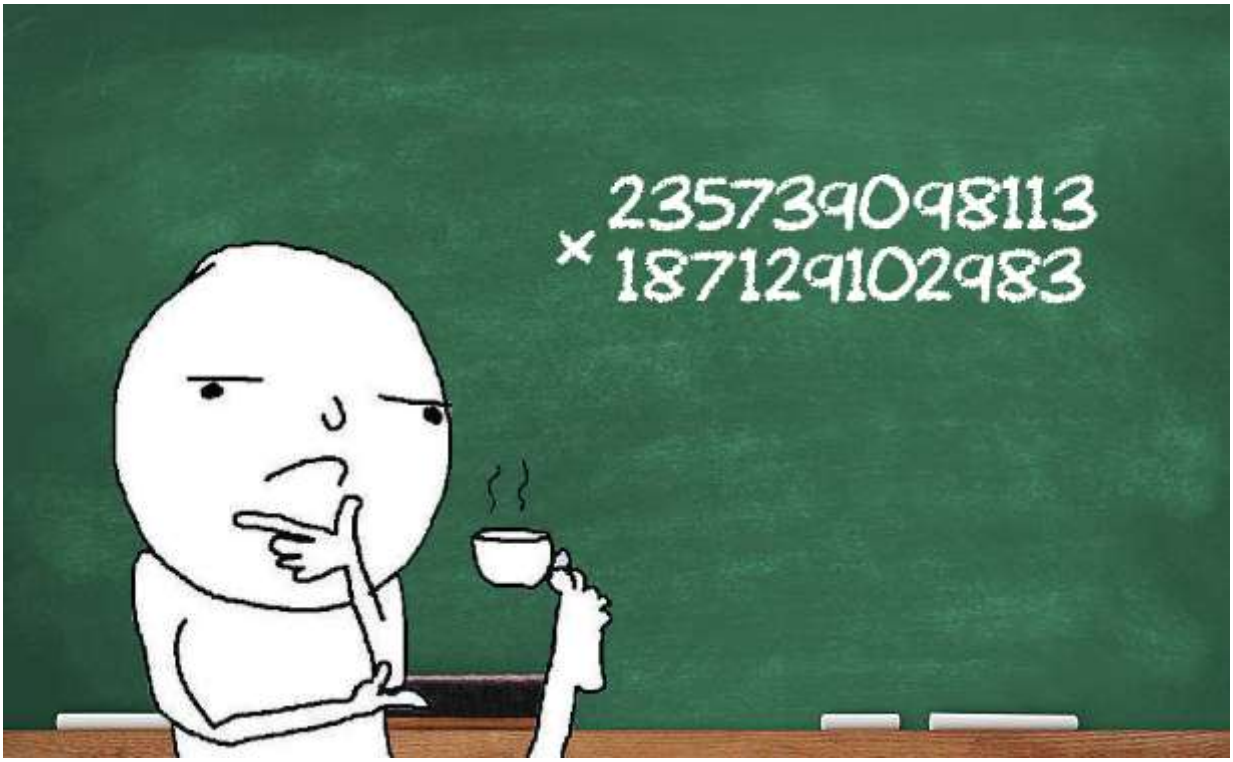
Алгоритмы быстрого умножения чисел: от столбика до Шенхаге-Штрассена

Средний **26 мин** **36K**

Python*Алгоритмы*Математика*



При написании высокоуровневого кода мы редко задумываемся о том, как реализованы те или иные инструменты, которые мы используем. Ради этого и строится каскад абстракций: находясь на одном его уровне, мы можем уместить задачу в голове целиком и сконцентрироваться на её решении.

И уж конечно, никогда при написании a * b мы не задумываемся о том, как реализовано умножение чисел a и b в нашем языке. Какие вообще есть алгоритмы умножения? Это какая-то нетривиальная задача?

В этой статье я разберу с нуля несколько основных алгоритмов быстрого умножения целых чисел вместе с математическими приёмами, делающими их возможными.

Оглавление

Зачем быстро умножать числа?

Программы постоянно перемножают числа. Перемножения 32-битных, 64-битных, а иногда и более длинных чисел встроены напрямую в арифметико-логические устройства микропроцессоров; генерация оптимальных цепей для кремния — отдельная инженерная наука. Но не всегда встроенных возможностей хватает.

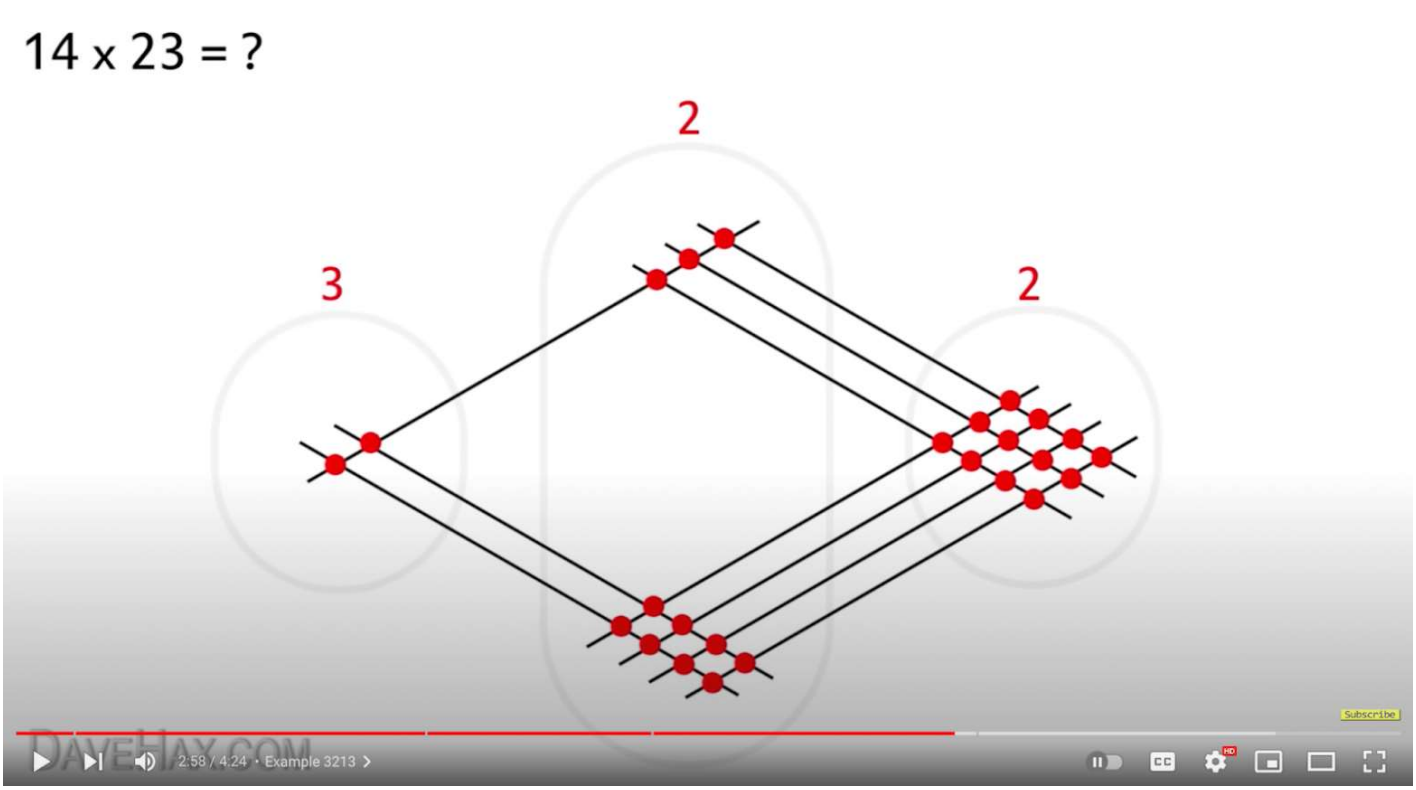
Например, для криптографии. Криптографические алгоритмы (вроде повсеместно используемого RSA) оперируют числами длиной в тысячи бит. Если мы сводим операции над 4096-битными числами к операциям над 64-битными словами, разница в количестве операций между алгоритмами за N^2 и $N \log N$ уже составляет десять раз!

Деление тоже сводится к умножению; в некоторых процессорах даже нет инструкции для целочисленного деления.

Но сразу признаюсь: не все существующие алгоритмы быстрого умножения достаточно практичны для широкого применения — по крайней мере на сегодняшний день. Помимо практического здесь силён академический интерес. Как вообще математически устроена операция умножения? Насколько быстро её можно делать? Можем ли мы найти оптимальный алгоритм и чему научимся в процессе поиска?

Его величество столбик

Когда-то очень давно мне попалось на глаза видео с кричащим названием "How to Multiply", описывающее так называемый японский метод умножения. Что меня удивило — так это то, что метод этот подаётся как более простой и интуитивно понятный. Оказывается, если умножение в столбик делать не в столбик, а занимая половину листа бумаги, получается проще и понятнее!



Да-да, принципиально это тот же самый алгоритм умножения столбиком. Можем посмотреть на запись в столбик и сравнить её с картинкой:

$$\begin{array}{r} 23 \\ \times\ 14 \\ \hline 92 \\ +\ 23 \\ \hline 322 \end{array}$$

92 — это две больших группы красных точек на нижне-правой диагонали. 23 — две маленьких группы на верхне-левой диагонали. Точно так же, как и в столбике, мы вынуждены перемножить попарно все разряды, потом сложить, и потом сделать перенос. Умножение в столбик — не что иное, как компактный на бумаге и *относительно* удобный для человеческого сознания способ проделать алгоритм, объединяющий в себе практически все придуманные до XX века способы умножения, за редким исключением вроде умножения египетских дробей.

Асимптотическая сложность умножения в столбик не зависит от того, в какой системе счисления мы производим умножение, и составляет $O(M \cdot N)$ операций, где M и N — количество разрядов в множителях. Каждый из M разрядов одного множителя нужно умножить на каждый из N разрядов второго множителя, после чего получившиеся MN маленьких чисел (в каждом не больше двух разрядов) сложить.

Почему асимптотика не зависит от системы счисления?

Обычно при анализе сложности под M и N подразумевается количество *двоичных* разрядов, а числа предполагаются примерно одинаковой длины; тогда формула сложности упрощается до $O(N^2)$.

Что означает запись O(N²)?

Разделяй и властвуй: алгоритм Карацубы

Первый шаг на пути ускорения умножения совершил в 1960-м году советский математик Анатолий Карацуба. Он заметил, что если длинные числа поделить на две части:

$$235739\,098113 \times 187129\,102983$$

$$\left(a_1 \cdot 10^6 + a_0\right) \times \left(b_1 \cdot 10^6 + b_0\right)$$

$$a_1 \cdot b_1 \cdot 10^{12} + \left(a_1 \cdot b_0 + a_0 \cdot b_1\right) \cdot 10^6 + a_0 \cdot b_0$$

То можно обойтись *тремя* умножениями этих более коротких частей друг на друга, а не четырьмя, как можно было бы подумать. Вместо прямого подсчёта $a_1 \cdot b_0 + a_0 \cdot b_1$, требующего двух умножений, достаточно посчитать $\left(a_1 + a_0\right) \times \left(b_1 + b_0\right)$ и вычесть из результата числа $a_1 \cdot b_1$ и $a_0 \cdot b_0$, нужные нам в любом случае для получения младших и старших разрядов.

После этих расчётов остаётся лишь пробежаться по разрядом справа налево и провести суммирование:

$$\begin{array}{rr} a_0 \cdot b_0 & \overleftarrow{10103971079} \\ a_1 \cdot b_0 + a_0 \cdot b_1 & 42636897014 \\ a_1 \cdot b_1 & 44113603331 \\ \hline a \cdot b & 4411364596790711797 1079 \end{array}$$

По сравнению с умножениями это быстрая операция, не заслуживающая особого внимания — как и два лишних сложения в скобках.

Для построения эффективного алгоритма осталось превратить это наблюдение в рекурсивную процедуру. Половинки чисел тоже будем делить на половинки и так далее, пока

не дойдём до достаточно коротких чисел, которые можно перемножить в столбик или, допустим, через lookup table.

Поскольку мы каждый раз делим задачу на три задачи с вдвое меньшими (по количеству бит) числами, для перемножения двух чисел длины 2^k нам потребуется рекурсия глубины k и суммарно 3^k умножений чисел наименьшей длины; отсюда получаем оценку сложности в $O(N^{\log_2 3}) \approx O(N^{1.58})$.

> Занятный факт — аналогичные фокусы с экономией за счёт одновременных умножений используются ещё в ряде мест. Так, два комплексных числа $a + bi$ и $c + di$ также можно перемножить за три вещественных умножения вместо четырёх, вычислив ac, bd и $(a + b) \cdot (c + d)$. А алгоритм Пана для быстрого перемножения матриц основывается на том, что можно одновременно вычислить произведения двух пар матриц XY и UV за меньшее число умножений, чем по отдельности [1, 2]. Не говоря уж о том, что само по себе перемножение матриц быстрее, чем за $O(N^3)$ — это быстрое одновременное умножение матрицы на N векторов.

Числа vs многочлены: алгоритмы Тоома-Кука

При виде магического сокращения вычислительной сложности при разбиении множителей на две части как-то сам собой возникает вопрос: а можно разбить множители на бóльшее число частей и получить бóльшую экономию?

Ответ — да; и подход, позволяющий это сделать, включает в себя алгоритм Карацубы как частный случай. Но для его формулировки нам придётся проделать некий фокус.

Давайте попробуем разбить те же самые числа на три части:

$$2357\,3909\,8113 \times 1871\,2910\,2983$$

$$\left(a_2 \cdot 10^8 + a_1 \cdot 10^4 + a_0\right) \times \left(b_2 \cdot 10^8 + b_1 \cdot 10^4 + b_0\right)$$

Если мы внимательно посмотрим на коэффициенты при степенях десятки, которые возникают при перемножении этих скобок:

$$
\begin{array}{r}
a_0 \cdot b_0 \\
a_1 \cdot b_0 + a_0 \cdot b_1 \\
a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2 \\
a_2 \cdot b_1 + a_1 \cdot b_2 \\
a_2 \cdot b_2 \\
\hline
a \cdot b
\end{array}
$$

То (при наличии опыта в алгебре) заметим, что где-то мы такое уже видели. При перемножении многочленов!

Если взять части наших чисел a и b и объявить их коэффициентами многочленов $f(x)$ и $g(x)$ при соответствующих степенях, числа в таблице выше будут не чем иным, как коэффициентами многочлена $f(x) \cdot g(x)$:

$$
\begin{aligned}
& f(x) \cdot g(x) = \\
& \quad = (a_0 + a_1 x + \ldots) \times (b_0 + b_1 x + \ldots) = \\
& \qquad = (a_0 \cdot b_0) \\
& \qquad + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot x \\
& \qquad + (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2) \cdot x^2 \\
& \qquad + (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3) \cdot x^3 \\
& \qquad + \ldots
\end{aligned}
$$

Сами же числа получаются из многочленов путём вычисления значения в некоторой точке. Какой именно — зависит от системы счисления и размера частей:

$$a = a_2 \cdot 10^8 + a_1 \cdot 10^4 + a_0 = f(10^4)$$

Хорошо, свели одну задачу к другой. В чём преимущество многочленов? Помимо того, что это некоторые формальные конструкции с параметрами, это также *функции*; чтобы вычислить $f(x) \cdot g(x)$ в произвольной точке, не нужно перемножать многочлены — достаточно вычислить оба значения в этой точке и перемножить их. Если бы я писал продакшн-код, в котором по какой-то причине нужно перемножать многочлены, я бы и вовсе сделал перемножение ленивым.

Но нам не нужно вычислять значение в точке. Нам нужны коэффициенты. А коэффициенты многочлена можно восстановить по значениям в точках, решив систему линейных уравнений:

$$\begin{cases} c0 + c_1 \cdot x_1 + \ldots = f(x_1) \cdot g(x_1) \\ c0 + c_1 \cdot x_2 + \ldots = f(x_2) \cdot g(x_2) \\ \vdots \end{cases}$$

Здесь x_i — это точки, в которых нам известны значения многочлена, в правой части — сами эти значения, а c_i — неизвестные нам коэффициенты многочлена. Количество неизвестных соответствует степени многочлена + 1; чтобы решение было единственным, нужно, соответственно, знать значения в таком же количестве точек. В нашем примере это пять точек, потому что у многочлена-произведения степень 4:

$$f(x) \cdot g(x) = a_2 \cdot b_2 \cdot x^4 + \ldots$$

Если переписать эту систему уравнений в матричном виде, получим

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots \\ 1 & x_3 & x_3^2 & x_3^3 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \times \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} f(x_1) \cdot g(x_1) \\ f(x_2) \cdot g(x_2) \\ f(x_3) \cdot g(x_3) \\ \vdots \end{bmatrix}$$

Соответственно, для создания алгоритма быстрого умножения чисел нам достаточно:

1. выбрать, на сколько частей мы разбиваем числа (можно даже разбить a и b на разное количество частей);
2. выбрать точки x_i, в которых мы будем вычислять значения многочленов;
3. построить по ним матрицу Вандермонда;
4. вычислить её обратную матрицу.

В ходе алгоритма нам нужно будет два раза умножить на прямую матрицу (по разу для каждого из чисел, которые мы хотим перемножить), перемножить результаты алгоритмом

умножения чисел меньшего размера, и один раз умножить получившийся вектор на обратную матрицу.

Давайте посмотрим на матрицу для точек $0, 1, -1, 2, -2$:

```
from sympy import Rational
from sympy.matrices import Matrix

# Точки, в которых будем вычислять значения многочленов:
x = [0, 1, -1, 2, -2]

# Матрица Вандермонда, построенная по этим точкам:
v = Matrix([
    [
        Rational(x_i ** j )
        for j, _ in enumerate(x)
    ] for x_i in x
])
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -2 & 4 & -8 & 16 \end{bmatrix}$$

(Серым я отметил числа, не имеющие значения, поскольку соответствующие коэффициенты в многочленах-множителях заведомо равны нулю.)

И на её обратную:

$$\frac{1}{24} \times \begin{bmatrix} 24 & 0 & 0 & 0 & 0 \\ 0 & 16 & -16 & -2 & 2 \\ -30 & 16 & 16 & -1 & -1 \\ 0 & -4 & 4 & 2 & -2 \\ 6 & -4 & -4 & 1 & 1 \end{bmatrix}$$

За исключением некоторой проблемы с делением на 3, подавляющая часть вычислений здесь представляется битовыми сдвигами, сложениями и вычитаниями «коротких» чисел, а это операции «простые» — имеющие линейную сложность.

Как проанализировать итоговую сложность получившегося алгоритма? Давайте обозначим количество операций через $C(N)$, где N — количество бит в наших множителях, и выведем рекуррентную формулу:

$$C(N) = 5 \cdot C\left(\frac{N}{3}\right) + \text{const} \cdot \frac{N}{3}$$

Что здесь написано: умножение двух чисел длиной N бит мы умеем сводить к 5 умножениям чисел длиной $N/3$ бит и ещё какому-то количеству простых операций над числами длиной $N/3$ бит. Применяем основную теорему о рекуррентных соотношениях и получаем итоговую сложность

$$C(N) = O(N^{\log_3 5}) \approx O(N^{1.46})$$

Таким образом мы ускорили умножение с $O(N^{1.58})$ до $O(N^{1.46})$.

Алгоритмы, получаемые таким образом, называются **алгоритмами Тоома-Кука**. Они активно используются на практике; в одной только библиотеке GMP поддержано 13 разных вариантов разбиения чисел.

Закономерный следующий вопрос: что дальше? Если число бьётся на M частей, сложность получается

$$C(N) = O\left(N^{\log_M(2M+1)}\right)$$

Показатель степени при большом M можно упростить:

$$\log_M(2M+1) \approx \log_M(2M) = \frac{\log_2 2 + \log_2 M}{\log_2 M} = 1 + \frac{1}{\log_2 M} \to 1$$

Получается, мы можем построить алгоритм со сколь угодно близкой к 1 степенью N в асимптотике? Увы, тут есть сложности.

Во-первых, это непрактично. Логарифм возрастает крайне медленно; асимптотика, соответственно, тоже будет падать медленно, а константа при этом будет быстро расти, потому что числа в матрицах будут всё больше и разнообразнее.

Во-вторых, это *неинтересно*. Да-да! Дальше мы будем обсуждать алгоритмы, сложность которых *лучше* $O(N^{1+\varepsilon})$, каким маленьким бы ни было ε.

> Занятный факт — аналогичной методу Тоома-Кука техникой строятся быстрые алгоритмы перемножения матриц. Однако процедура перемножения матриц по своей математической природе оказалась намного сложнее перемножения многочленов; поэтому до сих пор нет уверенности, что можно перемножать матрицы за $O(N^{2+\varepsilon})$ для сколь угодно малого ε. Лучшая асимптотика на сегодняшний день составляет примерно $O(N^{2.37})$.

Многочлены vs преобразование Фурье: алгоритм Шенхаге-Штрассена

Раз рекуррентное деление на несколько частей нам не интересно, давайте попробуем сделать радикальный шаг. Что, если делить числа не на фиксированное количество частей, а на части фиксированной длины?

Например, на десятичные разряды. В таком случае нам нужно будет $2N - 1$ точек для N-разрядного числа; в остальном подход как будто бы тот же самый, что в предыдущем разделе. Что ж, давайте возьмём $x_i = i$ и проведём численный эксперимент, чтобы убедиться, что всё работает!

```
import math
import numpy as np

# Числа, которые мы изначально собирались перемножить:
a = 235739098113
b = 187129102983
m = len(str(a))

# Коэффициенты соответствующих многочленов в порядке
# возрастания степени. Поскольку результирующий многочлен
# будет степени 2*m-1, добиваем нулями до нужной ширины:
a_coefs = [int(a_i) for a_i in str(a)[::-1]] + [0] * (m - 1)
b_coefs = [int(b_i) for b_i in str(b)[::-1]] + [0] * (m - 1)
```

```
    n = len(a_coefs)

    # Точки, в которых будем вычислять значения многочленов:
    x = np.arange(n)

    # Матрица Вандермонда, построенная по этим точкам:
    v = np.vander(x, increasing=True)

    # Вычисление значения многочлена в точке — не что иное, как
    # умножение этой матрицы на вектор коэффициентов:
    a_y = v @ a_coefs
    b_y = v @ b_coefs

    # Поточечно перемножаем значения, чтобы получить
    # значения многочлена-произведения:
    c_y = a_y * b_y

    # Восстанавливаем коэффициенты многочлена-произведения:
    c_coefs = np.linalg.solve(v, c_y)

    # Для сверки считаем коэффициенты "в лоб":
    actual_c_coefs = [
        sum(a_coefs[j] * b_coefs[i-j] for j in range(i+1))
        for i in range(n)
    ]

    # Считаем длину вектора-разности и делим на длину настоящего,
    # чтобы получить относительную погрешность:
    print(np.linalg.norm(c_coefs - actual_c_coefs) /
        np.linalg.norm(actual_c_coefs))
```

Запускаем и…

```
    2.777342817120168e+17
```

Хо-хо, да это не просто мимо, это *фантастически* мимо! Но почему?

Оказывается [3], у матриц Вандермонда есть неприятное свойство — решать системы с ними в подавляющим большинстве случаев очень плохая идея, потому что погрешность результата растёт *экспоненциально* с размером матрицы. Можно улучшить ситуацию, взяв вместо $x_i = i$ комплексные точки на единичной окружности:

```
x = np.exp(1j * np.random.uniform(0, math.pi, size=n))

...

0.00015636299542432733
```

Но лучшим выбором оказывается взять точки, равномерно распределённые на единичной окружности и являющиеся корнями из единицы степени N.

```
x = np.exp(-1j * np.linspace(0, 2*math.pi, n, endpoint=False))
```

> Что ещё за корни из единицы?

Запускаем новый вариант кода и получаем

```
1.5964929527133826e-15
```

Отлично! Мы дошли до предела машинной точности.

Но самое изумительное — при таком выборе иксов матрица представляет собой не что иное, как матрицу дискретного преобразования Фурье! А значит, мы можем умножать векторы на эту матрицу и решать систему уравнений с ней алгоритмом быстрого преобразования Фурье за $O(N \log N)$ операций сложения и умножения — вместо наивного алгоритма за $O(N^2)$.

> Что такое дискретное преобразование Фурье?

> Что такое быстрое преобразование Фурье?

Делить числа на части по десятичным разрядам — это, конечно, не самый эффективный способ. Тут даже нет как таковой рекурсии — после первого разделения на части мы сразу приходим к числам от 0 до 9, которые можно перемножить по таблице умножения.

Наилучшая асимптотика в таком алгоритме получается, если делить N-битные числа на части длиной примерно $\log N$ бит.

Полностью **алгоритм Шенхаге-Штрассена** выглядит так.

1. Берём на вход два числа, которые хотим перемножить; числа эти длины N бит. На практике это значит, что бóльшее из чисел имеет N бит.

$$\overbrace{1001010011000001}^{N} \times \overbrace{1100000001001110}^{N}$$

2. Делим двоичную запись чисел на фрагменты, каждый длиной примерно $\log N$ бит:

$$\overbrace{\underbrace{1001}_{\log N}\underbrace{0100}_{\log N}\underbrace{1100}_{\log N}\underbrace{0001}_{\log N}}^{N} \times \overbrace{\underbrace{1100}_{\log N}\underbrace{0000}_{\log N}\underbrace{1001}_{\log N}\underbrace{1110}_{\log N}}^{N}$$

3. Эти фрагменты — целые числа с $\log N$ бит — объявляем коэффициентами многочленов, которые нужно перемножить:

$$\left(1001 \cdot x^3 + 0100 \cdot x^2 + 1100 \cdot x + 0001\right) \times \left(1100 \cdot x^3 + 0000 \cdot x^2 + 1001 \cdot x + 1110\right)$$

4. Теперь у нас на руках два вектора коэффициентов многочленов. К этим векторам применяем дискретное преобразование Фурье:

$$FFT \times \begin{bmatrix} 0001 \\ 1100 \\ 0100 \\ 1001 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad FFT \times \begin{bmatrix} 1110 \\ 1001 \\ 0000 \\ 1100 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

   Для этого нам потребуются вычисления с плавающей точкой, гарантирующие достаточно точный результат — на это потребуется порядка $O(\log N)$ бит. Умножения делаем рекурсивно этим же алгоритмом.

5. Полученные векторы перемножаем поэлементно. Для перемножения элементов также используем рекурсивно этот алгоритм (или простые алгоритмы, если числа уже достаточно маленькие). Рекурсия будет очень небольшой глубины, потому что на каждом

шаге мы от входных данных размера $O(N)$ переходим к $O(\log N)$; например, для чисел, занимающих гигабайт (!) каждое, глубина рекурсии будет всего 6.

6. К получившемуся вектору применяем обратное преобразование Фурье. Умножения делаем рекурсивно этим же алгоритмом.

$$IFFT \times \begin{bmatrix} 780 \\ -146.17 - 93.35i \\ 22 - 58.43i \\ -217.82 + 41.36i \\ \vdots \end{bmatrix}$$

7. Наконец, из получившегося вектора коэффициентов многочлена-произведения восстанавливаем число-результат умножения, суммируя в столбик справа налево.

$$\begin{array}{r} \overleftarrow{1110} \\ 11001100 \\ 1101000 \\ 10011010 \\ 10110100 \\ 110000 \\ 1101100 \\ \hline 11011111011111000010011011001110 \end{array}$$

Формула асимптотической сложности получившегося алгоритма довольно страшная, в целом асимптотика получается чуть-чуть хуже $O(N \log N \log\log N)$. Но можно оценить полёт мысли и количество математических конструкций, требующихся для достижения такого результата.

Модульная арифметика и второй алгоритм Шенхаге-Штрассена

Хотя первый алгоритм Шенхаге-Штрассена круто улучшает асимптотику по сравнению с ранее существовавшими методами, у него всё ещё есть зоны роста. Во-первых, на каждом шаге рекурсии происходит экспоненциальное уменьшение размеров чисел — с N до $\log N$. Из-за этого глубина рекурсии очень мала и воспользоваться ускорением получается не в полном объёме. Во-вторых, необходимость производить расчёты с плавающей точкой разной точности весьма непрактична — мы все привыкли к float и double, а реализация чисел с плавающей точкой настраиваемой ширины весьма нетривиальна. (Здесь читатель может возразить, что многое из вышеописанного было нетривиально. Могу только согласиться.)

Второй алгоритм Шенхаге-Штрассена лишён этих двух недостатков, более быстр и в теории, и в практике и наиболее широко используется — например, в библиотеке GMP.

Для решения первой проблемы давайте попробуем разбить числа на более крупные фрагменты — например, размера $\sqrt{N}$ вместо $\log N$. Здесь мы незамедлительно столкнёмся с проблемой: для дискретного преобразования Фурье в таком случае потребуется $O(\sqrt{N}\log\sqrt{N})$ операций умножения чисел из $\sqrt{N}$ бит; каждое такое умножение будет стоить *в лучшем случае* $O(\sqrt{N}\log\sqrt{N})$ (напомню, все алгоритмы выше стоили ещё дороже), что даёт нам суммарную сложность не меньше $O(\sqrt{N}\log^2 N)$, что *уже* хуже предыдущего алгоритма. Что-то здесь нужно ускорить.

Для решения второй проблемы можно вспомнить, что в модульной арифметике тоже можно делать быстрое преобразование Фурье. Тогда мы будем иметь дело только с целыми числами! А если подобрать в качестве элементов матрицы Фурье степени двойки, то на них можно будет умножать быстрее, чем за $O(\sqrt{N}\log\sqrt{N})$. В идеале вообще за $O(\sqrt{N})$, потому что умножение на степень двойки — это просто битовый сдвиг.

Что такое модульная арифметика?

Наблюдение первое: самая удобная арифметика

Для того, чтобы быстро делать умножения во время быстрого преобразования Фурье, нам нужно, чтобы некие числа сочетали два свойства: были степенями двойки — чтобы можно было быстро на них умножать, и были корнями из единицы — чтобы быстрое преобразование Фурье в принципе работало. Значит, привычная всем программистам n-битная арифметика не подходит — потому что это фактически вычисления по модулю 2^n, а значит, любая степень двойки при возведении в достаточно большую степень обращается в ноль и никак не может быть корнем из единицы:

```
num_bits = 5
[pow(2, i, 2**num_bits) for i in range(2*num_bits)]
# [1, 2, 4, 8, 16, 0, 0, 0, 0, 0]
```

Если же основание арифметики кратно двум, но не степени двойки, мы не получим ноль, но и никогда не получим единицу:

```
mod = 6
[pow(2, i, mod) for i in range(2*mod)]
```

```
# [1, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2]
```

Потому что двойка является делителем нуля (в примере выше её можно умножить на три и получить 0 по модулю 6) и не является членом мультипликативной группы.

Гораздо лучше дела обстоят с основаниями, взаимно простыми с двойкой:

```
mod = 5
[pow(2, i, mod) for i in range(2*mod)]
# [1, 2, 4, 3, 1, 2, 4, 3, 1, 2]
```

Поскольку в этом случае двойка состоит в мультипликативной группе, всегда есть степень, в которую её можно возвести, чтобы получить 1 — и это справедливо также для любой степени двойки. У нас есть неограниченный запас простых чисел, мы могли бы просчитать их наперёд с запасом и делать вычисления по модулю достаточно большого простого числа… Но умножать числа по произвольному модулю быстро не получится даже на степень двойки, потому что после битового сдвига нужно брать остаток от деления, а это *медленно*.

Время для **первого наблюдения** — идеально для наших нужд подходит арифметика по модулю $2^n + 1$. Это число гарантированно взаимно простое с двойкой (потому что нечётное), и по нему легко брать остаток от деления. Как?

Допустим, мы умножили некое число x на некую степень двойки 2^y, сделав битовый сдвиг. Если результат уместился в первые n бит, всё отлично. Если результат равен 2^n, тоже неплохо. Если больше — то есть случилось переполнение — все биты, начиная с $(n+1)$-го, надо обнулить. И для этого есть простой алгоритм!

В вычислениях по модулю $2^n + 1$ имеем $2^n \equiv -1 \mod (2^n + 1)$. Домножая равенство на 2^k, получим $2^{n+k} \equiv -2^k \mod (2^n + 1)$. Соответственно, каждый лишний $(n+k)$-й бит мы можем превратить в *вычитание* k-го бита. Собираем по битам число, которое надо вычесть, и вычитаем по модулю — эта операция стоит $O(n)$. Et voilà!

$$\begin{array}{r} \overbrace{1010101001}^{n} \\ \downarrow \\ -\ 101001 \\ \quad 1010 \\ \downarrow \\ 011111 \end{array}$$

И дополнительный приятный факт: $2^{2n} = -2^n = +1$. Двойка является корнем из единицы заведомо известной нам степени $2n$.

Что там с асимптотикой?

Окей, мы решили, что сводим задачу умножения длинных чисел к нескольким задачам умножения чисел по модулю $2^n + 1$ посредством быстрого преобразования Фурье. Более формально, если на входе у нас были числа длиной N бит, мы делим их на m частей по n бит, $N = m \cdot n$, по уже хорошо знакомой нам процедуре:

$$\overbrace{\underbrace{1001}_{n}\underbrace{0100}_{n}\underbrace{1100}_{n}\underbrace{0001}_{n}}^{N = m \cdot n \text{ бит}} \times \overbrace{\underbrace{1100}_{n}\underbrace{0000}_{n}\underbrace{0100}_{n}\underbrace{1110}_{n}}^{N = m \cdot n \text{ бит}}$$

Дальше снова превращаем их в многочлены степени $m - 1$:

$$\left(1001 \cdot x^3 + 0100 \cdot x^2 + 1100 \cdot x + 0001\right) \times \left(1100 \cdot x^3 + 0000 \cdot x^2 + 0100 \cdot x + 1110\right)$$

Коэффициенты этих многочленов рассматриваем в арифметике по модулю $2^{n'} + 1$, где $n' \approx 2n + \log m$ — количество бит, способное вместить сумму m произведений двух чисел длины n бит, чтобы не было переполнения. Именно таким может быть самый большой коэффициент в произведении этих двух многочленов:

$$c_{m-1} = \underbrace{a_0 b_{m-1} + a_1 b_{m-2} + \ldots + a_{m-1} b_0}_{m \text{ слагаемых}} \leq m \cdot 2^n \cdot 2^n$$

Мы уже знаем, что можно свести умножение таких многочленов к умножению $2m-1$ чисел в $\mathbb{Z}_{2^{n'}+1}$ посредством быстрого преобразования Фурье; эти умножения поменьше легко можно провести этим же алгоритмом, а взять потом остаток от деления на $2^{n'}+1$, как мы уже выяснили, можно легко и быстро. Само сведение — быстрое преобразование Фурье в модульной арифметике с быстрыми корнями из единицы — обойдётся нам в $O(m\log m)$ сложений и умножений стоимости $O(n')$, то есть суммарно $O(m\log m\cdot n')=O(mn\log m)=O(N\log m)$ битовых операций. Итого сложность $C(N)$ совершения умножения имеет вид

$$C(N)=(2m-1)\cdot C(2n+\log m)+O(N\log m)$$

Насколько крутое ускорение мы можем получить? Сейчас я проведу сильно упрощённый анализ «на пальцах»; более строгий можно глянуть, например, в [5].

Если выбрать $n\approx m\approx\sqrt{N}$ и учесть, что $\log m$ очень мало по сравнению с $2n$, имеем

$$C(N)\approx 2\sqrt{N}\cdot C(2\sqrt{N})+O(N\log N)$$

Сделав замену $C(N)=c(N)\cdot N\log N$, получим более простую для анализа формулу:

$$c(N)\cdot N\log N\approx 2\sqrt{N}\cdot c(2\sqrt{N})\cdot 2\sqrt{N}\log 2\sqrt{N}+O(N\log N)$$

$$c(N)\cdot N\log N\approx 2N\cdot c(2\sqrt{N})\cdot \cancel{2}\cdot\frac{1}{\cancel{2}}\log N+\underbrace{O(N\log N)}_{+\ldots\cdot\log 2}$$

$$c(N)\cdot \cancel{N\log N}\approx 2\cdot c(2\sqrt{N})\cdot \cancel{N\log N}+O(\cancel{N\log N}^{1})$$

$$c(N)\approx 2\cdot c(2\sqrt{N})+\mathrm{const}$$

Так, рекуррентное соотношение. От N бит переходим к $2\sqrt{N}$, потом к $2\sqrt{2}\sqrt[4]{N}$, потом к $2\sqrt{2}\sqrt[4]{2}\sqrt[8]{N}$ и так далее. Раскрывая k раз, получаем

$$c(N) \approx 2^k c(2\sqrt{2}\sqrt[4]{2}\ldots\sqrt[2^{k-1}]{2}\sqrt[2^k]{N}) + \ldots + 4 \cdot \text{const} + 2 \cdot \text{const} + \text{const}$$

$\sqrt[2^k]{N}$ убывает очень быстро, и как только мы дойдём до какого-то достаточно малого числа, рекурсия остановится. Какая будет глубина рекурсии? Можем заметить, что на k-м шаге длина чисел, с которыми мы имеем дело, имеет вид

$$2\sqrt{2}\sqrt[4]{2}\ldots\sqrt[2^{k-1}]{2}\sqrt[2^k]{N} = \underbrace{2^{\overbrace{\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}}\right)}^{2}}}_{4} \cdot N^{\frac{1}{2^k}}$$

Соответственно, решая уравнение $N^{\frac{1}{2^k}} = \text{const}$, получаем глубину рекурсии $k \approx \log\log N$. Складывая степени двойки от нулевой до $\log\log N$ (в равенстве выше они умножаются на const), получаем $c(N) = \log N$ и суммарную сложность

$$C(N) = O(N \cdot \log^2 N)$$

И это *медленно*! В первом алгоритме Шенхаге-Штрассена мы уже добились сложности чуть хуже $O(N \log N \log\log N)$. Нужно ускорить ещё.

Наблюдение второе: самые короткие многочлены

В выражении, оценивающем сложность

$$c(N) \approx 2^k c(2\sqrt{2}\sqrt[4]{2}\ldots\sqrt[2^{k-1}]{2}\sqrt[2^k]{N}) + \ldots + 4 \cdot \text{const} + 2 \cdot \text{const} + \text{const}$$

можно заметить, что основной вклад в слишком большой результат вносят множители-степени двойки. Они там потому, что разбивая число на m частей, мы сводим задачу к $\approx 2m$ умножений. Это нужно, потому что при умножении двух k-битных чисел получается $(2k-1)$-битное число.

Но нам не нужно $(2k-1)$-битное число. Как только мы сделали первый шаг рекурсии, все вычисления у нас происходят по модулю 2^k+1. А при вычислениях по модулю длина чисел не может изменяться! Можно ли как-то вместо удвоения длины и взятия остатка от деления обойтись расчётами на k битах?

Если мы хотим при этом сохранить преимущества быстрого преобразования Фурье, от перемножения многочленов мы отказаться не можем. Есть ли в мире многочленов что-то, похожее на арифметику по модулю? Есть!

Арифметика многочленов по модулю???

При обычном перемножении многочленов коэффициент при x^m соответствует битам произведения чисел, начиная с $(m \cdot n)$-го. Если же мы делаем расчёты по модулю x^m+1, коэффициент при x^m становится эквивалентен нулевому коэффициенту (при x^0) со знаком минус:

$$\begin{gathered} x^m+1 \equiv 0 \mod (x^m+1) \\ \Downarrow \\ x^m \equiv -1 \mod (x^m+1) \\ \Downarrow \\ ax^m \equiv -a \mod (x^m+1) \end{gathered}$$

А в задаче умножения по модулю $2^{mn}+1$ $(m \cdot n)$-й бит… тоже соответствует нулевому биту со знаком минус!

$$2^{mn} \equiv -1 \mod (2^{mn}+1)$$

Это и есть наше ключевое **второе наблюдение**: перемножение многочленов по модулю x^m+1 эквивалентно перемножению чисел по модулю 2^{mn}.

Теперь у нашего многочлена-произведения есть только m значащих коэффициентов. Но есть нюанс: как только мы начинаем рассматривать многочлены по модулю, фраза *значение многочлена в точке* x теряет смысл. Многочлен в арифметике по модулю x^m+1 — это не конкретная функция $f(x)$, а совокупность функций вида

$$f(x) + (d_0 + d_1 x + d_2 x^2 + \ldots) \cdot (x^m + 1)$$

Коэффициенты d_i могут быть произвольными, и могут изменять значения многочлена в разных точках.

Мы можем быстренько провести вычислительный эксперимент, чтобы проверить, насколько хорошо заработает наш проверенный подход с матрицей Вандермонда, если мы попытаемся перемножать многочлены по модулю, просто урезав длину вектора коэффициентов. Попробуем взять разные наборы точек и посмотрим, что будет.

```
# Будем делить числа на 4 куска по 2 бита.
m = 4
n_ = 4  # n' с запасом

# Основание арифметики коэффициентов:
mod = 2**n_ + 1  # 17

# Два 8-битных множителя:
a_coefs = [0b01, 0b01, 0b10, 0b01]  # 101
b_coefs = [0b01, 0b11, 0b00, 0b01]  # 77


def construct_vandermonde_matrix_and_inverse(xs: List[int]):
    # Строим матрицу Вандермонда "в лоб",
    # потому что нам нужна модульная арифметика:
    v = np.array([
        [pow(xs[i], j, mod) for j in range(m)]
        for i in range(m)
    ])

    # Дальше хитрый фокус, чтобы обратить матрицу
    # в модульной арифметике. Да, мне было лень писать
    # метод Гаусса или конвертировать матрицу из sympy.
    det = int(round(np.linalg.det(v)))
    det_inv = pow(det, mod - 2, mod)
    v_inv_real = np.linalg.inv(v) * det * det_inv
    v_inv = np.array(np.round(v_inv_real), dtype=int) % mod

    # Проверяем, что действительно получилась
    # обратная матрица в модульной арифметике:
    assert np.all((v @ v_inv) % mod == np.eye(m, dtype=int))

    return v, v_inv
```

```
# Перебираем разные наборы точек:
for xs in [
    [2, 8, 15, 9],
    [2, 4, 8, 16],
    [3, 5, 7, 11],
    [5, 7, 11, 13],
]:
    v, v_inv = construct_vandermonde_matrix_and_inverse(xs)
    a_y = (v @ a_coefs) % mod
    b_y = (v @ b_coefs) % mod
    c_y = (a_y * b_y) % mod
    c_coefs = (v_inv @ c_y) % mod
    print(c_coefs)

# [14  2  4  8]
# [ 2 10  4 16]
# [ 4  4  0  8]
# [ 0  6  4 13]
```

В зависимости от выбранных точек получаем разные коэффициенты произведения! Но, в отличие от прошлого раза, теперь дело заведомо не в вычислительной погрешности — уж перемножать целые числа мы умеем точно.

Дело как раз в том, что мы никак не учли здесь, что мы умножаем по модулю $x^m + 1$. Как это учесть?

Посмотрим ещё раз на многочлен по модулю:

$$f(x) + (d_0 + d_1 x + d_2 x^2 + \ldots) \cdot (x^m + 1)$$

На самом деле *есть несколько точек*, в которых фраза *значение многочлена в точке* не бессмысленна. Эти точки — корни многочлена $x^m + 1$! В этих точках «произвольная» часть этой обобщённой функции всегда обращается в ноль, благодаря чему значение фиксируется.

Какие это точки в примере выше?

```
def xmp1(x):
    return (pow(x, m, mod) + 1) % mod

{i: xmp1(i) for i in range(mod)}

# { 0:  1,
#   1:  2,
#   2:  0,   <--
#   3: 14,
#   4:  2,
#   5: 14,
#   6:  5,
#   7:  5,
#   8:  0,   <--
#   9:  0,   <--
#  10:  5,
#  11:  5,
#  12: 14,
#  13:  2,
#  14: 14,
#  15:  0,   <--
#  16:  2}
```

А это — нечётные степени двойки, взятые по модулю:

```
[pow(2, i, mod) for i in range(1, mod, 2)]
# [2, 8, 15, 9, 2, 8, 15, 9]
```

В более общем случае это нечётные степени числа $\omega = 2^{n'/m}$: $\omega, \omega^3, \ldots$.

Нечётные степени не очень удобны для быстрого преобразования Фурье. Тут остаётся последний шаг — факторизовать матрицу Вандермонда для чисел $\omega, \omega^3, \ldots$, превратив в произведение матрицы Фурье и диагональной (которая не мешает, потому что на неё можно быстро умножать):

$$\begin{bmatrix} 1 & \omega & \omega^2 & \omega^3 & \dots \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots \\ 1 & \omega^5 & \omega^{10} & \omega^{15} & \dots \\ 1 & \omega^7 & \omega^{14} & \omega^{21} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots \\ 1 & \omega^4 & \omega^8 & \omega^{12} & \dots \\ 1 & \omega^6 & \omega^{12} & \omega^{18} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \times \begin{bmatrix} 1 & & & & \\ & \omega & & & \\ & & \omega^2 & & \\ & & & \omega^3 & \\ & & & & \ddots \end{bmatrix}$$

Итак, с перемножением многочленов по модулю разобрались — вернёмся к анализу сложности алгоритма. Поскольку мы перешли от многочленов порядка $2m - 1$ к многочленам порядка $m - 1$, множитель 2 в формуле сложности алгоритма исчезает:

$$c(N) \approx \cancel{2} \cdot c(2\sqrt{N}) + \text{const}$$

Глубина рекурсии остаётся порядка $\log \log N$, но без накапливающегося множителя-двойки $c(N)$ также становится порядка $\log \log N$:

$$c(N) \approx \underbrace{\cancel{2^k} \cdot c(2\sqrt{2}\sqrt[4]{2}\dots\sqrt[2^{k-1}]{2}\sqrt[2^k]{N}) + \dots + \cancel{4} \cdot \text{const} + \cancel{2} \cdot \text{const} + \text{const}}_{\log \log N \text{ слагаемых}}$$

Суммарная сложность теперь имеет вид

$$C(N) = O(N \cdot \log N \cdot \log \log N)$$

Что наконец-то превосходит первый алгоритм Шенхаге-Штрассена!

Собираем алгоритм

Итак! Перейдём к полному описанию второго алгоритма.

1. Формулируем изначальную задачу как задачу умножения по модулю $2^N + 1$. Какие бы у нас ни были изначально числа, всегда можно взять N достаточно большим, чтобы результат умножения влез в N бит. Для удобства рекурсии выбираем в качестве N степень двойки.

$$\overbrace{1001010011000001}^{N=2^k\text{ бит}} \times \overbrace{1100000001001110}^{N=2^k\text{ бит}} \mod (2^N+1)$$

2. Выбираем числа m и n такие, что $N = m \cdot n$ и $m, n \approx \sqrt{N}$. Есть хитрые формулы для наилучшего выбора, я их приводить не буду; можно подсмотреть в [5]. Делим входные числа, которые нужно перемножить, на m частей по n бит.

$$\overbrace{\underbrace{1001}_{n}\underbrace{0100}_{n}\underbrace{1100}_{n}\underbrace{0001}_{n}}^{N=m\cdot n\text{ бит}} \times \overbrace{\underbrace{1100}_{n}\underbrace{0000}_{n}\underbrace{0100}_{n}\underbrace{1110}_{n}}^{N=m\cdot n\text{ бит}} \mod (2^N+1)$$

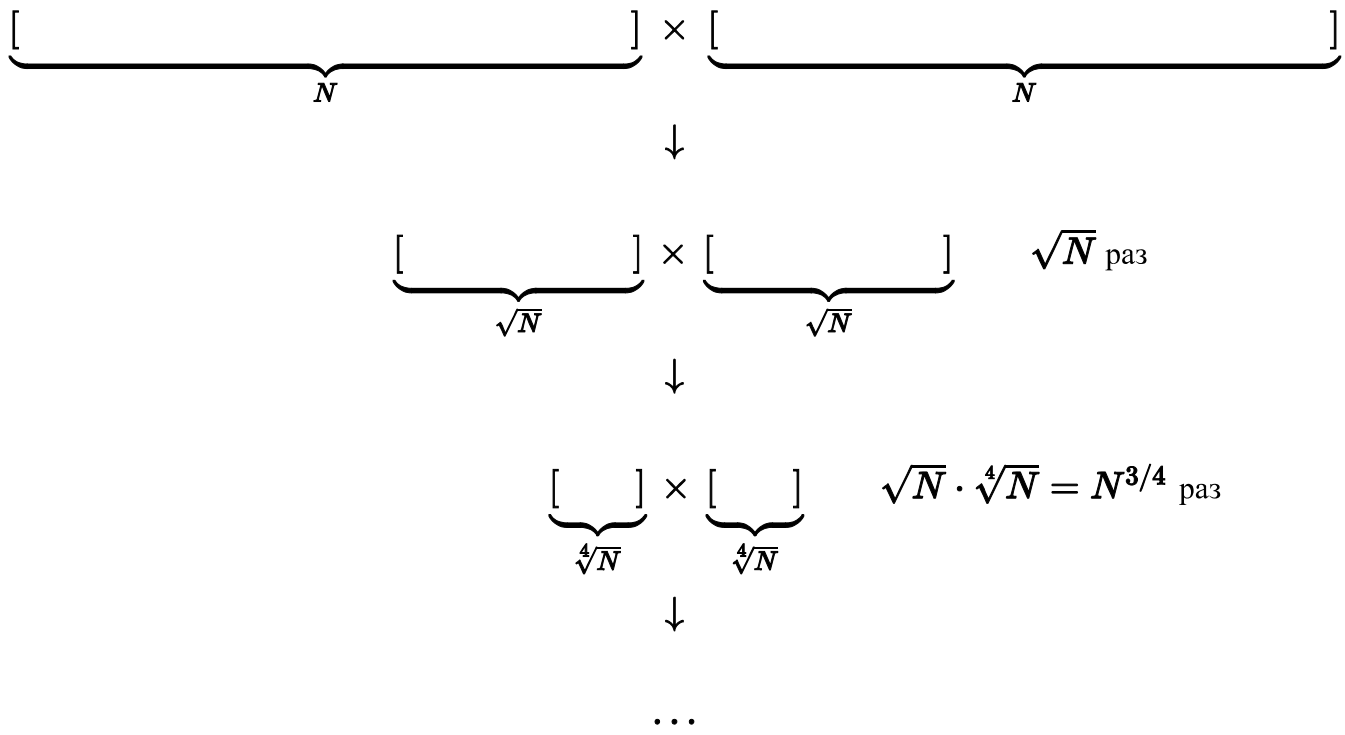
3. Превращаем в многочлены с коэффициентами, содержащими $n' \approx 2n + \log m$ бит, то есть добавляем несколько нулевых бит, чтобы промежуточные вычисления помещались без переполнения.

$$(\underbrace{001001}_{n'} \cdot x^3 + \underbrace{000100}_{n'} \cdot x^2 + \ldots) \times (\underbrace{001100}_{n'} \cdot x^3 + \underbrace{000000}_{n'} \cdot x^2 + \ldots)$$

4. Теперь у нас на руках два вектора коэффициентов многочленов. Векторы длины $m+1$, числа в них длины n' бит. Эти векторы умножаем сперва на диагональную матрицу (в формулах выше) за $O(m \cdot n') = O(N)$, потом делаем быстрое преобразование Фурье за $O(m \cdot n' \cdot \log m) = O(N \log N)$.

$$FFT \times \begin{bmatrix} 1 & & & & \\ & \omega & & & \\ & & \omega^2 & & \\ & & & \omega^3 & \\ & & & & \ddots \end{bmatrix} \times \begin{bmatrix} 000001 \\ 001100 \\ 000100 \\ 001001 \\ \vdots \end{bmatrix}, \quad FFT \times [\ldots] \times \begin{bmatrix} 001110 \\ 000100 \\ 000000 \\ 001100 \\ \vdots \end{bmatrix}$$

5. Полученные векторы перемножаем поэлементно. Для перемножения чисел по модулю $2^{n'}+1$ используем рекурсивно этот же самый алгоритм (или простые алгоритмы, если числа уже достаточно маленькие). Здесь будет рекурсия глубины $O(\log \log N)$.



6. К получившемуся вектору применяем обратное преобразование Фурье за $O(N \log N)$, потом умножаем на обратную диагональную матрицу за $O(N)$.

$$\begin{bmatrix} 1 & & & & \\ & \omega^{-1} & & & \\ & & \omega^{-2} & & \\ & & & \omega^{-3} & \\ & & & & \ddots \end{bmatrix} \times IFFT \times \begin{bmatrix} 001001 \\ 110111 \\ 1000000 \\ 011001 \\ \vdots \end{bmatrix}$$

(Обратите внимание на слишком длинный элемент 1000000 в этом примере — это ровно $2^{n'}$, наибольшее число, разрешённое модульной арифметикой!)

7. Из получившегося вектора коэффициентов многочлена-произведения восстанавливаем число-результат умножения, суммируя в столбик справа налево за $O(N)$.

$$\begin{array}{r} \overleftarrow{1110} \\ 11001100 \\ 1101000 \\ 10011010 \\ 10110100 \\ 110000 \\ 1101100 \\ \hline 11011111011111000010010110011110 \end{array}$$

Вот и всё! Все гениальные приёмы второго алгоритма Шенхаге-Штрассена лежат у нас перед глазами. Конечно, для реализации этого алгоритма, эффективной на практике, нужно учесть ещё много нюансов и применить много приёмов; но на уровне идейном мы освоили его целиком.

В совсем не таком уж далёком 1960 году Андрей Колмогоров, один из величайших математиков своего времени, выдвинул гипотезу, что невозможно умножать числа быстрее, чем за $O(N^2)$. И совершенно изумительно видеть, насколько быстрее оказалось возможно умножать числа, чем это изначально предполагалось. В этой статье мы рассмотрели четыре основополагающих алгоритма, перейдя от $O(N^2)$ к $O(N \log N \log\log N)$; для их понимания нам пришлось разобраться в теории Фурье, модульной арифметике и алгебре многочленов.

За рамками статьи остался алгоритм Фюрера и его варианты, а также опубликованный в 2020-м году [6] алгоритм с заявленной сложностью $O(N \log N)$; если эта статья окажется востребована, возможно, когда-нибудь мы разберём и их.

Спасибо за внимание!

P. S.

Эту статью я старался написать максимально популярно, избегая присущей учебникам математической строгости. Некоторые формулы намеренно упрощены, где-то вообще удалось обойти без них (мало где); в их оформлении там, где они остались, я сделал акцент на читаемость с ходу, без разбора с бумажкой и ручкой (выделение цветом, зачёркивание сокращённых величин); также я старался избегать математической терминологии, где это возможно («арифметика по модулю» вместо «в кольце вычетов по модулю»). Судьёй того, насколько моя задумка удалась, предстоит быть тебе, читатель; мне же остаётся лишь надеяться, что кто-то сможет почерпнуть из этого текста свежий взгляд на математику, обычно такую возвышенно-труднодоступную.

Литература

1. Pan, Victor. "How can we speed up matrix multiplication?." SIAM review 26.3 (1984): 393-415. PDF
2. Pan, Victor. "Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations." 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). IEEE, 1978.
3. Pan, Victor. "How bad are Vandermonde matrices?." SIAM Journal on Matrix Analysis and Applications 37.2 (2016): 676-694. PDF

4. Pharr, Matt, and Randima Fernando. GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems). Addison-Wesley Professional, 2005. HTML
5. Kruppa, Alexander. "A GMP-based implementation of Schonhage-Strassen’s large integer multiplication algorithm." PDF
6. Harvey, David, and Joris Van Der Hoeven. "Integer multiplication in time O (n log n)." *Annals of Mathematics* 193.2 (2021): 563-617. PDF

Библиотеки

1. NumPy использовалась для большинства численных экспериментов в статье.
2. SymPy использовалась для нескольких символьных вычислений.
3. GMP неоднократно упоминалась как стандартная реализация длинной арифметики.