

Быстрое преобразование Фурье и многочлены

Moscow International Workshop ACM ICPC 2017

Александр Кульков

Содержание

1 Быстрое умножение	1
Метод Карацубы	1
Умножение многочленов	2
Интерполяция	2
Дискретное преобразование Фурье	2
Схема Кули-Тьюки	3
Обратное преобразование	3
Интерлюдия	4
2 Применения и вариации преобразования	5
Свёртки и корреляции	5
Преобразование в кольце по модулю	5
Chirp Z-transform	5
Одновременное преобразование вещественных многочленов	5
Умножение по произвольному модулю	6
Многомерное преобразование Фурье	6
Преобразование Уолша-Адамара и другие свёртки	6
Метод Ньютона для функций над многочленами	7
Разделяй и властвуй	8
Деление и интерполяция	8
3 Упражнения	9
Рюкзак	9
Степенной ряд	9
Общая схема Кули-Тьюки	9
Арифметические прогрессии	9
Расстояние между точками	9
Сопоставление шаблонов	9
Линейные рекурренты*	9
Степень многочлена*	9

1 Быстрое умножение

Метод Карацубы Рассмотрим такую распространённую операцию как умножение двух чисел. Со школы все знают алгоритм, работающий за $O(n^2)$: умножение в столбик. Долгое время предполагалось, что ничего быстрее придумать нельзя. Первым эту гипотезу опроверг Карацуба, хотя считается, что преобразование Фурье в своих работах использовал ещё Гаусс.

Алгоритм Карацубы лаконичен и прост. Пусть мы перемножаем $A = a_0 + a_1x$ и $B = b_0 + b_1x$. Тогда:

$$\begin{aligned} A \cdot B &= a_0b_0 + (a_0b_1 + a_1b_0)x + a_1b_1x^2 = \\ &= a_0b_0 + [(a_0 + b_0)(a_1 + b_1) - a_0b_0 - a_1b_1]x + a_1b_1x^2 \end{aligned}$$

Пусть для простоты числа нам даны в двоичной системе счисления и имеют длину n . Тогда если мы возьмём $x = 2^k$, $k \approx n/2$, то мы сведём задачу к трём вызовам той же задачи, но в два раза меньшего размера: для a_0b_0 , a_1b_1 и $(a_0 + b_0)(a_1 + b_1)$. Для времени работы в таком случае будет иметь место оценка

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3}) \approx O(n\sqrt{n})$$

Умножение многочленов Чтобы прийти к алгоритму с лучшей оценкой, мы должны обратить внимание на то, что любое число можно считать многочленом $A(2) = a_0 + a_1 \cdot 2 + \dots + a_n \cdot 2^n$. Чтобы перемножить два числа, мы можем перемножить соответствующие им многочлены, а затем произвести нормировку.

```

1  const int base = 10;
2  vector<int> normalize(vector<int> c) {
3      int carry = 0;
4      for(auto &it: c) {
5          it += carry;
6          carry = it / base;
7          it %= base;
8      }
9      while(carry) {
10         c.push_back(carry % base);
11         carry /= base;
12     }
13     return c;
14 }
15
16 vector<int> multiply(vector<int> a, vector<int> b) {
17     return normalize(poly_multiply(a, b));
18 }

```

Прямая формула для произведения многочленов имеет вид

$$\left(\sum_{i=0}^n a_i x^i \right) \cdot \left(\sum_{j=0}^m b_j x^j \right) = \sum_{k=0}^{n+m} x^k \sum_{i+j=k} a_i b_j$$

Её подсчёт требует $O(n^2)$ операций, что нас не устраивает.

Интерполяция Пусть есть набор точек x_0, \dots, x_n . Многочлен степени n однозначно задаётся своими значениями в этих точках. Можно явным образом задать многочлен, которые принимает данные значения в данных точках (интерполяционный многочлен Лагранжа):

$$y(x) = \sum_{i=0}^n y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Заметим, что если у нас есть значения двух многочленов в наборе точек, то мы можем за $O(n)$ посчитать значения произведения многочленов в этих точках. При этом степень произведения многочленов степени n и m равна $n + m$, поэтому нам достаточно просто посчитать значения каждого многочлена в каких-то $n + m$ точках.

```

1  void align(vector<int> &a, vector<int> &b) {
2      int n = a.size() + b.size() - 1;
3      while(a.size() < n) {
4          a.push_back(0);
5      }
6      while(b.size() < n) {
7          b.push_back(0);
8      }
9  }
10
11 vector<int> poly_multiply(vector<int> a, vector<int> b) {
12     align(a, b);
13     auto A = evaluate(a);
14     auto B = evaluate(b);
15     for(int i = 0; i < A.size(); i++) {
16         A[i] *= B[i];
17     }
18     return interpolate(A);
19 }

```

К сожалению, непосредственное вычисление значений требует $O(n^2)$ операций, а интерполяция и того больше, но мы можем улучшить эту оценку если будем рассматривать точки x_i с особыми свойствами.

Дискретное преобразование Фурье Пусть в поле, в котором мы работаем есть элемент w такой что

$$\begin{cases} w^k = 1 & k = n \\ w^k \neq 1 & k < n \end{cases}$$

Будем называть его образующим корнем степени n из единицы. Такой элемент обладает очень полезным свойством, на которое мы будем опираться в дальнейшем. Во-первых, все w^i различны для i от 0 до

$k - 1$, во-вторых $w^m = w^{m \bmod n}$. Значит, степени w образуют группу остатков целых чисел от деления на n . Вычисление значений многочлена в таких точках и называется дискретным преобразованием Фурье.

Чаще всего, используют такие корни из поля комплексных чисел. Исходя из формулы Эйлера

$$e^{i\varphi} = \cos \varphi + i \sin \varphi$$

можно заключить, что все они имеют вид $w^k = e^{i\frac{2\pi}{n}k}$. Кроме этого при умножении многочленов с целыми коэффициентами можно использовать корни из единицы в полях остатков по простым модулям, что будет рассмотрено позже.

Схема Кули-Тьюки Представим многочлен в виде $P(x) = A(x^2) + xB(x^2)$, где $A(x)$ состоит из коэффициентов при чётных степенях x , а $B(x)$ – из коэффициентов при нечётных. Пусть $n = 2k$. Тогда

$$w^{2t} = w^{2t \bmod 2k} = w^{2(t \bmod k)}$$

Кроме того, что нетрудно проверить, w^2 является образующим корнем степени k из единицы. Значит,

$$P(w^t) = A(w^{2(t \bmod k)}) + w^t B(w^{2(t \bmod k)})$$

Данная формула за $O(n)$ сводит дискретное преобразование размера n к двум дискретным преобразованиям размера $\frac{n}{2}$, следовательно, общее время вычислений с использованием данной формулы составит

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

Заметим, что в данной формуле существенную роль играло предположение о делимости n на 2. Значит, n должно быть чётным на каждом уровне, кроме последнего, из чего необходимо следует, что n – степень двойки. Приведём код, производящий требуемые вычисления.

```

1 typedef complex<double> ftype;
2 const double pi = acos(-1);
3
4 template<typename T>
5 vector<ftype> fft(vector<T> p, ftype w) {
6     int n = p.size();
7     if(n == 1) {
8         return vector<ftype>(1, p[0]);
9     } else {
10         vector<T> AB[2];
11         for(int i = 0; i < n; i++) {
12             AB[i % 2].push_back(p[i]);
13         }
14         auto A = fft(AB[0], w * w);
15         auto B = fft(AB[1], w * w);
16         vector<ftype> res(n);
17         ftype wt = 1;
18         int k = n / 2;
19         for(int i = 0; i < n; i++) {
20             res[i] = A[i % k] + wt * B[i % k];
21             wt *= w;
22         }
23         return res;
24     }
25 }
26
27 vector<ftype> evaluate(vector<int> p) {
28     while(__builtin_popcount(p.size()) != 1) {
29         p.push_back(0);
30     } // p.size() has to be the power of 2
31     return fft(p, polar(1., 2 * pi / p.size()));
32 }

```

Обратное преобразование После того, как мы посчитали требуемые значения и попарно умножили значения первого многочлена на значения второго, нужно сделать обратное преобразование. Можно заметить, что все действия, которые мы совершали при прямом преобразовании были обратимы и можно просто проделывать обратные операции перед заходом в рекурсию.

Но есть ещё более простой способ. При вычислении мы фактически применяем матрицу к вектору:

$$\begin{pmatrix} w^0 & w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & w^3 & \dots & w^{-1} \\ w^0 & w^2 & w^4 & w^6 & \dots & w^{-2} \\ w^0 & w^3 & w^6 & w^9 & \dots & w^{-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{-1} & w^{-2} & w^{-3} & \dots & w^1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Рассмотрим сумму $\sum_{k=0}^{n-1} (w^i w^j)^k = \sum_{k=0}^{n-1} w^{(i+j)k}$. Любое число вида w^i удовлетворяет

$$w^n = 1 \implies 1 - w^n = (1 - w)(1 + w + w^2 + \dots + w^{n-1}) = 0$$

Значит,

$$\sum_{i=0}^{n-1} (w^i)^k = \begin{cases} n, & i = 0 \\ 0, & i \neq 0 \end{cases}$$

Поэтому означенная сумма равна n если $i + j = 0$ или 0 в противном случае. Отсюда следует, что

$$\frac{1}{n} \begin{pmatrix} w^0 & w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^{-1} & w^{-2} & w^{-3} & \dots & w^1 \\ w^0 & w^{-2} & w^{-4} & w^{-6} & \dots & w^2 \\ w^0 & w^{-3} & w^{-6} & w^{-9} & \dots & w^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^1 & w^2 & w^3 & \dots & w^{-1} \end{pmatrix}$$

Это обратная матрицей к той, на которую мы умножаем при прямом преобразовании. Значит, при обратном преобразовании мы должны посчитать преобразование Фурье от w^{-1} и разделить на n .

```
1 vector<int> interpolate(vector<ftype> p) {
2     int n = p.size();
3     auto inv = fft(p, polar(1., -2 * pi / n));
4     vector<int> res(n);
5     for(int i = 0; i < n; i++) {
6         res[i] = round(real(inv[i]) / n);
7     }
8     return res;
9 }
```

Пройденный к этому моменту путь позволяет перемножить два числа за $O(n \log n)$.

Интерлюдия Приведённый выше код, являясь корректным и имея асимптотику $O(n \log n)$, едва ли пригоден для использования в контекстах. Он имеет большую константу и далеко не так численно устойчивый, чем оптимальные варианты написания быстрого преобразования Фурье. Он такой, какой он есть, так как без всяких окольных путей делает именно то, что написано и лучше всего подходит для иллюстрации.

Перед переходом к следующей части читателю рекомендуется самостоятельно задуматься о том, как можно улучшить время работы и точность вычислений. Из наиболее важного здесь – внутри преобразования не должно происходить выделения памяти, работать желательно с указателями, а не векторами, а корни из единицы должны быть посчитаны наперёд. Также следует избавиться от операций взятия остатка по модулю. Также можно обратить внимание на то, что вместо вычисления преобразования с w^{-1} можно вычислить преобразование с w , а затем развернуть элементы массива со второго по последний. [Здесь](#) приведена одна из условно пригодных реализаций.

2 Применения и вариации преобразования

Свёртки и корреляции Пусть есть $\{a_i\}_{i=0}^n$ и $\{b_j\}_{j=0}^m$. Тогда свёрткой называют $\{c_k\}_{k=0}^{m+n}$:

$$c_k = \sum_{i=0}^n a_i b_{k-i}$$

Как мы видим, это просто k -ый коэффициент из произведения. Корреляцией же называют $\{d_k\}_{-n}^m$:

$$d_k = \sum_{i=0}^n a_i b_{k+i}$$

В обоих случаях мы предполагаем, что вне допустимых индексов последовательности равны нулям. Корреляцию можно интерпретировать двумя способами. С одной стороны, это коэффициент в произведении $A(x) \cdot B(x^{-1})$, т.е. сдвинутая свёртка первой последовательности и развёрнутой второй. С другой стороны, d_k – в точности скалярное произведение последовательности a_i и отрезка последовательности b_j , начинающегося в позиции k . Именно свёртка и корреляция являются теми величинами, которые чаще всего нужно считать в задачах на преобразование Фурье.

Преобразование в кольце по модулю Как было сказано выше, помимо комплексных корней из единицы, можно рассматривать корни из единицы в каком-нибудь поле. В данном случае нас интересуют поля остатков по модулю простых чисел. Известно, что в любом таком поле есть образующий элемент – такое число, что его степени пробегают все элементы, кроме нуля.

Значит, для любого простого p в поле остатков от деления на него есть корень g степени $p-1$ из единицы. Если при этом $(p-1) = c \cdot 2^k$, то g^c будет корнем степени 2^k , что позволяет применять метод Кули-Тьюки. Отсюда следует, что $p = c \cdot 2^k + 1$. Практика показывает, что чисел такого вида очень много.

Chirp Z-transform Пусть нам дано некоторое число z и мы хотим вычислить значение многочлена в числах вида $\{z^i\}_{i=0}^{n-1}$, т.е. множество чисел $y_k = \sum_{i=0}^{n-1} a_i z^{ik}$. Для этого сделаем замену $ik = \frac{i^2 + k^2 - (i-k)^2}{2}$, после которой получим, что нам нужно вычислить

$$y_k = z^{\frac{k^2}{2}} \sum_{i=0}^{n-1} \left(a_i z^{\frac{i^2}{2}} \right) z^{-\frac{(i-k)^2}{2}}$$

Что с точностью до множителя $z^{\frac{k^2}{2}}$ является свёрткой двух последовательностей

$$u_i = a_i z^{\frac{i^2}{2}}, \quad v_i = z^{-\frac{i^2}{2}}$$

Которая считается через произведение многочленов с такими коэффициентами. Но следует учесть, что здесь v_i определена также для отрицательных номеров. Данный метод среди прочего позволяет за $O(n \log n)$ посчитать преобразование Фурье произвольной длины.

Одновременное преобразование вещественных многочленов Пусть есть два многочлена

$$A(x) = \sum_{i=0}^{n-1} a_i x^i, \quad B(x) = \sum_{i=0}^{n-1} b_i x^i$$

С вещественными коэффициентами. Рассмотрим $P(x) = A(x) + iB(x)$ и сопряжённый к нему.

$$\overline{P(w^k)} = A(\overline{w^k}) - iB(\overline{w^k}) = A(w^{n-k}) - iB(w^{n-k})$$

Отсюда следует выражение для преобразования Фурье $A(x)$ и $B(x)$:

$$\begin{cases} A(w^k) = \frac{P(w^k) + \overline{P(w^{n-k})}}{2}, \\ B(w^k) = \frac{P(w^k) - \overline{P(w^{n-k})}}{2i} \end{cases}$$

Одновременное преобразование можно произвести и в обратную сторону, рассматривая последовательность $P(w^k) = A(w^k) + iB(w^k)$. После обратного преобразования мы получим $P(x) = A(x) + iB(x)$.

Умножение по произвольному модулю Нам нужно перемножить два многочлена, а затем вывести коэффициенты результата по модулю M , не являющимся подходящим для быстрого преобразования Фурье. При этом достаточно большому, чтобы обычному преобразованию не хватало точности. Для разрешения данной ситуации представим многочлены в виде

$$A(x) = A_1(x) + A_2(x) \cdot 2^k$$

$$B(x) = B_1(x) + B_2(x) \cdot 2^k$$

где $2^k \approx \sqrt{M}$. Тогда все коэффициенты будут $O(\sqrt{M})$, а произведение разложится как

$$A \cdot B = A_1 B_1 + (A_1 B_2 + A_2 B_1) \cdot 2^k + A_2 B_2 \cdot 2^{2k}$$

Такое представление позволяет нам уменьшить вдвое длину чисел, с которыми работаем, при этом, с учётом прошлого пункта, можно обойтись двумя прямыми и двумя обратными вызовами преобразования. Альтернативный подход, умножение по нескольким модулям, а затем использование китайской теоремы об остатках, является более сложным в написании, но при этом работает медленнее.

Многомерное преобразование Фурье Ранее мы работали с многочленами от одной переменной. Но аналогичные конструкции работают для многочлена от нескольких переменных. Считать значения многочлена теперь нужно в точках $(w_1^{k_1}, w_2^{k_2}, \dots, w_m^{k_m})$. Оказывается, для такого преобразования достаточно поочерёдно сделать одномерное преобразование Фурье вдоль каждой координаты. В двумерном случае, например, нужно сначала сделать одномерное преобразование каждой строки, а затем каждого столбца.

Докажем это для двумерного случая. Мы хотим получить набор чисел

$$P_{uv} = P(w_1^u, w_2^v) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij} w_1^u w_2^v$$

Изначально мы имеем таблицу $A_{uv} = a_{uv}$, после преобразования строк, мы получим

$$A'_{uv} = P_u(w_2^v) = \sum_{j=0}^{m-1} A_{uj} w_2^v = \sum_{j=0}^{m-1} a_{uj} w_2^v$$

После последующего преобразования столбцов же мы получим

$$A''_{uv} = P'_v(w_1^u) = \sum_{i=0}^{n-1} A'_{iv} w_1^u = \sum_{i=0}^{n-1} P_i(w_2^v) w_1^u = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_1^u w_2^v$$

Такое преобразование позволяет быстро вычислять двумерные свёртки $C(x, y) = A(x, y) \cdot B(x, y)$ вида

$$c_{uv} = \sum_{\substack{i_1+j_1=u \\ i_2+j_2=v}} \sum a_{i_1 i_2} b_{j_1 j_2}$$

Преобразование Уолша-Адамара и другие свёртки Вычисляя значения многомерного многочлена в некоторых особых точках, мы можем научиться считать свёртки с другими условиями суммирования:

$$c_k = \sum_{i|j=k} a_i b_j, \quad c_k = \sum_{i \oplus j=k} a_i b_j, \quad c_k = \sum_{i \& j=k} a_i b_j$$

Здесь $|$, $\&$ и \oplus соответствуют операциям побитового *or*, *and* и *xor* соответственно.

1. *xor*. Рассмотрим значения многочлена в точках гиперкуба $x \in \{-1, 1\}^k$. Для таких точек верно соотношение $x_i^a x_i^b = x_i^{a \oplus b}$, поэтому произведения значений многочленов в этих точках будут равны значениям многочлена, в котором мономы умножаются с учётом данного условия.

Иначе говоря, если рассматривать степень x_i в мономе, как i -ый бит номера данного коэффициента, мы можем считать, что при произведении двух мономов мы получаем моном, чьему номеру соответствует *xor* номеров исходных мономов.

Заметим, что такое вычисление есть ни что иное как вычисление многомерного преобразования Фурье в корнях степени 2 из единицы. Оно также называется преобразованием Уолша-Адамара. Здесь есть некоторое упрощение по сравнению с обычным преобразованием Фурье: во-первых, все вычисления можно производить в целых числах, во-вторых, $w^{-1} = w = -1$, поэтому для обратного преобразования можно просто применить прямое и разделить всё на n .

```

1 void transform(int *from, int *to) {
2     if(to - from == 1) {
3         return;
4     }
5     int *mid = from + (to - from) / 2;
6     transform(from, mid);
7     transform(mid, to);
8     for(int i = 0; i < mid - from; i++) {
9         int a = *(from + i);
10        int b = *(mid + i);
11        *(from + i) = a + b;
12        *(mid + i) = a - b;
13    }
14 }

```

2. *or*. Теперь рассмотрим значения в точках $x \in \{0, 1\}^k$. Для них имеет место $x_i^a x_i^b = x_i^{a \text{ or } b}$, из чего следует, что при произведении мономов можно трактовать результат как моном с номером, равным побитовому *or* их номеров. Отдельно заметим, что посчитанное значение многочлена в точке это сумма его коэффициентов по всем подмаскам номера данной точки.

```

1 void transform(int *from, int *to) {
2     if(to - from == 1) {
3         return;
4     }
5     int *mid = from + (to - from) / 2;
6     transform(from, mid);
7     transform(mid, to);
8     for(int i = 0; i < mid - from; i++) {
9         *(mid + i) += *(from + i);
10    }
11 }
12
13 void inverse(int *from, int *to) {
14     if(to - from == 1) {
15         return;
16     }
17     int *mid = from + (to - from) / 2;
18     inverse(from, mid);
19     inverse(mid, to);
20     for(int i = 0; i < mid - from; i++) {
21         *(mid + i) -= *(from + i);
22     }
23 }

```

3. *and*. Чтобы посчитать свёртку по данной операции, нужно либо поменять все маски на их дополнения, посчитать свёртку по *or*, а потом вернуться, либо воспользоваться идеей из прошлого пункта и провести суммирование по всем надмаскам. Это будет соответствовать значению многочлена в тех же точках, но с неявной перенумерацией, соответствующей переходу к дополнениям.

Заметим, что данные идеи обобщаются на случай когда числа представлены в системе с основанием, отличным от двух и нам нужно совершить свёртку относительно поразрядных операций сложения по модулю основания, максимума или минимума.

Метод Ньютона для функций над многочленами Хотим решить уравнение $f(x) = 0$. $f(x)$ можно представить в виде $f(x) = f(x_0) + f'(x_0)\Delta x + O(\Delta x^2)$. Будем последовательно искать её нули, приближая линейной $g(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$ на каждом шаге. Решая $g(x_{n+1}) = 0$, приходим к

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

При этом $f(x_{n+1}) = O((x_{n+1} - x_n)^2) = O\left(\frac{f(x_n)^2}{f'(x_n)^2}\right)$. В случае обратимой производной это $O(f(x_n)^2)$.

Если x — многочлен, это значит, что используя метод Ньютона, мы будем на каждом шаге удваивать число точно известных коэффициентов. Наиболее распространённые функции от многочленов:

1. Обратный ряд. Надо решить $PQ = 1 \Rightarrow f(P) = Q - P^{-1}$ и $P_{n+1} = P_n - \frac{Q - P_n^{-1}}{P_n^{-2}} = P_n(2 - QP_n)$.
2. Экспонента. $Q = \ln P \Rightarrow f(P) = Q - \ln P$ и $P_{n+1} = P_n - \frac{Q - \ln P_n}{-P_n^{-1}} = P_n(1 + Q - \ln P_n)$.

3. Корень. $Q = P^k \Rightarrow f(P) = Q - P^k$ и $P_{n+1} = P_n + \frac{Q - P_n^k}{kP_n^{k-1}} = P_n \left(\frac{k-1}{k} + \frac{Q}{kP_n^k} \right)$.

В выражении для экспоненты есть логарифм, для его вычисления следует воспользоваться тем, что $(\log P)' = P'P^{-1}$, что позволит восстановить коэффициенты при положительных степенях, а коэффициент при нулевой степени можно посчитать встроенными методами. Отметим, что все указанные алгоритмы работают за $O(n \log n)$.

Разделяй и властвуй Дано уравнение $AX = B$ и мы хотим посчитать первые n коэффициентов X . При этом B нам дан не весь, а подаётся по мере того, как мы узнаём коэффициенты X (например, B может зависеть от X , см. пример ниже). Пусть мы нашли первые $m \approx n/2$ коэффициентов и хотим найти следующие m , тогда

$$\begin{cases} X = X_1 + x^m X_2, \\ A = A_1 + x^m A_2, \\ B = B_1 + x^m B_2, \\ AX_1 = B_1 + x^m B'_2 \end{cases} \implies AX \bmod x^{2m} = B_1 + x^m(A_1 X_2 + B'_2) = B_1 + x^m B_2$$

$$A_1 X_2 = B_2 - B'_2 \bmod x^m$$

Значит, если запомнить B'_2 , который потом нужно будет вычесть из B_2 , можно свести вычисление оставшихся коэффициентов к этой же задаче размера $\approx n/2$. Что даст алгоритм за $O(n \log^2 n)$.

В качестве примера, этим методом также можно считать экспоненту, так как

$$e^A = X \implies X' = A'X$$

На практике это проще, чем по схеме Ньютона и скрытая константа должна быть меньше, т.к. не нужно вызывать такие затратные функции как логарифм, который требует подсчёта обратного ряда. Заметим, что аналогичная схема может быть применена для подсчёта $X = AB$, где A задан заранее, а B зависит от X .

$$AB_1 = X_1 + x^m X'_2 \implies AB \bmod x^{2m} = X_1 + x^m(A_1 B_2 + X'_2)$$

$$X_2 = A_1 B_2 + X'_2 \bmod x^m$$

Деление и интерполяция В завершение научимся делить многочлены с остатком, а также делать то, с чего всё началось – интерполировать многочлен и вычислять его на произвольных точках.

1. Деление с остатком. Нам нужно представить $A(x) = B(x)D(x) + R(x)$, $\deg R(x) < \deg B(x)$. Пусть $\deg A = n$, $\deg B = m$. Тогда $\deg D = n - m$. При этом с учётом $\deg R < m$ приходим к выводу, что коэффициенты при $\{x^k\}_{k=m}^n$ не зависят от $R(x)$. Получается, мы имеем систему из $n - m + 1$ линейных уравнений на $n - m + 1$ неизвестных (коэффициенты D).

Рассмотрим $A^r(x) = x^n A(x^{-1})$, $B^r(x) = x^m B(x^{-1})$, $D^r(x) = x^{n-m} D(x^{-1})$ – многочлены, в которых коэффициенты идут в обратном порядке. Для $n - m + 1$ старших коэффициентов исходных многочленов с учётом того, что $P(x) \bmod z^k$ – первые k коэффициентов $P(x)$ имеем систему

$$A^r(x) = B^r(x)D^r(x) \bmod z^{n-m+1}$$

Её решением будет $D^r(x) = A^r(x)[B^r(x)]^{-1} \bmod z^{n-m+1}$, что позволяет найти $D(x)$ и из него $R(x)$.

2. Многоточечное вычисление. Нужно вычислить $P(x_i)$ для $\{x_i\}_{i=1}^n$. Учитывая $P(x_i) = P \bmod (x - x_i)$, вычислим $P \bmod \prod_{i=1}^{n/2-1} (x - x_i)$ и $P \bmod \prod_{i=n/2}^n (x - x_i)$ и запустимся рекурсивно. Получим $O(n \log^2 n)$.
3. Интерполяция. Дан набор $\{(x_i, y_i)\}_{i=0}^{n-1}$, нужно найти $P : P(x_i) = y_i$. Пусть мы нашли многочлен P_1 для первых $n/2$ точек. Тогда $P = P_1 + P_2 \prod_{i=0}^{n/2-1} (x - x_i) = P_1 + P_2 Q$. Нахождение P_2 сведём к интерполяции и многоточечному вычислению: $P_2(x_i) = \frac{y_i - P_1(x_i)}{Q(x_i)}$ для $i > n/2$. Получим $O(n \log^3 n)$.

3 Упражнения

Рюкзак Есть n типов предметов. Предмет i -го типа имеет стоимость s_i . Пусть $s = \sum_{i=1}^n s_i$. Предложите алгоритм, который для каждого $w \leq s$ находит число способов выбрать подмножество предметов ровно с таким весом за $O(s \log s \log n)$.

Степенной ряд Даны числа k и n . Найдите $\sum_{m=0}^n m^k$ за $O(k \log k)$.

Общая схема Кули-Тьюки Пусть $n = pq$. Придумайте алгоритм, сводящий преобразование Фурье размера n к p преобразованиям Фурье размера q за $O(n)$ дополнительных операций.

Арифметические прогрессии Дано множество из n чисел от 0 до m . Найдите число арифметических прогрессий длины 3 в этом множестве за $O(m \log m)$.

Расстояние между точками Даны n точек в прямоугольнике $A \times B$. Для каждой возможной пары $(\Delta x, \Delta y)$ посчитайте сколько есть пар точек, таких что разность по x -координате между ними равна Δx , а по y -координате соответственно Δy за $O(AB \log AB)$.

Сопоставление шаблонов Даны две строки s и t . В них могут встречаться символы из множества Σ , а также знаки вопроса. Найдите все позиции i такие, что если приложить строку t к строке s начиная с i , то в любой позиции соответствующие символы в s и t должны либо совпадать, либо хотя бы один из них должен быть знаком вопроса за $O(\Sigma n \log n)$ и за $O(n \log n)^*$.

Линейные рекурренты* Последовательность F_n задана как $F_n = \sum_{i=1}^k a_{k-i} F_{n-i}$. Даны коэффициенты $\{a_i\}_{i=0}^{k-1}$ и начальные величины $\{F_i\}_{i=0}^{k-1}$. Предложите алгоритм, вычисляющий F_n за $O(k \log k \log n)$.

Степень многочлена* Дан $P(x) = \sum_{i=0}^n a_i x^i$. Нужно найти первые n коэффициентов $P^k(x)$ за $O(n \log n)$.