- На прошлой лекции мы узнали, как раскладывать числа на множители.
- Теперь попробуем решать другую базовую задачу криптографии: дискретный логарифм.
- *Дискретный логарифм*: в циклической группе G по $g \in G$ и $y \in G$ найти такой x, что $g^x = y$.
- Этот x определяется с точностью до порядка g; если $\langle g \rangle = G$, то логарифм определён с точностью до $|G| = n$. Мы будем считать, что $\langle g \rangle = G$.

Сложность общей задачи

- Известно, что если не пользоваться ничем, кроме групповой операции и взятия обратного, то ничего лучше, чем $\sqrt{n}$, не будет: когда алгоритм обращается за определёнными умножениями, можно по ходу строить группу так, что ему придётся обращаться $\Omega(\sqrt{p})$ раз, где p — наибольший простой делитель n [Shoup, 1997].
- Мы сначала рассмотрим методы, достигающие этой цели, а потом перейдём к специфически числовым методам, работающим не во всех группах.

Тривиальный подход

- Тривиальный подход: возводить $g, g^2, g^3, \ldots$, пока не наткнёмся на y.
- Требует примерно $\frac{n}{2}$ операций, имеет смысл только для маленьких n.

Атака на гладкие модули

- Пусть $n = p_1^{k_1} p_2^{k_2} \dots p_l^{k_l}$.
- Заметим, что для каждого из этих p порядок элемента g^{n/p^k} равен p^k, и порядок элемента y^{n/p^k} не превосходит p^k.
- Иначе говоря, g^{n/p^k} порождает подгруппу G порядка p^k, а y^{n/p^k} лежит в этой подгруппе.
- И если мы можем найти логарифм в этой подгруппе:

$$\left(g^{n/p^k}\right)^{x'} = y^{n/p^k}, \text{ то, с другой стороны,}$$

$$\left(g^{n/p^k}\right)^{x} = y^{n/p^k}, \text{ и тем самым}$$

$$x' \equiv x \pmod{p^k}.$$

Атака на гладкие модули

- Тогда, если мы найдём логарифмы по модулям простых чисел

$$\begin{aligned} \left(g^{n/p_1^{k_1}}\right)^{x_1} &= y^{n/p_1^{k_1}}, \\ \left(g^{n/p_2^{k_2}}\right)^{x_2} &= y^{n/p_2^{k_2}}, \\ \vdots & \quad \vdots, \\ \left(g^{n/p_l^{k_l}}\right)^{x_l} &= y^{n/p_l^{k_l}}, \end{aligned}$$

то сможем по китайской теореме об остатках восстановить x, потому что

$$\begin{aligned} x &\equiv x_1 \pmod{p_1^{k_1}}, \\ x &\equiv x_2 \pmod{p_2^{k_2}}, \\ \vdots & \quad \vdots \quad \vdots, \\ x &\equiv x_l \pmod{p_l^{k_l}} \end{aligned}$$

Атака на гладкие модули

- Оказывается, найти логарифм по модулю p^k для маленького простого p легко, даже если k большое.
- Разложим предполагаемый логарифм x' по основанию p:

$$x' = z_0 + z_1 p + z_2 p^2 + \ldots + z_k p^k.$$

- Положим сначала $y_0 = y^{n/p}$, $g_0 = g^{n/p}$. Порядок g_0 не больше p, значит,

$$y_0 = y^{n/p} = g^{x \cdot n/p} = g_0^x = g_0^{z_0}.$$

- Тем самым мы нашли z_0. Теперь можно его вычесть, положить $y_1 = \left(y g_0^{-z_0}\right)^{n/p^2}$ и продолжать.
- В итоге найдём логарифм по модулю p^k за k поисков логарифма по модулю p.

Атака на гладкие модули

- Значит, гладкие модули использовать нельзя.
- Нужно выбирать такие n, у которых есть большие простые делители.
- Либо, в крайнем случае, разложение n неизвестно, но есть причины полагать, что большие простые делители есть.

Baby-step–Giant-step

- Shanks, 1973: алгоритм, работающий за $O(\sqrt{n})$; стандартный time-space tradeoff.
  1. Запишем x в виде $x = im + j$ для какого-то m. Тогда $y \cdot (g^{-m})^i = g^j$.
  2. Предвычислим g^j и будем перебирать i, умножая y на g^{im} и проверяя, нет ли его среди g^j.
- Если записать g_j в хеш-таблицу, можно считать, что проверка на равенство происходит в среднем за константное время.

Baby-step–Giant-step

- Алгоритм записывает два массива. Первый (giant steps):

$$S = \left\{ \left(i, g^{i\lceil\sqrt{n}\rceil} \right) \mid i = 0..\left\lceil\sqrt{n}\right\rceil \right\}.$$

- Второй (baby steps):

$$T = \left\{ \left(j, y \cdot g^{j} \right) \mid j = 0..\left\lceil\sqrt{n}\right\rceil \right\}.$$

- Как только списки пересекутся, логарифм можно будет найти как

$$\log_g y \equiv i\left\lceil\sqrt{n}\right\rceil - j \pmod{n}.$$

- Однако этот алгоритм требует экспоненциальной памяти.

ρ-метод Полларда

- Pollard, 1978. Суть — «birthday paradox»: мы выбираем псевдослучайную последовательность элементов в группе и ждём цикла. Цикл будет в среднем через $O(\sqrt{n})$ элементов.
- Разобьём группу на три части (не подгруппы) S_1, S_2, S_3. Будем вычислять

$$a_{i+1} = \begin{cases} y \cdot a_i, & \text{если } a_i \in S_1, \\ a_i^2, & \text{если } a_i \in S_2, \\ g \cdot a_i, & \text{если } a_i \in S_3. \end{cases}$$

ρ-метод Полларда

- Если в последовательности найдётся цикл, это с большой вероятностью приведёт к тому, что мы найдём дискретный логарифм, потому что мы найдём соотношение вида $g^a y^b = g^c y^d$.
- Но, казалось бы, всё равно надо хранить всю последовательность, и с памятью лучше не становится. Что делать?

Алгоритм Флойда для поиска цикла

- Алгоритм Флойда, он же «tortoise-and-hare algorithm».
- Общая постановка: хотим найти цикл в последовательности $a_{i+1} = f(a_i)$.
- Давайте будем хранить всего два указателя: u и v, причём $u_i = a_i$ (черепаха), а $v_i = a_{2i}$ (заяц).
- Если в последовательности есть цикл периода r, начинающийся с позиции s ($a_i = a_{i+r}$ для $i \geq s$), то для любого $i \geq s$, делящегося на r, $a_i = a_{2i}$.
- Т.е. нам придётся искать не более чем на длину периода (т.е. примерно вдвое) дольше.

Алгоритм Брента

- Другой алгоритм для того же самого — алгоритм Брента.
- Теперь черепаха останавливается на степенях двойки, а заяц прыгает шаг за шагом к следующей степени.
  1. Пока tortoise≠hare:
     - если $i == pow$, то
       $\text{tortoise} := \text{hare}$
       $pow := 2 \cdot pow$
       $\mathrm{i} := 0$
     - hare$= f(\text{hare})$
     - $++i$
- Шагов в любом случае не больше, чем в алгоритме Флойда, но каждый шаг — это одно вычисление f, а не три.

λ-метод Полларда

- Раньше были зайцы и черепахи, теперь — кенгуру.
- λ-метод Полларда ещё называется «kangaroo method».
- Предположим, что мы знаем некий интервал $[a, b]$, на котором должен лежать неизвестный логарифм x.
- Как это использовать?

λ-метод Полларда

- Определим хеш-функцию h, делящую G на r множеств $S_1, S_2, \ldots, S_r$: $S_i = h^{-1}(i)$.
- Поставим каждому множеству в соответствие расстояние $d_1, d_2, \ldots, d_r$ и длину прыжка $g^{d_1}, g^{d_2}, \ldots, g^{d_r}$.
- Теперь путь кенгуру определяется как

$$c_{i+1} = c_i \cdot g^{d_{h(c_i)}}.$$

λ-метод Полларда

- Нам будут нужны два кенгуру: дикий и ручной.
- Ручной кенгуру начнёт прыгать из какой-нибудь точки внутри интервала $[a, b]$, например, $g^{\frac{a+b}{2}}$.
- Дикий кенгуру начнёт прыгать из неизвестной точки y.
- Однако, суммируя d_i, мы можем хранить общее пройденное расстояние для обоих кенгуру.

λ-метод Полларда

- Когда ручной и дикий кенгуру встретятся, причём ручной пройдёт к тому времени расстояние t, а дикий — расстояние w, у нас получится, что

$$g^{\frac{a+b}{2}} g^t = g^x g^w, \text{ и } x = \frac{a+b}{2} + t - w.$$

- Пересечение можно найти, например, храня только $t_1, t_2, t_4, t_8, \ldots$ и $w_1, w_2, w_4, w_8, \ldots$, потому что после пересечения пути кенгуру сойдутся навсегда.
- В результате (без доказательства) ожидаемое время работы получается $O(\sqrt{b-a})$.

ρ- и λ-методы

- Почему ρ- и λ-методы названы этими буквами?

ρ- и λ-методы

- Почему ρ- и λ-методы названы этими буквами?
- Потому что то, что происходит в алгоритмах, похоже на эти буквы:
  - ρ-метод строит последовательность элементов, которая в какой-то момент возвращается к одному из промежуточных значений, создавая цикл;
  - λ-метод строит две последовательности элементов, которые в какой-то момент сливаются и затем совпадают.

Outline

От общих групп к частным случаям

- Алгоритмов лучше, чем вышеописанные довольно простые соображения, для общих групп не известно.
- Однако можно сделать лучше при дополнительных предположениях на структуру группы.
- Они выполняются, в частности, в группах чисел $\mathbb{Z}_p$.

От общих групп к частным случаям

- Предположения простые: можно выбрать разумную базу факторизации $p_1, \ldots, p_s$, для которой многие элементы будут представляться в виде
$$r = p_1^{k_1} p_2^{k_2} \ldots p_s^{k_s}.$$
- Для чисел это легко: берём простые числа, меньшие B; «многие» — это в точности B-гладкие элементы.
- В дальнейшем будем считать, что мы работаем над $\mathbb{Z}_p$.

Общая идея index calculus

- Алгоритм index calculus очень похож на алгоритм факторизации, использующий квадратичное решето.
- Так что заодно в каком-то смысле и повторим прошлую лекцию.
- Мы знаем свойства логарифма, а именно

$$\log_g(ab) = \log_g a + \log_g b,$$
$$\log_g(a^e) = e \log_g a.$$

Общая идея index calculus

- Общая идея: логарифм гладкого элемента можно представить как

$$\log_g r \equiv k_1 \log_g p_1 + k_2 \log_g p_2 + \ldots + k_s \log_g p_s \pmod{p-1}.$$

- Если мы знаем $\log_g r$ (например, сами выбирали u и вычисляли $r = g^u$) и наберём достаточно много таких соотношений, у нас получится линейная система на $\log_g p_i$.
- Её можно решить и найти $\log_g p_i$, а затем с их помощью найти $\log_g y$.

Общая идея index calculus

- Итак, получаются три фазы.
  1. Найти достаточно много соотношений на $\log_g p_i$.
  2. Решить линейную систему.
  3. Найти логарифм интересующего нас y, зная логарифмы p_i.
- Линейные системы будем решать так же, как в алгоритме факторизации.
- А остальные фазы сейчас рассмотрим.

Гладкие числа

- Нам нужно выбрать границу гладкости B, а затем найти кучу соотношений на $\log_g p_i$, $p_i \leq B$, при помощи гладких чисел u.
- Иначе говоря, нужно проверить кучу чисел на гладкость.
- Мы начнём с методов проверки индивидуальных чисел на гладкость (тоже пригодится), а потом вспомним метод полиномиального решета.

Метод Полларда

- Если просто проверять на B-гладкость перебором, сложность будет порядка $O(\pi(B))$.
- Можно воспользоваться методом, очень похожим на ρ-метод Полларда: определим последовательность чисел

$$a_{i+1} \equiv a_i^2 + 1 \pmod{n}, \text{ где } n \text{ — интересующее нас число.}$$

- По birthday paradox, она начнёт повторяться в среднем через $O(\sqrt{n})$.
- Более того, если у n есть простой делитель q, то в среднем через $O(\sqrt{n})$ начнёт повторяться последовательность $a_i \pmod{q}$.

Метод Полларда

- Мы не знаем q, но можем проверять просто каждый раз a_i и a_{2i}, не даёт ли
$$\gcd(n, a_{2i} - a_i) \text{ или } \gcd(n, a_{2i} + a_i)$$
чего-нибудь интересного. При таком подходе мы ожидаем найти делитель n за $O(\sqrt{q})$, где q — наименьший простой делитель n.
- Значит, на гладкость проверить ожидаем за $O(\sqrt{B})$; если через $O(\sqrt{B})$ шагов совпадений не найдено, можно просто предположить с большой вероятностью, что не гладкое.

Алгоритм Ленстры

- Мы знаем эффективные алгоритмы разложения чисел на множители.
- У нас были алгоритмы, работающие за время $L_n\left[\frac{1}{2};\sqrt{2}\right]$ и даже $L_n\left[\frac{1}{2};1\right]$.
- Но непонятно, как их обобщить так, чтобы оценка зависела от размера простых делителей (от B), а не от n.

Алгоритм Ленстры

- Алгоритм Ленстры (ECM, elliptic curve method) делает как раз это. Он основан на эллиптических кривых, и мы его разбирать не будем.
- Важно, что работает он за время

$$O\left(e^{\sqrt{(2+o(1))\log B \log\log B}}(\log n)^2\right) = L_B\left[\frac{1}{2}; \sqrt{2}\right].$$

Итоги

- Итак, у нас есть два разумных подхода к проверке *одного* числа на гладкость:
    - метод Полларда проверяет на B-гладкость за $O(\sqrt{B})$;
    - ECM проверяет на B-гладкость за $L_B\left[\frac{1}{2};\sqrt{2}\right] = O\left(e^{\sqrt{(2+o(1))\log B \log\log B}}\right)$.

Задача

- Нам нужно на первой фазе породить много соотношений вида
$$\log_g r = k_1 \log_g p_1 + k_2 \log_g p_2 + \ldots + k_s \log_g p_s, \quad p_i \le B.$$
- Для этого нужно проверить массу чисел на B-гладкость. Вообще говоря, мы должны выбрать много случайных u, а потом проверить $g^u \pmod p$ на B-гладкость.

Квадратичное решето

- Мы для подобной задачи знаем метод квадратичного решета.
- Рассмотрим последовательность $Q(x) = x^2 - n$ для $x = x_0 = \lceil \sqrt{n} \rceil, x_0 + 1, \ldots$.
  - Если n — квадрат по модулю p, то $x^2 - n \equiv 0 \pmod{n}$ iff $x \equiv a$ или $b \pmod{p}$, где a и b — корни из n по модулю p.
  - Если n — не квадрат $\pmod{p}$, то делиться никогда не будет.
- Значит, можно просто так же вычёркивать те $Q(x)$, для которых x делится на a или b.
- Причём этот алгоритм можно применить к любому многочлену (нам нужны будут квадратичные и линейные).
- Сложность этого алгоритма: $O\left(\pi(B)(1 + \log B)^{o(1)} + N \log\log B\right)$, где N — количество

Проблема

- Но сейчас у нас не всё так просто.
- Если выбирать u, то g^u, которые нужно проверять на гладкость, не похожи ни на какой многочлен, и так просто всё не получится.
- Как обойти эту проблему?

Решение

- Рассмотрим $H = \lceil \sqrt{p} \rceil$ и будем рассматривать последовательность $(H+c_1)(H+c_2)$ для маленьких c_1 и c_2.
- Тогда для $p_i \leq B$ получаются соотношения вида

$$\log_g (H+c_1)(H+c_2) = k_1 \log_g p_1 + k_2 \log_g p_2 + \ldots + k_s \log_g p_s.$$

- Если $H^2 = p + J$, то

$$(H+c_1)(H+c_2) \equiv J + (c_1+c_2)H + c_1c_2 \pmod{p},$$

  и это линейный многочлен, к которому можно применить решето (если в каждый конкретный момент фиксировать c_1 и варьировать c_2).
- Но ведь мы по прежнему не знаем $\log_g (H+c_1)(H+c_2)$, и отдельных $\log_g (H+c_1)$ тоже не знаем. :) Чем же нам стало лучше?

Решение

- Нам стало лучше тем, что теперь с одними и теми же c_1 и c_2 получаются сразу много соотношений!
- Мы просто добавляем $\log_g(H + c_i)$ как новые неизвестные.
- Но количество уравнений растёт быстрее, чем количество неизвестных, и на практике получается, что для базы B нужно не больше $4\pi(B)$ уравнений.
- А затем мы их решим при помощи алгоритма Видеманна, за время $\pi(B)^2$.
- Будем варьировать $0 \leq c_1 < c_2 \leq C$, C выберем позже.

Outline

Промежуточный итог

- Итак, по итогам первых двух фаз мы вычислили $\log_g p_i$ для $p_i \leq B$. Как теперь найти $\log_g y$?
- Мы будем брать случайные числа w, пока yg^w не станет достаточно гладким.
- Но здесь «достаточно» не B-гладкости, а U-гладкости для некоторого $U > B$ (все константы выберем потом, когда будем сложность оценивать).

Идея третьей фазы

- Итак, выбираем w и проверяем yg^w на U-гладкость (заодно раскладывая на множители).
- Затем, когда yg^w станет U-гладким, задача сведётся к логарифмированию нескольких простых чисел «среднего размера» (от B до U). Такое простое m мы логарифмируем так.
  1. Начиная с $u = \lceil \sqrt{p}/m \rceil$ и увеличивая u, найдём B-гладкое u.
  2. Начиная с $v = H = \lceil \sqrt{p} \rceil$ и увеличивая v, найдём B-гладкое
  $$n \equiv uvm \pmod p.$$
  3. Теперь $\log_g m = \log_g n - \log_g u - \log_g v$, и все логарифмы справа мы знаем.
- Оба числа u и v можно найти полиномиальным решетом (оба многочлена линейные).

О равномерной сложности дискретного логарифма

- Обратите внимание: все дискретные логарифмы искать одинаково трудно.
- Если какой-нибудь $\log_g y$ было бы труднее вычислить, чем для большинства других y, достаточно было бы брать случайные w, пока yg^w не стало бы легко логарифмировать.
- А логарифмы по другому основанию, если умеем искать логарифмы по основанию g, тоже искать несложно, ведь

$$\log_h a \equiv \frac{\log_g a}{\log_g h} \pmod{p-1}.$$

Какие есть параметры

- Итак, мы хотим найти оптимальные параметры для алгоритма index calculus.
- Параметры — это:
  - B — базовая оценка гладкости;
  - C — число, до которого варьируются $0 \leq c_1 < c_2 \leq C$ в решете;
  - U — новая оценка гладкости на последнем этапе.
- Для начала предположим, что третья фаза быстрее первых двух, и соптимизируем B и C.

Числа $L_p[s; c]$

- Вспомним обозначения:

$$L_p[s; c] = e^{c(\log n)^s(\log\log n)^{1-s}}.$$

- Мы сейчас всё будем делать в терминах $L_p[s; c]$, поэтому сначала установим простые свойства $L_p[s; c]$.
- Замечание: мы будем включать все константные множители внутрь L_p, т.е. читать L_p как $O(\dots)$.

Числа $L_p[s; c]$

- Крайние случаи:

  если $s = 0, L_p[s; c] = (\log p)^c$ (полиномиальная сложность);

  если $s = 1, L_p[s; c] = e^{c \log p}$ (экспоненциальная сложность).

- Сумма:

$$L_p[s_1; c_1] + L_p[s_2; c_2] = L_p[\max\{s_1, s_2\}; \max\{c_1, c_2\} + o(1)]$$

  (на самом деле $\max\{c_1, c_2\}$ — это только для случая $s_1 = s_2$, но в любом случае это верхняя оценка, и нам её хватит).

- Произведение:

$$L_p[s_1; c_1] \cdot L_p[s_2; c_2] = L_p[\max\{s_1, s_2\}; c_1 + c_2 + o(1)]$$

  (то же замечание про $c_1 + c_2$).

Количество гладких чисел

- Итак, будем оптимизировать B и C.
- Сначала повторим и расширим некоторые рассуждения из прошлой лекции.
- Теорема из теории чисел (без доказательства): для любого $\epsilon > 0$, если $X \to \infty$, $u \to \infty$, причём $X^{1/u} > (\log X)^{1+\epsilon}$, то

$$\frac{\psi(X, X^{1/u})}{X} = u^{-(1+o(1))u},$$

где $\psi(X, B)$ — количество B-гладких чисел от 1 до X.
- Если $B = X^{1/u}$, значит, $u = \frac{\log X}{\log B}$.

Количество гладких чисел

- Нас интересуют B и X вида $L_p[s; c]$; подставим $X = L_p[s; c]$ и $B = L_p[s_B; c_B]$ в эту формулу:

$$\frac{\psi(X, B)}{X} = u^{-(1+o(1))u} =$$

$$= \left(\frac{c(\log p)^s(\log\log p)^{1-s}}{c_B(\log p)^s_B(\log\log p)^{1-s_B}}\right)^{-\frac{c(\log p)^s(\log\log p)^{1-s}}{c_B(\log p)^s_B(\log\log p)^{1-s_B}}+o(u)} =$$

$$e^{(s-s_B)\frac{c}{c_B}(\log p)^{s-s_B}(\log\log p)^{-s+s_B}(\log\log p+O(\log\log\log p))} =$$

$$L_p\left[s - s_B; -(s - s_B)\frac{c}{c_B} + o(1)\right].$$

- Это вероятность того, что случайное число от 1 до X будет B-гладким. Как обычно, про значения многочленов мы ничего не знаем, только предполагаем.

Количество гладких чисел

- Всего в нашей базе факторизации $\pi(B) \approx \frac{B}{\log B}$ простых чисел.
- Итого, если нам нужны $\frac{DB}{\log B}$ соотношений, а гладким будем каждое u^u число, мы должны выполнить
$$\frac{DBu^u}{\log B}$$
тестов на гладкость.
- Здесь мы, конечно, воспользуемся решетом и получим, что общее время на генерацию системы соотношений равно
$$\frac{DBu^u}{\log B} \log\log B.$$
- Найдём минимум этого значения по B.

Оптимизация

- Перейдём к логарифму: минимизируем теперь

$$\log D + \log B + u \log u - \log\log B + \log\log\log B.$$

- Возьмём производную по B и приравняем нулю:

$$\frac{1}{B} + \frac{du}{dB}\log u + \frac{du}{dB} = 0.$$

- Вспомним, что $u = \frac{\log X}{\log B}$:

$$\frac{1}{B} - \frac{\log X \log u}{B(\log B)^2} - \frac{\log X}{B(\log B)^2} = 0,$$
$$\log X(1 + \log\log X - \log\log B) = (\log B)^2.$$

Оценка

- Мы получили, что
$$\log X(1+\log\log X-\log\log B)=(\log B)^2.$$
- Поскольку $1<\log\log B<\log\log X$,
$$\log X<\log X(1+\log\log X-\log\log B)<\log X\log\log X, \text{ и}$$
$$e^{\sqrt{\log X}}<B<e^{\sqrt{\log X\log\log X}}.$$
- Раз уж мы ищем B в виде $L_p[s_B;c_B]$, это значит, что оптимальный выбор — что-то в духе
$$B=L_p\left[\frac{1}{2};c_B\right]$$
для некоторого c_B.

Сколько же на самом деле проверок

- Мы там ничего не говорили о D; а оно связано с C и, в конечном счёте, B.
- Поэтому сейчас оценим поточнее. Пусть $B = L_p[s_B; c_B + o(1)]$, $C = L_p[s_C; c_C + o(1)]$; напоминаю, что C — это оценка на c_1 и c_2.
- Мы проверяем все $0 \le c_1 < c_2 \le C$, то есть всего будет проверок
$$\frac{1}{2}C^2 = L_p[s_C; 2c_C + o(1)].$$
- А всего гладких чисел нужно найти
$$B + C = L_p[s_B; c_B + o(1)] + L_p[s_C; c_C + o(1)] = \\ = L_p[\max\{s_B, s_C\}; \max\{c_B, c_B\} + o(1)].$$

Вывод точной оценки

- Если P_{sm} — вероятность обнаружить гладкое число, то нужно выбрать B и C так, чтобы

$$\frac{1}{2}C^2 P_{sm} \geq B + C.$$

- Какого порядка будут эти числа? Мы брали числа вида $x = (H + c_1)(H + c_2)$, где $H = \lceil \sqrt{p} \rceil = \lceil L_p \left[1; \frac{1}{2}\right] \rceil$. Поскольку $J = H^2 - p \leq 2H$:

$$x = J + (c_1 + c_2)H + c_1 c_2 \leq (2 + c_1 + c_2)H + c_1 c_2 \leq$$
$$\leq 2L_p[s_C; c_C + o(1)] L_p\left[1; \frac{1}{2}\right] + L_p[s_C; 2c_C + o(1)] = L_p\left[1; \frac{1}{2} + o(1)\right].$$

Вывод точной оценки

- А вероятность P_{sm}, как мы уже говорили,

$$P_{sm} = \frac{\psi(x, B)}{x} = L_p\left[1 - s_B; \frac{-(1 - s_B)}{2c_B} + o(1)\right].$$

- Тогда условие $\frac{1}{2}C^2P \geq B + C$ превращается в

$$L_p[s_C; 2c_C + o(1)]L_p\left[1 - s_B; \frac{-(1 - s_B)}{2c_B} + o(1)\right] \geq$$
$$\geq L_p[\max\{s_B, s_C\}; \max\{c_B, c_B\} + o(1)], \text{ то есть}$$

$$L_p[s_C; 2c_C + o(1)] \geq$$
$$\geq L_p[\max\{s_B, s_C\}; \max\{c_B, c_C\} + o(1)]L_p\left[1 - s_B; \frac{(1 - s_B)}{2c_B} + o(1)\right].$$

- Отсюда, как минимум (точнее позже),
$s_C \geq \max\{s_B, s_C, 1 - s_B\}$.

Оптимизация

- С другой стороны, давайте вернёмся к времени работы.
- Решето наше C раз проверяет по C чисел (фиксирует c_1 и варьирует c_2), то есть работает время

$$C \cdot \left(\pi(B)(1+\log B)^{o(1)} + C \log\log B\right) =$$
$$= L_p[s_C; c_C]\left(L_p[s_B; c_B] + L_p[s_C; c_C]\right) =$$
$$= L_p[\max\{s_B, s_C\}; c_C + \max\{c_B, c_C\} + o(1)].$$

- А на линейную алгебру нужно время

$$(B+C)^2 = L_p[\max\{s_B, s_C\}; \max\{2c_B, 2c_C\} + o(1)].$$

Оптимизация

- В итоге первая и вторая фазы занимают

$$L_p[\max\{s_B, s_C\}; \max\{2c_B, 2c_C\} + o(1)].$$

- Нужно минимизировать в первую очередь $\max\{s_B, s_C\}$ при условии

$$s_C \geq \max\{s_B, 1 - s_B\}.$$

- Получается $s_B = s_C = \frac{1}{2}$. При этом

$$P_{sm} = L_p\left[1 - s_B; \frac{-(1 - s_B)}{2c_B} + o(1)\right] = L_p\left[\frac{1}{2}; -\frac{1}{4c_B} + o(1)\right].$$

Оптимизация

- Т.к. $P_{sm} = L_p\left[\frac{1}{2}; -\frac{1}{4c_B} + o(1)\right]$, условие на достаточное количество гладких чисел $\frac{1}{2}C^2P \geq B + C$ превращается в

$$L_p[\frac{1}{2}; 2c_C + o(1)] \geq L_p\left[\frac{1}{2}; \max\{c_B, c_C\} + o(1)\right] L_p\left[\frac{1}{2}; \frac{1}{4c_B} + o(1)\right],$$

$$\text{то есть } 2c_C \geq \max\{c_B, c_C\} + \frac{1}{4c_B}.$$

Оптимизация

- А суммарное время работы алгоритма превращается в
$$L_p[\max\{s_B, s_C\}; \max\{2c_B, 2c_C\} + o(1)].$$
- Оптимизируя это при условии $2c_C \geq \max\{c_B, c_C\} + \frac{1}{4c_B}$, получим $c_B = c_C = \frac{1}{2}$.
- В итоге $B = C = L_p\left[\frac{1}{2}; \frac{1}{2} + o(1)\right]$, а суммарное время работы первой и второй фаз составляет
$$L_p\left[\frac{1}{2}; 1 + o(1)\right].$$

Время работы третьей фазы алгоритма

- Мы предполагали, что третья фаза будет быстрее первых двух. Верно ли это?
- Напоминаю, что мы выбираем w и проверяем yg^w на U-гладкость, пока не попадём.
- Давайте оценим; у нас теперь новый параметр $U = L_p\left[s_U; c_U + o(1)\right]$, а вероятность найти подходящее число w будет P_w:

$$P_w = \frac{\psi(p, U)}{p} = L_p\left[1 - s_U; -\frac{1 - s_U}{c_U} + o(1)\right].$$

Время работы третьей фазы алгоритма

- Если мы пользуемся ECM, то каждое число проверяется за

$$e^{\sqrt{(2+o(1))\log U \log\log U}}(\log n)^2 =$$
$$= L_p\left[\frac{s_U}{2}; \sqrt{2s_Uc_U}+o(1)\right].$$

- А нам нужно провести $\frac{1}{P_W}$ таких тестов, т.е. общее время на поиск w равно

$$L_p\left[\frac{s_U}{2}; \sqrt{2s_Uc_U}+o(1)\right]\cdot L_p\left[1-s_U; \frac{1-s_U}{c_U}+o(1)\right] =$$
$$= L_p\left[\max\{\frac{s_U}{2}, 1-s_U\}; \frac{1-s_U}{c_U}+\sqrt{2s_Uc_U}+o(1)\right].$$

Время работы третьей фазы алгоритма

- Итак, нужно оптимизировать

$$L_p\left[\max\{\frac{s_U}{2}, 1-s_U\}; \frac{1-s_U}{c_U} + \sqrt{2s_Uc_U} + o(1)\right].$$

- Минимизируя $\max\{\frac{s_U}{2}, 1-s_U\}$, получим $s_U = \frac{2}{3}$, а минимизируя $\frac{1}{3c_U} + 2\sqrt{c_U/3}$, получим $c_U = \left(\frac{1}{3}\right)^{1/3}$. Итак:

$$U = L_p\left[\frac{2}{3}; \left(\frac{1}{3}\right)^{1/3} + o(1)\right],$$

а общее время работы третьей фазы составляет

$$L_p\left[\frac{1}{3}; 3^{1/3} + o(1)\right].$$

Анализ

- У нас получилось $L_p\left[\frac{1}{3}; 3^{1/3}+o(1)\right]$, что гораздо быстрее, чем $L_p\left[\frac{1}{2}; 1+o(1)\right]$ (время первой и второй фазы).
- Но так получилось только благодаря ECM; если использовать для проверки на гладкость метод Полларда, получится то же самое $L_p\left[\frac{1}{2}; 1+o(1)\right]$, а с тривиальным алгоритмом проверки (пробным делением) и вовсе $L_p\left[\frac{1}{2}; \sqrt{2}+o(1)\right]$.

Упражнение. Доказать эти оценки.

Но это ещё не всё

- Нужно ещё оценить логарифмирование «среднего размера» простых чисел.
- Нам для каждого такого простого m надо найти B-гладкое $u > \sqrt{p}/m$. Здесь u — число порядка $L_p[1; \frac{1}{2}]$, а вероятность выбрать гладкое u — $L_p\left[\frac{1}{2}; -\frac{1}{2} + o(1)\right]$.
- Т.е. нужно прогнать через решето $L_p\left[\frac{1}{2}; \frac{1}{2} + o(1)\right]$ вариантов; это быстрее первой и второй фазы.
- А самый последний шаг — найти такое $v > \sqrt{p}$, что $uvm \pmod{p}$ будет B-гладким. Здесь v тоже порядка $L_p[1; \frac{1}{2}]$, и точно так же получается сложность $L_p\left[\frac{1}{2}; \frac{1}{2} + o(1)\right]$.
- Так что этот шаг оказался сложнее, чем «основная часть» третьей фазы, но всё равно быстрее первой и второй фазы.

Теперь всё

- Теперь всё. Уффф.
- Важное замечание: одни и те же результаты первой и второй фазы можно использовать для вычисления многих дискретных логарифмов; каждый новый логарифм будет стоить как третья фаза, а не как первая+вторая, что дешевле.

Outline

Решето числового поля

- Полиномиальное решето — не предел мечтаний.
- Ещё эффективнее оказывается метод *решета числового поля* (number field sieve).
- По сути метод аналогичен квадратичному решету, но теперь всё происходит над другими кольцами.
- Мы рассмотрим только основную идею, безо всяких доказательств.

Идея

- Мы рассмотрим решето числового поля для задачи разложения чисел на множители.
- Мы хотим разложить n. Предположим, что у нас есть неприводимый многочлен $f(x)$ и число m, такое, что $f(m) \equiv 0 \pmod{n}$.
- Рассмотрим комплексный корень α многочлена $f(x)$ и кольцо $\mathbb{Z}[\alpha]$.
- $f(m) \equiv 0 \pmod{n}$ и $f(\alpha) = 0$, следовательно, есть естественный гомоморфизм колец $\varphi : \mathbb{Z}[\alpha] \to \mathbb{Z}_n$, который отображает α в m.

Идея

- Теперь предположим, что у нас есть множество таких пар чисел (a, b), что:
  - произведение всех $(a - \alpha b)$ — квадрат в кольце $Z[\alpha]$, скажем, γ^2;
  - произведение всех $(a - mb)$ — квадрат в $\mathbb{Z}$, скажем, v^2.
- Заменим в выражении для γ α на m; получим $\varphi(\gamma) \equiv u \mod n$. Теперь

$$u^2 \equiv \varphi(\gamma)^2 = \varphi\left(\gamma^2\right) = \varphi\left(\prod(a - \alpha b)\right) =$$
$$= \prod\left(\varphi\left(a - \alpha b\right)\right) = \prod(a - mb) = v^2 \pmod{n},$$

и мы тем самым сможем разложить n на множители.

Многочлен f

- Но откуда взять f? Он сам собой появится.
  - Выберем степень d, положим $m = \lfloor n^{1/d} \rfloor$.
  - Запишем n по основанию m: $n = m^d + c_{d-1}m^{d-1} + \ldots + c_0$.
  - Вот и многочлен: $f(x) = x^d + c_{d-1}x^{d-1} + \ldots + c_0$.
- Отдельный вопрос: будет ли он неприводимым? Если не будет, то $n = f(m) = g(m)h(m)$, и мы уже (с высокой вероятностью) разложили n. Так что будет. :)

Числа a и b

- Откуда взять a и b? Из такого же решета.
- Чтобы $\prod(a - mb)$ было квадратом, нужно решить линейную систему на коэффициенты, как раньше.
- Чтобы $\prod(a - \alpha b)$ было квадратом, нужно решить линейную систему на коэффициенты в кольце $\mathbb{Z}[\alpha]$, если это хорошее кольцо (с единственностью разложения). Хорошее кольцо можно добыть (без д-ва).
- Теперь можно просто объединить две системы — нам нужно, чтобы оба свойства выполнялись.

Оценка сложности

- Чем хорошо решето числового поля?
- Наши оценки были основаны на X — количестве чисел, из которых можно сделать квадрат.
- У нас было $X = n^{1/2+\epsilon}$.
- А в number field sieve получается $X = e^{c(\log n)^{2/3}(\log\log n)^{1/3}}$, что даёт общую оценку сложности

$$L_n\left[\frac{1}{3}; c\right] = e^{(c+o(1))(\log n)^{1/3}(\log\log n)^{2/3}}.$$

- Теоретический рекорд: $c \approx 1,902$, из анализа нашего алгоритма получилось бы

$$L_p\left[\frac{1}{3}; \left(\frac{64}{9}\right)^{1/3} + o(1)\right] \approx L_p\left[\frac{1}{3}; 1,923 + o(1)\right].$$

- Но главное — основная асимптотика стала лучше.

Number field sieve для дискретного логарифма

- Аналогичные соображения проходят и для задачи дискретного логарифма, и тоже время работы получается

$$L_p\left[\frac{1}{3};\left(\frac{64}{9}\right)^{1/3}+o(1)\right]\approx L_p\left[\frac{1}{3};1,923+o(1)\right].$$

- На практике решето числового поля начинает выигрывать, где-то начиная со 100-значных чисел.

Thank you!

Спасибо за внимание!