Алгоритм Евклида нахождения НОД (наибольшего общего делителя)

Даны два целых неотрицательных числа a и b. Требуется найти их наибольший общий делитель, т.е. наибольшее число, которое является делителем одновременно и a, и b. На английском языке "наибольший общий делитель" пишется "greatest common divisor", и распространённым его обозначением является $\gcd$:

$$\gcd(a, b) = \max_{k=1...\infty \ : \ k|a \ \& \ k|b} k$$

(здесь символом "$|$" обозначена делимость, т.е. "$k|a$" обозначает "k делит a")

Когда оно из чисел равно нулю, а другое отлично от нуля, их наибольшим общим делителем, согласно определению, будет это второе число. Когда оба числа равны нулю, результат не определён (подойдёт любое бесконечно большое число), мы положим в этом случае наибольший общий делитель равным нулю. Поэтому можно говорить о таком правиле: если одно из чисел равно нулю, то их наибольший общий делитель равен второму числу.

Алгоритм Евклида, рассмотренный ниже, решает задачу нахождения наибольшего общего делителя двух чисел a и b за $O(\log \min(a, b))$.

Данный алгоритм был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.), хотя, вполне возможно, этот алгоритм имеет более раннее происхождение.

Алгоритм

Сам алгоритм чрезвычайно прост и описывается следующей формулой:

$$\gcd(a, b) = \begin{cases} a, & \text{if b=0} \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Реализация

```
int gcd (int a, int b) {
        if (b == 0)
                return a;
        else
                return gcd (b, a % b);
}
```

Используя тернарный условный оператор C++, алгоритм можно записать ещё короче:

```
int gcd (int a, int b) {
        return b ? gcd (b, a % b) : a;
}
```

Наконец, приведём и нерекурсивную форму алгоритма:

```
int gcd (int a, int b) {
        while (b) {
                a %= b;
                swap (a, b);
        }
        return a;
}
```

Доказательство корректности

Сначала заметим, что при каждой итерации алгоритма Евклида его второй аргумент строго убывает, следовательно, посколько он неотрицательный, то алгоритм Евклида **всегда завершается**.

Для **доказательства корректности** нам необходимо показать, что $\gcd(a, b) = \gcd(b, a \bmod b)$ для любых $a \geq 0, b > 0$.

Покажем, что величина, стоящая в левой части равенства, делится на настоящую в правой, а стоящая в правой — делится на стоящую в левой. Очевидно, это будет означать, что левая и правая части совпадают, что и докажет корректность алгоритма Евклида.

Обозначим $d = \gcd(a, b)$. Тогда, по определению, $d|a$ и $d|b$.

Далее, разложим остаток от деления a на b через их частное:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Но тогда отсюда следует:

$$d \mid (a \bmod b)$$

Итак, вспоминая утверждение $d|b$, получаем систему:

$$\begin{cases} d \mid b, \\ d \mid (a \bmod b) \end{cases}$$

Воспользуемся теперь следующим простым фактом: если для каких-то трёх чисел p, q, r выполнено: $p|q$ и $p|r$, то выполняется и: $p \mid \gcd(q, r)$. В нашей ситуации получаем:

$$d \mid \gcd(b, a \bmod b)$$

Или, подставляя вместо d его определение как $\gcd(a, b)$, получаем:

$$\gcd(a, b) \mid \gcd(b, a \bmod b)$$

Итак, мы провели половину доказательства: показали, что левая часть делит правую. Вторая половина доказательства производится аналогично.

Время работы

Время работы алгоритма оценивается **теоремой Ламе**, которая устанавливает удивительную связь алгоритма Евклида и последовательности Фибоначчи:

Если $a > b \geq 1$ и $b < F_n$ для некоторого n, то алгоритм Евклида выполнит не более $n - 2$ рекурсивных вызовов.

Более того, можно показать, что верхняя граница этой теоремы — оптимальная. При $a = F_n, b = F_{n-1}$ будет выполнено именно $n - 2$ рекурсивных вызова. Иными словами, **последовательные числа Фибоначчи — наихудшие входные данные** для алгоритма Евклида.

Учитывая, что числа Фибоначчи растут экспоненциально (как константа в степени n), получаем, что алгоритм Евклида выполняется за $O(\log \min(a, b))$ операций умножения.

НОК (наименьшее общее кратное)

Вычисление наименьшего общего кратного (least common multiplier, lcm) сводится к вычислению $\gcd$ следующим простым утверждением:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Таким образом, вычисление НОК также можно сделать с помощью алгоритма Евклида, с той же асимптотикой:

```
int lcm (int a, int b) {
        return a / gcd (a, b) * b;
}
```

(здесь выгодно сначала поделить на $\gcd$, а только потом домножать на b, поскольку это поможет избежать переполнений в некоторых случаях)

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]

Решето Эратосфена

Решето Эратосфена — это алгоритм, позволяющий найти все простые числа в отрезке $[1; n]$ за $O(n \log \log n)$ операций.

Идея проста — запишем ряд чисел $1 \dots n$, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем делящиеся на 3, кроме самого числа 3, затем на 5, затем на $7, 11$, и все остальные простые до n.

Реализация

Сразу приведём реализацию алгоритма:

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
        if (prime[i])
                if (i * 1ll * i <= n)
                        for (int j=i*i; j<=n; j+=i)
                                prime[j] = false;
```

Этот код сначала помечает все числа, кроме нуля и единицы, как простые, а затем начинает процесс отсеивания составных чисел. Для этого мы перебираем в цикле все числа от 2 до n, и, если текущее число i простое, то помечаем все числа, кратные ему, как составные.

При этом мы начинаем идти от i^2, поскольку все меньшие числа, кратные i, обязательно имеют простой делитель меньше i, а значит, все они уже были отсеяны раньше. (Но поскольку i^2 легко может переполнить тип int, в коде перед вторым вложенным циклом делается дополнительная проверка с использованием типа $long\ long$.)

При такой реализации алгоритм потребляет $O(n)$ памяти (что очевидно) и выполняет $O(n \log \log n)$ действий (это доказывается в следующем разделе).

Асимптотика

Докажем, что асимптотика алгоритма равна $O(n \log \log n)$.

Итак, для каждого простого $p \le n$ будет выполняться внутренний цикл, который совершит $\frac{n}{p}$ действий. Следовательно, нам нужно оценить следующую величину:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{n}{p} = n \cdot \sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{1}{p}.$$

Вспомним здесь два известных факта: что число простых, меньше либо равных n, приблизительно равно $\frac{n}{\ln n}$, и что k-ое простое число приблизительно равно $k \ln k$ (это следует из первого утверждения). Тогда сумму можно записать таким образом:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}.$$

Здесь мы выделили первое простое из суммы, поскольку при $k = 1$ согласно приближению $k \ln k$ получится 0, что приведёт к делению на нуль.

Теперь оценим такую сумму с помощью интеграла от той же функции по k от 2 до $\frac{n}{\ln n}$ (мы можем производить такое приближение, поскольку, фактически, сумма относится к интегралу как его приближение по формуле прямоугольников):

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk.$$

Первообразная для подынтегральной функции есть $\ln \ln k$. Выполняя подстановку и убирая члены меньшего порядка, получаем:

$$\int\limits_2^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n.$$

Теперь, возвращаясь к первоначальной сумме, получаем её приближённую оценку:

$$\sum_{\substack{p \le n, \\ p\ is\ prime}} \frac{n}{p} \approx n \ln \ln n + o(n),$$

что и требовалось доказать.

Более строгое доказательство (и дающее более точную оценку с точностью до константных множителей) можно найти в книге Hardy и Wright "An Introduction to the Theory of Numbers" (стр. 349).

Различные оптимизации решета Эратосфена

Самый большой недостаток алгоритма — то, что он "гуляет" по памяти, постоянно выходя за пределы кэш-памяти, из-за чего константа, скрытая в $O(n \log \log n)$, сравнительно велика.

Кроме того, для достаточно больших n узким местом становится объём потребляемой памяти.

Ниже рассмотрены методы, позволяющие как уменьшить число выполняемых операций, так и значительно сократить потребление памяти.

Просеивание простыми до корня

Самый очевидный момент — что для того, чтобы найти все простые до n, достаточно выполнить просеивание только простыми, не превосходящими корня из n.

Таким образом, изменится внешний цикл алгоритма:

```
for (int i=2; i*i<=n; ++i)
```

На асимптотику такая оптимизация не влияет (действительно, повторив приведённое выше доказательство, мы получим оценку $n \ln \ln \sqrt{n} + o(n)$, что, по свойствам логарифма, асимптотически есть то же самое), хотя число операций заметно уменьшится.

Решето только по нечётным числам

Поскольку все чётные числа, кроме 2, — составные, то можно вообще не обрабатывать никак чётные числа, а оперировать только нечётными числами.

Во-первых, это позволит вдвое сократить объём требуемой памяти. Во-вторых, это уменьшит число делаемых алгоритмом операций примерно вдвое.

Уменьшение объёма потребляемой памяти

Заметим, что алгоритм Эратосфена фактически оперирует с n битами памяти. Следовательно, можно существенно сэкономить потребление памяти, храня не n байт — переменных булевского типа, а n бит, т.е. $n/8$ байт памяти.

Однако такой подход — "**битовое сжатие**" — существенно усложнит оперирование этими битами. Любое чтение или запись бита будут представлять из себя несколько арифметических операций, что в итоге приведёт к замедлению алгоритма.

Таким образом, этот подход оправдан, только если n настолько большое, что n байт памяти выделить уже нельзя. Сэкономив память (в 8 раз), мы заплатим за это существенным замедлением алгоритма.

В завершение стоит отметить, что в языке C++ уже реализованы контейнеры, автоматически осуществляющие битовое сжатие: vector<bool> и bitset<>. Впрочем, если скорость работы очень важна, то лучше реализовать битовое сжатие вручную, с помощью битовых операций — на сегодняшний день компиляторы всё же не в состоянии генерировать достаточно быстрый код.

Блочное решето

Из оптимизации "просеивание простыми до корня" следует, что нет необходимости хранить всё время весь массив $prime[1 \dots n]$. Для выполнения просеивания достаточно хранить только простые до корня из n, т. е. $prime[1 \dots \sqrt{n}]$, а остальную часть массива $prime$ строить поблочно, храня в текущий момент времени только один блок.

Пусть s — константа, определяющая размер блока, тогда всего будет $\left\lceil \frac{n}{s} \right\rceil$ блоков, k-ый блок ($k = 0 \dots \left\lfloor \frac{n}{s} \right\rfloor$) содержит числа в отрезке $[ks; ks + s - 1]$. Будем обрабатывать блоки по очереди, т.е. для каждого k-го блока будем перебирать все простые (от 1 до $\sqrt{n}$) и выполнять ими просеивание только внутри текущего блока. Аккуратно стоит обрабатывать первый блок — во-первых, простые из $[1; \sqrt{n}]$ не должны удалить сами себя, а во-вторых, числа 0 и 1 должны особо помечаться как не простые. При обработке последнего блока также следует не забывать о том, что последнее нужное число n не обязательно находится в конце блока.

Приведём реализацию блочного решета. Программа считывает число n и находит количество простых от 1 до n:

```
const int SQRT_MAXN = 100000; // корень из максимального значения N
const int S = 10000;
bool nprime[SQRT_MAXN], bl[S];
int primes[SQRT_MAXN], cnt;

int main() {

        int n;
        cin >> n;
        int nsqrt = (int) sqrt (n + .0);
        for (int i=2; i<=nsqrt; ++i)
                if (!nprime[i]) {
                        primes[cnt++] = i;
                        if (i * 1ll * i <= nsqrt)
                                for (int j=i*i; j<=nsqrt; j+=i)
                                        nprime[j] = true;
                }

        int result = 0;
        for (int k=0, maxk=n/S; k<=maxk; ++k) {
                memset (bl, 0, sizeof bl);
                int start = k * S;
                for (int i=0; i<cnt; ++i) {
                        int start_idx = (start + primes[i] - 1) / primes[i];
                        int j = max(start_idx,2) * primes[i] - start;
                        for (; j<S; j+=primes[i])
                                bl[j] = true;
                }
                if (k == 0)
                        bl[0] = bl[1] = true;
                for (int i=0; i<S && start+i<=n; ++i)
                        if (!bl[i])
                                ++result;
        }
        cout << result;

}
```

Асимптотика блочного решета такая же, как и обычного решета Эратосфена (если, конечно, размер s блоков не будет совсем маленьким), зато объём используемой памяти сократится до $O(\sqrt{n} + s)$ и уменьшится "блуждание" по памяти. Но, с другой стороны, для каждого блока для каждого простого из $[1; \sqrt{n}]$ будет выполняться деление, что будет сильно сказываться при меньших размерах блока. Следовательно, при выборе константы s необходимо соблюсти баланс.

Как показывают эксперименты, наилучшая скорость работы достигается, когда s имеет значение приблизительно от 10^4 до 10^5.

Улучшение до линейного времени работы

Алгоритм Эратосфена можно преобразовать в другой алгоритм, который уже будет работать за линейное время — см. статью "Решето Эратосфена с линейным временем работы". (Впрочем, этот алгоритм имеет и недостатки.)

Расширенный алгоритм Евклида

В то время как "обычный" алгоритм Евклида просто находит наибольший общий делитель двух чисел a и b, расширенный алгоритм Евклида находит помимо НОД также коэффициенты x и y такие, что:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

Т.е. он находит коэффициенты, с помощью которых НОД двух чисел выражается через сами эти числа.

Алгоритм

Внести вычисление этих коэффициентов в алгоритм Евклида несложно, достаточно вывести формулы, по которым они меняются при переходе от пары (a, b) к паре $(b\%a, a)$ (знаком процента мы обозначаем взятие остатка от деления).

Итак, пусть мы нашли решение (x_1, y_1) задачи для новой пары $(b\%a, a)$:

$$(b\%a) \cdot x_1 + a \cdot y_1 = g,$$

и хотим получить решение (x, y) для нашей пары (a, b):

$$a \cdot x + b \cdot y = g.$$

Для этого преобразуем величину $b\%a$:

$$b\%a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a.$$

Подставим это в приведённое выше выражение с x_1 и y_1 и получим:

$$g = (b\%a) \cdot x_1 + a \cdot y_1 = \left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right) \cdot x_1 + a \cdot y_1,$$

и, выполняя перегруппировку слагаемых, получаем:

$$g = b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right).$$

Сравнивая это с исходным выражением над неизвестными x и y, получаем требуемые выражения:

$$\begin{cases} x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1, \\ y = x_1. \end{cases}$$

Реализация

```
int gcd (int a, int b, int & x, int & y) {
        if (a == 0) {
                x = 0; y = 1;
                return b;
        }
        int x1, y1;
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
}
```

Это рекурсивная функция, которая по-прежнему возвращает значение НОД от чисел a и b, но помимо этого — также искомые коэффициенты x и y в виде параметров функции, передающихся по ссылкам.

База рекурсии — случай a. Тогда НОД равен b, и, очевидно, требуемые коэффициенты x и y равны 0 и 1 соответственно. В остальных случаях работает обычное решение, а коэффициенты пересчитываются по вышеописанным формулам.

Расширенный алгоритм Евклида в такой реализации работает корректно даже для отрицательных чисел.

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы**: **Построение и анализ** [2005]

Числа Фибоначчи

Определение

Последовательность Фибоначчи определяется следующим образом:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

Несколько первых её членов:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ ,\ 55,\ 89,\ \ldots$$

История

Эти числа ввёл в 1202 г. Леонардо Фибоначчи (Leonardo Fibonacci) (также известный как Леонардо Пизанский (Leonardo Pisano)). Однако именно благодаря математику 19 века Люка (Lucas) название "числа Фибоначчи" стало общеупотребительным.

Впрочем, индийские математики упоминали числа этой последовательности ещё раньше: Гопала (Gopala) до 1135 г., Хемачандра (Hemachandra) — в 1150 г.

Числа Фибоначчи в природе

Сам Фибоначчи упоминал эти числа в связи с такой задачей: "Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов за год может произвести на свет эта пара, если известно, что каждый месяц, начиная со второго, каждая пара кроликов производит на свет одну пару?". Решением этой задачи и будут числа последовательности, называемой теперь в его честь. Впрочем, описанная Фибоначчи ситуация — больше игра разума, чем реальная природа.

Индийские математики Гопала и Хемачандра упоминали числа этой последовательности в связи с количеством ритмических рисунков, образующихся в результате чередования долгих и кратких слогов в стихах или сильных и слабых долей в музыке. Число таких рисунков, имеющих в целом n долей, равно F_n.

Числа Фибоначчи появляются и в работе Кеплера 1611 года, который размышлял о числах, встречающихся в природе (работа "О шестиугольных снежинках").

Интересен пример растения — тысячелистника, у которого число стеблей (а значит и цветков) всегда есть число Фибоначчи. Причина этого проста: будучи изначально с единственным стеблем, этот стебель затем делится на два, затем от главного стебля ответвляется ещё один, затем первые два стебля снова разветвляются, затем все стебли, кроме двух последних, разветвляются, и так далее. Таким образом, каждый стебель после своего появления "пропускает" одно разветвление, а затем начинает делиться на каждом уровне разветвлений, что и даёт в результате числа Фибоначчи.

Вообще говоря, у многих цветов (например, лилий) число лепестков является тем или иным числом Фибоначчи.

Также в ботанике известно явление "филлотаксиса". В качестве примера можно привести расположение семечек подсолнуха: если посмотреть сверху на их расположение, то можно увидеть одновременно две серии спиралей (как бы наложенных друг на друга): одни закручены по часовой стрелке, другие — против. Оказывается, что число этих спиралей примерно совпадает с двумя последовательными числами Фибоначчи: 34 и 55 или 89 и 144. Аналогичные факты верны и для некоторых других цветов, а также для сосновых шишек, брокколи, ананасов, и т.д.

Для многих растений (по некоторым данным, для 90% из них) верен и такой интересный факт. Рассмотрим какой-нибудь лист, и будем спускаться от него вниз до тех пор, пока не достигнем листа, расположенного на стебле точно так же (т.е. направленного точно в ту же сторону). Попутно будем считать все листья, попадавшиеся нам (т.е. расположенные по высоте между стартовым листом и конечным), но расположенными по-другому. Нумеруя их, мы будем постепенно совершать витки вокруг стебля (поскольку листья расположены на стебле по спирали). В зависимости от того, совершать витки по часовой стрелке или против, будет получаться разное число витков. Но оказывается, что число витков, совершённых нами по часовой стрелке, число витков, совершённых против часовой стрелки, и число встреченных листьев образуют 3 последовательных числа Фибоначчи.

Впрочем, следует отметить, что есть и растения, для которых приведённые выше подсчёты дадут числа из совсем других последовательностей, поэтому нельзя сказать, что явление филлотаксиса является законом, — это

скорее занимательная тенденция.

Свойства

Числа Фибоначчи обладают множеством интересных математических свойств.

Вот лишь некоторые из них:

- Соотношение Кассини:

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$$

- Правило "сложения":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- Из предыдущего равенства при $k = n$ вытекает:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

- Из предыдущего равенста по индукции можно получить, что

  F_{nk} всегда кратно F_n.

- Верно и обратное к предыдущему утверждение:

  если F_m кратно F_n, то m кратно n.

- НОД-равенство:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

- По отношению к алгоритму Евклида числа Фибоначчи обладают тем замечательным свойством, что они являются наихудшими входными данными для этого алгоритма (см. "Теорема Ламе" в Алгоритме Евклида).

Фибоначчиева система счисления

Теорема Цекендорфа утверждает, что любое натуральное число n можно представить единственным образом в виде суммы чисел Фибоначчи:

$$N = F_{k_1} + F_{k_2} + \ldots + F_{k_r}$$

где $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, \cdots, k_r \geq 2$ (т.е. в записи нельзя использовать два соседних числа Фибоначчи).

Отсюда следует, что любое число можно однозначно записать в **фибоначчиевой системе счисления**, например:

$$\begin{aligned} 9 &= 8 + 1 = F_6 + F_1 = (10001)_F, \\ 6 &= 5 + 1 = F_5 + F_1 = (1001)_F, \\ 19 &= 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F, \end{aligned}$$

причём ни в каком числе не могут идти две единицы подряд.

Нетрудно получить и правило прибавления единицы к числу в фибоначчиевой системе счисления: если младшая цифра равна 0, то её заменяем на 1, а если равна 1 (т.е. в конце стоит 01), то 01 заменяем на 10. Затем "исправляем" запись, последовательно исправляя везде 011 на 100. В результате за линейное время будет получена запись нового числа.

Перевод числа в фибоначчиеву систему счисления осуществляется простым "жадным" алгоритмом: просто перебираем числа Фибоначчи от больших к меньшим и, если некоторое $F_k \leq n$, то F_k входит в запись числа n, и мы отнимаем F_k от n и продолжаем поиск.

Формула для n-го числа Фибоначчи

Формула через радикалы

Существует замечательная формула, называемая по имени французского математика Бине (Binet), хотя она была известна до него Муавру (Moivre):

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

Эту формулу легко доказать по индукции, однако вывести её можно с помощью понятия образующих функций или с помощью решения функционального уравнения.

Сразу можно заметить, что второе слагаемое всегда по модулю меньше 1, и более того, очень быстро убывает (экспоненциально). Отсюда следует, что значение первого слагаемого даёт "почти" значение F_n. Это можно записать в строгом виде:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right],$$

где квадратные скобки обозначают округление до ближайшего целого.

Впрочем, для практического применения в вычислениях эти формулы мало подходят, потому что требуют очень высокой точности работы с дробными числами.

Матричная формула для чисел Фибоначчи

Нетрудно доказать матричное следующее равенство:

$$\begin{pmatrix} F_{n-2} & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} & F_n \end{pmatrix}.$$

Но тогда, обозначая

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

получаем:

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}.$$

Таким образом, для нахождения n-го числа Фибоначчи надо возвести матрицу P в степень n.

Вспоминая, что возведение матрицы в n-ую степень можно осуществить за $O(\log n)$ (см. Бинарное возведение в степень), получается, что n-ое число Фибоначчи можно легко вычислить за $O(\log n)$ с использованием только целочисленной арифметики.

Периодичность последовательности Фибоначчи по модулю

Рассмотрим последовательность Фибоначчи F_i по некоторому модулю p. Докажем, что она является периодичной, и причём период начинается с $F_1 = 1$ (т.е. предпериод содержит только F_0).

Докажем это от противного. Рассмотрим $p^2 + 1$ пар чисел Фибоначчи, взятых по модулю p:

$$(F_1, F_2),\ (F_2, F_3),\ \ldots,\ (F_{p^2+1}, F_{p^2+2}).$$

Поскольку по модулю p может быть только p^2 различных пар, то среди этой последовательности найдётся как минимум две одинаковые пары. Это уже означает, что последовательность периодична.

Выберем теперь среди всех таких одинаковых пар две одинаковые пары с наименьшими номерами. Пусть это пары с некоторыми номерами (F_a, F_{a+1}) и (F_b, F_{b+1}). Докажем, что $a = 1$. Действительно, в противном случае для них найдутся предыдущие пары (F_{a-1}, F_a) и (F_{b-1}, F_b), которые, по свойству чисел Фибоначчи, также будут равны друг другу. Однако это противоречит тому, что мы выбрали совпадающие пары с наименьшими номерами, что и требовалось доказать.

Литература

- Рональд Грэхэм, Дональд Кнут, Орен Паташник. **Конкретная математика** [1998]