Алгортимы, V семестр

Осень 2016, лектор: Копелиович Сергей

Авторы: 3 курс

Собрано: 18 января 2017 г. 21:36

Оглавление

Глава 1

FFT

$\omega_n = e^{2\pi i/n}$
$\omega_n^j = e^{2\pi ij/n}$
2 задачи:

- Экстерполяция(по многочлену - значения в точках).

  $P(x) \xrightarrow{FFT} P(\omega_n^0), P(\omega_n^1), ...$
  $P \cdot Q(\omega_n^j) = P(\omega_n^j) \cdot Q(\omega_n^j)$ // $\mathcal{O}(n)$

- Интерполяция(по точкам - многочлен).

  $P(\omega_n^0), P(\omega_n^1), ... \xrightarrow{FFT^{-1}} P(x)$

$FFT^{-1} =$

1. FFT
2. reverse(a + 1, a + n). $a[0]$ остаётся на месте
3. $a[i]/ = n$

Псевдокод:

```
FFT(n, p) { // $n = 2^k, \omega_n^j - we want to count in this roots$
    if (n == 1) { return p[0] }
    // $P(x) = P_0(x^2) + xP_1(x^2)$ - divide degrees on even and odd
    for i = 0..n - 1
        A[i % 2].push_back(p[i])
    F_0 = FFT(n/2, A[0])
    F_1 = FFT(n/2, A[1])
    for i = 0..n - 1
        res[i] = F_0[i % (n/2)] + w_n^i \cdot F_1[i % (n/2)] // P_0(w_n^{2i}) = P_0(w_{n/2
```

$FFT = \theta(n \log n) \Rightarrow Mul = \theta(n \log n)$

1.1. Разделяй и властвуй

1.1.1. Перевод из одной системы счисления в другую

Например, их десятичной в двойчную($10 \to 2$).
$A = A_0 \cdot 10^{n/2} + A_1$

$B_0 = Get_2(A_0)$
$B_1 = Get_2(A_1)$
$C = Get_2(10^n/2)$
$B = B_0 \cdot C(FFT - n\log n) + B_1$ - всё в двоичной системе счисления.
Так мы сможем решить задачу, но тут три рекурсивных вызова, а мы хотим 2! ⇒ предподсчёт!
$n = 2^k$
$10^1, 10^2, 10^3, ..., 10^{2^k}$ - просто считаем втупую.

$$\sum_{\text{По степеням двойки}} i\log i = \theta(n\log n)$$

Итого, Работа $= T(n) = 2T(n/2) + MUL(n\log n for FFT)$
Предподсчёт $= \sum i\log i = \theta(n\log n)$
Мастер-теорема:
Пусть $T(n) = aT(n/b) + n^c\log^d n = aT(n/b) + f(n)$.

- $a < b^c \to T(n) = f(n)$
- $a = b^c \to T(n) = f(n)\cdot\log n$
- $a > b^c \to T(n) = a^{\log_b n} = n^{\log_a b}$

Подсчёт времени на перевод:

- Если MUL = FFT, то T(n) = ?. По теореме $T(n) = n\log^2 n$(a = 2, b = 2, c = d = 1)
- Если MUL = Карацуба, то умножение занимает: $T(n) = 3T(n/2) + n = n^{1.58}$
  Таким образом, $T(n)$ для перевода: $T(n) = n^{1.58} = f(n)$ (a = 2, b = 2, c = 1.58)

На практике непонятно, что лучше. Чем больше n, тем FFT лучше. Например, при n = 100 000, FFT уже лучше.
// Java 1.7 → Java 1.8 быстрый BigInteger! Не хуже Карацубы.
Python MUL = Карацуба, но 2 → 10 работает за n^2, поэтому вывод такой долгий. //

1.1.2. Деление

A(x) = B(x)Q(x) + R(x) $\in \mathbb{R}[x]$, $\deg R < \deg B$.
По A, B хотим найти Q и R.
$T(n) = 2T(n/2) + n\log n$ - хотим применить метод разделяй и властвуй, чтобы получилось такое уравнение.
Если получим Q, R потом посчитать несложно.
Утверждается, что чтобы найти Q, достаточно знать несколько первых коэффициентов A и B. К примеру, если $deg(A) = 100, deg(B) = 100$, достаточно всего одного старшего коэффициента у A и у B. А если $deg(A) = 100, deg(B) = 98$, то достаточно всего три старших коэффициента. Отсюда:
Th:
x - количество старших коэффициентов у A и B, которые нам необходимо знать. Тогда $x = k_1 - k_2 + 1$, где $k_1 = deg(A), k_2 = deg(B)$.
Док-во:
Деление в столбик.
Пусть Div(n, A, B) - находит n старших коэффициентов частного A и B. $n = 2^k$. Если частное имеет степень меньшую n, то мы считаем также несколько коэффициентов при отрицательных степенях частного - ничего страшного. Изначально n - первая степень двойки, большая или равная $k_1 - k_2 + 1$.
Псевдокод:

```
Div(n, A, B) {
    if (n == 1): return A[0]/B[0]
    C1 = Div(n/2, A[:n/2], B[:n/2])
    A -= B*C_1 // Now first n/2 coefficients A is null. Here's FFT.
    C2 = Div(n/2, A[n/2:], B[:n/2])
    return [C1, C2] //concatenation
}
```

Посчитаем время. $\mathrm{T(n)} = 2\mathrm{T(n/2)} + \left[\begin{array}{c} n\log n - -FFT \\ n^{1.58} - -\text{Карацуба} \end{array}\right.$

1.1.3. Значения в точках

$x_1, x_2, \dots, x_n$ - произвольные точки. Хотим посчитать значения в них.
План действий:

1. $P(x) \bmod (x-x_1)(x-x_2)(x-x_3)\dots(x-x_n) =: Q(x)$
2. $Q(x_1, \dots, x_{n/2})$
3. $Q(x_{n/2+1}, \dots, x_n)$

Псевдокод:

```
Calc(P, n, x1, x2, x3, .., xn) {
    if (n == 1): return P(x1)
    P %= (x - x1)(x - x2)..(x - xn) //after this, deg P < n
    A1 = Calc(P, n/2, x1, x2, .., xn/2)
    A2 = Calc(P, n/2, x(n/2 + 1), .., xn)
    return A1 concat A2
}
```

Остался единственный вопрос - как мы быстро находим многочлен $(x-x_1)(x-x_2)\dots(x-x_n)$?
Дерево отрезков! (Картинка)
Считаем время. Mod = Div + Mul
$\mathrm{T(n)} = 2\mathrm{T(n/2)}$ + Mod + Mul(<- на каждом уровне ДО FFT, мы его включаем в уравнение тоже на каждом уровне)
Mod $= n\log^2 n \Rightarrow n\log^3 n = T(n)$ - FFT
Mod $= n^{1.58} \Rightarrow n^{1.58} = T(n)$ - Карацуба
Здесь разница стала уже не такой большой. Поэтому,
Правильный разделяй и властвуй:

$$\left[\begin{array}{c} FFT - -n \geqslant 32 \\ \text{Карацуба} - -n \leqslant 16 \end{array}\right.$$

1.1.4. Интерполяция

Задача аналогична предыдущей, только наоборот. Есть произвольные точки, а нам надо восстановить многочлен. Для этого мы вспоминаем интерполяционный многочлен Ньютона и его построение.
$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) \to P(x)$
Псевдокод:

```
Calc(n, x1, y1, x2, y2, ..)
    P = Calc(n/2, x1, y1, x2, y2, ..)
    z(x) = (x - x1)(x - x2)..(x - xn/2) // we can count this with help segment tree
    yi‘ = (yi - P(yi))/z(yi) // for all i = n/2 + 1..n
    Q = Calc(n/2, x(n/2 + 1), y‘(n/2 + 1), .., xn, yn‘)
    Ans = P + z(x)Q
```

$T(n) = 2T(n/2) + Mul +$ Экстраполяция(когда считали P(yi)), если считать всё в терминах FFT, то $T(n) = n \log^4 n$.

Упражнение:

Посчитать $\sqrt{P(x)}$ методом разделяй и властвуй. Корень многочлена - это такой многочлен Q, степени не меньше n/2, что $\deg(Q^2 - P) \to \min$.

1.2. Деление за $n \log n$

- Деление чисел = метод Ньютона.

  $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ Тогда, если в x_i было k верных знаков, то в x_{i+1} примерно 2k верных знаков.

  Из матана мы знаем: $f(x) = f(0) + xf'(0) + 1/2x^2 f''(\xi)$, для какой то $\xi \in [0, x]$. Если x - корень f, то уравнение получается таким: $0 = f(0) + xf'(0) + 1/2x^2 f''(\xi)$, таким образом, $x = \frac{1/2x^2 f''(\xi) + f(0)}{f'(0)}$.

  Таким образом, если изначальная разница была $x - 0 = x$, то после того, как мы сделали переход по правилу и получили новый x', $x - x' = 1/2x^2 \frac{f''(\xi)}{f'(0)}$. Если $M = \frac{max f''}{min f'} \frac{1}{2}$, то разница в x переходит в разницу Mx^2. А если в терминах точных знаков, то k точных знаков становятся $2k - \log M$ точных знаков.

  Чтобы M ничего не портило, надо начинать с достаточно большого приближения.

  Но всё, что наверху относимо в общем к принципу Ньютона. Мы хотим научиться делить числа. Деление чисел = взятие обратного + умножение, так что научимся брать обратный. То есть $a \to a^{-1}$.

  Так как Фурье умеет работать только с целыми, то вещественные числа храним так: Integer number + Integer pos. При перемножении number перемножаются, а pos складываются.

  В методе Ньютона для наших целей $f(x) = \frac{1}{x} - a$. $x \to x - \frac{f(x)}{f'(x)} = x + \frac{1/x - a}{\frac{1}{x^2}} = 2x - ax^2$.

  Откуда начинаем? Начинаем со значения x^{-1} в double. Там будут 15 точных знаков. Можно считать, что каждый раз количество точных знаков увеличивается в полтора раза, чтобы не оценивать M. Каждый раз отрубаем по количеству точных знаков.

  $T(n) = \sum_k 1.5^k \log(1.5^k) = \theta(n \log n)$, где n - количество точных знаков, которые мы хотим получить.

- Многочлены.

  A(x) $\in \mathbb{R}[[x]]$ - формальный степенной ряд. Чтобы у него был обратный, $a_o \neq 0$.

  Обратим бесконечный ряд. $A(x) = a_0 + a_1 x + \ldots$, deg A = n. $\frac{1}{A(x)}$ может не иметь степени, поэтому надо заранее понимать, какое количество коэффициентов нам надо.

  Тупо, коэффициенты можно посчитать по формуле из Дискретки:

  $b_0 = \frac{1}{a_0}$

  $b_1 = -b_0 a_1 / a_0$

$b_2 = -(b_0 a_2 + b_1 a_1)/a_0$

...

$b_i = - ...$

Длина i-го уравнения - это i $\Rightarrow$ Втупую работает за n^2.

А теперь метод разделяй и властвуй. Хотим $n \log n$.

```
Inv(A) {
    if (n == 1): return {1/a[0]}
    R = Inv(n/2, A)
    Q = A*R(FFT) - 1 and shift
    Ans = -Q*R
}
```

Объяснение:

$A * R = 1 + x^{n/2} Q$

$(R + C * x^{n/2})A = 1 + x^n T$ - нам надо найти ещё C.

$1 + x^{n/2} Q + AC \cdot x^{n/2} = 1 + x^n T$

$Q + AC = x^{n/2} T$

$AC \approx -Q$

$C = (-QR)$

Время работы:

$T(n) = T(n/2) + Mul = T(n/2) + n \log n = n \log n$

- Многочлены.

  Итак, для рядов мы умеем делать следующее:

  $A \in \mathbb{R}[[x]], a_0 \neq 0$ $\frac{1}{A}$**первые k цифр** $\xrightarrow{\Theta(k \log k)}$ $\frac{1}{A}$первые 2k цифр

  $\Leftrightarrow$ n цифр за $\Theta(n \log n)$.

  А теперь мы хотим научиться делить многочлены с остатком за $\Theta(n \log n)$.

  A(x) = B(x)Q(x) + R(x), deg R $<$ deg B.

  A, B $\xrightarrow{хотим}$ Q, R

  A^R - развернём A, как массив. $A(x) = x^n A^R(1/x)$.

  Тогда равенство можно переписать так:

  $x^n A^R(1/x) = x^n B^R(1/x) Q^R(1/x) + R^R(1/x) x^{m-1}$ - мы считаем, что степень B равна m - 1, иначе можно просто добить старшие коэффициенты 0, где n = deg A, m = deg B.

  $1/x = z$, равенство $* = x^{-n}$.

  $A^R(z) = B^R(z) Q^R(z) + R^R(z) z^{n-m+1}$

  $\Leftrightarrow$ $A^R(z) \equiv B^R(z) Q^R(z)$, то есть последние n - m + 1 коэффициент у них одинаковые. $\Leftrightarrow$ $A^R(z) B^{R^{-1}}(z) \equiv (mod z^{n-m+1}) Q^R(z)$ - мы умеем за $\theta(n - m + 1) \log(n - m + 1)$ высчитывать первые n - m + 1 коэффициент -1 степени. А затем умножение слева - Фурье за $\theta(n \log n)$.

  $B^R(z)[a_0] \neq 0$, так как $B(z)$[старший коэффициент] $\neq 0$.

  Алгоритм:

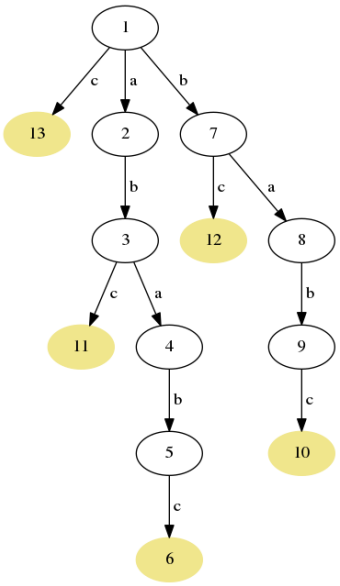
1. $(B^R)^{-1}$
2. $Q^R = A^R(B^R)^{-1}$ (первые n - m + 1), $n \log n$.
3. $R = A - BQ$, $n \log n$.

Глава 2

Суффиксный автомат

2.1. Определение, связь с суффиксным деревом, единственность, конечные вершины

Суффиксное дерево: Построем **несжатое** суффиксное дерево для строки **ababc**:



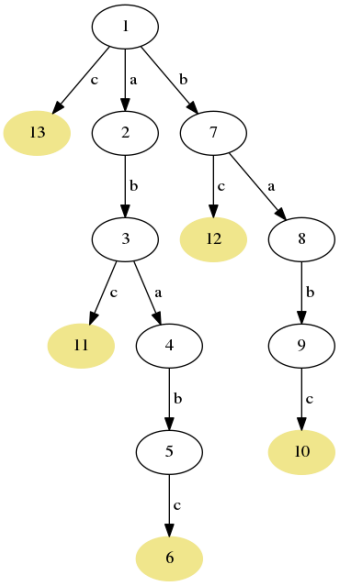
У дерева есть вершины, в которых заканчиваются суффиксы, то есть терменальные вершины и есть корень, то есть стартовая вершина.

Замечание 2.1.1. Суффиксное дерево является детерменированным конечным автоматом(ДКА), который принимает суффиксы строки и только их.

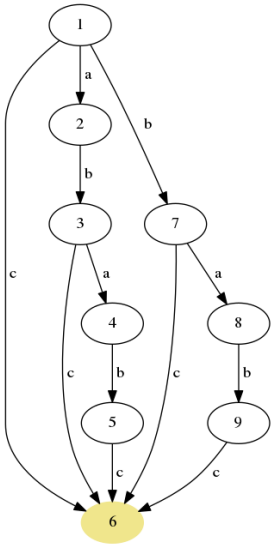
Определение суфавтомата:

𝔇𝔢𝔣 2.1.1. Суффиксный автомат — минимальный по числу вершин ДКА, который принимает все суффиксы данной строки и только их.

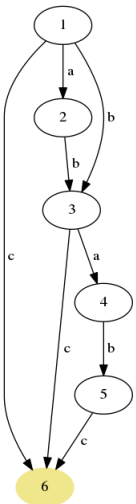
Из дерева в автомат: Отличие дерева от автомата, что в дереве очень много лишних вершин. Например, следующие 4 вершины абсолютно одинаковые, их можно склеить:



Есть еще пара вершин, которые совсем одинаковые, из ни выходит по одному ребру с символом c и оно ведет в терминальную вершину. Тоже сожмем:

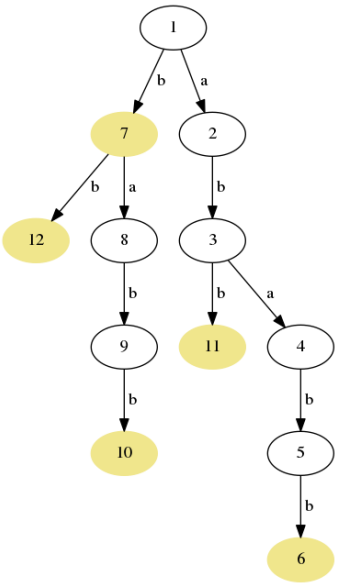


Ну и еще две вершины тоже равны, тоже можно склеить:



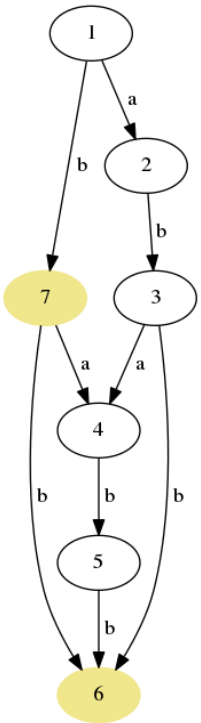
Проверяем, что в получином автоматие есть все суффиксы и нет ничего лишнего. Пока не понятно, почему этот автомат минимальный. Это мы поймем чуть позже.

Еще один пример: Теперь построим дерево для строчки **ababb**, что бы понять важность терминальных вершин:



Это дерево принципиально отличается тем, что есть терминальная вершина, которая не в листе.

Здесь попробуем так же склеить.



Заметим, что мы **не** можем склеить терминальную и не терминальную вершину. Поэтому в этом автомате вершинок получилось чуть больше, чем раньше.

Правые контексты:

Def 2.1.2. Правый контекст v относительно S это следующее множество:
$R_S(v) = \{w : vw — \text{суффикс } S\}$
То есть множество всех строк, котрые нужно дописать к v, что бы получить суффикс S.

Теперь, что бы понять как писать автомат, хочется понять в каком случае две вершины являются одинаковыми. Вершины одинаковые, если их правые контексты одинаковые.

С точки зрения дерева, это все пути из вершины, которые ведут до терминального состояния. То есть, если есть две вершины u, v, такие что $R_S(v) = R_S(u)$, то эти две вершины надо слить.

Теорема об устройстве суффиксного автомата:

Теорема 2.1.1. $V_A = \{u : R_S(u) = A\}$ — множество строк с правым контекстом A.
Каждому непустому классу V_A должна соответствовать ровно одна вершинка в автомате.

Так же V_A можно воспринимать как множество строк, которые закончатся в соответствующей вершине.

Вершину назовем A.

▶ Доказывать данную теорему не будем, в связи с существанием у нас курса по формальным языкам, где данное утверждение уже было доказано. Это, кстати, не значит, что его не надо уметь доказывать, просто доказательство можно найти в другом конспекте. ◀

Ребра автомата: Однозначно можно понять, как между вершинами проводить ребро.

$$\left.\begin{array}{rl} S \in & V_A \\ Sa \in & V_B \end{array}\right\} \Rightarrow A \xrightarrow{a} B$$

Берем какую-то строчку, у которой правый контекст A, добавляем символ a, смотрим какой правый контекст у новой строчки и туда проводим ребро.

Тогда наша задача просто найти эти правые контексты, автомат будем строит по **индукции**:

База: Есть одна вершинка last, s - пустая.
Вершина last - это вершина, которая соответствует всей строке.

Переход: Построили автомат для s, теперь перестроим для строки sa, где s - строка, a - символ.
И что бы нам это сделать, нужно сначала понять, как будут меняться правые контексты.

Устройство классов V_A:

Лемма 2.1.1. Был у нас праввый контекст $R_S(v)$, теперь хотим посчитать $R_{Sa}(v)$. Правый контекст получается из правого дописыванием символа a ко всем строчкам и, возможно, добавлением ϵ

$R_{Sa}(v) = \{za | z \in R_S(v)\} + ?\{\epsilon\}$

Лемма 2.1.2. $R_S(v) = R_S(u) \Rightarrow v$ и u — либо v суффикс u, либо u суффикс v.

Def 2.1.3. Обозначение: $v \leqslant u \Leftrightarrow v$ — суффикс u

Лемма 2.1.3. $v \leqslant w \leqslant u$ и $R_s(v) = R_s(u) \Rightarrow R_s(w) = R_s(v)$.

Def 2.1.4. Обозначение: $v \subset S$ — v подстрока S.

▶ Если v подстрока S, то правый контекст v не пустое множество. Размер правого контекста равен количеству вхождений строчки v в строчку S.
Теперь, если мы говорим, что два не пустых правых контекста равны, то там есть хотя бы одна строчка z. Возьмем строчку v, допишем z, получили $vz \leqslant S$, но так же и $uz \leqslant S$. Пусть v короче u, тогда из этого следует, что $vz \leqslant uz \Rightarrow v \leqslant u$. Что и требовалось. ◀

▶ Теперь доказательство второй леммы. Есть строчка u, есть v, между ними вклинилась строчка w.
Пусть есть строчка z, которую мы можем дописать к u и получить суффикс строки S, но w суффикс u, тогда uz суффикс S.
И в другую сторону, если мы что-то могли дописать к w, то тоже самое можно дописать и к v и так же получить суффикс S. ◀

Лемма 2.1.4. V_A — отрезок суффиксов.

Замечание 2.1.2. Самую длинную строку в V_A, то остальные строки в V_A будут ее суффиксами, причем подряд идущеми.

Пример 2.1.1. Есть строчка ababa: $R(aba) = R(ba) \neq R(a)$. $R(aba) = R(ba) = \{\epsilon, ba\}$ $R(a) = \{\epsilon, ba, baba\}$.

То есть вершины автомата — это классы эквивалентности, каждому классу эквивалентности соответствует отрезок суффиксов.

Конечные вершины: Давай-те понимать, где у автомата конечные вершины. У автомата точно есть вершина, в которой заканчивается строчка S. Это вершина $V_{\{\epsilon\}}$.

Эта вершина конечная. Этот класс — это некоторые суффиксы. Суффикс S и какие-то поменьше. И есть какой-то еще суффикс x, который принадлежит другой вершине V_A. И как нам находить эту вершину, мы будем поддерживать связь в виде суффиксной ссылки.

𝔇𝔢𝔣 2.1.5. $suf[V_{\{\epsilon\}}] = V_A$ — суффиксная ссылка.

$suf[v]$ — это вершина, которая соответствует наибольшему из суффиксов строки вершины v, котрый уже данному классу не принадлежит.

Конечными вершинами тогда являются : $last, suf[last], suf[suf[last]], \cdots$.

𝔇𝔢𝔣 2.1.6. Также будем поддерживать длину максимального суффикса, который заканчивается в этой вершине.
$len[v] = max|s| \colon s \in V$

Длину минимальной, если нам вдруг понадобится можно считать так: $len[suf[v]] + 1$.

2.2. Онлайн алгроитм построения за $O(n)$

В каких случаях разделяется вершина: Вершина это какой-то правый контекст $R_S(v)$. Теперь дописали символ a и правый контекст изменился следующим образом $R_{Sa}(v) = \{za | z \in R_S(v)\} + ?\epsilon$

От сюда видно, что если две строки заканчивались в разных вершинах, то если к строке дописать символ a, то они все еще будут заканчиваться в разных вершинах.

Если строки заканчивались в одной вершине, то что мы можем сказать, про то, заканчиваются ли они в одной вершине в новом автомате, то есть если $R_S(x) = R_S(y)$, равны ли $R_{Sa}(x)$ и $R_{Sa}(y)$.

Они могут быть не равны только если в одном из них есть ϵ, в другом нет.

Пусть $x \leqslant y$ и $R_{Sa}(x) \neq R_{Sa}(y)$. Значит в классе $R_{Sa}(x)$ содержится ϵ, а в $R_{Sa}(y)$ — нет.

На две поделится максимум одна вершина:

Лемма 2.2.1. Существует не более 1 $R_s(v)$, которая поделится попалам.

▶ Почему на два? Потому что он мог преобразоваться только двумя способами: с эпсилоном и без, то есть больше, чем на две части он точно не мог поделиться.
Рассмотрим x, такое что $|x|$ — max и содержит в правом контексте ϵ.
Это значит x — самый длинный суффикс Sa. А более короткие суффиксы Sa мы можем получить по суффиксной ссылке от этой вершины.
$V_{R_S}(x) \Leftrightarrow V[x]$.
То есть вершины, которые еще могут подойти под эту конструкцию заканчиваются в $suf[v[x]], suf[suf[v[x]], \cdots$.
Лемма 2.2.2. $R_S(v)$ — поделится на две $\Leftrightarrow \exists x \in R_S(v), y \in R_S(v)$
x — суффикс Sa
y — не суффикс Sa

◀

Про ребра: Если две вершинки не поменялись, то и ребра между ними не поменялись.
Почему? Если между вершинами было ребро, это значит, была какая-та страка x, которая заканчивалась в этой вершине. И была какая-та строка xb, которая заканчивает в соседней вершине. Вершина — это множество строчек, когда мы гворим, что вершина не поменялась, это значит, строка x все еще заканчивается в этой вершине. Вторая вершина тоже не поменялась, значит xb все еще заканчивается во второй вершине. Значит должно быть ребро из x в xb.
Нужно менять ребра только с новыми вершинами. А новых вершин будет максимум три. Старая поделилась на две и еще одна совсем новая.

Алгоритм поиска max|x| x — максимальный суффикс Sa и $x \subset S$.

Разделить нужно на несколько классов суффиксы Sa. Они удобны тем, что их легко перебирать. Если x суффикс Sa, то он заканчивается на a. Значит есть в вершину x какое-то ребро по a из вершины, например, w.

В более верхних слоях не могло быть ребра по символу a, иначе бы мы получили бы более длинный суффикс строки Sa.

То есть нужно идти по суффиксным ссылкам, пока не встретим первое ребро по букве a.

Как проверить, что все строки в данной вершине являются суффиксами Sa: То есть, нужно ли раздваивать вершину. Мы знаем, что |x| = |w| + 1.

Пусть строчка w лежит в вершине p, а x в q.

w максимальная строчка в своем классе, то есть |w| + 1 = Len[p] + 1. А максимальный размер строки в классе q равен Len[q].

Ну и остается проверить, совпадает ли он с длиной x.

Если равны, то раздваивать вершину не надо, если не равны, то надо.

2.3. Написание алгоритма

```
1 int last, suf[maxn], len[maxn], next[maxn][alf];
2
3 last = 1,
```

```
suf[last] = 0,
len[last] = 0;

void add(a) {
    p = last, last = new Vertex;
    Len[last] = len[p] + 1;
    while(p != 0 && Next[p][a] == 0) {
        next[p][a] = Last;
        p = suf[p];
    }

    p = suf[p];

    if (p == 0) {
        suf[last] = 1;
    } else {
        q = next[p][a];
        if (len[q] = len[p] + 1) {
            suf[last] = q;
        } else {
            r = new Vertex
            suf[r] = suf[q];
            suf[q] = r;
            suf[last] = suf[q] = r;
            len[r] = len[p] + 1;
            next[r] = next[q]
            while (p != 0 && next[p][a] = q)  {
                next[p][a] = r;
                p = suf[p];
            }

        }
    }
}
```

2.4. Оценка размера автомата

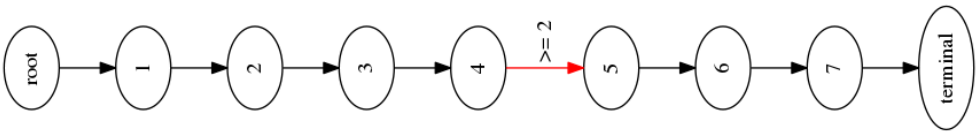
Теорема 2.4.1. В автомате $\leqslant 2n$ вершин и $\leqslant 3n-2$ ребра.

▶ С вершинами все просто. когда пустая строка - две вершины, а дальше по индукции будет добавляться не более двух вершин.

С ребрами чуть интереснее. Разобъем ребра на два класса.

1. Короткие ребра. Это такое ребро $AA \xrightarrow{a} B$, что $len[B] = len[A] + 1$
   Короткие ребра образуют дерево. Во-первых, поймем почему нет циклов. Пусть из A есть два пути по коротким ребрам до B. Эти два пути одинаоковой длины, потому что мы просто через len можем их узнать. Но это значит, что в B заканчиваются две строчки одинаковой длины, которые не равны. Противоречие.
   Еще надо понять, что это дерево, а не лес. Граф связен по построению. Добавляется новая вершина, она привешивается за единичное ребро. Перевешиваем мы не единичные ребра. И когда вершину раздваиваем к новой ведет еденичное ребро. Значит все дерево.

A в дереве на 2n вершинах 2n - 1 ребро.

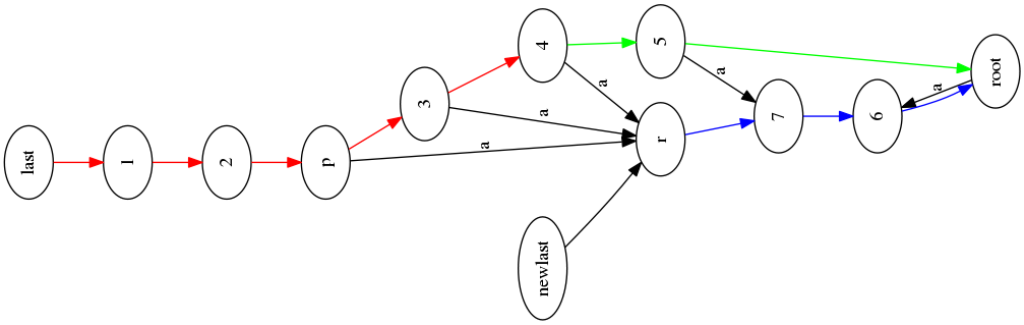
2. 

Длинные ребра. Рассмотрим ребро длины хотя бы два. От двух ее концов можно дойти до началькой и конечной вершины по единичным ребрам. То есть можем каждому ребру сопоставить суффикс. Причем каждому ребру свой суффикс, так как каждый суффикс задается единственным путем.
Значит длиных ребер $\leqslant n-1$.

Значит всего ребер $\leqslant 3n-2$. ◀

2.5. Оценка времени работы

Теорема 2.5.1. Время построения автомата $O(n)$.



▶

Обозначение:

1. красных ребер k штук.
2. зеленых - A штук.
3. синих - B штук.

Лемма 2.5.1. $B \leqslant A+1$

▶ Доказываем по принцепу Дирихле. В нижней часте все B вершин являются суффиксами строки Sa, значит в каждую должно вести ребро a. При этом в каждую из своей вершины. Значит в верхней части их хотя бы B - 1. ◀

$\Phi_{old} = A + k$
$\Phi_{new} = B + 1 \leqslant A + 2$

$a_i = t_i + \Delta\Phi = O(1)$

◀

2.6. Простейшие свойства

Мы научились строить автомат, доказали, что он быстро работает, осталось научиться его использовать.

У нас был текст, построили от него суффиксный автомат за линейное время ($text \to S.A(text)$).

Провекра, входит ли строка в текст Есть строчка S, хотим проверить, входит ли S в текст за $O(|S|)$.

Подстрока текста, это префикс какого-то ее суффикса. То есть, нужно просто съесть строчку S автоматом, и если были все переходы по дорогие и в конце мы оказались в какой-то, не обязательно терминальной, вершине, значит входит.

Подсчет количества вхождений строки S Обозничим v(S) — вершину, в которую мы попали, когда съели строчку S.

Теперь хотим посчитать, размер правого контекста вершины v(S). То есть, если мы прочитали строчку S, сколько способов дописать символы, что бы получить суффикс text.

Для этого посчитаем динамику. Пусть x_i все возможные вершины в которые можно попасть за один шаг из вершины v. $f[v]$ — размер правого контекста вершины. Тогда $f[v] = \sum_{x_i} + is_term[v]$.

Самое левое вхождение строки S При добавление вершины при построение суффиксного автомата, запомним в какой позиции мы добавили эту вершину. При расщипление вершины, позиция самого левого вхождения не меняется.

```
for (int i = 0; i < n; ++i) {
    add(text[i]);
    left[last] = i;
}
```

При расщипление `left[r] = left[q]`

Самое правое вхождение Это считается динамикой по автомату. Самое правое вхождение у последней вершины мы знаем. У остальных вершин надо взять максимум по соседям и вычесть 1.

2.7. Примеры задач

2.7.1. Наибольшая общая подстрока k строк

Наивный алгоритм: Можно строить автомат от, например, конкатинации. Взять строчки слить, еще через разные разделительные символы. Но так не делают, нужно строить автомат от минимльной строки, он будет есть куда меньше памяти и как следствие - быстрее работать.

Описание алгоритма:
1. Строим суффиксный автомат от минимальной из k строк. S.A(минимальна s_i)
2. Съедаем последующие строчки атоматом, переходя по суффиксным ссылкам, если не можем пройти по символу и в каждой вершине помним длину наибольшей общей строчки.

Обработка крайних случаев: Что бы не было неудобство с крайними случаями и всегда можно было откатиться до состояния, когда есть переход, у нас будет фиктивная вершин, из начала мы откатываемся именно в нее, а потом по любому символу переходим вперед.

Как обновлять длину проходя по суфссылкам: Если два раза прошлись по одной и той же вершине нужно запомнить первое или последнее вхождение? Нужно брать максимаьную длину из всех вхождений.

Есть текущая длина x, мы точно занаем, что эта длина x не больше чем len[v], но она может быть меньше. Мы же в вершину могли не по самой длинной строке прийти, а по любой. Когда мы откатываемся по суффиксной ссылке мы делаем $x = min(len[suf[v]], x)$.

Обощение, когда строчек все-таки не две, а k: Но это будет верно, только если мы пропускаем первую строчку через автомат. Теперь не первую.

Тогда будем хранить, ограничение с предыдущих строк $max[v]$ — максимальная длина подстроки, которая являлвсь общей для предыдущих вершин.

Изначально вершина принимает строки от некоторой минимальной длины, до максимальной, но общими являются только подстроки, которые не больше, чем $max[v]$.

Изначально $max[v] = len[v]$. Потому что строка была всего одна и все строки являются общими. Когда я прохожу через вершины автомата, я пересчитываю x.

Реализация: w[v] — максимальная общая подстрока текущей и нулевой строки которая заканчивается в вершине v.

Надо не забыть в конце пробросить по суффиксным ссылкам в обе стороны.

```
max[v] = len[v]
for (int g = 1; g < k; ++g) {
    v = root, x = 0, w = {0}
    for (int j = 0; j < s[g].size(); ++j) {
        while (go[v][s[g][j]] == -1) {
            v = suf[v];
            x = min(x, max[v]);
        }
        v = go[v][s[g][j]];
        ++x;
        w[v] = max(w[v], x);
    }

    vector<int> vs[len]; //список вершин с данной длиной.
    for (int i = len[last]; i >= 0; --i) {
        for (v : vs[i]) {
            max[v] = min(max[v], w[v]);
            max[suf[v]] = min(max[suf[v]], max[v]);
        }
    }
}
```

Время работы и память: Заметим, что первая часть деалется за $\Theta(s_k)$, а вторая часть за $\Theta(N)$(колич вершин в автомате, которое пропорционально длине наименьшей строки).

Время работы $O(\sum_{k=1}^{m} s_k + Nm)$

Память $O(N)$.

2.7.2. LZSS за O(n)

Определение: Есть строчка s. Для каждого i находм такое j < i, что k = LCP(s[j:], s[i:]) максимально. И выписываем j k, если k = 0, то выписываем симовол s[i].

Пример: Это метод сжатия.

Например для следующей строчки. s = aaaaabbbaaabb

Сначала я напишу букву a, потом повторю с 0 символа 4 буквы, потом зпишу символ b, потом начиная с позиции 5 повторю 2 буквы, потом начиная с позиции 2 повторю 5 букв.

lzss:
a
0 4
b
5 2
2 5

Алгоритм От автомата нам нужно, что когда мы уже выписали i букв находить такое j для которого k максимально.

В чем сложность, когда выписали первых i букв нужно максимум взять только по j, которые меньше чем i.

Заметим, что для i имеем право найти ответ за $O(k)$. Потому что после этого мы сделаем i += k. Значит суммарное время будет линейное.

Мы будет от начала автомата идти вперед. Пропуская i-ый суффикс. Пуская мы уже выписали k первых символов i-ого суффикса, и нам нужно проверить, что у строки, которую мы написали - самое левое вхождение левее чем i.

Строим автомт от строки и считаем left[v].

Что бы найти ответ для i идем по автомату до тех пор, пока left[v] $<$ i + k - 1. То есть проверяем, что левое вхождение строки левее позиции i.

2.8. Суффиксное дерево по автомату

Хотим сказать, что дерево может быть в чем-то удобнее автомата, но автомат нам дает дерево.

Теорема 2.8.1. Ребра из v в suf[v] — это ребра суффиксного дерева перевернутой строки S.

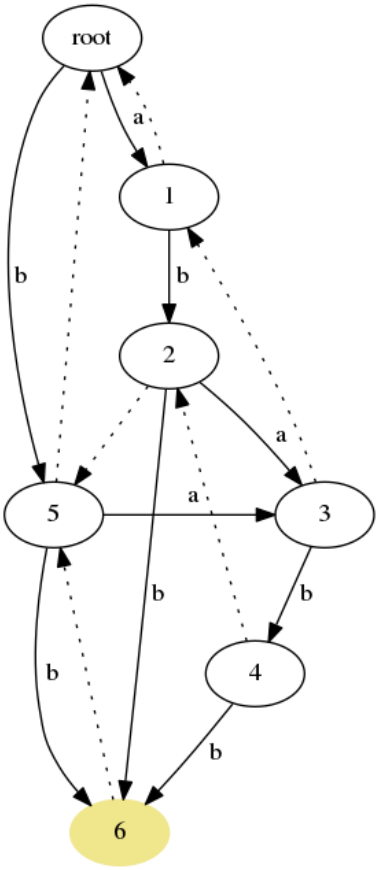
Если хотим дерево от строки S, то строим автомат от обратной строки, берем ребра из v в suf[v] и вот суффиксное дерево.

Заметим, что суффдерево можно не строить, если мы не собираемся его обходит DFS вниз.

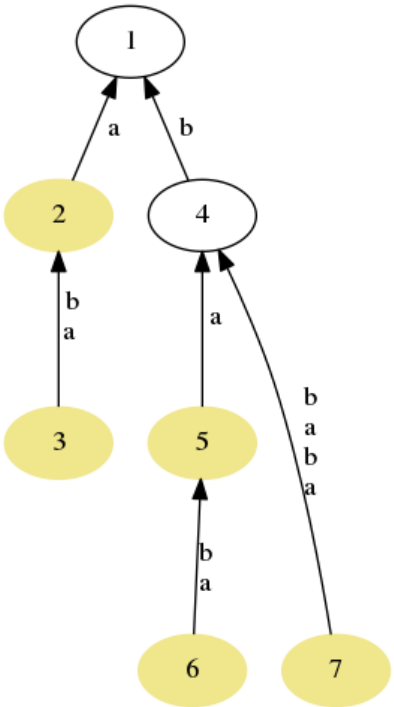
После того, как мы построили автомат у нас уже есть ссылки. Уже есть суффиксное дерево, которое просто другим способом задано.

Например, что хорошего есть в суффиксном дереве, чего нет в автомате. Там есть LCA = LCP. LCA в суффиксном дереве - это LCP двух суффиксов. Через автомат мы не можем взять два суффикса и как-то быстро найти LCP. Если мы хотим быстро код, то у нас есть массив отцов, и мы на нем насчитываем двоичные подъемы.

Пример: строим автомат от #ababb
и получаем суф дерево для строки: bbaba#



И рисуем суффиксные ссылки и по ним дерево.



Что написано на этом ребре. На этой вершине написано ababb, а на этой вершине написано b, значит на самом ребре написано baba. И именно она была отрезана.

Суффиксная ссылка которая переходит в строчку a отрезает ab, занчит написать надо ba.

Смотрите, как у нас интересно получилось. У нас есть суффикс a, ba, aba, baba, bbaba. Что в этом суфдереве может показаться странным. У нас есть вершина по середине ребра. Что бы такого не было, добавим в начало решетку.

Эта вершина по середине ребра означает, что у нас есть суффикс a и суффикс ba и они заканчиваются по середине ребер.

Строим от этой строчки автомат. # нужно, что бы листья не были по середине ребра.

Давай-те напишем код, как мы получали, какая строка написана на ребере.

У нас есть вершина v и вершина suf[v]. У них есть len[v] и len[suf[v]]. То есть мы знаем длину того, что нужно написать. Осталось только понять, где эта штука находится в строке.

v - это какая-та страка, в suf[v] это какой-то ее суффикс. А написать нужно строчку, которая их различает.

То есть нужно взять место, в котором начинается строка, которая написана на вершине v и выписать первые len[v] - len[suf[v]] символов. Мы умеем считать уже и первое и последнее вхождение, то есть какое-то мы можем взять.

И когда мы считали и первое и последнее вхождение - мы считали позицию конца. Что бы получить позицию начала - нужно вычесть длину.

В суффиксном дереве удобно знать, где заканчиваются суффиксы.

Получается что в вершинах. Это просто last по ходу построения. Действительно. Автомат у нас от обратной строки. То есть в автомате нам нужно смотреть на префиксы. Нужно знать, где префиксы заканчиваются - это last. После добавления каждого символа - запоминаем где был last, и множество last - это терминальные состояния.

И ни один last не является суффиксом другого last из-за #.

Доказательство теоремы: Давай-те теорему докажем, мы ее сформулировали, надо доказать.

Лемма 2.8.1. Последнее вхождение в любой строки - это первое вхождение в перевернутой. $Last_S(u) = First_{S^T}(rev(u)) = ST.path(rev(u))$

Тут Сережа запутался в переворачивание строк и пошло доказательсвто на пальцах..

У нас была строчка u и у нее была суффиксная ссылка. Теперь предствим перевернутый мир. Есть подстрочка rev(u) и есть suf[rev(u)]. А теперь рисуем суффиксное дерево, теперь суффиксная ссылка - это ссылка на префикс. Где-то в суффиксном дереве есть место, которое соответствует suf[rev(u)]. А из этого места мы можем еще дописать сколько-то символов и попасть в строчку rev(u). Значит, как минимум, они лежат на одном пути от корня в суффиксном дереве. Осталось сказать, что это вершины.

Каким образом у нас получаются вершины, есть другая строчка x. Так что суффиксная ссылка от x совпадает с suf[u].

Вот была строчка x и u и у них есть суффиксная ссылка, в перевернутом мире - это префикс. У них есть общий префикс.

Самый важный факт, что suf лежит на пути, а про вершины - это просто понимание.

У suf[v] более большой правый контекст. Нарисуем тот правый контекст, которого не было. Появилось новое вхождение. А теперь это новое вхождение продолжим на один символ. Либо

отличие в eps, тогда появится новая терминальная вершина, либо мы можем продолжить на один символ. Хотим сказать, что вершина, полученная дописыванием одного символа имеет такую же суффиксную ссылку.

Взяли вхождение v перешли по суф ссылки, у суфссылки больше правый контест, значит есть какое-то новое вхождение, взяли это новое вхождение.

Мы пытаемся доказать, что на suf[v] кто-то еще ссылается, это обязательно развилка.

Возбмем правый контекст suf[v] возьмем новое вхождение и расширем его влево. Суфиксная ссылка полученной строки совпадает с suf[v].

Невсегда можно расширить слева. Слева нельзя расширить, если слева ничего нет. Но тогда мы как раз терминал.

2.9. Перебор всех вхождений строки в текст

У нас есть текст, мы как всегда построили автомат от текста, нам подсовывают строчку s, а мы хотим найти все вхождения в текст за O(|S| + |Answer|).

Пока мы умеем делать одна операцию, взять из строки s пройти вперед и оказаться в некоторой вершине v. От вершины v мы могли что-то предподсчитать, но от вершины v есть очень много разных путей в терминалы. Каждый путь от вершины v в терминал задает новое вхождение строки s. И так как их много мы не можем просто так их хранить, нам нужно как-то их быстро получить.

В решение этой задачи как раз помогает неявное построение суффиксного дерева.

Представим, что у нас есть суфдерево. Тогда что бы вывести все вхождения строки s нужно просто обойти поддерево строки s.

Обойдем поддерево мы просто за размер поддерева.А вхождений хотябы размер поддерева/2. Потому что каждый лист задает новое вхождение.

Мы знаем связь между автоматом и деревом.

Берем суфиксное дерево. Которое мы построили методом взяли автомат перевернутой строки и запомнили `list[suf[v]].push(v)`.

Теперь нужно обойти поддерево дерева суфиксных ссылок.

Но при этом, кстати, не обязательно переворачивать явно, стока s входит тогда и только тогда, когда входяит в перевернутую строчку s перевернутое. Поэтому можно спокойно жить в перевернутом мире.

Глава 3

Автоматы

𝔇𝔢𝔣 3.0.1. Изоморфизм — если есть биекция по тому, как мы ходим.

𝔇𝔢𝔣 3.0.2. Эвкивалентность — принимают одни и те же строчки.

3.1. Быстрая проверка на эквивалентность

Если мы хотим проверить два автомата на эквивалентность, и один из них минимальный, то мы просто можем сделать синхронный dfs, запоминая, что для каждой вершины i автомата A она входит в класс, эквивалентный вершине j автомата $\min(A)$. Если по дороге конфликт – очень жаль.

3.2. Эквивалентность

Можно сделать оба минимальными, потом изоморфизм сделать. Но это грустно.

Сделаем так: нарисуем фиктивный корень, из него по специальной букве (или по ε, но буква `a` удобнее) два перехода: в начало первого и начало второго автоматов.

Минифицируем его. Если стартовые вершины исходных автоматов сжались в один класс — эквивалентны.

3.3. Минификация

Изначально есть два класса вершин — терминалы и нетерминалы. Далее, мы дробимся по классу R_1. Возьмём его прообраз по букве $V = a^{-1}(R_1)$, разделим его по всем классам:

$$\begin{matrix} V \cap R_1 & \bar{V} \cap R_1 \\ V \cap R_2 & \bar{V} \cap R_2 \\ \vdots & \vdots \end{matrix}$$

Осталось понять, как это делать быстро.

```
Q.push(T), Q.push(Q / T);
while (!Q.empty()) {
    R_i = Q.pop();
    for a in Σ {
        V = a^{-1}(R_i)
        // ???
```

```
    }
}
```

Мы стремимся к времени работы $O(|\Sigma| \cdot |E| \log |Q|)$. Как делаем поиск прообраза в лоб: перебором обратных рёбер. Мы хотим перебирать рёбра вершины $O(\log |Q|)$ раз. Таким образом, каждая вершина должна попасть в очередь $O(\log |Q|)$ раз.

Делаем так: когда мы захотим разбить класс на два, класть в очередь меньший класс. Почему? Потому что когда мы будем по этому классу резать, вторая половина получится сама собой.

Осталось дописать код:

```
Q.push(T), Q.push(Q / T);
while (!Q.empty()) {
    R_i = Q.pop();
    for a in Σ {
        V = a^{-1}(R_i)
        for u in V {
            list[class[u]].pb(u);
            non_zero_cnt.pb(class[u]);
        }
        for class in non_zero_cnt {
            cnt = list[class].size();
            if (cnt != size[class]) {
                // Если мы нашли не ВСЕ вершины данного класса
                if (cnt < size - cnt) {
                    // Добавить новый класс
                } else {
                    // Добавить новый класс, взяв его дополнение
                }
            }
        }
    }
}
```

Новый кусок работает за размер $O(V)$.

Глава 4

Паросочетания

У нас была лемма о дополняющем пути: в немаксимальном паросочетании есть дополняющий путь.

Также, есть лемма Куна: если из какой-то вершины не было дополняющего пути, то и не появится.

4.1. Алгоритм Эдмунса поиска паросочетания в произвольном графе.

Взяли свободную вершину v. Из неё идут рёбра в какие-то уже взятые вершины, по ним можно перейти в парные, из них снова по рёбрам в другие занятые и так далее. Если наш алгоритм не находит нечётный цикл — то всё как в двудольных графах и алгоритме Куна, нашли путь — есть путь.

Def 4.1.1. Соцветие: какой-то путь от свободной v, заканчивающийся вершиной, из которой идёт два незанятых ребра, называемый стеблем, и нечётный цикл на нём (включающий те два ребра), называемый циклом. **TODO** картинка!

Как видно, не все нечётные циклы подходят под это определение: **TODO** картинки!

Как же мы это ищем? Если поиск дополняющего пути нашёл нечётный цикл, мы начинаем откатываться к v до первого пересечения, вот и выписали соцветие.

Теорема 4.1.1. Можно сжать цикл в соцветии, дополняющий путь не пропадёт, а потом можно будет расжать обратно.

▶ Пусть мы нашли дополняющий путь после сжимания цикла. Посмотрим на то, входит ли в паросочетание ребро, выходящее из сжатого цикла. В зависимости от ответа выбираем правильную ветку цикла.

В другую сторону: из графа G сделаем G', инвертировав стебель. Размер паросочетания не поменялся, значит доппуть всё ещё есть. Сожмём $\bar{G'}$, там цикл — свободная вершина. Возьмём любой дополняющий путь в G'. При сжатии он или не задет, или его можно закончить в сжатом цикле. Теперь покажем, что и в $\bar{G}$ он есть: переключим стебель (xor-им). Получим дополняющий путь в $\bar{G}$. Успех. ◀

Можно доказать, что в том числе будет и путь из v.

Теперь сам алгоритм поиска пути за $O(NM)$:

- Запускаем поиск пути из V
- Нашли дополняющийся путь — радуемся

- Иначе, если нашли цикл в виде ребра из первой доли в первую долю — сжали соцветие, смотри пункт 1.

Сожмём не более n раз. Само сжатие можно сделать тупо за $O(M)$: перевыписать все нужные рёбра.

4.2. Реализация Габова за $O(N^3)$.

```
int match[N]; // парная вершина в паросочетании
int p[N]; // ссылки по восстановлению обратного пути; из 2 доли ведут только в 1
```

Пока ищем путь, входя в вершину второй доли (занятую) выставялем ей p. Нашли незанятую — радуемся, по p откатились.

Нашли нечётный цикл. Теперь выставляем p: по всему циклу выставляем ещё значения p-шек, встречные к существующим. Это даёт возможность по этим ссылкам вернуться из любой точки цикла.

```
queue q = { v:  p[match[v]] != -1 }
```

Инвариант: ссылки $v \to p[match[v]]$ образуют дерево, по которым мы умеем корректно возвращаться.

Далее, заведём массив оснований цикла $base$ — для каждой вершины где основание её цветка. Как находить? Ищем LCA в дереве $v \to p[match[v]]$ за линейное время, в лоб. Далее, на путях от замыкающего цикл ребра и включая нечётные петли на них, уже сжатые до этой компоненты, надо выставить новый $base$. Как это делать? Это ровно все те вершины, которые имеют встреченный по дороге старый $base$.

```
for x on path:
    visited[base[x]] = true;
for base_v in visited:
    base[base_v] = new_base
```

Ура, весь код! Там bfs, проще при изменении ссылок обрабатывать: тут код от Серёжи.

При перезаписи пешек: видяшка.

Время кубическое.

Небольшое упрощение: когда выставляем ссылки по уже сжатому циклу, можно не перевыставлять, а сразу в базу прыгнуть. Это выкинет слагаемое.

Глава 5

Планарные графы

𝔇𝔢𝔣 5.0.1. Планарный, плоский граф.

Теорема 5.0.1. Всякую грань можно сделать внешней.

Укладка на сферу.

Теорема 5.0.2 (Эйлера)**.**

$$E + C + 1 = V + G$$

Следствие 5.0.2.1. Если у каждой грани степень хотя бы 3, в каждой компоненте хотя бы 3 вершины, то $E \leq 3V - G$.

Теорема 5.0.3 (Куратовский, 1930)**.** Граф планарен тогда и только тогда, когда несуществует подграфа, гомеоморфного[1] K_5 или $K_{3,3}$.

Теорема 5.0.4 (Вагнер, 1937)**.** То же самое, но через стягивание[2].

Разные алгоритмы стягивания видят разные графы. Тут Куратовский видит $K_{3,3}$, а Вагнер — K_5.

Теорема 5.0.5 (Форш, 1948)**.** Существует укладка плоского прямыми отрезками.

Теорема 5.0.6 (Шнаудер, 1989)**.** Тем более, можно уложить вершины на сетку $(V-2)^2$.

5.1. Алгоритмы

1. Демукрон, 1964, $O(N^2)$. Planarity Testing. В контесте будет задачка, можно пользоваться и лекцией, и исходной статьёй.
2. Тарьян, Хопкрофт, 1974. $O(N)$. Ужасен по всему, но первый)
3. Boyer, 2004; Brandes 2011. $O(N)$, пишется и шустрое.
4. Теорема Татта: любой вершинно трёхсвязный можно уложить так, чтобы все рёбра — отрезки, следующим методом: каждую внешнюю и сопостовляем выпуклые многоугольники, и каждая не внешняя вершина — центр масс соседей, система линейных уравнений.

[1] удаление вершины степени 2 на ребро

[2] замена ребра на вершину, смежную со всеми их соседями

Побочный эффект всех алгоритмов: грани дают.

Пусть есть разбиение на грани двусвязного графа, хотим сделать временно трёхсвязным. Сделаем триангуляцию: взяли, и или провели диагональные рёбра, или добавили новую вершину в центр.

Как уложить несвязный? Покомпонентно. Как уложить связный? Или укладывать по точкам сочленения, или взять все компоненты вокруг, выбрали внешние циклы в каждой, соединили их по кругу.

Осталось решить систему линейных уравнений. Метод Гаусса — $O(V^3)$, жирно. Есть метод итераций, может помочь.

Пусть есть несколько вершин, лежат вперемешку, внутри внешней грани. Посчитали, где она должна лежать, передвинули. Сойдётся быстро, получается что-то вида $O(kE)$, нам ведь не честное решение нужно, а уложить граф.

После этого всего удаляем лишнее и радуемся жизни.

5.2. Демукрон

Есть две версии, из статьи и эта:

1. Выбрали цикл, «нарисовали».
2. Теперь взяли рёбра и разбили их на компоненты относительно цикла. Их можно укладывать независимо друг от друга, лишь бы выбрать, снаружи или внутри.
3. По одну сторону две компоненты можно, если концы, касающиеся цикла, одной их компонент лежат целиком между парой соседних концов другой компоненты. **TODO** . Это делаем так: взяли одну компоненту, расположили по циклу, перебрали другие компоненты вторым указателем, умещается ли она в текущей паре. Это делается за $O(K^2 + V)$.
4. Раскрасим этот граф совместимости в два цвета. $O(K^2)$. Если не получилось... Ну очень жаль. Дальше в каждой независимо.
5. Теперь, взяли компоненту, хотим её уложить. Взяли путь между двумя вершинами цикла, нарисовали его, разбили по двум цветам относительно него рёбра компоненты (какие можно слева, какие можно справа), продолжаем рекурсивно на всём слева и справа.

5.3. Планарные граф — окончание

5.3.1. Физическая модель

Рассмотрим физическую модель соответствующую планарному графу: пусть вершины, соединённые рёбрами, соответствуют некоторым точкам, которые связаны пружинками. Пусть длина пружинки 1, тогда

- Если $len > 1$, то точки притягиваются и $F \sim len^2$
- Если $len < 1$, то пусть точки отталкиваются с $F \sim e^{1/d} - 1$

Теперь хотим красивый плоский граф. Для этого будем пересчитывать состояние ситемы так (из-за экспоненты, в какой-то момент, система прийдёт в равновесие):

Пусть $R_k = R_0 \cdot e^{-t \cdot const}$, t — время

```
for i = 1..n:
    x[i] = (sum_j F[i][j]) * R[k]
```

Чтобы не образовывались сгустки вершин, граф иногда можно "встряхнуть"

```
x[i] += random * R[k]
```

5.4. Пересечение невыпуклых многоугольнков

Многоугольники — это плоские графы.
Пусть было 2 многоугольника по N вершин, напишем оценки на размер пересечения:

- $V \leq N^2 + 2N$ — максимальное число пересечений отрезков, второе число вершин в каждом исходном
- $E \leq 3V - 6$

Теперь сам алгоритм:

1. Берём каждое ребро первого многоугольника и пересекаем его со вторым многоугольником. Точки пересечения сортируем по расположению на отрезке, соседние нужно соединить ребром. Кроме того, вокруг каждой вершины нужно упорядочить рёбра по углу.
2. Исходный граф плоский, значит грани пересечения — это циклы того, что нарисовали. Начинаем идти по ребру в каком-то направлении, зашли в вершину, выходим по следующему ребру в отсортированном порядке. Когда зациклимся получим грань. Если пойдём по ребру в противоположном направлении, то получится другая грань. Нужно заметить, что у внешней грани ориентация будет отличаться от остальных.

```
for edge:
    if (!used[edge]):
        cc++
        for (i = edge; !used[i]; i = next[i]):
            used[i] = cc;
```

Ещё нужно запоминать какой грани какое ребро принадлежит, чтобы построить двойственный граф.

3. Теперь осталось только понять, какие грани входят в пересечение, какие нет. Для каждого ребра посчитаем маску в какую фигуру оно входит (т.е. число от 0 до 4, но 0 никогда не бывает). Теперь нам нужен двойственный граф (вершины — грани, смежные соединены рёбрами). В нём мы начинаем идти с внешней грани, DFS-ом. Для каждой грани тоже заведём маску какой фигуре оно принадлежит (для внешней она 0) Тогда маска грани определяется в обходе DFS $mask(p) \otimes mask(edge)$, где p — это грань из которой мы идём.

Оценка на время: $T = N^2 \log N + N^2 \cdot |\frac{K}{w}| + KN^2 \log N + KN^2$

5.5. Локализация точки

Дан плоский граф, нужно определить какой грани оно пренадлежит.

5.5.1. Online

Будем делать scanline, поэтому вначале запрещаем вертикальные отрезки (либо честно выбираем хороший угол, либо выбираем случайный и поворачиваем на него) и разбиваем граф на грани.

Преподсчитаваем структуру: для каждой вертикальной прямой список отрезков, которые её пересекают, упорядоченные по y. сначала удаляем входящие рёбра, потом добавляем исходящие.

TODO рисунок, с ним всё будет понятно

Нужно быть аккуратным если несколько вершин имеют одинаковый x, нужно будет сначала удалить все входящие во все вершины, а потом добавить все исходящие.

Теперь провели вертикальную прямую, через точку пересечения, она пересекает, какие-то рёбра, вершины, их можно получить из значений для прямой, в нашей структуре которая левее локализуемой точки:

Если локализуемая точка крайняя, то грань внешняя. Иначе мы смотрим на рёбра, ограничивающие её снизу и сверху, смотрим какой грани они принадлежат.

5.5.2. Offline

Теперь у нас есть какой-то набор локализуемых точек. Событими будут: $x_1 < x_2 ... < x_n$.

Поступим почти аналогично предыдущему случаю: заведём персистентное дерево по y, для рёбер пересекающих прямую.

```
// x*, y* - кооординаты локализуемой точки
get(x*, y*):
    i = lowerbound(x, x*); // находим нужный вариант состояния дерева
    root[i] -> get(y*);      // находим какими отрезками ограничена наша точка
```

Глава 6

Рандомизированные алгоритмы

6.1. Лас-Вегас

тянем случайный и проверяем правильность

6.1.1. Простой пример

Есть массив, хотим взять элемент, который в отсортированном стоит на позиции $\geq \frac{n}{2}$ Возьмём случайный элемент с вероятностью $P \geq \frac{1}{2}$ он подойдёт.

6.1.2. Арифметическая прогрессия

Дан массив, нужно выбрать подмножество $|A| \geq \frac{n}{2}$ элементов, образующих арифметическую прогрессию.

Ткнём в 3 элемента. С вероятностью $P > \frac{1}{8}$, они образуют арифметическую прогрессию.

Вероятность, что x входит в арифметическую прогрессию:

$$x = a + dk \Leftrightarrow x - a = dy \Rightarrow y = gcd(|a_j - a_i|, |a_k - a_i|)$$
$$P\{gcd(j - i, k - i) = 1\} > 0$$

Значит можно вытаскивать случайно элемент, проверять, что он входит в прогрессию и если нет, то вытаскивать снова.

6.2. Монте-Карло

посчитать что-нибудь

6.2.1. Площадь пересечения

Классический пример: пусть есть две фигуры на плоскости в какомто ограничивающем квадрате хотим найти пересечение. Умеем проверять принадлежность точки пересечению. Будем много раз тыкать в наш квадрат и будем приближать вероятность как отношение попаданий ко всем попыткам. Тогда площадь можно считать как мат. ожидание: $S_I = \frac{hit}{N} S$, где N — все попытки, hit — попадания, S — площадь ограничивающего множества.

Давайте посмотрим на погрешность, нам нужна дисперсия: $D = Np(1-p)$, т.к. $X \sim Bin(p, q)$

Теперь оценим погрешность так: $\frac{\sqrt{Np(1-p)}}{Np} = \theta(\frac{1}{\sqrt{N}})$ Т.е. при большом N всё хорошо.

6.2.2. Площадь пересечения, детерминированно

Теперь рассмотрим другое решение, будем делать что-то похожее на Жорданову меру: поместим наши фигуры в большой квадрат, и будем его дробить по серединам сторон, пока дробление не станет меньше какого-то ϵ. При дроблении у нас возможны следующие варианты:

1. квадрат находится внутри фигуры: для выпуклых многоугольников достаточно проверить, что все вершины квадрата внутри.
2. квадрат пересекает фигуру: это происходит, если есть нунулевая проекция на одну из сторон квадрата.
3. иначе квадрат содержит фигуру.

Тогда можно написать такую рекурсивную функцию:

```
go(Root)
if Root is Inside:
    return 1
if Root is Outside:
    return 0
if S(Root) < eps:
    return S(Root)/2;
return 1/4(go + go + go + go)
```

Погрешность $\frac{1}{N}$, где N — это число листьев завершившихся на 3-ем шаге.

Зачем про него вспомнили: при малых размерностях детерминированный вариант лучше: нет рандома и погрешность меньше. Но нам приходится делать 2^n вызовов если $\dim = n$, что становится плохо при больших размерностях. С большими размерностями придётся столкнуться при машинном обучении, поэтому там лучше использовать Монте-Карло.

6.3. Хеширование-Кукушка

Добавление за рандомизированное $O(1)$, удаление и поиск за детерминированное, амортизированное $O(1)$.

Идея: у нас будет 2 хеш-функции (где брать хорошие? см совершенное хеширование)

```
find(x):
    return X[H1(x)] == x || X[H2(x)] == x
```

```
del(x):
    if X[H1(x)] == x:
        x[H1(x)] = -1
```

Добавление: как обрабатывать коллизии? Мы перекладываем элемент, с которым столкнулись, в ячейку $H1(y) + H2(y) - H1(x)$. Если там образовалась коллизия, то мы продолжаем перекладывать. Возможны несколько исходов:

- Мы нашили свободную ячейку и всё радуемся жизни.
- Цепочка вытеснений вытеснила элемент из таблицы.
- Цепочка вытеснений зациклилась. Как научиться понимать, что мы зациклились?

– Идём двумя указателями с шагами разной длины, если они внезапно оказались в одной точке, то мы зациклились.
– Если есть память, то запомнить цепочку ходов

Утверждение 6.3.1. Не зациклимся с большой вероятностью

▶ Пусть $0 < H_1, H_2 < n$ и мы хотим добавить m элементов. Если m маленькое, то циколов не будет: $n > 5m \Rightarrow E(\text{кол-во циклов}) \le \frac{1}{2}$ $E = \sum_{\text{по всем циклам полного графа}} P(\text{цикл есть})$
TODO ◀

Следствие 6.3.0.1. За $O(n)$ рандомизированное найдём по $x_1, \dots, x_n$ такую f, что $f : x_i \mapsto i$ за $O(1)$.

Если случилась плохая ситуация, то меняем хеш-функции.

6.4. Пересечение полуплоскостей

У нас есть система неравенств, хотим найти все x, что

$$\begin{cases} Ax \le 0 \\ cx \leftrightarrow 0 \end{cases}$$

(dim = 2)

Ниже решение за рандомизированное $O(n)$:

Замечание 6.4.1. Живём в bound box

1. random shuffle полуплоскостей
2. добавляем по одной полуплоскости, возможные варианты:
   - текущий ответ лежит в полуплоскости, тогда ничего делать не нужно.
   - ответ лежит на прямой, задающей полуплоскость

```
for i = 0..n-1:
    if (ans not in halfPlane):  // P <= 2/i
        ans = solve(i, 0..i-1); // Theta(i)
```

здесь `solve` решает задачу в меньшей размерности: нужный кусок границы — это пересечение лучей от точек пересечения старого ответа и новой прямой.

TODO рисунок

Лемма 6.4.1. вероятность, что ответ лежит на прямой $P \le \frac{2}{i}$

▶ точка пересечения, входящая в ответ задаётся 2-мя полуплоскостями, значит последняя, должна задавать эту точку. ◀

6.4.1. Обобщение на большие размерности

В текущем виде алгоритм обобщается на большие размерности, поменяется только оценка на время:

Теорема 6.4.1. Для размерности dim = d мы решаем задачу за $n \cdot d!$

▶ Доказательство по индукции, база $d = 2$, переход: $T_d = \sum_{i=1}^{n} \frac{d}{i} T_{d-1}(i) = \sum_{i=1}^{n} \frac{d}{i} \cdot (d-1)! \cdot i = n \cdot d!$
◀

6.5. Покрывающий круг

Даны точки на плоскости, нужно найти круг минимально радиуса, который накроет их всех.

Аналогично предыдущей задаче алгоритм состоит из 2х частей: random shuffle точек и итеративное изменение ответа:

```
solve_1():
    // строим стартовую окружность на P[1], P[2] P[3]
    // нумерация с 1
    for i = 4..|A|:
        if (P[i] not in ans):  // P <= 2/i
            ans = solve_2(i, i-1) // Theta(i)
```

☒ Пусть есть множество точек X и для них известен оптимальный ответ и есть точка P которая, не покрывается оптимальным ответом для X, тогда P лежит на границе оптимального ответа для $X \cup \{P\}$.

▶ Пусть не так, тогда P лежит внутри оружности, но мы знаем, что окружность задаётся либо 3-мя точками – это противоречит тому, что P не покрывалось оптимальным ответом для X, либо 2-мя точками, это либо противоречит оптимальности ответа, если не на диаметре, то можно чуть-чуть уменьшить, либо снова тому, что ответ для X не покрывал P. ◀

```
solve_2(x, A):
    // инвариант: точка "x" уже на границе ответа
    // строим стартовую окружность на x, P[1], P[2]
    for i = 3..A:
        if (P[i] not in ans):  // P <= 2/i
            ans = solve_3(x, i, i-1) // Theta(i)
    return ans

solve_3(x, y, A):
    // инвариант: точка "x", "y" уже на границе ответа
    // строим стартовую окружность на x, y, P[1]
    for i = 2..A:
    if (P[i] not in Ans):  // P <= 2/i
            ans = solve_4(x, y, i) // Theta(i)
    return ans

solve_4(x, y, z):
    // x, y, z однозначно задают окружность
    // находим центр центр описанной окружности
    retrun ans
```

Лемма 6.5.1. вероятность, что очередная точка не принадлежит ответу $P \leq \frac{3}{i+1}$

▶ окружность однозначно можно задать 3-мя точками ◀

Мат. ожидание времени работы оценивается так: $\sum \frac{3}{i} \cdot i = \theta(n)$ Вначале для `solve_3`, затем для `solve_2` и для `solve_1`. Правда так считать можно, только если функции независимы, т.е. если мы делаем shuffle в начале каждой из них. Но даже если сделать только 1 раз, всё хорошо работает.

6.6. Рёберная 3-связность

Граф рёберно 3-связан тогда и только тогда, когда нет 2-разреза.

Рассмотрим произвольное остовное дерево, расставим на рёбрах неостова произвольным образом числа 0 и 1. Хотим, чтобы было чётное число нулей у всех вершин. Для этого красим остальные рёбра дерева. Такая расраска есть: пойдём с листьев, а дальше по индукции.

☒ Посмотрим на 2-разрез. Рёбра покрашены в один цвет.

▶ **TODO** ◀

Алгоритм:

1. рандомно покрасили неостов
2. жадно докрасили остов
3. повторим k раз
4. для каждого ребра храним историю раскрасок
5. если у какой-то пары одинаковая история, то говорим, что это 2-разрез

Алгоритм с одосторонней ошибкой: $P[error\ (e_1, e_2)] = (\frac{1}{2})^k$

Повторим столько раз, чтобы ошибка стала нулевой $P[error] < E = \frac{m(m-1)}{2}(\frac{1}{2})^k) < \epsilon$, т.е. $k \sim \log m$

Итого, время $O(m \log m)$ вероятность ошибки $P < \frac{1}{2}$

6.7. Random walk

Найдем паросочетание в d-регулярном двудольном графе рандомизированно за $O(n \log n)$

По лемме Холла совершенное паросочетание существует.

Будем искать дополняющий путь случайным блужданием, то есть из свободных вершин идем по случайному ребру.

Пусть текущий размер паросочетания k, тогда будем запускать Random walk на глубину $\frac{3n}{n-k}$, пока он не найдет дополняющий путь.

Вероятность того, что один такой Random walk найдет путь больше некоторого $\epsilon > 0$ (без доказательства). Значит матожидание числа запусков для одного k равно $\frac{n}{\epsilon}$

Время работы это $\sum\limits_{k=0}^{n-1} \frac{3n}{n-k}$, т.е. $O(n \log n)$

Глава 7

Полуплоскости

7.1. Нахождение пересечения полуплоскостей

Хотим найти пересечение полуплоскостей за $O(n \log n)$

Найдем отдельно пересечение всех полуплоскостей вида $y \leqslant k_i x + b_i$ (нижние полуплоскости) и отдельно всех полуплоскостей вида $y \geqslant k_i x + b_i$, а затем получим из них общее пересечение.

Научимся искать пересечение полуплоскостей вида $y \leqslant k_i x + b_i$

Отсортируем полуплоскости по k_i, это то же самое что сортировка по углу. Затем сделаем проход в отсортированном порядке со стеком.

Будем поддерживать стек прямых и координат точек пересечения двух соседних прямых в пересечении первых i полуплоскостей. Чтобы было меньше случаев, положим в начало стека вертикальную прямую далеко слева. При добавлении следующей полуплоскости достаем со стека полуплоскости, пока их точки пересечения лежат в новой полуплоскости

```
$while (y.top() > k_i \cdot x.top() + b_i) {
    x.pop();
    y.pop();
    lines.pop();
}
x.push(intersectionX);
y.push(intersectionY);
lines.push(newLine);$
```

Лемма 7.1.1. $x_i, y_i \in Z \cap [-C; C] \Rightarrow$ чтобы не было проблем с точностью, в худшем случае нужен тип хранящий числа порядка C^3. Без доказательства.

Этот алгоритм похож на построение верхней части выпуклой обоочки.

7.1.1. Биекция с задачей нахождения выпуклой оболочки

Чтобы построить верхнюю часть выпуклой оболочки, мы сортируем все точки по координатам, затем делаем проход со стеком, доставая точки, пока угол с предыдущим отрезком больше 2π.

Теорема 7.1.1. Существует биекция между задачами нахождения пересечения полуплоскостей вида $y \geqslant k_i x + b_i$ и нахождения верхней части выпуклой оболочки.

Построим сначала биекцию между прямой $y = kx + b_i$ и точкой (x, y):

$$(x, y) \to (k = x, b = -y)$$

$$(k, b) \to (x = k, y = -b)$$

Лемма 7.1.2. Биекция сохраняет отношение "ниже"

▶ Точка (p_x, p_y) лежала под прямой $y = kx+b$, значит после преобразования, прямая $y = p_x x - p_y$ лежит под точкой $(k, -b)$, т.к. $p_y < kp_x + b$ ◀

Лемма 7.1.3. Прямая с точкой лежащей на ней переходит в точку лежащую на прямой.

Лемма 7.1.4. Точка пересечения двух прямых переходит в прямую проходящую через точки, в которые перешли прямые.

Лемма 7.1.5. Верхняя часть выпуклой оболочки переходит в пересечение полуплоскостей вида $y \geqslant k_i x + b_i$

▶ Следует из предыдущих лемм.

Прямая из пересечения полуплоскостей переходит в точку выпуклой оболочки и наоборот. А прямая между соседними точками выпуклой оболочки переходит в точку пересечения соседних прямых из пересечения полуплоскостей.

Точка внутри выпуклой оболочки ниже всех прямых между соседними точками, значит она перейдет в прямую, которая будет выше всех точек пересечения соседних прямых и эта прямая не будет принадлежать границе пересечения полуплоскостей. ◀

Следствие.

Умеем пересекать полуплоскости за то же время, что и строить выпуклую оболочку. Например для выпуклой оболочки есть алгоритм "заворачивания подарка работающий за $O(nk)$, где k это число точек на выпуклой оболочке.

7.1.2. Общий случай пересечения полуплоскостей

Мы научились отдельно искать пересечение верхних полуплоскостей и нижних полуплоскостей, теперь получим общее пересечение.

Чтобы не было вертикальных прямых, повернем все на случайный угол, точнее применим любое невырожденное афинное преобразование $(a, b, c, d \neq 0)$:

$$x \to ax + by$$

$$y \to cx + dy$$

Идем двумя указателями слева направо по полуплоскостям верхней и нижней части. Пересекаем отрезки.

Можно таким же методом найти все точки пересечения двух монотонных слева направо ломаных за время $O(n + Answer), Answer < 2n$

Глава 8

Выпуклые многоугольники, динамическая выпуклая оболочка

8.1. Простые задачки на выпуклые многоугольники

8.1.1. Точка внутри многоугольника?

1. Выберем самую левую точку многоугольника. Проведем из нее лучи во все остальные точки. Таким образом аля триангулировали многоугольник. Теперь проведем луч в точку запроса. Хотим понять, в каком треугольники она потенциально может лежать. Бин-поиск по углу.

2. Нашли треугольник. Теперь за $O(1)$ проверяем, внутри ли она (какое-нибудь в.п.)

8.1.2. Крайняя точка по направлению

Задача: Дан выпуклый многоугольник и направление. Грубо говоря нужно найти такую точку в многоугольнике, что по заданному направлению, остальные будут "за ней".

Итак, есть в.м., зафиксируем его обход (например против часовой стрелки). Направление задается нормалью. Рассмотрим прямую, перпендикулярную нормали (пусть у нее угол B), и $\alpha[i]$ - углы, соответствующие ребрам, выходящим из ввершин. Если вершины многоугольника пронумерованы таким образом, что угол у нулевой вершины минимален, то нужная вершина - это $lower\_bound(\alpha[], B)$.

8.1.3. Касательная к многоугольнику из точки

Строим касательные из точки A. Зафиксируем ориентацию и точку многоугольника (a). Поймем, эта точка лежит на ближней к нам стороне, или на дальней (в.п.). Пусть на дальней. Таким образом вершины разбились на те, что справа (если смотреть на a) и на те, что слева.

Найдем правую касательную (соответствующую ей точку B). Сделаем бин-поиск по "правым" вершинам. Пусть текущаяя вершина x. Тогда до B знак в.п. луча (A, x) и ребра исходящего из x будет одним, а после B - другим.

Проверять, что мы попали в левую долю можно векторным произведением относительно (A, a)

```
L = 0, R = n;
while (R - L > 1) {
    M = (L + R) / 2;
    if ((p[M] - A)x(P[0] - A) < 0        // Проверка того, что мы в правой части
        && (P[M] - A)x(P[M + 1] - P[M]) > 0)    // Где мы относительно B
        L = M;
```

```
7       } else {
8           R = M;
9       }
```

8.1.4. Пересечение многоугольника и прямой

Возьмем далеко на прямой v точку A. Посмотрим на вершину многоугольника x и знак в.п. прямой с (A, x). Кайф. Все вершины разбились на те, что с одной стороны и те, что с другой. Фактически имеем что-то вроде $[1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]$. Нам надо найти границы. Просто бин-поиск сделать не можем, т.к. не факт, что первая и последняя вершины многоугольника имеют разный знак. Проведем касательные из A и по ним "разрежем" массив $[1, 1, 1, 1, 1, 0, 0, 0, 0]$ и $[0, 0, 0, 1, 1, 1, 1]$. Теперь можем сделать бин-поиск.

8.2. Общие касательные к двум в.м.

Многоугольники не пересекаются и не лежат друг в друге

8.2.1. Ближайшая к точке вершина многоугольника

Есть в.м. и точка A. Найдем ближайшую к ней вершину. Для этого из A проведем касательные к многоугольнику, тем самым разобьем его на две части: ближнюю и дальнюю. Рассмотрим ближнюю. На ней расстояние от A до точек многоугольника выпукло, значит можно сделать тернарный поиск. На паре формально доказывалось, почему это так. Рассуждения велись от: "Пойду в одну сторону, расстояние уменьшится, т.к. угол тупой; пойду в другую - угол острый, расстояние увеличится".

8.2.2. Ближайшие точки для двух многоугольников

За $O(\log^2 n)$.

1. Возьмем x_0 из первого многоугольника и найдем для него ближайший y_0 из второго.
2. Проведем касательные из y_0 и рассмотрим ближнюю половинку первого многоугольника.
3. С этой половинки запустим тернарник, минимизирующий расстояние до второго многоугольника. Успех.

8.2.3. Общие касательные

1. Найдем x_{ans} и y_{ans} предыдущим алгоритмом.
2. Проведем через них прямую, по которой "разрежем" многоугольники.
3. Построим отдельно касательные для верхних и нижних половинок за $O(\log^2 n)$.

Научимся строить общую касательную для верхней половинки. Касательная будет задаваться парой точек (A, B) - из первого и второго многоугольника. Таким образом мы можем легко проверить, является ли прямая касательной: ребра смежные с A и смежные с B должны находиться по одну сторону от (A, B).

Сделаем бинарный поиск по вершинам из первого многоугольника (вернее его верхней половинки). Из очередной вершины строим касательную и смотрим:

1. Если полученная прямая действительно является касательной, то нашли.
2. Если это дальняя до B вершина, то двигаем правую границу (можем проверить, если зададим ориентацию многоугольника).
3. Иначе левую.

Полезно знать, что существует алгоритм, находящий их за $O(\log n)$

8.3. Динамическая выпуклая оболочка

Хотим за $Poly(\log)$ уметь:

- Add(x, y)
- Del(x, y)
- Количество вершин в в.о. (например)

У нас будет два дерева отрезков: для верхней части в.о. и для нижней. Деревья отрезков по X. Считаем, что все X различны.

Будем говорить о верхней части. В вершинах д.о. хранятся декартовы деревья для выпуклых оболочек (в данном случае для их верхних частей). Чтоб получить значение в корне, нужно найти верхнюю общую касательную для сыновей, посплитить по вершинам этой касательной и добавить в корень их объединение (merge).

Чтоб избежать персистентности, в детях будем оставлять грызки - то, что осталось после подсчета значения для корня. Теперь, при спуске в детей, нужно родителя посплитить по соответствующей координате и, таким образом, их починить.

Оценим время работы очередного запроса: $O\left(\log C \left(\log n + \text{общ. касательная}\right)\right)$

Если не думать, то общую касательную мы находим за $O(\log^3 n)$. Если понять, что бин-поиск - это спуск по дереву, то за $O(\log^2 n)$. Держим в уме, что существует алгоритм, делающий это за $O(\log n)$.

Оценка на память: $O(n \log C)$.

8.4. Линейные рекурренты

8.4.1. Решение линейных рекуррент

Хотим решить линейную рекурренту вида $f_n = \sum\limits_{i=1}^{k} a_i f_{n-i}$.

Умеем решать тремя способами (два из которых мы знали и раньше):

1. $O(nk)$

   Просто считаем очередное значение f_i с помощью цикла длины k.

2. $O(k^3 \log n)$

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ \vdots \\ f_{n-k+1} \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \\ \vdots \\ f_{n-k} \end{pmatrix}$$

3. $O(k \log k \log n)$

   Заметим, что f_n как-то выражается через первые k членов нашей последовательности f:

   $f_n = f_{k-1}x_{k-1} + f_{k-2}x_{k-2} + \cdots + f_0 x_0$.

   Теперь наша задача — быстро найти x_i.

   $f_n \to \sum_{i=1}^{k} a_i f_{n-i}$.

   Давайте переобозначим f_n за z^n.

   $z^n \to \sum_{i=1}^{k} a_i z^{n-i}$.

   В частности, $z^k \to \sum_{i=1}^{k} a_i z^{k-i}$.

   То есть, на самом деле $z_k \to z_k \mod P(z)$, где $P(z) = z^k - \sum_{i=1}^{k} a_i z^{k-i}$.

   Это верно для любого n: $z_n \to z_n \mod P(z)$.

   Заметим, что $\deg P = k$, поэтому все операции с этим многочленом (например, деление и взятие по модулю) мы умеем делать за $O(k \log k)$.

   Теперь осталось быстро посчитать $z^n \mod P(z)$, сделаем это с помощью бинарного возведения в степень. То есть каждый шаг мы возводим текущий многочлен в квадрат (или умножаем на z), а затем берем по модулю $P(z)$. В любой момент времени степень нашего многочлена $O(k)$, а всего операций будет $O(\log n)$, поэтому итого все будет работать за $O(k \log k \log n)$.

8.5. Диаграмма Вороного

8.5.1. Постановка задачи

На плоскости дано n точек. Для каждой из точек есть часть плоскости (возможно, бесконечная), для которой эта точка — ближайшая среди всех. Наша задача — разбить всю плоскость на такие области.

Пример 8.5.1. Если нам дано две точки, то нам просто нужно разбить плоскость на две полуплоскости, проведя серединный перпендикуляр между исходными точками.

Этот частный случай поможет нам решить задачу для произвольного n.

Кроме того, любая диаграмма Вороного на самом деле представляет из себя планарный (даже плоский) граф, в каждой грани которого находится одна из n исходных точек.

8.5.2. Алгоритм за $O(n^2 \log n)$

Зафиксируем точку i, хотим найти часть области, относящуюся к ней (она также называется "ячейкой диаграммы"). Как мы поняли для двух точек, их всегда разделяет серединный перпендикуляр.

То есть для каждой из оставшихся точек j проведем серединный перпендикуляр между точками i и j и направим нормаль по направлению к точке i. Теперь нам осталось просто пересечь все получившиеся полуплоскости, а это мы умеем делать за $O(n \log n)$.

Итого для каждой из n точек мы пересекаем $n-1$ полуплоскость и получаем итоговую асимптотику $O(n^2 \log n)$.

8.5.3. Алгоритм за $O(n^2)$

Но, как оказывается, существует алгоритм за $O(n^2)$ (и который на практике проще пишется).

Он основан на идее "заворачивания подарка". Сразу отметим, чтобы избежать проблем с точками на бесконечности при реализации, удобно воспользоваться идеей Bounding Box.

Давайте вновь, как и в прошлом алгоритме, для каждой точки выделять ее ячейку диаграммы.

Зафиксируем точку i и рассмотрим ближайшую к ней j. Заметим, что точка $\frac{P_i+P_j}{2}$ точно лежит на границе ячейки (т.к. точка j ближайшая к i).

Будем идти от этой точки по направлению серединного перпендикуляра и заворачиваться по часовой стрелке. Пусть мы уже прошли какой-то путь, у нас сейчас есть какое-то направление. Из всех серединных перпендикуляров между i-й и k-й точками выберем тот, точка пересечения которого с лучом (именно лучом, а не прямой) направления — ближайшая к текущему концу пути (если таких несколько, то выберем из всех минимальный по углу, то есть тот, при котором мы "заворачиваем" больше всего).

Итого, просто на очередном шаге мы рассматриваем $n-1+4$ прямые ($n-1$ серединных перпендикуляра и 4 прямые Bounding Box-а) и выбираем наилучшую.

Казалось бы, работать это будет за $O(n^3)$, но нет, итоговая асимптотика $O(n^2)$ из-за того, что в диаграмме Вороного $O(n)$ ребер (а именно столько раз мы и делаем выбор очередной прямой).

Лемма 8.5.1. В диаграме Вороного $O(n)$ ребер.

▶ Как мы упоминали в самом начале, граф, соответствующий диаграме, планарен, поэтому в нем выполняется формула Эйлера: $V+G=E+2$.

Кроме того, вершины в диаграме появляются только, когда соприкасаются $\geqslant 3$ ячейки, поэтому $\forall v : \deg v \geqslant 3$.

$2E = \sum\limits_v \deg v \geqslant 3V \Rightarrow E \geqslant \frac{3V}{2}$.

$V+G = E+2 \geqslant \frac{3V}{2}+2 \Rightarrow n = G \geqslant \frac{V}{2}+2 \Rightarrow V \leqslant 2n-4 \Rightarrow E \leqslant 3n$.

Итого, показали, что ребер $O(n)$. ◀

Стоит отметить, что существует алгоритм за $O(n \log n)$, о котором нам должны рассказать на курсе вычислительной геометрии.

8.6. Факторизация

8.6.1. Факторизация чисел

Хотим научиться факторизовать (то есть раскладывать на простые) числа за $2^{o(\log n)}$.

Уже умеем это делать за $n^{1/4} = 2^{\log n/4}$ с помощью алгоритма ρ-Полларда.

Также есть вероятностный алгоритм проверки на простоты — тест Миллера-Рабина, который работает за $\log n$. Примерно так работает $isProbablePrime()$ в $Java$ и похожая функция в $NumPy$.

8.6.2. Алгоритм Крайчика

Наша текущая задача — найти любой нетривиальный делитель n, то есть такое x, что $\gcd(n,x) \neq 1$ и $\gcd(n,x) \neq n$. После этого мы можем запуститься от x и n/x, например, рекурсивно.

Рассмотрим последовательность $x_i = \lfloor \sqrt{n} \rfloor + i$ и $y_i = x_i^2 \bmod n$, понятно, что можно оценить y_i, примерно как $2i\sqrt{n}$.

Мы хотим найти такой набор $\{i_k\}$, что $\prod\limits_{i_k} y_{i_k} = z^2$ для какого-то z.

$z^2 = \prod\limits_{i_k} y_{i_k} = \prod\limits_{i_k} x_{i_k}^2 \underset{n}{\equiv} (\prod\limits_{i_k} x_{i_k})^2$.

Если обозначить $\prod\limits_{i_k} x_{i_k}$ за α, то получим, что $\alpha^2 \underset{n}{\equiv} z^2$, то есть $(\alpha - z)(\alpha + z) \vdots n$. Утверждается, что тогда с вероятностью, близкой к 1, $\gcd(\alpha - z, n) = x$ — нетривиальный делитель n.

Что значит, что $\prod y_i$ — квадрат? Значит, в записи $\prod y_i = \prod prime_j^{B_j}$ все B_j — четные. То есть нас интересует только четность степени вхождения каждого простого.

Давайте рассмотрим только B-гладкие y_i (то есть такие, у которых все простые в разложении $\leqslant B$). Их мы, очевидно, можем факторизовать за $O(B + \log y_i)$, просто перебрав все простые до B и попробовав на них поделить. Итого, теперь каждое y_i задается своим вектором четностей простых до B, длина этого вектора равна $m = \frac{B}{\ln B}$.

Выберем первые $m + \varepsilon$ B-гладких y_i и запустим на них Гаусса. Так как векторов больше, чем размерность, то они будут линейно зависимы. Давайте в ε свободных переменных запишем случайные значения, остальные выводятся из них. Как мы знаем, вероятность того, что мы сразу найдем нетривиальный делитель близка к 1, но не равна ей, поэтому мы и выбираем изначально $m + \varepsilon$ чисел (чтобы было просто сгенерировать новый набор y_i и проверить на нем).

Пример, иллюстрирующий вектора четностей простых:

Пример 8.6.1.

	2	3	5	7
6	1	1	0	0
7	0	0	0	1
8	1	0	0	0
10	1	0	1	0
15	0	1	1	0

Можем выбрать три линейно зависимых строки: соответствующие числам 6, 10 и 15. И действительно, $6 \cdot 10 \cdot 15 = 900 = 30^2$.

Гаусс работает за $\frac{m^3}{word\_size}$, но есть алгоритм Видемана, который умеет решать систему с k единичками за $O(km)$. В нашем случае мы можем оценить количество единичек как $O(m \log m)$, т.к. в каждой строчке задействовано $\leqslant \log m$ простых. Итоговая асипмптотика будет $O(m^2 \log m)$.

Утверждается, что можно подобрать такое B, что алгоритм будет работать корректно и при этом время работы будет $e^{C\sqrt{ln n}}$.

На практике для n порядка 2^{64} подходит $B = 100$.

8.6.3. Факторизация многочленов

Хотим научиться факторизовать многочлены:

1. над $\mathbb{R}$
2. над $\mathbb{C}$
3. над $\mathbb{F}_p$

Давайте научимся решать эти задачи.

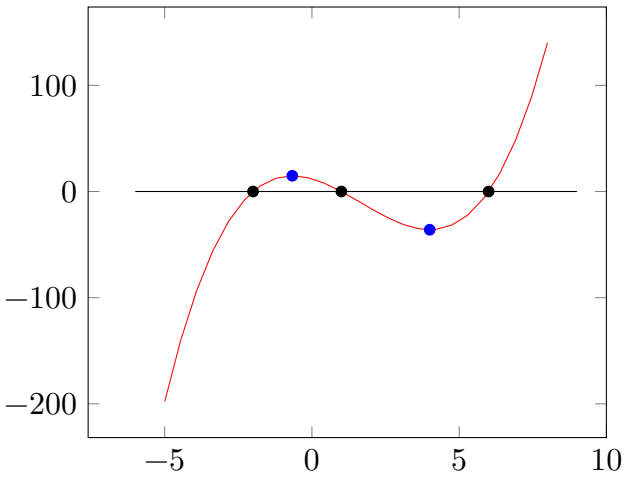
1. Нам дан многочлен P над $\mathbb{R}$, мы хотим найти все его вещественные корни.

   Пусть мы знаем корни производной P'. Тогда заметим, что корни производной соответствуют точкам экстремума многочлена P, а значит между двумя соседними корнями производной P монотонен. Поэтому мы можем найти корень обычным бинпоиском.

   То есть если мы знаем корни производной то мы можем найти все вещественные корни нашего многочлена за $O(n^2 \log C)$, т.к. мы запускаем бинпоиск на n участках, при этом в каждом $\log C$ итераций, в каждой из которых мы за $O(n)$ вычисляем значение в точке.

Осталось понять, как же вычислить корни производной? Рекурсивно! То есть просто запускаем нашу функции поиска корней рекурсивно, пока не дойдем до линейного многочлена (его корень мы можем вычислить сами), а затем откатываемся обратно.

В этой рекурсии будет n шагов (каждый раз степень многочлена уменьшается на 1), поэтому итоговая асимптотика $O(n^3 \log C)$. Кроме того, стоит отметить, что из-за рекурсии у нас теряется точность вычислений, поэтому по-хорошему нам нужно еще хранить $\sim n$ знаков, тогда асимптотика будет $O(n^4 \log C)$.



Картинка, иллюстрирующая то, как мы ищем корни: корни производной разбивают все на три отрезка: от $-\infty$ до первого корня, от первого до второго и от второго до ∞. На каждом из них все монотонно, поэтому просто ищем бинпоиском новый корень.

2. Нам дан многочлен P над $\mathbb{C}$, мы хотим найти все его комплексные корни.

   Если мы нашли хотя бы один корень x_*, то мы можешь поделить на $x - x_*$ и запуститься рекурсивно, поэтому наша текущая задача — найти один корень нашего многочлена.

   - Случайный поиск.

     Возьмем какую-нибудь случайную точку и относительно большой радиус R. Будем итерационно выполнять следующие действия: попробуем потыкаться в C рандомных точек в шаре радиуса R с центром в текущей точке, выберем из всех ту, у которой $|P(x)|$ минимален, перейдем в нее, а радиус изменим по формуле $R_{t+1} = R_t \cdot d, d < 1$.

     На практике можно брать $C \sim 100$, $d \sim 0.9$.

     Утверждается, что во многих случаях такой алгоритм находит корни.

   - Метод Ньютона.

     Вспомним, как работает метод Ньютона для одномерного случая (в нашем случае $f(x) = P(x)$).

     Мы берем какое-то первое приближение ответа x_0, а затем на каждой итерации изменяем x по формуле $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ (в геометрической интерпретации удобно понимать сходимость к корню через проведение касательных).

     Оказывается, что для комплексного случая формула $z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}$ тоже работает. Кроме того, есть следующая формула: $z_{n+1} = z_n - \frac{|f(z)|\nabla f(z)}{|\nabla f(z)|^2}$.

3. Нам дан модуль p и $a < p$, наша задача найти такой x, что $x^2 \underset{p}{\equiv} a$ или сказать, что такого x не существует.

   Алгоритм:

(a) Если $a^{\frac{p-1}{2}} \neq 1$, то решений нет.

(b) Пока не нашли корень:

- Выбираем случайное i.
- Рассмотрим многочлен $A(x) = ((x+i)^{\frac{p-1}{2}} - 1) \bmod (x^2 - a)$.
- Он, очевидно, первой степени, то есть имеет вид $A(x) = bx + c$. Если $b \neq 0$, то возвращаем $x = -c \cdot b^{-1}$.

▶ Доказательство коррекности:

(a) Заметим, что у любого числа либо ни одного, либо ровно два квадратных корня. Действительно, пусть $a^2 \underset{p}{\equiv} b^2$, значит $(a-b)(a+b) \vdots p$, то есть либо $a \underset{p}{\equiv} b$, либо $a \underset{p}{\equiv} -b$.

(b) Всего у нас $p-1$ квадрат: 1^2, 2^2, ...$(p-1)^2$, при этом из предыдущего пункта все разбивается на пары, значит квадратный корень существует ровно у $\frac{p-1}{2}$ чисел.

(c) Группа остатков по просто модулю циклична, то есть $\exists z : z^0, z^1, \dots, z^{p-2}$ — различные остатки по модулю p

(d) $\forall x : 0 < x < p \ \exists t : z^t = x$.
Если t чётное, то $x^{\frac{p-1}{2}} = (z^{\frac{t}{2}})^{p-1} = 1$, если нечетное, то $x^{\frac{p-1}{2}} = (z^{\frac{t-1}{2}})^{p-1} \cdot z^{\frac{p-1}{2}} = z^{\frac{p-1}{2}} = -1$, т.к. $z^{\frac{p-1}{2}}$ — квадратный корень из 1.

(e) Если $a^{\frac{p-1}{2}} = -1$, то $1 = x^{p-1} = (x^2)^{\frac{p-1}{2}} = a^{\frac{p-1}{2}} = -1$, противоречие, то есть для таких a действительно решений нет. Всего таких a ровно $\frac{p-1}{2}$, соответственно для всех остальных корень существует.

(f) Обозначи корни уравнения $x^2 = a \mod p$ за z_* и $-z_*$. Рассмотрим $P(x) = (x+i)^{\frac{p-1}{2}} - 1$, его корни — это числа вида $z - i$, где z — квадратичный вычет по модулю p.
Также обозначим $Q(x) = (x - z_*)(x + z_*)$.
Теперь, если оба числа z_* и $-z_*$ являются корнями $P(x)$, то $P(x) \mod Q(x) = 0$, и по нашему алгоритму x мы не найдем.
Если ни одно из чисел z_* и $-z_*$ не является корнем $P(x)$, то по КТО $P(x) \mod Q(x) = const \neq 0$, поэтому x мы снова не найдем.
А вот если корнем является ровно одно из чисел, например z_*, то $P(x) \mod Q(x) = R(x)(x - z_*) \mod (x + z_*)(x - z_*) = c \cdot (x - z_*)$, $c \neq 0$. Получили ненулевой коэффициент при x, значит по алгоритму найдем корень z_*.

◀

Оценка времени работы:

(a) Вероятность того, что случайное i подойдет, равна $\frac{1}{2}$.

Как мы поняли из доказательства, мы хотим найти такое i, что ровно одно из чисел $z_* - i$ и $-z_* - i$ является квадратичным вычетом. Иначе говоря $\left(\frac{z_*+i}{p}\right) \neq \left(\frac{-z_*+i}{p}\right)$ (тут мы заменили i на $-i$ для чуть более простой записи).

На всякий случай, $\left(\frac{a}{p}\right)$ — обозначение символа Лежандра.
$\left(\frac{z_*+i}{p}\right) \neq \left(\frac{-z_*+i}{p}\right) \Rightarrow \left(\frac{(z_*+i)/(z_*-i)}{p}\right) = -1$.

Заметим, что $i \to \frac{z_*+i}{z_*-i}$ — биекция. Действительно, пусть $\frac{z_*+i}{z_*-i} = \frac{z_*+j}{z_*-j}$, то есть $(z_*+i)(z_*-j) = (z_*+j)(z_*-i)$, значит $2iz_* = 2jz_* \Rightarrow i = j$.

Мы знаем, что невычетов ровно $\frac{p-1}{2}$, а случайное i дает нам случайное $\frac{z_*+i}{z_*-i}$, поэтому с вероятностью $\frac{1}{2}$ $\frac{z_*+i}{z_*-i}$ будет невычетом, а значит ровно одно из чисел z_* и $-z_*$ будет корнем $P(x)$, и наш алгоритм завершится на этом шаге.

(b) $A(x)$ мы вычисляем с помощью бинарного возведения в степень, как в алгоритме для решение рекуррентных соотношений (только здесь у нас степень всегда $O(1)$, поэтому работать будет за $O(\log p)$).

Итого, получили оценку матожидания времени работы $O(\log p)$.

4. Теперь мы хотим разложить многочлен $P(x), \deg P = n$ над $\mathbb{F}_p$, причем нам известно, что $P(x)$ является произведением нескольких неприводимых многочленов степени d.

   Опять же, достаточно найти хотя бы один нетривиальный делитель $T(x)$, а дальше можно запуститься рекурсивно от $T(x)$ и $P(x)/T(x)$ рекурсивно.

   Алгоритм:

   (a) Возьмем случайный многочлен $Q(x), \deg Q < n$.

   (b) Если $\gcd(P(x), Q(x)) \neq 1$, то мы нашли нетривиальный делитель.

   (c) Рассмотрим $R(x) = (Q(x)^{\frac{p^d-1}{2}} - 1) \bmod P(x)$, как и раньше, считаем мы это бинарным возведением в степень и всегда берем по модулю $P(x)$.

   (d) Утверждается, что с вероятностью $\frac{1}{2^{n-1}}$ $\gcd(R(x), P(x))$ является нетривиальным делителем $P(x)$.

   Стоит отметить, что это в некотором смысле обобщение предыдущего алгоритма: когда мы искали квадратный корень, мы полагали $d = 1, n = 2$.