СПб ВШЭ, 3-й курс, весна 2025
Конспект лекций по алгоритмам

Собрано 3 марта 2025 г. в 11:42

Содержание

Лекция #1: FFT и его друзья

1-я пара, весна 2025

1.1. FFT

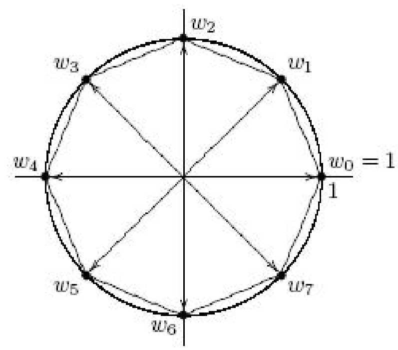
1.1.1. Прелюдия

Пусть есть многочлены $A(x) = \sum a_i x^i$ и $B(x) = \sum b_i x^i$.
Посчитаем их значения в точках $x_1, x_2, \ldots, x_n$: $A(x_i) = fa_i, B(x_i) = fb_i$.
Значения $C(x) = A(x)B(x)$ в точках x_i можно получить за линейное время:

$$fc_i = C(x_i) = A(x_i)B(x_i) = fa_i fb_i$$

Схема быстрого умножения многочленов:

$$a_i, b_i \xrightarrow{\mathcal{O}(n \log n)} fa_i, fb_i \xrightarrow{\mathcal{O}(n)} fc_i = fa_i fb_i \xrightarrow{\mathcal{O}(n \log n)} c_i$$



Осталось подобрать правильные точки x_i.

FFT расшифровывается Fast Fourier Transform и за $\mathcal{O}(n \log n)$ вычисляет значения многочлена в комплексных точках $w_j = e^{\frac{2\pi i j}{n}}$ для $n = 2^k$ (то есть, только для степеней двойки).

Что нужно помнить про комплексные числа?
При умножении комплексных чисел углы складываются, длины перемножаются.
В частности, если обозначить $w = e^{\frac{2\pi i}{n}} = \cos\frac{2\pi i}{n} + i \sin\frac{2\pi i}{n}$, то $w_j = w^j$ (все корни из единицы – это степени главного корня, и они образуют циклическую группу). Также $w^{-j} = w^{n-j}$.

1.1.2. Собственно FFT

$A(x) = \sum a_i x^i = (a_0 + x^2 a_2 + x^4 a_4 + \ldots) + x(a_1 x + a_3 x^3 + a_5 x^5 + \ldots) = B(x^2) + xC(x^2)$ – обозначили все чётные коэффициенты многочлена A многочленом B, а нечётные соответственно C.

Посчитаем рекурсивно $B(w_j)$ и $C(w_j)$, зная их, за $\mathcal{O}(n)$ посчитаем $A(w_j) = B(w_j) + w_j C(w_j)$.

Заметим, что $\forall j\ w_j = w_{j \bmod n} \Rightarrow \forall j\ w_j^2 = w_{n/2+j}^2 \Rightarrow B$ и C нужно считать только в $\frac{n}{2}$ точках.

Итого алгоритм:

```
def FFT(a):
    n = len(a)
    if n == 1: return a[0] # посчитать значение многочлена A(x) ≡ a[0] в точке x = 1
    for j=0..n-1: (j%2 ? c : b)[j / 2] = a[j]
    b, c = FFT(b), FFT(c) # самое важное - две ветки рекурсии
    for j=0..n-1: a[j] = b[j % (n/2)] + exp(2πi*j/n) * c[j % (n/2)]
    return a
```

Время работы $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

1.1.3. Крутая нерекурсивная реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все $w_j = e^{\frac{2\pi ij}{n}}$.

Заметим, что b и c можно хранить прямо в массиве a. Тогда получается, что на прямом ходу рекурсии мы просто переставляем местами элементы a, на обратном ходу рекурсии делаем какие-то полезные действия. Число a_i перейдёт на позицию $a_{rev(i)}$, где $rev(i)$ – перевёрнутая битовая запись i. Кстати, $rev(i)$ мы уже умеем считать динамикой для всех i.

При реализации на C++ можно использовать стандартные комплексные числа `complex<double>`, но свои рукописные будут работать немного быстрее.

```
const int K = 20, N = 1 << K; // N - ограничение на длину результата умножения многочленов
complex<double> root[N];
int rev[N];
void init():
    for (int j = 0; j < N; j++):
        root[j] = exp(2πi*j/N); // cos(2πj/N), sin(2πj/N)
        rev[j] = rev[j >> 1] + ((j & 1) << (K - 1));
```

Теперь, корни из единицы степени k хранятся в `root[j*N/k]`, $j \in [0, k)$.
Доступ к памяти при этом не последовательный, проблемы с кешом.
Чтобы посчитать все корни, мы $2N$ раз вычисляли тригонометрические функции.

• **Улучшенная версия вычисления корней**

```
for (int k = 1; k < N; k *= 2):
    num tmp = exp(πi/k);
    root[k] = {1, 0}; // в root[k..2k) хранятся первые k корней степени 2k
    for (int i = 1; i < k; i++)
        root[k+i] = (i&1) ? root[(k+i) >> 1] * tmp : root[(k+i) >> 1];
```

Теперь код собственно преобразования Фурье может выглядеть так:

```
FFT(a): // a → f = FFT(a)
    vector<complex> f(N);
    for (int i = 0; i < N; i++) // прямой ход рекурсии превратился в один for =)
        f[rev[i]] = a[i];
    for (int k = 1; k < N; k *= 2) // пусть уже посчитаны FFT от кусков длины k
        for (int i = 0; i < N; i += 2 * k) // [i..i+k) [i+k..i+2k) → [i..i+2k)
            for (int j = 0; j < k; j++): // оптимально написанный главный цикл FFT
                num tmp = root[k + j] * f[i + j + k]; // root[] из «улучшенной версии»
                f[i + j + k] = f[i + j] - tmp; // w_{j+k} = −w_j при n = 2k
                f[i + j] = f[i + j] + tmp;
    return f;
```

1.1.4. Обратное преобразование

Обозначим $w = e^{2\pi i/n}$. Нам нужно из
$f_0 = a_0 + a_1 + a_2 + a_3 + \ldots$
$f_1 = a_0 + a_1 w + a_2 w^2 + a_3 w^3 + \ldots$
$f_2 = a_0 + a_1 w^2 + a_2 w^4 + a_3 w^3 + \ldots$

научиться восстанавливать коэффициенты $a_0, a_1, a_2, \ldots$

Заметим, что $\forall j \neq 0 \ \sum\limits_{k=0}^{n-1} w^{jk} = 0$ (сумма геометрической прогрессии).
И напротив при $j = 0$ получаем $\sum\limits_{k=0}^{n-1} w^{jk} = \sum 1 = n$.
Поэтому $f_0 + f_1 + f_2 + \dots = a_0 n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = \boxed{a_0 n}$
Аналогично $f_0 + f_1 w^{-1} + f_2 w^{-2} + \dots = \sum_k a_0 w^{-k} + a_1 n + a_2 \sum_k w^k + \dots = \boxed{a_1 n}$
И в общем случае $\sum_k f_k w^{-jk} = \boxed{a_j n}$
Рассмотрим $F(x) = f_0 + x f_1, x^2 f_2 + \ldots \Rightarrow F(w^{-j}) = a_j n$, похоже на `FFT(f)`.
Осталось заметить, что множества чисел $w^{-j} = w^{n-j} \Rightarrow$

```
FFT_inverse(f): // f → a
    a = FFT(f)
    reverse(a + 1, a + N) // w^j ↔ w^{-j}
    for (int i = 0; i < N; i++) a[i] /= N;
    return a;
```

1.1.5. Два в одном

Часто коэффициенты многочленов – вещественные или даже целые числа.
Если у нас есть многочлены $A(x), B(x) \in \mathbb{R}[x]$, возьмём числа $c_j = a_j + ib_j$, коэффициенты $C(x) = A(x) + iB(x)$, посчитаем $fc = FFT(c)$. Тогда по f за $\mathcal{O}(n)$ можно восстановить fa и fb.

Для этого вспомним про сопряжения комплексных чисел:
$\overline{x+iy} = x - iy$, $\overline{u \cdot v} = \overline{u} \cdot \overline{v}$, $w^{n-j} = w^{-j} = \overline{w^j} \ \Rightarrow \ \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C}(w^j) = A(w^j) - iB(w^j) \ \Rightarrow$ $fc_j + \overline{fc_{n-j}} = 2A(w^j) = 2 \cdot fa_j$. Аналогично $fc_j - \overline{fc_{n-j}} = 2B(w^j) = 2i \cdot fb_j$.

Итого для умножения двух многочленов можно использовать $\boxed{\text{не 3 вызова FFT, а 2}}$.

1.1.6. Умножение чисел, оценка погрешности

Число длины n в системе счисления $10 \to$ система счисления $10^k \to$ многочлен длины n/k.
Умножения многочленов такой длины будет работать за $\frac{n}{k} \log \frac{n}{k}$.
Отсюда возникает вопрос, какое максимальное k можно использовать?
Коэффициенты многочлена-произведения будут целыми числами до $(10^k)^2 \cdot \frac{n}{k}$.
Чтобы в типе `double` целое число хранилось с погрешностью меньше 0.5 (тогда мы его сможем правильно округлить к целому), оно должно быть не более 10^{15}.
Получаем при $n \leqslant 10^6$, что $(10^k)^2 \cdot 10^6/k \leqslant 10^{15} \Rightarrow k \leqslant 4$.
Аналогично для типа `long double` имеем $(10^k)^2 \cdot 10^6/k \leqslant 10^{18} \Rightarrow k \leqslant 6$.
Это оценка сверху, предполагающая, что само FFT погрешность не накапливает... на самом деле эта оценка очень близка к точной.

1.2. Разделяй и властвуй

1.2.1. Перевод между системами счисления

Задача: перевести число X длины $n=2^k$ из a-ичной системы счисления в b-ичную.

Разобьём число X на $\frac{n}{2}$ старших цифр и $\frac{n}{2}$ младших цифр: $X = X_0 \cdot a^{n/2} + X_1 \Rightarrow$

$$F(X) = F(X_0)F(a^{n/2}) + F(X_1)$$

Умножение за $\mathcal{O}(n \log n)$ и сложение за $\mathcal{O}(n)$ выполняются в системе счисления b.
Предподсчёт $F(a^1), F(a^2), F(a^4), F(a^8), \dots, F(a^n)$ займёт $\sum_k \mathcal{O}(2^k k) = \mathcal{O}(n \log n)$ времени.
Итого $T(n) = 2T(n/2) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$.

1.2.2. Деление многочленов с остатком

Задача: даны $A(x), B(x) \in \mathbb{R}[x]$, найти $Q(x), R(x)\colon\ \deg R < \deg B \ \wedge\ A(x) = B(x)Q(x) + R(x)$.

Зная Q мы легко найдём R, как $A(x) - B(x)Q(x)$ за $\mathcal{O}(n \log n)$. Сосредоточимся на поиске Q.
Пусть $\deg A = \deg B = n$, тогда $Q(x) = \frac{a_n}{b_n}$. То есть, $Q(x)$ можно найти за $\mathcal{O}(1)$.
Из этого мы делаем вывод, что Q зависит не обязательно от всех коэффициентов A и B.

<u>Lm</u> 1.2.1. $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$, и Q зависит только от $m{-}n{+}1$ коэффициентов A и $m{-}n{+}1$ коэффициентов B.

Доказательство. У A и $B \cdot Q$ должны совпадать $m{-}n{+}1$ старший коэффициент ($\deg R < n$). В этом сравнении участвуют только $m{-}n{+}1$ старших коэффициентов A. При домножение B на $x^{\deg Q}$, сравнятся как раз $m{-}n{+}1$ старших коэффициентов A и B. При домножении B на меньшие степени x, в сравнении будут участвовать лишь какие-то первые из этих $m{-}n{+}1$ коэффициентов. ■

Теперь будем решать задачу: даны n старших коэффициентов A и B, найти такой C из n коэффициентов, что у A и BC совпадает n старших коэффициентов. Давайте считать, что младшие коэффициенты лежат в первых ячейках массива.

```
Div(int n, int *A, int *B)
    C = Div(n/2, A + n/2, B + n/2) // нашли старших n/2 коэффициентов ответа
    A' = Subtract(n, A, n + n/2 - 1, Multiply(C, B))
    D = Div(n/2, A', B + n/2) // сейчас A' состоит из n/2 не нулей и n/2 нулей
    return concatenate(D, C) // склеили массивы коэффициентов
```

Здесь `Subtract` – хитрая функция. Она знает длины многочленов, которые ей передали, и сдвигает вычитаемый многочлен так, чтобы старшие коэффициенты совместились.

1.2.3. Вычисление значений в произвольных точках

Задача. Дан многочлен $A(x)$, $\deg A = n$ и точки $x_1, x_2, \dots x_n$. Найти $A(x_1), A(x_2), \dots A(x_n)$.

Вспомним теорему Безу: $A(w) = A(x) \bmod (x - w)$.
Обобщение: $B(x) = A(x) \bmod (x - x_1)(x - x_2)\dots(x - x_n) \ \Rightarrow\ \forall j\ B(x_j) = A(x_j)$

```
def Evaluate(n, A, x[]): # n = 2^k
    if n == 1: return list(A[0])
    return Evaluate(n/2, A mod (x-x_1)·...·(x-x_{n/2}), [x_1, ... , x_{n/2}]) +
        Evaluate(n/2, A mod (x-x_{n/2+1})·...·(x-x_n), [x_{n/2+1}, ... , x_n])
```

Итого $T(n) = 2T(n/2) + \mathcal{O}(\mathtt{div}(n))$. Если деление реализовано за $\mathcal{O}(n \log n)$, получим $\mathcal{O}(n \log^2 n)$.

1.2.4. Интерполяция

Задача. Даны пары $(x_1, y_1), \dots, (x_n, y_n)$. Найти многочлен A: $\deg A = n-1$, $\forall i\ A(x_i) = y_i$.

Сделаем интерполяцию по Ньютону методом разделяй и властвуй.
Сперва найдём интерполяционный многочлен B для $(x_1, y_1), (x_2, y_2), \dots, (x_{n/2}, y_{n/2})$.

$$A = B + C \cdot D, \text{ где } D = \prod_{j=1..\frac{n}{2}} (x - x_j), \text{ а } C \text{ нужно найти}$$

Подгоним правильные значения в точках $x_{n/2+1}, \dots, x_n$, вычислим b_j, d_j – значения B и D в точках $x_{n/2+1}, \dots, x_n \Rightarrow C$ – интерполяционный многочлен точек $(x_j, -\frac{b_j}{d_j})$ при $j = \frac{n}{2}+1 \dots n$.

Итого $T(n) = 2T(n/2) + 2\mathcal{O}(\mathtt{evaluate}(n/2))$. При $\mathtt{evaluate}(n) = \mathcal{O}(n \log^2 n)$ имеем $\mathcal{O}(n \log^3 n)$.

1.2.5. Извлечение корня

Дан многочлен $A(x)$: $\deg A \equiv 0 \bmod 2$. Задача – найти $R(x)$: $\deg(A - R^2)$ минимальна.
Пусть мы уже нашли старшие k коэффициентов R, обозначим их R_k. Найдём $2k$ коэфф-тов: $R_{2k} = R_k x^k + X, R_{2k}^2 = R_k^2 x^{2k} + 2R_k X \cdot x^k + X^2$. Правильно подобрав X, мы можем “обнулить” k коэффициентов $A - R_{2k}^2$, для этого возьмём $X = (A - R_k^2 x^{2k})/(2R_k)$. В этом частном нам интересны только k старших коэффициентов, поэтому переход от R_k к R_{2k} происходит за $\mathcal{O}(\mathtt{mul}(k) + \mathtt{div}(k))$. Итого суммарное время на извлечение корня – $\mathcal{O}(\mathtt{div}(n))$.

1.3. Литература

[sankowski]. Слайды по FFT и всем идеям разделяй и властвуй.
[e-maxx]. Про FFT и оптимизации к нему.
[codeforces]. Задачи на тему FFT.
[vk]. Краткий конспект похожих идей от Александра Кулькова.

Лекция #1: Действия над многочленами

1-я пара, весна 2025

1.4. Над $\mathbb{F}_2$

Умножение/деление/gcd можно делать битовым сжатием за $\approx \frac{nm}{w}$, где w – word size.

- **Хранение**

$A(x) = a_0 + a_1x + \dots + a_nx^n \;\to$ `N = n + 1; bitset<N> a`
Обобзначения: $n = \deg A$, $m = \deg B$, $N = \deg A + 1$, $M = \deg B + 1$.
Многочлен степени n = массив коэффициентов длины N.

- **Умножение**

```
for i=0..n
    if a[i]
        c ^= b << i
```

Время работы: $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$. Можно n заменить на «*число не нулей в* `a`».

- **Деление**

```
for i=n..m
    if a[i]
        a ^= b << (i-m), c[i-m] = 1
```

Результат: в «`a`» лежит остаток, в «`c`» частное.
Время работы: $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil)$.

- **gcd**

Запускаем Евклида. Один шаг Евклида – деление. Деление работает за $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil)$ и уменьшает n на $n-m \Rightarrow$ суммарно все деления отработают за $\mathcal{O}((n-m) \cdot \lceil \frac{m}{w} \rceil + \frac{m^2}{w}) = \mathcal{O}(\frac{nm}{w})$.

1.5. Умножение многочленов

Над произвольным кольцом умеем за $\mathcal{O}(nm)$.
Точнее за «(*число не нулей в* `a`) $\cdot$ (*число не нулей в* `b`)».

- **Карацуба**

Пусть $N = 2^k$. Если нет, дополним оба массива нулями (нулевые старшие коэффициенты).
Делим многочлены на две части: $A = A_0 + x^{N/2}A_1,\; B = B_0 + x^{N/2}B_1$

$$A \cdot B = A_0B_0 + x^NA_1B_1 + x^{N/2}(A_0B_1 + A_1B_0) = C_0 + x^NC_1 + x^{N/2}((A_0{+}B_0)(A_1{+}B_1) - C_0 - C_1)$$

Умножение многочленов длины N – сложение, вычитание и 3 умножения многочленов длины $\frac{N}{2}$.

$$T(n) = 3T(\tfrac{n}{2}) + n = \boxed{\Theta(n^{\log 3})}$$

Такой способ умножения работает над произвольным кольцом.

- **Оптимальное над $\mathbb{F}_2$**

У нас уже есть Карацуба и битовое сжатие. Соединим. Внутри Карацубы реализуем сложение, вычитание и разделение многочлена на две части за $\lceil \frac{n}{w} \rceil$. Казалось бы мы ускорили всё в w раз, но нет, время работы равно числу листьев рекурсии.

$$T(n) = 3T(\tfrac{n}{2}) + \lceil \tfrac{n}{w} \rceil = \Theta(n^{\log 3})$$

Асимптотическая оптимизация: в рекурсии при $N \leqslant w$ будем умножать за $\mathcal{O}(n)$. Улучшили $w^{\log 3}$ до $w \Rightarrow$ новое время работы в $w^{(\log 3)-1}$ раз меньше.

• Над $\mathbb{Z}$, над $\mathbb{R}$, над $\mathbb{C}$

Фурье за $\mathcal{O}(n \log n)$. Смотри главу про Фурье (FFT).

• Над конечным полем

Все конечные поля изоморфны $\Rightarrow$ умножить над $\mathbb{F}_p \Leftrightarrow$ умножить над $\mathbb{Z}/p\mathbb{Z}$.
Умножим в $\mathbb{Z}$ (Карацуба или FFT), затем возьмём по модулю.
Для некоторых p, например $p = 3 \cdot 2^{18} + 1$, можно напрямую применить «Фурье по модулю».

Лекция #2: Деление многочленов

2-я пара, весна 2025

2.1. Быстрое деление многочленов

Цель – научиться делить многочлены за $\mathcal{O}(n \log n)$.
Очень хочется считать частное многочленов $A(x)/B(x)$, как $A(x)B^{-1}(x)$. К сожалению, у многочленов нет обратных. Зато обратные есть у рядов, научимся сперва искать их.

• Обращение ряда

Задача. Дан ряд $A \in [[\mathbb{R}]],\ a_0 \neq 0$. Найти ряд $B\colon A(x)B(x) = 1$.
Первые n коэффициентов B можно найти за $\mathcal{O}(n^2)$:
$b_0 = 1/a_0$
$b_1 = -(a_1 b_0)/a_0$
$b_2 = -(a_2 b_0 + a_1 b_1)/a_0$
...
А можно за $\mathcal{O}(n \log n)$.
Обозначим $B_k(x) = b_0 + b_1 x + \dots + b_{k-1}x^{k-1}$. Заметим, что $\forall k\ A(x)B_k(x) = 1 + x^k C_k(x)$.
$B_1 = b_0 = 1/a_0$. Научимся делать переход $B_k \to B_{2k}$ за $\mathcal{O}(k \log k)$.
$B_{2k} = B_k + x^k Z \Rightarrow A \cdot B_{2k} = 1 + x^k C_k + x^k A \cdot Z = 1 + x^k(C_k - A \cdot Z)$.
Выберем $Z = B_k \cdot C_k \Rightarrow C_k - A \cdot Z = C_k - C_k(A \cdot B_k) = C_k - C_k(1 + x^k C_k) = -x^k C_k^2$.
Итого $B_{2k} = B_k + B_k(x^k C_k) = B_k + B_k(1 - A \cdot B_k) \Rightarrow \boxed{B_{2k} = B_k(2 - A \cdot B_k)}$

Два умножения = $\mathcal{O}(k \log k)$. Общее время работы $n \log n + \frac{n}{2} \log \frac{n}{2} + \frac{n}{4} \log \frac{n}{4} + \dots = \mathcal{O}(n \log n)$.
Конечно, мы обрежем B_{2k}, оставив лишь $2k$ первых членов.

• Деление многочленов

A^R – reverse многочлена. $a_0 + a_1 x + \dots + a_n x^n \to a_n + a_{n-1} x + \dots + a_0 x^n$.
Умножение: $A^R B^R = (AB)^R$ (доказательство: в c_{ij} += $a_i b_j$ поменяли индексы на $n-i$ и $m-j$).
Новое определение деления: по A, B хотим $C\colon A^R \equiv (BC)^R \bmod x^n$.
Здесь n – число коэффициентов у A, у B ровно столько же.

Обращение ряда нам даёт умение по многочлену $Z\colon z_0 \neq 0$ строить $Z^{-1}\colon Z \cdot Z^{-1} \equiv 1 \bmod x^n$.

$$C^R = (B^R)^{-1} A^R$$

Время работы: обращение ряда + умножение = $\mathcal{O}(n \log n)$.
Над кольцом делить странно, а вот над произвольным полем Фурье может не работать, тогда деление работает за $\mathcal{O}(\texttt{mul}(n) + \texttt{mul}(\frac{n}{2}) + \dots) = \mathcal{O}(\texttt{mul}(n))$ для $\texttt{mul}(n) = \Omega(n)$.

2.2. (*) Быстрое деление чисел

Для нахождение частного чисел, достаточно научиться с большой точностью считать обратно.
Рассмотрим метод Ньютона поиска корня функции $f(x)$:
$x_0 =$ `достаточно точное приближение корня`
$x_{i+1} = x_i - f(x_i)/f'(x_i)$
Решим с помощью него уравнение $f(x) = x^{-1} - a = 0$.
$x_0 =$ `обратное к старшей цифре` a
$x_{i+1} = x_i - (\frac{1}{x_i} - a)/(-\frac{1}{x_i^2}) = x_i + (x_i - a \cdot x_i^2) = x_i(2 - ax_i)$.
Любопытно, что очень похожую формулу мы видели при обращении формального ряда...
Утверждение: каждый шаг метода Ньютона удваивает число точных знаков x.
Итого, имея x_i с k точными знаками, мы научились за $\mathcal{O}(k \log k)$ получать x_{i+1} с $2k$ точными знаками. Суммарное время получения n точных знаков $\mathcal{O}(n \log n)$.

2.3. (*) Быстрое извлечение корня для чисел

Продолжаем пользоваться методом Ньютона.

$$x_{i+1} = \tfrac{1}{2}(x_i + \tfrac{a}{x_i})$$

Если у x_i k_i верных знаков, то $k_{i+1} = k_i + \Theta(1)$, а
$x_i \to x_{i+1}$ вычисляется одним делением многочленов длины k_i за $\mathcal{O}(k_i \log k_i)$.

2.4. (*) Обоснование метода Ньютона

Цель: доказать, что каждый шаг удваивает число точных знаков x.

Сдалем замену переменных, чтобы было верно $f(0) = 0 \Rightarrow$ корень, который мы ищем, – 0.
Сейчас находимся в точке x_i. По Тейлору $f(0) = f(x_i) - x_i f'(x_i) + \frac{1}{2}x_i^2 f''(\alpha)$ $(\alpha \in [0..x_i])$.
Получаем $\frac{f(x_i)}{f'(x_i)} = x_i + \frac{1}{2}x_i^2 \frac{f''(\alpha)}{f'(x_i)}$. Передаём Ньютону $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - x_i - \frac{1}{2}x_i^2 \frac{f''(\alpha)}{f'(x_i)}$.
Величина $\frac{f''(\alpha)}{f'(x_i)}$ ограничена сверху константой C.
Получаем, что если $x_i \leqslant 2^{-n}$, то $x_{i+1} \leqslant 2^{-2n+\log C}$.
То есть, число верных знаков почти удваивается.

2.5. Линейные рекуррентые соотношения

Задача. Дана последовательность $f_0, f_1, \dots, f_{k-1}, \forall n \geqslant k\ f_n = f_{n-1}a_1 + \dots + f_{n-k}a_k$, найти f_n.

Вычисления «в лоб» можно произвести за $\mathcal{O}(nk)$.

2.5.1. Через матрицу в степени

$$A = \begin{bmatrix} a_1 & a_2 & \dots & a_k \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \end{bmatrix},\ \forall i\ A \cdot \begin{bmatrix} f_{i-1} \\ f_{i-2} \\ \dots \\ f_{i-k} \end{bmatrix} = \begin{bmatrix} f_i \\ f_{i-1} \\ \dots \\ f_{i-k+1} \end{bmatrix} \Rightarrow A^n \cdot \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ \dots \\ f_0 \end{bmatrix} = \begin{bmatrix} f_{n+k-1} \\ f_{n+k-2} \\ \dots \\ f_n \end{bmatrix}$$

Умножать матрицы умножаем за $\mathcal{O}(k^3) \Rightarrow$ общее время работы $\mathcal{O}(k^3 \log n)$.

2.5.2. Через умножение многочленов

На самом деле можно возводить в степень не матрицу, а многочлен по модулю...
Умножение матриц за $\mathcal{O}(k^3)$ заменяется на умножение многочленов по модулю за $\mathcal{O}(k \log k)$.

Будем выражать f_n через $f_i\colon i < n$. В каждый момент времени $f_n = \sum_j f_j b_j$.

Изначально $f_n = f_n \cdot 1$. Пока $\exists j \geqslant k\colon b_j \neq 0$, меняем $f_j b_j$ на $\left(\sum\limits_{i=1}^{k} f_{j-i} a_i\right) \cdot b_j$. (*)

Посмотрим, как меняются коэффициенты b_j. Пусть $B(x) = \sum b_j x^j$, $A(x) = x^k - \sum\limits_{i=1}^{k} x^{k-i} a_i$.

Тогда (*) – переход от $B(x)$ к $B(x) - A(x) x^{j-k} b_j$.
Изначально $B(x) = x^n \Rightarrow$ наш алгоритм – вычисление $x^n \bmod A(x)$.
Возведение многочлена x в степень n по модулю $A(x)$ – $\mathcal{O}(\log n)$ умножений и взятий по модулю.

Итого: $\mathcal{O}(k \log k \log n)$.

Лекция #2: Применение умножения многочленов

2-я пара, весна 2025

2.6. Факторизация целых чисел

• **Вычисление** $n! \bmod m$

Возьмём $k = \lfloor\sqrt{n}\rfloor$, рассмотрим $P(x) = x(x+k)(x+2k)...(x+k(k-1))$.
$P(1)P(2)P(3)...P(k) = (k^2)!$, чтобы дополнить до $n!$, сделаем руками $\mathcal{O}(k)$ умножений.
Посчитать $P(x)$ мы можем методом разделяй и властвуй за $O(k \log^2 k)$.

• **FFT над** $\mathbb{Z}/m\mathbb{Z}$

Умножим в $\mathbb{Z}$ (то же, что $\mathbb{R}$, что $\mathbb{C}$), в конце возьмём по модулю m.
Если не хватает точности обычного вещественного типа ($m = 10^9 \Rightarrow$ `long double` уже слишком мал), то представим многочлен с коэффициентами до m, как сумму двух многочленов с коэффициентами до $k = \lceil m^{1/2}\rceil$: $P(x) = P_1(x) + k \cdot P_2(x), Q(x) = Q_1(x) + k \cdot Q_2(x)$.
Сделаем 3 умножения, как в Карацубе:

$$P(x)Q(x) = P_1Q_1 + k^2P_2Q_2 + k((P_1{+}Q_1) \cdot (P_2{+}Q_2) - P_1Q_1 - P_2Q_2)$$

• **Факторизация**

Найдём $\min d$: $gcd(d!, n) \neq 1$. Тогда d – минимальный делитель n. Можно искать бинпоиском, можно «двоичными подъёмами», тогда время $= \mathcal{O}(calc(n^{1/2}!)) = \mathcal{O}(n^{1/4} \log^2 n)$.

2.7. CRC-32

Один из вариантов хеширования последовательности бит $a_0, a_1, \dots, a_{n-1}$ – взять остаток от деления многочлена $A(x) \cdot x^k$ на $G(x)$, где $G(x)$ – специальный многочлен, а $k = \deg G + 1$.

В *CRC-32-IEEE-802.3* (в v.42, mpeg-2, png, cksum) $k = 32$, $G(x) = 0\text{xEDB88320}$ (32 бита).
Если размер машинного слова $\geqslant k$, CRC вычисляется за $\mathcal{O}(n)$, в общем случаем за $\mathcal{O}(n \cdot \lceil \frac{k}{w} \rceil)$.

```
for i = n-1..k:
    if a[i] != 0:
        a[i..i-k+1] ^= G
```

Упражнение 2.7.1. $\texttt{CRC(A\^{}B)} = \texttt{CRC(A)\^{}CRC(B)}$, $\texttt{CRC(concat(A, 1))} = (\texttt{CRC(A)} \cdot 2{+}1) \bmod G$

2.8. Кодирование бит с одной ошибкой

Контекст. По каналу хотим передать n бит.
В канале может произойти не более 1 ошибки вида «замена бита».

Детектирование ошибки. Передадим $a_1, a_2, \dots, a_n$ и $b = \texttt{XOR}(a_1, a_2, \dots, a_n)$.
Если b' равно $\texttt{XOR}(a'_1, a'_2, \dots, a'_n)$, ошибки при передачи не было.

Исправление ошибки.
Удвоение бит не работает – и из 0, и из 1 в результате одной ошибки может получиться 01.
Работает утроение бит ($3n$) или «удвоение бит и дополнительно передать `XOR`» ($2n{+}1$).
Раскодирование для $2n{+}1$: $00 \to 0,\ 11 \to 1,\ 01/10 \Rightarrow$ подгоняем, чтобы сошёлся `XOR`.

Исправление ошибки за $\lceil\log_2 n\rceil+1$ дополнительных бит.
Передадим два раза $\texttt{XOR}(a_1, a_2, \dots, a_n)$. Теперь мы знаем, есть ли ошибка среди $a_1, a_2, \dots, a_n$.
Если ошибка есть, её нужно исправить. $\forall b \in [0, \lceil\log_2 n\rceil)$ передадим $\texttt{XOR}_{i:\, bit(i,b)=0}(a_i)$.
В итоге мы знаем все $\lceil\log_2 n\rceil$ бит позиции ошибки.

2.9. Коды Рида-Соломона

Задача. Кодируем n элементов конечного поля $\mathbb{F}_q$.
Канал допускает k ошибок. Хотим научиться исправлять ошибки после передачи.

Замечание 2.9.1. Конченые поля имеют размер p^k ($p \in Prime, k \in \mathbb{N}$). Поле размера p – $\mathbb{Z}/p\mathbb{Z}$.
Поле размера p^k – остатки по модулю неприводимого многочлена над $\mathbb{F}_p$ степени k.
Для кодирования битовых строк удобно использовать $q = 2^k$ при $k \in \{32, 64, 4096\}$.
Пример $\mathbb{F}_{2^{32}}$: остатки по модулю $x^{32}+x^{22}+x^2+x+1$. Один из алгоритмов поиска неприводимого степени n: ткнуть в случайный, попробовать факторизовать (Бэрликэмп за n^3).

Lm 2.9.2. Если обозначить расстояние Хэмминга как $D(s,t)$, а $f(s)$ – код строки s, канал передачи допускает k ошибок, то исправить ошибки можно iff $\forall s \neq t \;\; D(f(s), f(t)) \geqslant 2k+1$.

Доказательство. С учётом ошибок строка s может перейти в любую точку шара радиуса k с центром в $f(s)$. Если такие шары пересекаются, $\forall$ точку пересечения не раскодировать. ■

Код Рида-Соломона. Данные $a_0, \dots a_{n-1}$ задают многочлен $A(x) = \sum a_i x^i$. Передадим значения многочлена в произвольных $n+2k+1$ различных точках (здесь мы требуем $p \geqslant n$).

Теорема 2.9.3. Корректность кодов Рида-Соломона.

Доказательство. $A(x) \neq B(x) \Rightarrow (A-B)(x)$ имеет не более n корней $\Rightarrow A(x)$ и $B(x)$ имеют не более n общих значений $\Rightarrow$ хотя бы $2k+1$ различных $\Rightarrow D(f(A), f(B)) \geqslant 2k+1$. ■

Выбор точек и q: хочется применить FFT. На практике можно отправлять данные такими порциями, что $n+2k+1 = 2^b$. Нужно q вида $a \cdot 2^b + 1$ и x: $ord(x) = 2^b$, точка $w_i = x^i$, $i \in [0, 2^b)$.

• Декодирование

Мы доказали возможность однозначного декодирования, осталось предъявить алгоритм.
Имели $A(x)$, $\deg A = n-1$, передали $A(w_0), A(w_1), \dots A(w_{n+2k})$. Получили на выходе данные с ошибками $A'(w_0), A'(w_1), \dots A'(w_{n+2k})$. Посчитали интерполяционный многочлен $A'(x)$.

Утверждение 2.9.4. Если у $A'(x)$ старшие $2k+1$ коэффициентов нули, ошибок не было.

Итого, если ошибок нет, декодирование при желании можно сделать за $\mathcal{O}(n \log n)$ через FFT.

Обозначим позиции ошибок $e_1, e_2, \dots, e_k$.
Может быть, ошибок меньше. Главное, что в позициях кроме e_i ошибок точно нет.
Многочлен ошибок $E(x) = (x - w_{e_1})(x - w_{e_2}) \dots (x - w_{e_k})$, $B(x) = A(x)E(x)$, $B'(x) = A'(x)E(x)$.
$\forall i \notin \{e_j\}$ $A(w_i) = A'(w_i) \Rightarrow \forall i$ $B(w_i) = B'(w_i)$. Запишем это, как СЛАУ $\forall i$ $B(w_i) = A'(w_i)E(w_i)$, где неизвестные – коэффициенты многочленов B ($n+k+1$ штук) и E (k штук).

Теорема 2.9.5. Для записанной СЛАУ $\exists!$ решение.

Следствие 2.9.6. Декодирование: решим СЛАУ, найдём $A(x) = \frac{B(x)}{E(x)}$.

Асимптотика декодирования: Гаусс $\mathcal{O}((n+k)^3)$. Бэрликэмп-Мэсси $\mathcal{O}((n+k)^2)$.
Ссылка на более крутые алгоритмы декодирования в разд. 2.11.

2.10. Применения FFT в комбинаторике

• **Возведение в степень**

2^n бинарным возведением в степень вычисляется за $T(n) = T(\frac{n}{2}) + n\log n = \mathcal{O}(n\log n)$.

• **Умножение многочленов от нескольких переменных**

Посчитать $C(x,y) = A(x,y)\cdot B(x,y)$, пусть $\deg_x \leqslant n, \deg_y \leqslant m$, тогда возьмём $y = x^{2n+1}$ и вычислим $C'(x) = A'(x)\cdot B'(x)$, мономы вида $ax^{(2n+1)i+j}, j{\leqslant}2n$ заменим на ax^jy^i. $\mathcal{O}(nm\log nm)$.

2.10.1. Покраска вершин графа в k цветов

Предподсчитаем за $\mathcal{O}(2^n)$ все независимые множества I (те, что можно покрасить в один цвет).

Рассмотрим любую корректную покраску в k цветов $I_1, I_2, \dots I_k\colon\ \sqcup I_j = V$,
заметим $\sum I_j = 2^n{-}1, \sum |I_j| = n$.
Рассмотрим $P(x,y) = \sum_I x^I y^{|I|}$, внимательно посмотрим на $P^k(x,y)$:

Теорема 2.10.1. Коэффициент в $P^k(x,y)$ монома $x^{2^n-1}y^n$ – это число покрасок в k цветов.

Lm 2.10.2. $A\cap B = \varnothing \Leftrightarrow |A| + |B| = |A\cup B|$

Lm 2.10.3. $A\cap B \neq \varnothing \Leftrightarrow A + B > A\cup B$
(сложение/сравнение множеств – операции с битовыми масками)

Алгоритм – возведение многочлена степени $2^n n$ в степень k.
Одно умножение работает за $\mathcal{O}(2^n n^2)$, возведение в степень за $\mathcal{O}(2^n n^2\log k)$.

2.10.2. Счастливые билеты

Задача. Массив из $2n$ цифр из множества $d_1, d_2, \dots, d_k$ называется счастливым, если $\sum$ первых n цифр совпадает с $\sum$ последних n. Найти число счастливых массивов из $2n$ цифр.

Рассмотрим $P(x) = (x^{d_1} + x^{d_2} + \dots + x^{d_k})^n$. Ответ – сумма квадратов коэффициентов P.
Алгоритм = возведение в степень. Время возведения в степень – $\mathcal{O}(\log n)$ умножений.
Точнее $\mathcal{O}(mul(N) + mul(\frac{N}{2}) + mul(\frac{N}{4}) + \dots) = \mathcal{O}(N\log N)$, где $\deg P = N = n\cdot\max d_i$.

2.10.3. 3-SUM

Задача. Даны n чисел $a_i \in \mathbb{Z}\cap[S]$, найти $i,j,k\colon a_i + a_j + a_k = S$.
Решение #1. Сортировка, далее $\forall i$ два указателя для j, k. $\mathcal{O}(n^2)$.
Решение #2. Возьмём $P(x) = \sum_i x^{a_i}$, рассмотрим P^3, возьмём коэффициент при x^S. $\mathcal{O}(S\log S)$.

2.10.4. Применение к задаче о рюкзаке

Простая задача. Subsetsum. Даны n целых $a_i > 0$, выбрать подмножество: $\sum_j a_j = S$.
Посчитаем $P(x) = \prod_i (1 + x^{a_i})$, возьмём коэффициент при x^S. n умножений $\Rightarrow \mathcal{O}(nS\log S)$.

Алгоритм 2.10.4. Сложная задача. *То же, но выбрать подмножество размера ровно k.*
Посчитаем $P(x,y) = \prod_i (1 + yx^{a_i})$, возьмём коэффициент при x^Sy^k.
Будем вычислять разделяйкой: $T(n) = 2T(\frac{n}{2}) + nS\log nS$ (nS – степень многочлена).
Получаем $\mathcal{O}(nS\log nS\log n)$, что лучше базовой динамики за $\mathcal{O}(n^2S)$ (`dp[i,size,sum]`).

2.10.5. (*) Сверхбыстрый рюкзак за $\tilde{O}(\sqrt{n}S)$

Решаем subsetsum. Пусть $A = \{a_i\}$. Если мы разобьём $A = A_1 \sqcup \dots \sqcup A_k$ и для каждого A_i насчитаем массив f_{ij} = можем ли мы набрать вес j, используя предметы из A_i, то останется только за $\mathcal{O}(kS\log S)$ перемножить $F_1(x)\cdot\ldots\cdot F_k(x)$, где $F_i(x) = \sum f_{ij}x^j$.

Возьмём $k \approx \sqrt{n}$, $A_i = \{x \in A\colon x \bmod k = i\}$. $x \in A_i \Rightarrow x = ky + i \Rightarrow$ рассмотрим $B_i = \{y\colon ky + i \in A_i\}$ и для B_i воспользуемся 2.10.4, который насчитает $\sum f_{ijt}x^jy^t$, где f_{ijt} = число способов набрать сумму ровно j, используя ровно t предметов из B_i, при этом $jk + it \leqslant S \Rightarrow j \leqslant \lfloor\frac{S}{k}\rfloor \Rightarrow$ 2.10.4 отработает за $\mathcal{O}(\frac{S}{k}n_i\log n_i\log nS)$, где $n_i = |A_i|$.

Итого: решили subsetsum за $\mathcal{O}(kS\log S) + \sum n_i\mathcal{O}(\frac{S}{k}n_i\log n_i\log nS) =$ $\mathcal{O}(kS\log S) + \mathcal{O}(\frac{S}{k}n\log n\log S) \Rightarrow$ оптимальное $k = \sqrt{n\log n}$, subsetsum за $\mathcal{O}(\sqrt{n\log n}\cdot S\log S)$. $\tilde{O}f = \mathcal{O}(f\cdot poly(\log))$.

2.10.6. (*) Сверхбыстрый рюкзак за $\tilde{O}(n+S)$

Будем решать задачу $f(A)$**:** определить $\forall s \in [0,S]$, можно ли набрать вес s.

Если есть ответы $f(A)$ и $f(B)$, то `fft` за $\mathcal{O}(S\log S)$ даёт ответ для $A+B$. Назовём это свёрткой. Если мы разделим множество предметов A на $A = A_1 \sqcup A_2 \sqcup \dots \sqcup A_k$, мы можем для каждой части A_i решить задачу и свёртками за $\mathcal{O}(kS\log S)$ получить ответ для A.

Хорошее разделение: $m = \lceil\log n\rceil$, $A_i = A \cap (\frac{S}{2^i}, \frac{S}{2^{i-1}}], i \in [1, m], A_{m+1} = A \cap [1, \frac{S}{2^m}]$.
Для A_{m+1} делаем разделяйку с `fft` даёт $T(n) = 2T(\frac{n}{2}) + \texttt{fft}(n\frac{S}{2^m}) = \mathcal{O}(S\cdot n\log n\cdot\log)$.
Для A_i рассмотрим множество-ответ X_i, заметим $|X_i| < 2^i$. Обозначим $k = 2^i$.
Поделим A_i случайным образом на k множеств: $A_i = \sqcup A_{ij}$, $\mathsf{E}(\max_j(X_i \cap A_{ij})) = \mathcal{O}(\log k) = \varepsilon$, чтобы решить задачу для A_{ij} разделим его на ε^2 случайных множеств A_{ijt}.

$Pr[\forall t\ |A_{ijt} \cap X_i| \leqslant 1] \geqslant \frac{1}{2} \Rightarrow$ ответ для A_{ijt} тривиален: мы можем взять $\leqslant$1 предмета $\Rightarrow$ можем набрать только суммы $s \in A_{ijt}$. Ответ для $A_{ij} = \varepsilon^2$ свёрток, при этому в ответе нам нужны суммы не до S, а до $\frac{S}{2^i}\varepsilon$, так как $\max A_{ij} \leqslant \frac{S}{2^i}$ и $|A_{ij} \cap X_i| \leqslant \varepsilon$. Осталось свернуть ответы для A_{ij} в ответ для A_i – разделяйка длины 2^i, где в листьях `fft` от длины $\frac{S}{2^i}\varepsilon \Rightarrow$ время на свёртки для A_i: $2^i\log(2^i)M\log M$, где $M = \frac{S}{2^i}\varepsilon \Rightarrow \mathcal{O}(S\log 2^i\log M\varepsilon) = \mathcal{O}(S\log^3)$. Итак $\forall i \Rightarrow \mathcal{O}(S\log^4) = \tilde{O}(S)$.

2.11. Литература

Про FFT.
[Shuhong,Gao'2002]. Декодирование Рида-Соломона через расширенного Евклида.

Про рюкзаки (и отчасти FFT).
[Koiliaris,Xu'2017]. Subset Sum in $\tilde{\mathcal{O}}(\sqrt{n}S)$.
[Birmingam'2017]. Subset Sum in $\tilde{\mathcal{O}}(n+S)$.
[Jin,Wu'2018]. Subset Sum in $\mathcal{O}((n+S)\log^2)$. Улучшили log-факторы предыдущего решения.
[Pissinger'1999]. Практически эффективный knapsack за $\mathcal{O}(n\cdot\max a_i)$.
[Bringmann'2021]. Тут описывают, когда рюкзак решается за $\tilde{\mathcal{O}}(n)$ и дают нижние оценки.
[Becker'2011]. Рюкзак за $\tilde{\mathcal{O}}(2^{0.291n})$.

Лекция #3: Автоматы

3-я пара, весна 2025

3.1. Определения, детерминизация

Def 3.1.1. *Детерминированный автомат – $\langle V, s, T, \Sigma, E\rangle, s \in V, T \subseteq V, D \subseteq V \times \Sigma,\ E\colon D \to V$. Обозначим $|V| = n$, $|E| = m$.*

Def 3.1.2. *Детерминированный автомат называется полным, если $D = V \times \Sigma$.*

Замечание 3.1.3. Чтобы сделать автомат полным, добавим фиктивную вершину «тупик», все $\nexists$ рёбра направим в «тупик», замкнём «тупик»: по всем символам из него торчат петли.

Def 3.1.4. *Недетерминированный автомат – $\langle V, s, T, \Sigma, E\rangle,\ s \in V,\ T \subseteq V,\ E \subseteq (V \times \Sigma) \times V$*

Def 3.1.5. *Автомат принимает строку w, если $\exists s = v_0, v_1, \dots v_{|w|}\colon \forall i\ (v_i, s_i, v_{i+1}) \in E$.*

Замечание 3.1.6. Принимает ли детерминированный автомат строку s, мы проверяем за $\mathcal{O}(|s|)$.

Алгоритм 3.1.7. *Принимает ли недетерминированный автомат строку s? После i символов поддерживаем множество вершин «где мы можем сейчас находиться?». Переход $i \to i{+}1$ за $\mathcal{O}(m) \Rightarrow \mathcal{O}(m|s|)$.*

• Детерминизация

Принимая строку s, недетерминированный автомат в момент времени t находится в одной из вершин множества A_t. «Множества вершин» – состояния детерминированного автомата $\langle V', E'\rangle$.

Можно взять $|V'| = 2^n$, можно оптимальнее – dfs-ом выбрать достижимые множества вершин.

Время детерминизации $\mathcal{O}(|V'| \cdot m)$.

3.2. Эквивалентность

• Простейший алгоритм

Проверяем эквивалентность детерминированных автоматов $\langle V_1, s_1, T_1, E_1\rangle$ и $\langle V_2, s_2, T_2, E_2\rangle$.
Найдём все пары состояний $v_1 \in V_1, v_2 \in V_2\colon v_1 \not\equiv v_2$. Все пары будем помещать в очередь.
База: $(v_1 \in T_1) \neq (v_2 \in T_2) \Rightarrow$ помечаем и помещаем $\langle v_1, v_2\rangle$ в очередь.
Переход: $v_1 \not\equiv v_2 \Rightarrow \forall c, x_1, x_2\colon E(x_1, c) = v_1, E(x_2, c) = v_2 \quad x_1 \not\equiv x_2$.
Реализация: `(v1,v2)= q.pop(); for c: for x1 in from(v1,c): for x2 in from(v2,c): toQueue(x1, x2)`
Каждую пару (v_1, v_2) переберём не более одного раза $\Rightarrow$ каждую пару рёбер $\Rightarrow \mathcal{O}(m^2)$.
Для корректности алгоритма автоматы должны быть полными.

• Через минимизацию

Чтобы проверить эквивалентность $A_1 = \langle V_1, E_1, T_1, s_1\rangle$, $A_2 = \langle V_2, E_2, T_2, s_2\rangle$, запустим минимизацию для $\langle V_1 \cup V_2, E_1 \cup E_2, T_1 \cup T_2\rangle$, и посмотрим попали ли s_1 и s_2 в один класс эквивалентности.

3.3. Минимизация

Задача: построить автомат, минимальный по числу вершин, эквивалентный данному. Перед тем, как рассматривать решения, поймём, как устроен минимальный автомат.

Def 3.3.1. $R_A(w)$ – *правый контекст строки* w. $R_A(w) = \{x \mid A$ *принимает* $wx\}$.

Теорема 3.3.2. Минимальный автомат, эквивалентный A, есть $A_{min} = \langle V, E, s, T\rangle$, где $V = \{R_A(w)$ по всем $w\}$, $E = \{R_A(w) \xrightarrow{c} R_A(wc)\}$, $s = R_A(\varepsilon)$, $T = \{\varnothing\}$.

Для недетерминированных есть алгоритм Бржозовского [wiki] [pdf] :

$$A_{min} = d(r(d(r(A)))) = drdrA$$

Где d – детерминизация автомата, r – разворот всех рёбер автомата и `swap(S, T)`.

Для детерминированных обычно пользуются алгоритмом Хопкрофта.

3.4. Хопкрофт за $\mathcal{O}(VE)$

```
# дополняем автомат до полного (next[v, char] - или конец ребра, или -1)
fictive = newVertex() # next[fictive, *] = -1, isTerminal[fictive] = 0
for v in [0, vertexN):
    for char:
        if next[v, char] == -1:
            next[v, char] = fictive

# строим обратные рёбра, инициализируем классы
for v in [0, vertexN):
    for char:
        prev[next[v, char], char].add(v)
    type[v] = isTerminal[v] # тип/класс вершины
    A[type[v]].add(v)       # A[type] - множество вершин типа type
typeN = 2 # изначально есть только терминалы и нетерминалы

# основной цикл с очередью
queue q; q.push(0); # любой из классов 0, 1
while !q.isEmpty():
    t = q.pop()
    for char:
        Split(t, char) # самая сложная процедура
```

Функция `Split(t, c)` должна разделить во всех существующих классах разделить вершины по предикату «*ведёт ли ребро по символу* c *в класс* t?» и положить в очередь новые классы.

Её несложно реализовать за $\mathcal{O}(E)$, тогда суммарное время работы алгоритма $\mathcal{O}(VE)$, так как `Split` вызовется не более $V-2$ раз.

3.5. Хопкрофт за $\mathcal{O}(E \log V)$

Можно реализовать `Split` оптимальнее. Главная идея:
1. реализовать `Split(t)` за $\mathcal{O}$(просматра входящих рёбер в t),
2. при разбиении класса на два добавлять в очередь только меньшую половину.

```
1  def Split(t, char):
2      cc++ # очищаем vertexMark[] за O(1)
3      allTypes = []
4      types = []
5      for v in A[t]:
6          for u in prev[v, char]:
7              if vertexMark[u] != cc:
8                  vertexMark[u] = cc
9              t0 = type[u]
10             allTypes.add(t0)
11             B[t0].add(u) # для каждого типа t0 помним посещённую половину
12             if B[t0].size == 0:          types.add(t0)
13             if B[t0].size == A[t0].size: types.remove(t0)
14
15     for t0 in types: # те классы, которые поделились относительно (t, char)
16         if B[t0].size * 2 > A[t0].size: # если B[t0] - большая половина
17             B[t0].clear()
18             for u in A[t0]: # тратим времени O(B[t0].size), то есть, O(уже потраченного)
19                 if vertexMark[u] != cc:
20                     B[t0].add(u)
21         # теперь B[t0] - точно меньшая половина
22         for u in B[t0]: # перекрашиваем половину B[t0]
23             type[u] = typeN # номер нового класса - автоинкремент
24             A[typeN].add(u)
25             A[t0].add(u)
26         # Старый класс t0 разбит на 2 новых (t0,typeN), кладём в очередь меньшую половину
27         q.add(typeN++)
28
29     for t0 in allTypes: # быстрое обнуление массивов B[]
30         B[t0].clear()
```

Время работы. Блок строк [15-26] работает за $\mathcal{O}$([5-13]). Блок [5-13] – перебор вершин $v \in A[t]$ и входящих в них рёбер. Если вершина $v \in A[t]$ на строке (5), то следующий раз мы положим её в очередь в составе в два раза меньшего класса $\Rightarrow$ переберём её не более $\log V$ раз $\Rightarrow$ каждое входящее рёбро переберём $\log V$ раз $\Rightarrow$ время работы $\mathcal{O}(E \log V)$.

Замечание 3.5.1. Здесь E – количество рёбер в автомате, дополненном до полного $\Rightarrow E = V \cdot |\Sigma|$.

3.6. Изоморфность

Def 3.6.1. *Изоморфизм автоматов: биекция на вершинах такая, что начальная → начальная, терминальные → терминальные, рёбра → рёбра.*

• **Проверка двух автоматов на изоморфизм за $\mathcal{O}(m_1 + m_2)$**

Пишем $dfs(v_1, v_2)$, который параллельно ходит по двум автоматам, запускаем $dfs(start_1, start_2)$.

• **Проверка автомата на эквивалентность минимальному за $\mathcal{O}((m_1 + m_2))$**

Пусть 1-й из двух минимальный. Оставим от 2-го только вершины, из которых достижимы терминальные. Теперь каждая вершина 2-го лежит в одном из классов эквивалентности = вершин 1-го. Пишем $dfs(v_1, v_2)$, который параллельно ходит по двум автоматам, и для каждой v_2, понимает, в каком классе v_1 она лежит. Если какая-то v_2 должна лежать сразу в двух классах, не эквивалентны. Если в одном из автоматов нет парного ребра, не эквивалентны. Запускаем $dfs(start_1, start_2)$.

3.7. Литература

[hopcroft,motwani,ulman'2001]. Книжка про автоматы. Минимизация в разделе 4.4, стр. 154.

Лекция #4: Суффиксный автомат

4-я пара, весна 2025

4.1. Введение, основные леммы

Будем обозначать «v – суффикс u» как $\boxed{v \subseteq u}$

Def 4.1.1. *Суффиксный автомат строки s, $SA(s)$ – $\min$ по числу вершин детерминированный автомат, принимающий ровно суффиксы строки s, включая пустой.*

Def 4.1.2. $R_s(u)$ *– правый контекст строки u относительно строки s.*

$$R_s(u) = \{x \mid ux \subseteq s\}$$

Пример: $s = abacababa \Rightarrow R_s(ba) = \{cababa, ba, \epsilon\}$
Мы будем рассматривать правые контексты только от подстрок $s \Rightarrow R_s(v) \neq \varnothing$.

Def 4.1.3. $V_A = \{u \mid R_s(u) = A\}$ *– все строки с правым контекстом A.*

Def 4.1.4. $V(w)$ *– вершина автомата, в которой заканчивается строка w ($w \in V_A$).*

Утверждение 4.1.5. $\forall A$ все строки V_A заканчиваются в одной вершине суффавтомата.
Собственно вершины автомата, как и в 3.3.2 – классы V_A.

Следствие 4.1.6. Рёбра между вершинами проводятся однозначно:
$(\exists\ x \in V_A,\ xc \in V_B) \Leftrightarrow$ (между вершинами V_A и V_B есть ребро по символу «`c`»).

<u>Lm</u> 4.1.7. $R_s(v) \cap R_s(u) \neq \varnothing,\ |v| \leqslant |u| \ \Rightarrow\ v \subseteq u.$

Доказательство. Возьмём $w \in R_s(v) \cap R_s(u)$, строки vw и uw – суффиксы s, отрежем w. ■

<u>Lm</u> 4.1.8. $R_s(v) = R_s(u),\ |v| \leqslant |u| \ \Rightarrow\ v \subseteq u.$

<u>Lm</u> 4.1.9. v – суффикс $u \Rightarrow R_s(u) \subseteq R_s(v)$ (у суффикса правый контекст шире).

<u>Lm</u> 4.1.10. $v \subseteq w \subseteq u,\ R_s(v) = R_s(u) \Rightarrow R_s(v) = R_s(w) = R_s(u)$ (непрерывность отрезания).

Следствие 4.1.11. $\forall A$ класс V_A определяется парой $s_{min} \subseteq s_{max}$: $V_A = \{w \mid s_{min} \subseteq w \subseteq s_{max}\}$.

Def 4.1.12. *Суффиксная ссылка $V(w)$ – вершина $V(z)$: $z \subseteq w,\ R_s(z) \neq R_s(w),\ |z| = \max$.*

`suf[V]` – суффиксная ссылка V_A
`len[V]` $= |s_{max}(V_A)|$

<u>Lm</u> 4.1.13. $|s_{min}(V_A)| =$ `len[suf[A]]` $+ 1$

Замечание 4.1.14. `suf[A]` корректно определена iff `len[A]` $\neq 0$.

<u>Lm</u> 4.1.15. У $SA(s)$ терминальными являются вершины `V(s)`, `suf[V(s)]`, `suf[suf[V(s)]]`, ...

Из 4.1.5 (вершины), 4.1.6 (рёбра), 4.1.15 (терминалы) мы представляем устройство суфавтомата.

Из лемм и 4.1.11 (вершина = отрезок суффиксов) 4.1.12 (суффссылка) мы получили инструменты для построения линейного алгоритма.

4.2. Алгоритм построения за линейное время

Алгоритм будет онлайн наращивать строку s. Начинаем с пустой строки $s = \varepsilon$.
Осталось научиться, дописывая к s символ a, от $SA(s)$ переходить к $SA(sa)$.

Будем в каждый момент времени поддерживать:

(a) `start` – $V(\varepsilon)$ (стартовая вершина)
(b) `last` – $V(s)$ (последняя вершина)
(c) `suf[V]` – для каждой вершины автомата суффссылку
(d) `len[V]` – для каждой вершины автомата максимальную длину строки
(e) `next[A,c]` – рёбра автомата

База: $s = \varepsilon$, `start` $=$ `last` $= 1$.

Для того, чтобы понять, как меняется автомат, нужно понять, как меняются его вершины – правые контексты. Переход: $s \to sa \Rightarrow R_s(v) = \{z_1, \dots, z_k\} \to R_{sa}(v) = \{z_1a, \dots, z_ka\}$ +? ε.

Пример 4.2.1. $s = abacabx$, $R_s(ab) = \{acabx, x\}$, $R_{sa}(ab) = \{acabxa, xa\}$

Пример 4.2.2. $s = abacab$, $R_s(ba) = \{cab\}$, $R_{sa}(ba) = \{caba, \varepsilon\}$

Lm 4.2.3. $(\varepsilon \in R_{sa}(v)) \Leftrightarrow (v \subseteq sa)$.

TODO

4.3. Реализация

```
template<const int N> // автомат от строки из N вершин
struct Automaton:
    static const int VN = 2 * N + 1; // число вершин будет 2N, и ещё 1 фиктивная.
    int root, last, n, len[VN], suf[VN];
    map<int, int> to[VN]; // храним рёбра по-простому, можно лучше: массив, хещ-таблица
    Automaton(): // конструктор
        n = 1, root = last = newV(0, 0); // 0 - фиктивная, 1 - корень
    int newV(int _len, int _suf): // создание новой вершины
        len[n] = _len, suf[n] = _suf; // уже знаем длину и суффссылку
        return n++;
    void add(int a): // добавляем один символ a, перестраиваем SA(s) → SA(s+a)
        int r, q, p = last; // p указывает на старый last
        last = newV(len[last] + 1, 0); // s+a заканчивается в новом last
        while (p && !to[p].count(a)) // пропускаем вершины, из которых нет ребра по a
            to[p][a] = last, p = suf[p]; // создаём им ребро по a
        if (!p) // если мы дошли до фиктивной вершины 0
            suf[last] = root;
        else if (len[q = to[p][a]] == len[p] + 1) // если не нужно раздваиваться
            suf[last] = q;
        else:
            r = newV(len[p] + 1, suf[q]); // r - вершина для части q, суффиксов sa
            suf[last] = suf[q] = r;
            to[r] = to[q]; // r - просто копия q
            while (p && to[p][a] == q) // все суффиксы sa должны заканчиваться в r
                to[p][a] = r, p = suf[p];
Automaton SA;
for (char c : str) SA.add(c);
```

4.4. Линейность размера автомата, линейность времени построения

Теорема 4.4.1. При $n \geqslant 3$ в автомате не более $2n{-}1$ вершин.
Доказательство. Для $n = 2$ имеем базу «три вершины».
Переход $n \to n+1$: добавится две вершины. ■

Замечание 4.4.2. $2n{-}1$ достигается на тесте «`abbbbb`...».

Теорема 4.4.3. При $n \geqslant 3$ в автомате не более чем $3n{-}4$ ребра.
Доказательство. Назовём ребро $p \to q$ коротким, если $len[q] = len[p] + 1$.
Короткие рёбра образуют дерево $\Rightarrow$ их не более $2n{-}2$.
Длинным рёбрам $e\colon p \xrightarrow{c} q$ сопоставим строки $u{+}c{+}w$: u – длиннейший путь в p, w – длиннейший из q. Пути длиннейшие $\Rightarrow$ $u{+}c{+}w$ из старта в терминал $\Rightarrow$ суффикс. Пути длиннейшие $\Rightarrow$ u и w состоят только из коротких рёбер $\Rightarrow$ $\forall e$ строки $u{+}c{+}w$ различны $\Rightarrow$ их не больше непустых суффиксов $\Rightarrow$ $\leqslant n{-}1$. Итого рёбер $\leqslant (2n{-}2) + (n{-}1) = 3n{-}3$.
Ещё -1 получаем т.к. число вершин $2n{-}1$ достигается *только* на тесте `abbbb`.... ■

Замечание 4.4.4. $3n{-}4$ достигается на тесте «`abbb`...`bbbc`».

4.5. Решение задач

4.5.1. LZSS за $\mathcal{O}(n)$

• LZSS

Мы можем использовать запись (n, i) «повторить n символов, начиная с i-й позиции» для сжатия данных. Например, «abababcbab» можно теперь записать, как «ab(4,0)c(3,1)».

Задача в том, чтобы выбрать код min длины. Чуть упростим задачу: пусть и один символ, и пара (n, i) записываются одинаковым числом байт, тогда (a) выгодно использовать только пары (n, i), (b) выбирать пару с максимальным n и любым i. Сделаем это суф.автоматом за $\mathcal{O}(n)$.

• Жадность

Построим автомат от всего текста t. $\forall$ вершины v автомата предподсчитаем $i[v]$ позицию начала самого длинного суффикса-терминала, достижимого из v.
Пусть мы уже выписали p символов. Пропускаем строку `t[p:]` через автомат, пока $i[v] < p$.
Пусть мы спустились в итоге n раз, остановились в v: $i[v] < p \Rightarrow$ возвращаем пару $(n, i[v])$.

• Практические нюансы

Исходный текст следует бить на куски, например 10^6 байт, чтобы автомат влезал в кэш.
Код, который мы нашли суф.автоматом, следует записать оптимально:
символ кодируется как $9 = 1 + 8$ бит, а пара (n, i) как $\min(9 \cdot n, 1 + \lceil\log(N{-}p)\rceil + \lceil\log p\rceil)$ бит, где p – сколько мы к паре (n, i) уже выписали символом.

4.5.2. Общая подстрока k строк

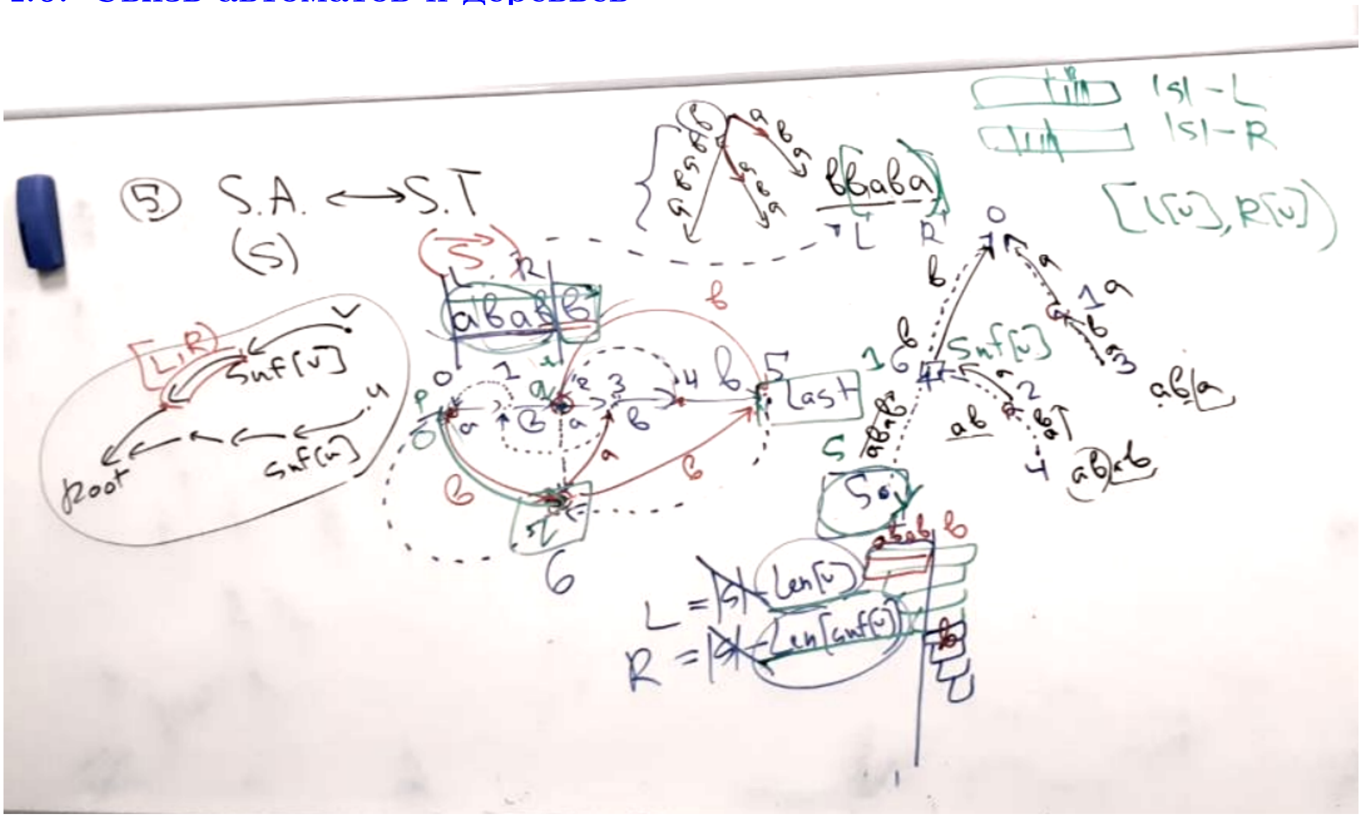
Построим суффавтомат A от минимальной из строк. Далее пропустим все остальные строки через A и для каждой вершины v, которой соответствует класс строк $(len[suf[v]], len[v]]$ будем поддерживать $m[v]$ – длину max общей подстроки, заканчивающейся в v.

Пересчёт $m[v]$. Пропускаем строку s через A:

```
v = root, k = 0
for (c in s)
    while (next[v][c] == 0) // нет ребра
        v = suf[v], k = len[v]
    v = next[v][c], k++
    // пропусктили p - префикс s через A
    // максимальный суффикс p, который есть в A, заканчивается в v
    // длина суффикса p ровно k
```

По всем вершинам v запоминаем $mk[v] = \max k$, а в конце проталкивкаем mk по суффиксным ссылкам. В итоге $m[v] = min(m[v], mk[v])$.

4.6. Связь автоматов и деревьев



4.7. Литература и история

История.

1973 | люди научились за $\mathcal{O}(n)$ растить суффдеревья, первым был алгоритм Вейнера
1983 | люди увидели связь дерева и автомата, $\mathrm{ST}(s).edges = \mathrm{SA}(s^R).suflinks$
1987 | заметили, что в автомате линия рёбер
1997 | придумали, как строить автоматы от строк напрямую, без деревьев
2001 | придумали, как строить автоматы уже от бора строк
2005 | суфф автоматы потихоньку переворачивают мир ICPC

[e-maxx]. Полное изложение суффавтомата.
[itmo]. Изложение суффавтомата без оценок размера и времени работы.
[wiki]. Полное изложение суффавтомата (автор adamant).
[codeforces]. Пост от adamant на тему суффавтоматов.
[cp-algorithms]. Ещё одно описание суффавтоматов.
[SK]. Код для копипаста.
[2009|pdf]. Статья Mohri,Moreno,Weinstein про «суффавтомат от бора» и «music identification».
[2009|pdf]. Preprint статьи выше.

Лекция #5: Линейное программирование

13 марта 2024

5.1. Применение LP и ILP

Def 5.1.1. *Задача LP. Поиск* x: $Ax \leqslant b,\ x \geqslant 0,\ \langle c, x\rangle \to \max,\ x \in \mathbb{R}^n$

Def 5.1.2. *Задача ILP. Поиск* x: $Ax \leqslant b,\ x \geqslant 0,\ \langle c, x\rangle \to \max,\ \boxed{x \in \mathbb{Z}^n}$

При этом мы помним, как приводить к виду LP несколько похожих задач.

• LP: Кратчайшее расстояние в графе (от s до всех)

x_i — расстояние до вершины i, $x_s = 0$
Для каждого ребра $a \xrightarrow{w} b$ добавляем условие $x_b \leqslant x_a + w$
$\sum_i x_i \to \max$

• LP: Максимальный поток $s \to t$

Обратные рёбра не создаём, x_e — поток по ребру e.
$0 \leqslant x_e \leqslant capacity_e$ (второе добавляем в список неравенств)
Для каждой вершины $v \neq s, t$ добавляем условие $\sum x_{e\in in(v)} = \sum x_{e \in out(v)}$
$\sum_{e\in out(s)} x_e \to \max$

Немного магии: в симплекс-методе можно $0 \leqslant x_e$ бесплатно заменить на $0 \leqslant x_e \leqslant c_e$.

• LP: Поток размера k минимальной стоимости $s \to t$

$\sum_{e\in out(s)} x_e = k$
$\sum_e x_e cost_e \to \min$

• LP: Мультипродуктовый поток

Мы решаем на одной сети одновременно k задач «пустить из s_i в t_i F_i единиц потока». По каждому ребру e течёт одновременно $f_{e_1}, f_{e_2}, \ldots, f_{e_k}$, и должно выполняться общее ограничение $\forall e \sum_i f_{e_i} \leqslant capacity_e$. В отличии от предыдущих проще чем «сведём к LP» задача не решается.

• ILP: Два непересекающихся пути $A \to B, C \to D$

Задача `NP`-трудна, через поток размера два (склеить A и B, C и D) не делается.
Зато является частным случаем *целочисленного мультипродуктового потока*.

• ILP: Паросочетание в произвольном графе максимального веса

Каждому ребру сопоставляем переменную x_e — взяли ли мы ребро в паросочетание. $x_e \geqslant 0$.
Каждой вершине условие $\sum_{e\in adj(v)} x_e \leqslant 1$.
$\sum_e x_e cost_e \to \max$.

Замечание 5.1.3. На самом деле эту задачу можно решить за полином через LP

• LP: Паросочетание в двудольном графе максимального веса

Та же сеть, что в предыдущей, но благодаря двудольности матрица A задачи LP будет обладать свойством *тотальной унимодулярности*, из чего следует, что симплекс автоматически найдёт целочисленное решение.

• **LP: Вершинное покрытие минимального веса**

$0 \leqslant x_v$ – взяли ли вершину v, для каждого ребра (a, b) имеем $x_a + x_b \geqslant 1$. Цель: $\sum_v x_v w_v \to \min$.
$x_v \in \mathbb{Z} \Rightarrow$ решаем ILP, возвращаем $\{i \mid x_i = 1\}$, получили точное решение.
$x_v \in \mathbb{R} \Rightarrow$ решаем LP, возвращаем $\{i \mid x_i \geqslant \frac{1}{2}\}$, получили 2-приближение.

5.2. Сложность задач LP и ILP

Симплекс метод решает `LP` на большинстве тестов за полином, но в худшем всё же за экспоненту. Есть полиномиальные решения `LP`. Одно из них – *метод эллипсоидов*, им мы займёмся сегодня.

Lm 5.2.1. `ILP` $\in$ `NP-hard`

Доказательство. Сведём SAT. $0 \leqslant x_i \leqslant 1$, для каждого дизъюнкта $\sum x_{i_j} \geqslant 1$. ∎

5.3. Нормальные формы задачи, сведения

Есть несколько форм задачи LP, равносильных стандартной форме $Ax \geqslant 0, x \geqslant 0, \langle c, x\rangle \to \max$:

- $Ax = b, x \geqslant 0$ (уравнения уместо неравенств).
- $\langle c, x\rangle \to \min$ (минимизация вместо максимизации).
- $Ax \leqslant b, \langle c, x\rangle \to \max$ (отсутствует $x \geqslant 0$).
- $Ax \leqslant b, x \geqslant 0$ (отсутствует максимизация/минимизация).
- $Ax \geqslant 0$.

Обозначим n – число неизвестных, m – число уравнений.
Будем сводить разные формы задачи друг к другу, следить, как меняются n и m.

$Ax = b \Leftrightarrow Ax \leqslant b \wedge -Ax \leqslant -b$. Свели равенства к неравенствам. $n, m \to n, 2m$.

$Ax \leqslant b \Leftrightarrow Ax + y = b, y \geqslant 0$. Добавили для каждого неравенства переменную $y_i \geqslant 0$, обращающую нер-во в равенство. Свели неравенства к равенствам. $n, m \to n + m, m$.

$\langle c, x\rangle \to \max \Leftrightarrow \langle -c, x\rangle \to \min$. Минимизация и максимизация равносильны.

Если мы умеем решать задачу $Ax \leqslant b$, то $Ax \leqslant b, x \geqslant 0$ – частный случай, а максимизацию можно прикрутить бинпоиском по ответу α, добавив одно неравенство: $\langle c, x\rangle \geqslant \alpha$.

Если у нас отсутствует $x \geqslant 0$, но очень хочется, можно ввести новые переменные:
$x_i = u_i - v_i$, $u_i, v_i \geqslant 0$. $n, m \to 2n, m$. На практике так, не делают, это теоретический трюк.

$Ax \geqslant 0$: **TODO**

5.4. Симплекс метод

5.4.1. Кошерный вид задачи

Пусть исходная задача была в стандартной форме $Ax \leqslant b, x \geqslant 0, \langle c, x\rangle \to \max$. И все $b_i \geqslant 0$.
Введём переменные y_j: $\langle a_j, x\rangle + y_j = b_j, y_j \geqslant 0$. Рассмотрим решение $x_i = 0, y_j = b_j$.

TODO

5.4.2. Поиск начального решения

Пусть $\exists b_i < 0$, возьмём $t = -\min b_i > 0$. Изменим наши неравенства вида $Ax \leqslant b$ на $Ax - t \leqslant b$ (из всех вычтем t). Заметим, что $x_i = 0$ под новые нер-ва подходит. Осталось решить $t \to \min$ при

$t \geqslant 0$: запустим симплекс-метод. Если $\min t = 0$, нашли начальное решение, иначе решений $\nexists$.

5.4.3. Основной шаг оптимизации

У базисных x_i $c_i =, 0$. Пусть у всех не базисных x_i $c_i \leqslant 0 \Rightarrow$ решение оптимально:
$\forall i\ x_i \geqslant 0, c_i \leqslant 0 \Rightarrow \forall x, c\ \langle c, x\rangle \leqslant 0 \Rightarrow \langle c, x\rangle = 0 = \max$ – оптимум.

5.5. Геометрия и алгебра симплекс-метода

Система неравенств $Ax \leqslant b$ — пересечение полупространств. Такой объект называется полиэдром. Полиэдр выпукл, максимум любой линейной функции на нём достигается в вершине.

Lm 5.5.1. Для системы $Ax = b, x \geqslant 0, \langle c, x\rangle \to \max$ из m уравнений $\exists$ оптимальное решение, содержащее не более m ненулевых x_i.

Доказательство. Рассмотрим оптимальное решение x^* с максимальным числом нулевых компонент. Пусть число нулей $k \geqslant m+1$. Мы хотим подвигать x^* так, чтобы нули остались нулями, а x^* осталось решением. Решение системы «m уравнений $Ax = b$ и $n-k$ уравнений $x_i = 0$» или пусто, или хотя бы прямая ($dim = n - m - (n-k) = k - m \geqslant 1$). x^* — решение $\Rightarrow \exists$ прямая, пойдём по ней в сторону увеличения $\langle c, x\rangle$, пока не упрёмся в ограничение $x_j = 0$.
Получили решение, у которого $\langle c, x\rangle$ не хуже, а нулей больше. Противоречие. ■

Теорема 5.5.2. Корректность симплекса

Доказательство. Чем занимается симплекс? Перебирает наборы из m столбцов, которые соответствуют ненулевым переменным. Зафиксировав эти m столбцов и занулив оставшиеся переменные, мы однозначно получаем кандидата на оптимальное решение.
Конечность алгоритма: есть всего $\binom{m}{n}$ выборок, важно, чтобы они не повторялись.
Для этого мы используем правило Блэнда (доказательство) — брать min столбец.
Оптимальность решения после остановки симплекса: $\forall i\ c_i \leqslant 0 \Rightarrow \langle c, x\rangle \leqslant 0 \Rightarrow 0$ – оптимум. ■

• Симплекс перебирает вершины полиэдра

Если исходная задача имела форму $Ax \leqslant b, x \geqslant 0$, то область допустимых решений — полиэдр, а оптимум $\langle c, x\rangle$ достигается в вершине полиэдра. Симплексу мы скармливаем систему $Ax+y = b, x \geqslant 0, y \geqslant 0$. При этом и $y_i = 0$, и $x_i = 0$ означает, что одно из исходных неравенств обратилось в равенство (полупространство $\to$ плоскость). Из $n+m$ переменных x_i, y_i n обратятся в ноль, чем породят n плоскостей, пересечение которых — вершина полиэдра.

• Когда у симплекса есть вероятность застрять?

Если вершина полиэдра – не оптимум, из неё есть ребро, по которому функция увеличивается. Но симплекс может остаться на месте, лишь поменяв множество столбцов. Это значит, что вершина полиэдра есть одновременное пересечение больше чем n плоскостей.

5.6. Литература, полезные ссылки

[CF]. LP: геометрия, двойственность
[cormen]. Доступное для школьников описание симплекса.
[test]. Тест, на котором симплекс работает за экспоненту (Klee–Minty cube).
[efficiency]. Чуть подробнее про время работы симплекса (Hirsch Conjecture and so on).
[bland]. Правило Блэнда.
[max-babenko]. Хороший видео курс про линейное программирование от Максима Бабенко.

5.7. Перебор базисных планов

Понимание симплекса дало нам простейшее решение для LP – перебрать все $\binom{m}{n}$ базисных планов и для каждого пустить Гаусса. Такое решение можно использовать для тестирование. Заметьте, что до этого мы не знали вообще никаких решений задачи LP кроме симплекса.

5.8. Обучение перцептрона

В фундаменте нейронных сетей живёт один нейрон, он же перцептрон. Для решения некоторых задач классификации достаточно сети, состоящей лишь из одного нейрона.

Задача: даны точки $a_i \in \mathbb{R}^n$ и значения $y_i \in \{\pm 1\}$, найти гиперплоскость $b, x_1, \dots x_n$, что $\forall i\ sign(b + a_{i_1}x_1 + a_{i_2}x_2 + \dots + a_{i_n}x_n) = sign(y_i)$

Сразу упростим задачу
1. Добавим $\forall i\ a_{i_0} = 1$, обозначим $x_0 = b$.
2. Для всех $y_i = -1$ заменим a_i на $-a_i$, а y_i на 1. 3. Нормируем все a_i.

Теперь мы просто ищем такой вектор x, что $\forall i\ \langle x, a_i \rangle > 0$.

Решение: начнём с $x_0 = 0$, пока $\exists i_k\ \langle x_k, a_{i_k} \rangle \leqslant 0$, переходим к $x_{k+1} = x_k + a_{i_k}$.

$$|a_i| = 1,\ |x_k|^2 = (x_{k-1} + a_{i_{k-1}})^2 = x_{k-1}^2 + 1 + 2\langle x_{k-1}, a_{i_{k-1}} \rangle \leqslant |x_{k-1}|^2 + 1 \leqslant k. \quad (1)$$

$$x^* \text{ – искомый ответ, } |x^*| = 1,\ \langle x_k, x^* \rangle = \textstyle\sum_j \langle a_{i_j}, x^* \rangle \geqslant k\alpha, \text{ где } \alpha = \min_i \langle a_i, x^* \rangle > 0. \quad (2)$$

$$k\alpha \overset{(2)}{\leqslant} \langle x_k, x^* \rangle \leqslant |x_k| \overset{(1)}{\leqslant} \sqrt{k} \Rightarrow \sqrt{k} \leqslant \tfrac{1}{\alpha} \Rightarrow \boxed{k \leqslant \alpha^{-1/2}} \quad \blacksquare$$

Проблема: уже при $n \geqslant 2$ мы можем построить тесты с α близким к нулю.

5.9. Метод эллипсоидов (Хачаян'79)

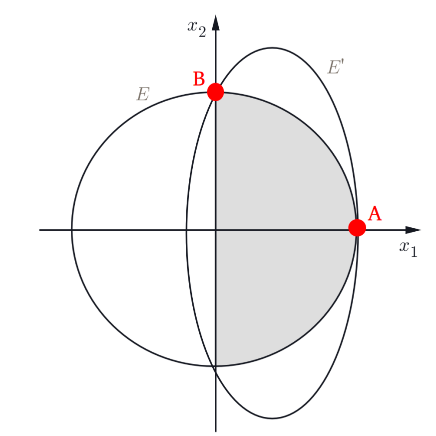
Решаем задачу $P = \{x \mid Ax \geqslant b\}$, найти $x \in P$, ограничение $P \neq \varnothing \Rightarrow Volume(P) > 0$.
Кроме A и b нам должны дать шар $E_0(x_0, r_0) \supseteq P$.

• Алгоритм

В каждый момент времени у нас есть эллипсоид $E_k(x_k, R_k)$: $x^T R x \leqslant 1$, содержащий P.
Если $x_k \in P$, счастье. Иначе выберем i_k: $\langle a_{i_k}, x \rangle < b$. $P \subseteq E_k \cap H_k$, где $H_k = \{x \mid \langle a_{i_k}, x \rangle \geqslant b\}$ (в половине эллипсоида по направлению a_{i_k} от центра). Теперь выпишем переход к $(k+1)$-му эллипсоиду в предположении, что E_k – единичная сфера, и $\forall i\ |a_i| = 1$.

$$x_{k+1} = x_k + \tfrac{1}{n+1} a_{i_k}$$
$$r_{k+1} = \left(\tfrac{n}{n+1}, \tfrac{n}{\sqrt{n^2-1}}, \dots, \tfrac{n}{\sqrt{n^2-1}}\right)$$
$$R_{k+1} = \begin{bmatrix} \frac{(n+1)^2}{n^2} & 0 & 0 \\ 0 & \frac{n^2-1}{n^2} & 0 \\ 0 & 0 & \ddots \end{bmatrix}$$

Где первая координата радиуса указана по направлению a_{i_k}.



• Математика: эллипсоиды

Эллипс: $\frac{x^2}{a^2} + \frac{y^2}{b^2} \leqslant 1$.
Добавим поворот и переход к $\mathbb{R}^n$: $z^t R z \leqslant 1$, где $z \in \mathbb{R}^n, R \in \mathbb{R}^{n \times n}, R \succ 0$.
Случай из $\mathbb{R}^2$ без поворота: $z = (x, y),\ R = \begin{bmatrix} \frac{1}{a} & 0 \\ 0 & \frac{1}{b} \end{bmatrix}$.

• Математика: матрицы

$R \succ 0$ (положительно определена) $\Rightarrow R = B^t D B$ (здравствуй, Жорданова форма), где
B – ортонормированная матрица собственных векторов,
D – диагональная матрица собственных чисел (с точки зрения геометрии $D_{ii} = \frac{1}{r_i^2}$).
Более того $B^t = B^{-1}$ (это нормально для ортонормированных матриц).

Lm 5.9.1. Растяжение системы координат по направлению **z**

(1) Важно помнить, что если $f(x, R) = x^t R x$, то
$f(x+y, R) = f(x, R) + f(y, R),\ f(x, R+\Delta R) = f(x, R) + f(x, \Delta R)$.
(2) Фокус: $\forall z \in \mathbb{R}^n \colon |z| = 1$ матрица $S = zz^t$ обладает свойством $f(z, S) = 1$, $\forall v \perp z\ f(v, S) = 0$.
(1),(2) $\Rightarrow \forall A, |z|{=}1, \alpha{\in}\mathbb{R}, S = A + \alpha z^t z \quad f(z, S) = f(z, A) + \alpha$, $\forall v \perp z\ f(v, S) = f(z, S)$

• Математика: системы координат

Почему мы предполагали, что E_k – единичная сфера?
Это удобно, и мы всегда можем перейти к такой системе координат, где это верно:
$R = B^t D B, x^t R x = 1 \Rightarrow S = B^T D^{-1/2}, x = Sy, y^2 = 1$ (y – точка на единичной сфере).

• Обоснование алгоритма

Радиус $r_1 = \frac{n}{n+1}$ подобран так, чтобы точка A была покрыта: $\frac{1}{n+1} + \frac{n}{n+1} = 1$.
Радиус $r_2 = \frac{n}{\sqrt{n^2-1}}$ подобран так, чтобы точка B была покрыта: $\frac{x_1^2}{r_1^2} + \frac{x_2^2}{r_2^2} = \left(\frac{1}{n+1}\right)^2 \left(\frac{n+1}{n}\right)^2 + \frac{n^2-1}{n^2} = 1$.
Посмотрим на частное объёмов. Объём эллипсоида равен $f(n) r_1 r_2 \ldots r_n$, поэтому $\frac{V_{k+1}}{V_k} = \frac{r_1 r_2 \ldots r_n}{1 \cdot 1 \cdot \ldots 1} =$
$= \frac{n}{n+1} \left(\frac{n}{\sqrt{n^2-1}}\right)^{n-1} = \ldots \leqslant e^{-1/2(n+1)} \Rightarrow$ за $2(n{+}1)$ шагов объём уменьшится хотя бы в e раз.

$\Rightarrow$ количество шагов не более $2(n{+}1) \cdot \ln(Volume(E_0)/Volume(P))$.
Хорошая новость: это полином от n.
Плохая новость: в худшем это $\mathcal{O}(n^4 \log(nU))$.
Пояснение: $Volume(E_0) \leqslant (nU)^{\Theta(n^2)}$, $Volume(P) \geqslant (nU)^{-\Theta(n^3)}$. U – ограничение на $a_{ij}, b_i \in \mathbb{Z}$.

• Время работы алгоритма

Один шаг работает за $mn + n^2$.
m – количество проверок $\langle a_i, x_k \rangle \geqslant b_i$.
n^2 – время пересчёта матрицы система координат.

Требуемая точность вычислений – $n^3 \log U$ цифр.

Итого: $\mathcal{O}^*(n^9)$, где $9 = 2 + 3 + 4$.

• Обобщение

Можно применить не только к $\cap$ полупространств, но и к $\cap$ $\forall$ выпуклых множеств P_i.
Нам достаточно уметь проверять $x_k \in P_i$ и искать опорную плоскость к P_i.

Так получается полиномиальное решение для SDP (semidefinite programming) [wiki].

• **Псевдокод**

```
# Наш эллипсоид задаётся уравнением (x − x_0)^t D^{-1}(x − x_0) = 1
# Напомним, что D ≻ 0 ⇒ D = A^T RA, D^{-1} = A^T R^{-1} A, где R - диагональная
x_0 = [0,0,0,...,0]
D[i,i] = 10^20 # ∞
while True: # Предполагаем, что ответ существует
    find i : ∑_j a[i,j]x_0[j] < b[i] # O(nm)
    if i does not exist : return x_0 # Нашли точку, удовлетворяющую всем неравенствам
    z = D*a_i # O(n^2), z - нормаль a_i в другой системе координат
    len = (a_i^t*D*a_i)^{1/2} # O(n^2)
    x_0 += 1/(n+1) z/len # O(n)
    D = n^2/(n^2-1)*(D - 2/(n+1) z*z^t/len^2) # O(n^2), растянули всё в n/sqrt(n^2-1) раз и
                                               # поменяли радиус по направлению z, см. 5.9.1
```

5.10. Литература, полезные ссылки

[MIT'09 | ellipsoid]. Краткое, но полное описание метода эллипсоидов.
[Florida'07 | ellipsoid]. Строгое описание метода эллипсоидов с кучей ссылок по теме.
[Кормен'2013, разделы 29.2 и 35.4]. Примеры задач на LP.