

SPb HSE, 2 курс, осень 2020/21  
Конспект лекций по алгоритмам

Собрано 26 августа 2021 г. в 23:55

---

## Содержание

|                                                                                                            |           |
|------------------------------------------------------------------------------------------------------------|-----------|
| <b>1. Теория чисел</b>                                                                                     | <b>1</b>  |
| 1.1. Решето Эратосфена . . . . .                                                                           | 1         |
| 1.2. Решето и корень памяти (на практике) . . . . .                                                        | 2         |
| 1.3. Вычисление мультипликативных функций на $[1, n]$ (на практике) . . . . .                              | 2         |
| 1.4. Определение . . . . .                                                                                 | 2         |
| 1.5. Расширенный алгоритм Евклида . . . . .                                                                | 2         |
| 1.6. Обратные в $(\mathbb{Z}/m\mathbb{Z})^*$ и $\mathbb{Z}/p\mathbb{Z}$ . . . . .                          | 3         |
| 1.7. Возведение в степень за $\mathcal{O}(\log n)$ . . . . .                                               | 3         |
| 1.8. Обратные в $\mathbb{Z}/p\mathbb{Z}$ для чисел от 1 до $k$ за $\mathcal{O}(k)$ (на практике) . . . . . | 4         |
| 1.9. Первообразный корень . . . . .                                                                        | 4         |
| 1.10. Дискретное логарифмирование . . . . .                                                                | 5         |
| 1.11. Корень $k$ -й степени по модулю . . . . .                                                            | 5         |
| 1.12. КТО . . . . .                                                                                        | 6         |
| 1.12.1. Использование КТО в длинной арифметике. . . . .                                                    | 6         |
| 1.13. Криптография. RSA. . . . .                                                                           | 7         |
| 1.14. Протокол Диффи-Хеллмана . . . . .                                                                    | 7         |
| <b>2. Линейные системы уравнений</b>                                                                       | <b>8</b>  |
| 2.1. Гаусс для квадратных невырожденных матриц. . . . .                                                    | 9         |
| 2.2. Гаусс в общем случае . . . . .                                                                        | 10        |
| 2.3. Гаусс над $\mathbb{F}_2$ . . . . .                                                                    | 11        |
| 2.4. Погрешность . . . . .                                                                                 | 11        |
| 2.5. Метод итераций . . . . .                                                                              | 11        |
| 2.6. Вычисление обратной матрицы . . . . .                                                                 | 12        |
| 2.7. Гаусс для евклидова кольца . . . . .                                                                  | 12        |
| 2.8. Разложение вектора в базисе . . . . .                                                                 | 13        |
| 2.8.1. Ортогонализация Грама-Шмидта . . . . .                                                              | 13        |
| 2.9. Вероятностные задачи (на практике) . . . . .                                                          | 13        |
| <b>3. Быстрое преобразование Фурье</b>                                                                     | <b>15</b> |
| 3.1. Прелюдия к FFT . . . . .                                                                              | 15        |
| 3.2. Собственно идея FFT . . . . .                                                                         | 15        |
| 3.3. Крутая реализация FFT . . . . .                                                                       | 16        |
| 3.4. Обратное преобразование . . . . .                                                                     | 17        |
| 3.5. Два в одном . . . . .                                                                                 | 17        |
| 3.6. Умножение чисел, оценка погрешности . . . . .                                                         | 17        |
| <b>4. Длинная арифметика</b>                                                                               | <b>17</b> |
| 4.1. Бинарная арифметика . . . . .                                                                         | 19        |

|                                                                                      |           |
|--------------------------------------------------------------------------------------|-----------|
| 4.2. Деление многочленов за $\mathcal{O}(n \log^2 n)$                                | 20        |
| 4.3. Деление чисел                                                                   | 20        |
| 4.4. Деление чисел за $\mathcal{O}((n/k)^2)$                                         | 21        |
| <b>5. Умножение матриц и 4 русских</b>                                               | <b>23</b> |
| 5.1. Четыре русских: общие слова                                                     | 23        |
| 5.2. Битовое сжатие в лоб                                                            | 23        |
| 5.3. Битовое сжатие строк над $\mathbb{F}_2$                                         | 23        |
| 5.4. Применяем идею 4-х русских                                                      | 23        |
| 5.5. Умножение матриц над $\mathbb{F}_2$ за $\mathcal{O}(n^3/(w \log n))$            | 24        |
| 5.6. НОП за $\mathcal{O}(n^2/\log^2 n)$ (на практике)                                | 24        |
| <b>6. Базовые алгоритмы на строках</b>                                               | <b>25</b> |
| 6.1. Обозначения, определения                                                        | 25        |
| 6.2. Поиск подстроки в строке                                                        | 25        |
| 6.2.1. C++                                                                           | 25        |
| 6.2.2. Префикс функция и алгоритм КМП                                                | 25        |
| 6.2.3. LCP                                                                           | 27        |
| 6.2.4. Z-функция                                                                     | 27        |
| 6.3. Полиномиальные хеши строк                                                       | 27        |
| 6.3.1. Алгоритм Рабина-Карпа                                                         | 28        |
| 6.3.2. Наибольшая общая подстрока за $\mathcal{O}(n \log n)$                         | 28        |
| 6.3.3. Оценки вероятностей                                                           | 30        |
| 6.3.4. Число различных подстрок (на практике)                                        | 30        |
| 6.3.5. Алгоритм Бойера-Мура                                                          | 31        |
| <b>7. Суффиксный массив</b>                                                          | <b>32</b> |
| 7.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами                                  | 33        |
| 7.2. Применение суффиксного массива: поиск строки в тексте                           | 33        |
| 7.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой | 33        |
| 7.4. LCP за $\mathcal{O}(n)$ : алгоритм Касаи                                        | 34        |
| 7.5. Быстрый поиск строки в тексте                                                   | 35        |
| <b>8. Ахо-Корасик и Укконен</b>                                                      | <b>36</b> |
| 8.1. Бор                                                                             | 36        |
| 8.2. Алгоритм Ахо-Корасик                                                            | 36        |
| 8.3. Суффиксное дерево                                                               | 38        |
| 8.4. Суффиксное дерево, решение задач (на практике)                                  | 38        |
| 8.5. Алгоритм Укконена                                                               | 39        |
| 8.6. LZSS                                                                            | 40        |
| <b>9. Потoki (база)</b>                                                              | <b>40</b> |
| 9.1. Основные определения                                                            | 41        |
| 9.2. Обратные рёбра                                                                  | 41        |
| 9.3. Альтернативное определение                                                      | 42        |
| 9.4. Декомпозиция потока (на практике)                                               | 42        |
| 9.5. Теорема и алгоритм Форда-Фалкерсона                                             | 43        |
| 9.6. Реализация, хранение графа                                                      | 44        |

|                                                                 |           |
|-----------------------------------------------------------------|-----------|
| 9.7. Двудольные графы (на практике)                             | 45        |
| 9.7.1. Паросочетание                                            | 45        |
| 9.7.2. Вершинное покрытие                                       | 45        |
| 9.8. Леммы, позволяющие работать с потоками                     | 45        |
| 9.9. Алгоритмы поиска потока                                    | 46        |
| 9.9.1. Эдмондс-Карп за $\mathcal{O}(VE^2)$                      | 46        |
| 9.9.2. Масштабирование за $\mathcal{O}(E^2 \log U)$             | 47        |
| <b>10. Потоки (быстрые)</b>                                     | <b>47</b> |
| 10.1. Алгоритм Диница                                           | 48        |
| 10.2. Алгоритм Хопкрофта-Карпа                                  | 49        |
| 10.3. Теоремы Карзанова                                         | 50        |
| 10.4. Диниц с link-cut tree (на практике)                       | 50        |
| 10.5. Глобальный разрез                                         | 51        |
| 10.5.1. Алгоритм Штор-Вагнера (на практике)                     | 51        |
| 10.5.2. Алгоритм Каргера-Штейна                                 | 51        |
| 10.6. (*) Алгоритм Push-Relabel                                 | 52        |
| <b>11. Mincost</b>                                              | <b>52</b> |
| 11.1. Mincost k-flow в графе без отрицательных циклов           | 53        |
| 11.2. Потенциалы и Дейкстра                                     | 54        |
| 11.3. Графы с отрицательными циклами                            | 54        |
| 11.4. Mincost flow (на практике)                                | 54        |
| 11.5. Полиномиальные решения                                    | 55        |
| <b>12. Паросочетания</b>                                        | <b>56</b> |
| 12.1. Определения                                               | 56        |
| 12.2. Сведения                                                  | 56        |
| 12.3. Дополняющие чередующиеся пути                             | 56        |
| 12.4. Поиск паросочетания в двудольном графе                    | 57        |
| 12.5. Реализация                                                | 57        |
| 12.6. Алгоритм Куна                                             | 57        |
| 12.7. Оптимизации Куна                                          | 58        |
| 12.8. Поиск минимального вершинного покрытия (на практике)      | 58        |
| 12.9. Обзор решений                                             | 59        |
| 12.10. Решения для произвольного графа                          | 59        |
| 12.10.1. Обзор                                                  | 59        |
| 12.10.2. Матрица Татта                                          | 59        |
| 12.10.3. Читерский подход                                       | 60        |
| <b>13. Паросочетания (продолжение)</b>                          | <b>60</b> |
| 13.1. Классификация рёбер двудольного графа                     | 61        |
| 13.2. Stable matching (marriage problem)                        | 61        |
| 13.3. Венгерский алгоритм                                       | 62        |
| 13.3.1. Реализация за $\mathcal{O}(V^3)$                        | 62        |
| 13.3.2. Псевдокод                                               | 63        |
| <b>14. (*) Суффиксный массив за <math>\mathcal{O}(n)</math></b> | <b>63</b> |

|                                                                             |           |
|-----------------------------------------------------------------------------|-----------|
| <b>15. Игры на графах</b>                                                   | <b>65</b> |
| 15.1. Основные определения . . . . .                                        | 66        |
| 15.1.1. Решение для ациклического орграфа . . . . .                         | 66        |
| 15.1.2. Решение для графа с циклами (ретроанализ) . . . . .                 | 67        |
| 15.2. Ним и Гранди, прямая сумма . . . . .                                  | 67        |
| 15.3. Вычисление функции Гранди . . . . .                                   | 68        |
| <b>16. (*) Число простых на <math>[1, n]</math> за <math>n^{2/3}</math></b> | <b>70</b> |
| <b>17. Хеширование</b>                                                      | <b>72</b> |
| 17.1. Универсальное семейство хеш функций . . . . .                         | 72        |
| 17.2. Оценки для хеш-таблицы с закрытой адресацией . . . . .                | 72        |
| 17.3. Фильтр Блюма . . . . .                                                | 73        |
| 17.4. (*) Совершенное хеширование . . . . .                                 | 73        |
| 17.4.1. Одноуровневая схема . . . . .                                       | 74        |
| 17.4.2. Двухуровневая схема . . . . .                                       | 74        |
| 17.4.3. Графовый подход . . . . .                                           | 74        |
| 17.5. Хеширование кукушки . . . . .                                         | 75        |

## Лекция #1: Теория чисел

1 сентября 2020

## 1.1. Решето Эратосфена

**Задача:** найти все простые от 1 до  $n$ .

Решето Эратосфена предлагает вычёркивать числа, кратные уже найденным простым:

```

1 vector<bool> is_prime(n + 1, 1); // Хорошо по памяти даже при  $n \approx 10^9$ !
2 is_prime[0] = is_prime[1] = 0;
3 for (int i = 2; i <= n; i++)
4     if (is_prime[i]) // Нашли новое простое!
5         for (int j = i + i; j <= n; j += i)
6             is_prime[j] = 0; // cnt++, чтобы определить константу

```

*Замечание 1.1.1.* Сейчас для каждого числа мы находим лишь один бит. Код легко модифицировать, чтобы для каждого числа находить наименьший простой делитель.

Данную версию кода можно оптимизировать в константу раз, пользуясь тем, что у любого не простого числа есть делитель не более корня.

```

1 for (int i = 2; i * i <= n; i++) // cnt++, чтобы определить константу
2     if (is_prime[i])
3         for (int j = i * i; j <= n; j += i)
4             is_prime[j] = 0; // cnt++, чтобы определить константу

```

Можно ещё оптимизировать: мы ищем только нечётные простые  $\Rightarrow$

внешний цикл можно вести только по нечётным  $i$ , а во внутреннем прибавлять  $2i$ .

Теперь у нас три версии решета, отличающиеся не большими оптимизациями.

Эмпирический запуск при  $n = 10^6$  даёт значения **cnt**:  $3.7752n$ ,  $2.1230n$ ,  $0.8116n$  соответственно.

**Теорема 1.1.2.** Обе версии работают за  $\Theta(n \log \log n)$ .

*Доказательство.* При достаточно больших  $k$  верно  $0.5 k \log k \leq p_k \leq 2 k \log k$  (без док-ва).

Так как  $\left\lceil \frac{n}{p_k} \right\rceil = \mathcal{O}(1) + \frac{n}{k \log k}$ , время работы 1-й версии равно

$$\begin{aligned}
 n + \mathcal{O}(n) + \sum_{k=\mathcal{O}(1)}^{p_k \leq n} \left\lceil \frac{n}{p_k} \right\rceil &= \Theta\left(n + \sum_{k=\mathcal{O}(1)}^{n/\log n} \frac{n}{k \log k}\right) = \\
 \Theta\left(n + n \int_{\mathcal{O}(1)}^n \frac{1}{x \log x} dx\right) &= \Theta(n + n \log \log n - \mathcal{O}(1)) = \Theta(n \log \log n)
 \end{aligned}$$

• **Более быстрое решение.**

Чтобы найти все простые от 1 до  $n$  за  $\mathcal{O}(n)$ , достаточно модифицировать алгоритм так, чтобы каждое составное  $x$  пометать лишь один раз, например, наименьшим простым делителем  $x$ .

Пусть  $d[x]$  – номер наименьшего простого делителя  $x$  ( $primes[d[x]]$  – собственно делитель).

Пусть  $x = primes[d[x]] \cdot y \Rightarrow (d[y] \geq d[x] \vee y = 1)$

Алгоритм: перебирать  $y$ , а для него потенциальные  $d[x]$  (простые не большие  $d[y]$ ).

```

1 vector<int> primes, d(n + 1, -1);
2 for (int y = 2; y <= n; y++)
3     if (d[y] == -1)
4         d[y] = primes.size(), primes.push_back(\red{y});
5     for (int i = 0; i <= d[y] && y * primes[i] <= n; i++)
6         d[y * primes[i]] = i; // x=y*primes[i], i=d[x]

```

## 1.2. Решето и корень памяти (на практике)

Пусть нам нужно найти все простые на промежутке  $(n - \sqrt{n}..n] = (l..r]$ .

Это можно сделать за  $\mathcal{O}(\sqrt{n} \log \log n)$  времени и  $\mathcal{O}(\sqrt{n})$  памяти.

1. У не простого числа до  $n$  есть простой делитель  $\leq \sqrt{n} \Rightarrow$  посчитаем все простые до  $\sqrt{n}$ .
2. Будем этими простыми “просеивать” нужный нам интервал... Для простого  $p$ , сперва нужно пометить  $l - (l \bmod p) + p$ , затем с шагом  $p$  до  $r$ , как в обычном решете.

## 1.3. Вычисление мультипликативных функций на $[1, n]$ (на практике)

**Def 1.3.1.** Функция мультипликативна  $\Leftrightarrow \forall a, b: (a, b) = 1 \Rightarrow f(ab) = f(a)f(b)$ .

Т.е. имея разложение числа на простые числа  $x = \prod_i p_i^{\alpha_i}$ , имеем  $f(x) = f(x/p_1^{\alpha_1})f(p_1^{\alpha_1})$

### • Примеры:

|                                       |                                              |                                                                  |
|---------------------------------------|----------------------------------------------|------------------------------------------------------------------|
| Простой делитель $n$                  |                                              | <code>p[x] = primes[d[x]]</code>                                 |
| Удобное обозначение                   |                                              | <code>y[x] = x / p[x]</code>                                     |
| Степень этого делителя                |                                              | <code>deg[x] = (d[y[x]] == d[x] ? deg[y[x]] + 1 : 1)</code>      |
| Результат отщепления $p_1^{\alpha_1}$ |                                              | <code>rest[x] = (d[y[x]] == d[x] ? rest[y[x]] : y[x])</code>     |
| Собственно $p_1^{\alpha_1}$           |                                              | <code>term[x] = x / rest[x]</code>                               |
| Число взаимнопростых                  | $\varphi(n) = n \prod_i \frac{p_i - 1}{p_i}$ | <code>phi[x] = phi[rest[x]] * (term[x]/p[x]) * (p[x] - 1)</code> |
| Число делителей                       | $\sigma_0(n) = \prod_i (\alpha_i + 1)$       | <code>s0[x] = s0[rest[x]] * (deg[x] + 1)</code>                  |

Чуть сложнее посчитать сумму делителей:  $\sigma_1(n) = \prod_i (p_i^0 + p_i^1 + \dots + p_i^{\alpha_i}) = \prod_i \frac{p_i^{\alpha_i+1} - 1}{p_i - 1}$ .

Итого: `s1[x] = s1[rest[x]] * (term[x] * p[x] - 1) / (p[x] - 1)`.

## 1.4. Определение

**Def 1.4.1.**  $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$  поле остатков по модулю  $p$ .

**Def 1.4.2.**  $(\mathbb{Z}/m\mathbb{Z})^*$  Группа по умножению  $\{a: (a, m) = 1 \wedge 1 \leq a < m\}$ .

**Def 1.4.3.** Линейное диофантово уравнение  $(a, b, c$  даны, нужно найти  $x, y)$ .

$$ax + by + c = 0; \quad x, y \in \mathbb{Z}$$

Деление:  $\frac{a}{b} = a \cdot b^{-1}$

### • Алгоритм Евклида

Используется школьниками для подсчёта gcd:

$\gcd(a, b) = \gcd(a - b, b)$ , повторяя вычитание много раз, получаем  $\gcd(a, b) = \gcd(a \bmod b, b)$

Итого: `int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }`

## 1.5. Расширенный алгоритм Евклида

Теперь задача – найти ещё и  $x, y: ax + by = \gcd(a, b)$

Действуя опять же рекурсивно, имея базу,  $a \cdot 1 + b \cdot 0 = a$

$$a \cdot 0 + b \cdot 1 = b$$

мы можем добавить переход из двух строк  $a \cdot x_i + b \cdot y_i = r_i$

$$a \cdot x_{i+1} + b \cdot y_{i+1} = r_{i+1}$$

в новую  $r_{i+2} = r_{i+1} \bmod r_i = r_{i+1} - kr_i$  ( $k$  – частное)

$$x_{i+2} = x_{i+1} - ky_i$$

$$y_{i+2} = y_{i+1} - ky_i$$

Заметим, что процесс на числах  $r_i$  – обычный алгоритм Евклида.

```

1 def euclid(a, b): # returns (x,y): ax + by = gcd(a, b)
2   if b == 0:
3     return 1, 0
4   x, y = euclid(b, a % b);
5   return y, x - (a // b) * y # целочисленное деление

```

### • Решение диофантового уравнения:

Если  $c \bmod \gcd(a, b) \neq 0$ , то решений нет.

Иначе найдём  $x, y: ax + by = \gcd(a, b)$  и домножим уравнение на  $c/\gcd(a, b)$ .

### • Свойства расширенного алгоритма Евклида (на практике)

Следующие утверждения обсуждались и доказывались на практике:

(a)  $\forall i$  в строке  $ax_i + by_i = r_i$  верно, что  $(x_i, y_i) = 1$

(b)  $\max |x_i| \leq |b|$  и  $\max |y_i| \leq |a| \Rightarrow$

$\forall$  типа  $T$ , если исходные данные помещаются в тип  $T$ , то и все промежуточные тоже.

Также на практике были решены следующие задачи:

(c) Найдите класс решений уравнения  $ax \equiv b \pmod{m}$

(d) Найдите  $x, y: ax + by = c, |x| + |y| \rightarrow \min$

## 1.6. Обратные в $(\mathbb{Z}/m\mathbb{Z})^*$ и $\mathbb{Z}/p\mathbb{Z}$

Задача:  $a$  и  $m$  даны, хотим найти  $x: a \cdot x \equiv 1 \pmod{m}$

Первый способ – решить диофантово уравнение  $ax + my = 1 = \gcd(a, m)$

Другой способ – воспользоваться малой теоремой Ферма или теоремой Эйлера:

$$a^{p-1} = 1 \pmod{p} \Rightarrow x = a^{p-2} \text{ (для простого)}$$

$$a^{\varphi(m)} = 1 \pmod{m} \Rightarrow x = a^{\varphi(m)-1} \text{ (для произвольного)}$$

Замечание 1.6.1. Функцию Эйлера считать долго!

Пусть  $n = \prod p_i^{\alpha_i} \Rightarrow \varphi(n) = n \prod \frac{p_i - 1}{p_i}$ , для вычисления нужна факторизация  $n$

## 1.7. Возведение в степень за $\mathcal{O}(\log n)$

Сводим к  $\mathcal{O}(\log n)$  умножениям. Считаем, что одно умножение работает за  $\mathcal{O}(1)$ .

```

1 def pow(x, n):
2   if n == 0: return 1
3   return pow(x**2, n // 2)**2 * (x if n % 2 == 1 else 1)

```

*Замечание 1.7.1.* При возведении в степень, если исходные данные в типе  $T$ , то при умножении по модулю мы можем столкнуться с переполнением  $T$ ...

Есть два решения: или умножение за  $\mathcal{O}(1)$  превратить в  $\mathcal{O}(\log n)$  сложений тем же алгоритмом, или использовать вещественные числа:

```

1 int64 mul(int64 a, int64 b, int64 m) { // 0 ≤ a, b < m
2   int64 k = (long double)a * b / m; // посчитано с погрешностью!
3   int64 r = a * b - m * k; // в знаковом типе формально это UB =(
4   while (r < 0) r += m;
5   while (r >= m) r -= m;
6   return r;
7 }
```

## 1.8. Обратные в $\mathbb{Z}/p\mathbb{Z}$ для чисел от 1 до $k$ за $\mathcal{O}(k)$ (на практике)

Сейчас одно обращение работает за  $\mathcal{O}(\log p) \Rightarrow$  задачу мы умеем решать только за  $\mathcal{O}(k \log p)$ .  
Время улучшать! Используем динамику: зная, обратные к  $1..i-1$  найдём  $i^{-1}$ .

**Теорема 1.8.1.**  $i^{-1} = -\lfloor \frac{m}{i} \rfloor \cdot (m \bmod i)^{-1}$

*Доказательство.*  $0 \equiv m = (m \bmod i) + i \cdot \lfloor \frac{m}{i} \rfloor$ . Домножим на  $(m \bmod i)^{-1}$ :  
 $0 \equiv 1 + (m \bmod i)^{-1} \cdot i \cdot \lfloor \frac{m}{i} \rfloor \Rightarrow i^{-1} \equiv - (m \bmod i)^{-1} \lfloor \frac{m}{i} \rfloor$  ■

## 1.9. Первообразный корень

Пусть  $p$  – простое. Работаем в  $(\mathbb{Z}/p\mathbb{Z})^*$  (группа по умножению по модулю  $p$ ).  
На алгебре вы доказывали (и хорошо бы помнить, как именно!), что

$$\exists g: \{1, 2, \dots, p-1\} = \{g^0, g^1, \dots, g^{p-2}\}$$

Такое  $g$  называется *первообразным корнем*.

### • Задача проверки.

Дан  $g$ , проверить, является ли он первообразным корнем.

Если не является, то  $\text{ord } g < p-1$ , при этом  $\text{ord } g \mid p-1$ . То есть, достаточно перебрать все  $d$  делители  $p-1$ , и для каждого возведением в степень проверить, что  $g^d \neq 1$ .

Возведение в степень работает за  $\mathcal{O}(\log p)$ , если  $p$  помещается в машинное слово.

На длинных числах умножение и деление с остатком по модулю работают  $\mathcal{O}(\log p \log \log p)$ , итого возведение в степень за  $\mathcal{O}(\log^2 p \log \log p)$ .

Делители перебирать нужно не все, а только вида  $\frac{p-1}{\alpha}$ , где  $\alpha$  – простой делитель  $p-1$ .

Обозначим  $p_k$  –  $k$ -е простое. Далее будем без доказательства пользоваться тем, что  $p_k \geq k \log k$ .

**Lm 1.9.1.** Пусть  $f(x)$  – число различных простых делителей у  $x \Rightarrow \forall x f(x) = \mathcal{O}(\frac{\log x}{\log \log x})$

*Доказательство.* Худший случай:  $x$  – произведение минимальных простых.

$x = \prod_{i=1..k} p_i \geq \prod_{i=1..k} i = k! \Rightarrow \log x \geq \Omega(k \log k)$ , то есть  $\geq ck \log k$  для какой-то константы  $c$ .  
Докажем, что  $k = \mathcal{O}(\frac{\log x}{\log \log x})$ .

Если бы  $k > f \frac{\log x}{c \log \log x}$  (константу  $f$  выберем позже), то

$$ck \log k > f \frac{\log x}{\log \log x} (\log f + \log \log x - \log \log \log x) > \dots$$



Оценим  $\log f + \log \log x - \log \log \log x = \Omega(\log \log x)$ , то есть  $> d \log \log x$  для какой-то константы  $d$ , не зависящей от  $f$ .

Тогда  $\dots > f^{\frac{\log x}{\log \log x}} (d \log \log x) = (fd) \log x > \dots$

Выбирая  $f > 1/d$ , получаем  $\dots > \log x$  — противоречие. ■

**Теорема 1.9.2.** Мы научились проверять кандидат  $g$  за  $\mathcal{O}(\text{ФАСТ} + \log^3 p)$ .

Факторизация нужна, как раз чтобы найти простые делители  $p-1$ .

### • Задача поиска.

И сразу решение: ткнём в случайное число, с хорошей вероятностью оно подойдёт.

Из детерминированных решений популярным является перебор в порядке  $1, 2, 3, \dots$

Shoup'92 доказал, что в предположении обобщённой Гипотезы Римана, нужно  $\mathcal{O}(\log^6 p)$  шагов.

Обозначим как  $G(p)$  множество всех первообразных корней для  $p$ .

**Теорема 1.9.3.**  $a$  — случайное от 1 до  $p-1 \Rightarrow \Pr[a \in G(p)] = \Omega(\frac{1}{\log \log p})$  (т.е. хотя бы столько).

*Доказательство.* Пусть  $g$  —  $\forall$  первообразный, тогда  $G(p) = \{g^i \mid (i, p-1) = 1\} \Rightarrow \frac{G(p)}{p-1} = \frac{\varphi(p-1)}{p-1}$ .

Осталось научиться оценивать  $\frac{\varphi(n)}{n} = \prod_{q|n} \frac{q-1}{q} \geq \prod \frac{i \log i - 1}{i \log i}$ , где  $q$  — простые делители  $n$ .

Логарифмируем:  $\log \frac{\varphi(n)}{n} \geq \sum \log(i \log i - 1) - \sum \log(i \log i) = \Theta(-\sum_{i=1}^k \frac{1}{i \log i}) = \Theta(-\log \log k)$

Здесь  $k$  — число простых делителей. Получаем  $\frac{\varphi(n)}{n} \geq \Theta(\frac{1}{\log k}) = \Theta(\frac{1}{\log \log n})$ . ■

*Замечание 1.9.4.* Пусть есть алгоритм  $T$ , который работает корректно, если дать ему правильный первообразный корень. Пусть мы ещё умеем проверять корректность результата  $T$ . Тогда проще всего вместо того, чтобы искать первообразный корень,  $\approx \log \log p$  раз запустить алгоритм, подсовывая ему случайные числа.

## 1.10. Дискретное логарифмирование

Задача схожа по записи с обычным логарифмированием:  $a^x = b \Rightarrow x = \log_a b$ .

Собственно её нам и предстоит решить, только все вычисления по модулю  $m$ .

Заметим, что  $x$  имеет смысл искать только в диапазоне  $[0, \varphi(m))$  и  $\varphi(m) < m$ .

Общая идея решения: корневая по  $x$ . Любую степень корнем поделим на две части...

Возьмём  $k = \lceil \sqrt{m} \rceil$ , построим множество пар  $B = \{\langle a^0, 0 \rangle, \langle a^k, k \rangle, \langle a^{2k}, 2k \rangle, \dots, \langle a^{(k-1)k}, (k-1)k \rangle\}$ .

Пусть  $x$  существует, поделим его с остатком на  $k$ :  $x = ik + j \Rightarrow a^x = a^{ki} a^j \wedge \langle a^{ki}, ki \rangle \in B$ .

Заметим, что  $a^{ki} a^j = b \Leftrightarrow a^{ki} = ba^{-j}$ . Осталось перебрать  $j$ :

```

1 a1 = inverse(a) # один раз за  $\mathcal{O}(\log p)$ 
2 for j in range(k):
3     if b in B: # с точки зрения реализации  $B$  - словарь
4         x = j + B[b] # словарь ;)
5         b = mul(b, a1) # по модулю! в итоге на  $j$ -й итерации у нас под рукой  $ba^{-j}$ 

```

Если  $B$  — хеш-таблица, то суммарное время работы  $\mathcal{O}(\sqrt{p})$ .

*Замечание 1.10.1.* Если для каждого  $b$  в  $B[b]$  хранить весь список индексов, код легко модифицировать так, чтобы он находил все решения.

## 1.11. Корень $k$ -й степени по модулю

Решаем уравнение вида  $x^k \equiv a \pmod{p}$ . Даны  $k, b, p \in \mathbb{P}$ .

Дискретно прологарифмируем по основанию первообразного корня:  $a = g^b$ ,  $x$  ищем в виде  $g^y$ .

Получаем  $g^{ky} \equiv g^b \pmod{p} \Leftrightarrow ky \equiv b \pmod{p-1} \Leftrightarrow y \equiv \frac{b}{k} \pmod{p-1}, x = g^y$ .  
 Время работы – логарифмирование + деление, т.е.  $\sqrt{p} + \log p$ .

## 1.12. КТО

Китайская Теорема об Остатках (К.Т.О.) была у вас на алгебре в простейшем виде:

**Теорема 1.12.1.**  $\begin{cases} x \equiv a_i(m_i) \\ \forall i \neq j (m_i, m_j) = 1 \end{cases} \Rightarrow \exists! a \in [0, M): x \equiv a(M), \text{ где } M = \prod m_i.$

Также показывалось, что  $a = \sum a_i e_i$ , где  $e_i$  подбиралось так, что  $e_i \equiv 1(m_i), \forall j \neq i e_i \equiv 0(m_j)$ . Собственно, если обозначить  $t = \prod_{j \neq i} m_j$ , то  $e_i = (t \cdot t^{-1}(m_i)) \pmod{M}$ .

Сейчас мы пойдём чуть дальше и рассмотрим случай произвольных модулей  $m_i$ .

Первым делом  $\forall i$  факторизуем  $m_i = \prod p_{ij}^{\alpha_{ij}}$ . И заменим сравнения на  $\forall j x \equiv a_i \pmod{p_{ij}^{\alpha_{ij}}}$ .

Для каждого простого  $p$  оставим только главное сравнение:  $p^\alpha$  с максимальным  $\alpha$ .

Нужно проверить, что любые сравнения по модулям вида  $p^\beta$  главному не противоречат.

Итого за  $\mathcal{O}(\text{ФАКТ})$  мы свели задачу к КТО, или нашли противоречие.

### 1.12.1. Использование КТО в длинной арифметике.

Пусть нам нужно посчитать  $X$  – значение арифметического выражения. Во время вычислений, возможно, появляются очень длинные числа, но мы уверены, что  $X \leq L$  ( $L$  дано).

Возьмём несколько случайных 32-битных простых  $p_i$ , чтобы их произведение было больше  $L$ .

Понадобится  $k \approx \log L$  чисел. Теперь  $k$  раз найдём  $X \pmod{p_i}$ , оперируя только с короткими числами, а затем по формулам из КТО соберём  $X \pmod{\prod p_i}$ .

Поскольку  $X \leq L < \prod p_i$ , мы получили верный  $X$ .

### 1.13. Криптография. RSA.

Два типа шифрования:

**Симметричная криптография.** Один и тот же ключ позволяет и зашифровать, и расшифровать сообщение. Примеры шифрования: хог с ключом; циклический сдвиг алфавита.

**Криптография с открытым ключом.** Боб хочет послать сообщение Алисе и шифрует его *открытым ключом* Алисы ( $e$ ), ключ ( $e$ ) знают все. Для расшифровки Алисе понадобится ее закрытый ключ ( $d$ ), который знает только она. Сами функции для шифровки и расшифровки открыты, их знают все.

#### • RSA.

Выберем два больших простых числа  $p, q$ . Посчитаем  $n = pq, \varphi(n) = (p-1)(q-1)$ .

Выберем случайное  $1 \leq e < \varphi(n)$ , посчитаем  $d: ed \equiv 1 \pmod{\varphi(n)}$ .

Итого: генерим случайно  $p, q, e$ ; вычисляем  $n, \varphi(n), d$ .

Тогда открытым ключом будет пара  $\langle e, n \rangle$ , а закрытым –  $\langle d, n \rangle$ .

Действия Боба для шифрования:  $m \rightarrow \mu = m^e \pmod n$

Действия Алисы для дешифровки:  $\mu \rightarrow m = \mu^d \pmod n$

Проверим корректность:  $(m^e)^d = m^{ed} = m^{\varphi(n) \cdot k + 1} \equiv 1^k \cdot m^1 = m$ .

Алгоритм надежен настолько, насколько сложна задача факторизации чисел.

Числа умеют факторизовать так ([более полный список на wiki](#)):

- (a)  $\mathcal{O}(n^{1/2})$  – тривиальный перебор всех делителей до корня.
- (b)  $\mathcal{O}(n^{1/4} \cdot \gcd)$  – Эвристика Полларда, была в главе про вероятностные алгоритмы.
- (c)  $L_n(1/2, 2\sqrt{2})$  – алгоритм Диксона-Крайчика (есть на 3-м курсе)
- (d)  $L_n(1/2, 2)$  – метод эллиптических кривых (алгоритм Ленстры)
- (e)  $L_n(1/3, (32/9)^3)$  – **SNFS**

Здесь  $L_n(\alpha, c) = \mathcal{O}(e^{(c+o(1))(\log n)^\alpha})$ .

При  $0 < \alpha < 1$  получаем  $L_n$  между полиномом и экспонентой.

При  $\alpha = 1$  получаем  $L_n$  – ровно полином  $n^{c+o(1)}$ .

Обычно в RSA используют ключ длины  $k = 2048$ .

При шифровке/расшифровке используют  $\mathcal{O}(k)$  операций деления по модулю,

её мы скоро научимся реализовывать за  $\mathcal{O}(k^2/w^2)$  и  $\mathcal{O}(k \log^2 k)$ , оптимальное время –  $\mathcal{O}(k \log k)$ .

Итого: простейшая реализация RSA даёт время  $\mathcal{O}(k^3)$ , оптимальная –  $\mathcal{O}(k^2 \log k)$  и  $\mathcal{O}(k^3/w^2)$ .

#### • Взлом RSA.

На практике разобраны два случая:

- (a) Вариант взлома при  $e = 3$ .
- (b) Вариант взлома через Оракул, который ломает случайные сообщения с вероятностью 1%.

### 1.14. Протокол Диффи-Хеллмана

Есть Алиса и Боб. Им и вообще всем людям на Земле известны числа  $g$  и  $p$ .

Алиса и Боб хотят создать неизвестный более никому секретный ключ.

Для этого они пользуются следующим алгоритмом (протоколом):

1. Алиса генерируют большое случайное число  $a$ , Боб  $b$ .

2. Алиса передаёт Бобу по открытому каналу (любой может его слушать)  $g^a \bmod p$
3. Боб передаёт Алисе по открытому каналу  $g^b \bmod p$
4. Алиса знает  $\langle a, g^b \rangle \Rightarrow$  может вычислить  $g^{ab} \bmod p$ , аналогично Боб. Ключ готов.

Предполагается, что злоумышленник вмешаться в процесс передачи данных.

Злоумышленник в итоге знает  $g, g^a, g^b, p$ , но не знает  $a$  и  $b$ . Оказывается, что задача “получить  $g^{ab}$  по этим данным” не проще дискретного логарифмирования, а она не проще факторизации.

# Лекция #2: Линейные системы уравнений

3 сентября 2020

СЛАУ – Система линейных алгебраических уравнений.

Решением СЛАУ мы сейчас как раз и займёмся. Заодно научимся считать определитель матрицы, обращать матрицу, находить координаты вектора в базисе.

## • Постановка задачи.

$$\text{Дана система уравнений} \begin{cases} a_{00}x_0 + a_{01}x_1 + \dots + a_{0n-1}x_{n-1} = b_0 \\ a_{10}x_0 + a_{11}x_1 + \dots + a_{1n-1}x_{n-1} = b_1 \\ \dots \\ a_{m-10}x_0 + a_{m-11}x_1 + \dots + a_{m-1n-1}x_{n-1} = b_{m-1} \end{cases}$$

Для краткости, будем записывать  $Ax = b$ , где  $A$  – матрица,  $b$  – вектор-столбец.

Мы – программисты, поэтому нумерация везде с нуля.

Задача – найти какой-нибудь  $x$ , а лучше всё множество  $x$ -ов.

Все  $a_{ij}$ ,  $b_i$  – элементы поля (например,  $\mathbb{R}$ ,  $\mathbb{C}$ ,  $\mathbb{Z}/p\mathbb{Z}$ ), т.е. нам доступны операции  $+$ ,  $-$ ,  $*$ ,  $\backslash$ .

## 2.1. Гаусс для квадратных невырожденных матриц.

В данной части мы увидим классического Гаусса. Такого, как его обычно описывают.

Наша цель – превратить матрицу  $A$  в треугольную или диагональную.

Наш инструмент – можно менять уравнения местами, вычитать уравнения друг из друга.

Для удобства реализации сразу начнём хранить  $b_i$  в ячейке  $a_{in}$

План: для каждого  $i$  в ячейку  $a[i, i]$  поставить ненулевой элемент, и пользуясь им и вычитанием строк занулить все другие ячейки в  $i$ -м столбце.

```

1 // этот код работает только для  $m = n$ ,  $\det A \neq 0$ 
2 // тем не менее, чтобы не путаться в размерностях, мы в разных местах пишем и  $m$ , и  $n$ 
3 vector<vector<F>> a(m, vector<T>(n + 1)); //  $b$  хранится последним столбцом  $a$ 
4 for (int i = 0; i < n; i++) {
5     int j = i;
6     while (isEqual(a[j][i], 0)) // isEqual для  $\mathbb{R}$  обязана использовать  $\epsilon$ 
7         j++; // ненулевой элемент точно найдётся из невырожденности
8     swap(a[i], a[j]); // меняем строки местами, кстати, за  $\mathcal{O}(1)$ 
9     for (int j = 0; j < n; j++) // перебираем все строки
10         if (j != i) // если хотим получить диагональную
11             if (j > i) // если хотим получить треугольную
12                 if (!isEqual(a[j][i], 0)) { // хотим в  $[i, j]$  поставить 0, вычтем  $i$ -ю строку
13                     F coef = a[j][i] / a[i][i];
14                     for (int k = i; k <= n; k++) // самая долгая часть
15                         a[j][k] -= a[i][k] * coef;
16                 }
17 }
```

Строка 14 – единственная часть, работающая за  $\mathcal{O}(n^3)$ , поэтому для производительности важно, что цикл начинается с  $i$  (так можно, так как слева от  $i$  точно нули), а не с 0.

Если результат Гаусса – диагональная матрица, то  $x_i = b_i/a_{ii}$ .

Из треугольной матрицы  $x$ -ы нужно восстанавливать в порядке  $i = n-1..0$ :

$x_i = (b_i - \sum_{j=i+1..n-1} x_j \cdot a_{ij})/a_{ii}$ . Время восстановления  $\mathcal{O}(n^2)$ .

**Пример 2.1.1.** Работа Гаусса.

$$\left[ \begin{array}{ccc|c} \boxed{1} & 1 & 2 & 5 \\ 3 & 3 & 8 & 5 \\ 2 & 5 & 1 & 5 \end{array} \right] \xrightarrow{i=0} \left[ \begin{array}{ccc|c} 1 & 1 & 2 & 5 \\ 0 & 0 & 2 & -10 \\ 0 & 3 & -3 & -5 \end{array} \right] \xrightarrow{\text{swap}} \left[ \begin{array}{ccc|c} 1 & 1 & 2 & 5 \\ 0 & \boxed{3} & -3 & -5 \\ 0 & 0 & 2 & -10 \end{array} \right] \xrightarrow{i=1} \left[ \begin{array}{ccc|c} 1 & 0 & 3 & 6\frac{2}{3} \\ 0 & 3 & -3 & -5 \\ 0 & 0 & \boxed{2} & -10 \end{array} \right] \xrightarrow{i=2} \left[ \begin{array}{ccc|c} 1 & 0 & 0 & 21\frac{2}{3} \\ 0 & 3 & 0 & -20 \\ 0 & 0 & 2 & -10 \end{array} \right]$$

Итого:  $x = [21\frac{2}{3}, -6\frac{2}{3}, -5]$

Оценим время работы в худшем случае (всегда заходим в `if` в 12-й строке):

Для превращения в треугольную  $\sum_i i^2 \approx n^3/3$

Для превращения в диагональную  $\sum_i ni \approx n^3/2$

⇒ если важна скорость, приводите к  $\Delta$ -ой. Когда на первом месте удобство, к диагональной.

**2.2. Гаусс в общем случае**

Если мы подойдём в вопросу чисто математически, придётся ввести трапецевидные и ступенчатые матрицы. Возможно, нам захочется менять столбцы (переменные) местами.

Мы хотим сразу удобный универсальный код. Поэтому задачу сформулируем так:

*По одному добавляются пары  $\langle a_i, b_i \rangle \in \langle \mathbb{F}^n, \mathbb{F} \rangle$ , нам нужно поддерживать базис пространства  $\text{linear}\{a_1, a_2, \dots, a_m\}$  и поддерживать множество решений системы  $Ax = b$ .*

```

1 vector<vector<F>> A; // текущий базис и прикрепленные b-шки
2 vector<int> col; // для каждого базисного вектора храним номер столбца, который он обнуляет
3 bool add(vector<F> a) { // a[0], ..., a[n-1], b
4     for (size_t i = 0; i < A.size(); i++)
5         if (!isEqual(a[col[i]], 0)) {
6             F coef = a[col[i]] / A[i][col[i]];
7             for (size_t k = 0; k < a.size(); k++)
8                 a[k] -= A[i][k] * coef;
9         }
10    size_t i = 0;
11    while (i < a.size() && isEqual(a[i], 0))
12        i++;
13    if (i == a.size()) return 1; // уравнение - линейная комбинация предыдущих
14    if (i == a.size() - 1) return 0; // выразили из данных уравнений «0+...+0 = 1»
15    A.push_back(a), col.push_back(i); // добавили в базис новый вектор
16    return 1; // система всё ещё разрешима
17 }
```

Время работы добавления  $m$  векторов, если  $\dim \text{linear}\{a_1, \dots, a_m\} = k$ , работает за  $\mathcal{O}(mnk)$ .

Восстановим решение. Свободные переменные – ровно те, что не вошли в `col`.

```

1 vector<F> getX() { //  $\mathcal{O}(n+k^2)$ , что для маленьких  $k$  гораздо быстрее обычного  $\mathcal{O}(n^2)$ 
2     vector<F> x(n, 0); // пусть свободные переменные равны нулю
3     for (int i = A.size() - 1; i >= 0; i--) {
4         x[col[i]] = A[i][n]; // A[i].size() == n + 1
5         for (size_t j = i + 1; j < A.size(); j++)
6             x[col[i]] -= A[i][col[j]] * x[col[j]];
7         x[col[i]] /= A[i][col[i]];
8     }
9     return x; // нашли какое-то одно решение
10 }
```

Теперь запомним начальное `s = getX()` и переберём те столбцы  $j$ , которые являются свободными переменными. Для каждого  $j$  начнём восстановление ответа с `x[j] = 1`, и после строки

4 учтём слагаемое  $A[i][j] * x[j]$ . Результат для  $j$ -й переменной обозначим  $v_j$ .

Теперь у нас есть всё пространство решений:  $s + \text{linear}\{v_1 - s, v_2 - s, \dots, v_{n-k} - s\}$ .

Время:  $\mathcal{O}((n-k)(n+k^2))$ . Где  $(n-k)$  – размерность пространства решений.

### 2.3. Гаусс над $\mathbb{F}_2$

Самая долгая часть Гаусса – вычесть из одной строки другую, умноженную на число.

В  $\mathbb{F}_2$  вычитание –  $\oplus$ , а умножение &. Любимый нами `bitset` обе операции сделает за  $\mathcal{O}(n/w)$ .

Полученное ускорение применимо к обоим версиям Гаусса, описанным выше.

### 2.4. Погрешность

При вычислениях в  $\mathbb{Z}/p\mathbb{Z}$ , естественно, отсутствует. При вычислениях в  $\mathbb{R}$  она зашкаливает.

Рассмотрим *матрицу Гильберта*  $G$ :  $g_{ij} = \frac{1}{i+j}; i, j \in [1, n]$ .

Попробуем решить руками уравнение  $Gx = 0$ :  $\det G \neq 0 \Rightarrow \exists! x = \{0, \dots, 0\}$ .

Теперь применим Гаусса, реализованного в типе `double` при  $\varepsilon = 10^{-12}$ .

Уже при  $n = 11$  в процессе выбора ненулевого элемента мы не сможем отличить 0 от не нуля.

Окей! `double` (8 байт)  $\rightarrow$  `long double` (10 байт), и  $\varepsilon = 10^{-15}$ . Та же проблема при  $n = 17$ .

#### • Решения проблемы:

Обычно, чтобы хоть чуть-чуть уменьшить погрешность выбирают, не любой ненулевой элемент в столбце  $i$ , а тах по модулю элемент в подматрице  $[i, n] \times [i, n]$  (*эвристика тах элемента*).

В инкрементальном способе (добавлять строки по одной) эту оптимизацию не применить.

Во многих языках реализованы длинные вещественные числа, например, Java: `BigDecimal`.

Но всё равно возникает вопрос, какую точность выбрать? Содержательный ответ можно будет извлечь из курса “вычислительные методы”, а простой звучит так:

1. Запустите Гаусса два раза с  $k$  и  $2k$  значащими знаками.
2. Если ответы недопустимо сильно отличаются,  $k$  слишком мало, его нужно увеличить.

Ну и никто не отменял “детского” способа: если нам дана 1 секунда, выберем тах возможную точность, чтобы уложиться в секунду.

На некотором классе матриц меньшую погрешность даёт *метод простой Итерации*.

### 2.5. Метод итераций

Пусть нам нужно решить систему  $x = Ax + b$ . При этом  $\|A\| < 1$  (вы ведь помните про нормы?).

Решение: начнём с  $x_0 = \text{random}$ , будем пересчитывать  $x_{i+1} = Ax_i + b$ .

Сделаем сколько-то шагов, последний выдадим, как ответ. Сложность  $\mathcal{O}(n^2 t)$ ,  $t$  – число шагов.

Если бы система имела более простой вид  $x = Ax$ , мы могли бы вычислять быстрее:

$A \rightarrow A^2 \rightarrow A^4 \rightarrow \dots \rightarrow A^{2^k} \rightarrow (A^{2^k})x$ , итого сложность  $\mathcal{O}(n^3 \log t)$ .

Попробуем такой же фокус проверить с исходной системой  $x = Ax + b$ .

$$x_0 \rightarrow Ax_0 + b \rightarrow A^2x_0 + Ab + b \rightarrow \dots \rightarrow A^kx_0 + \underbrace{A^{k-1}b + \dots + Ab + b}_{\text{Обозначим } S_{i,k=2^i}}$$

Заметим:  $S_k = A^{k/2}S_{k-1} + S_{k-1} = (A^{k/2} + E)S_{k-1}$ .

База:  $T_0 = A, S_0 = b$ . Переход:  $\begin{cases} S_{i+1} = (T_i + E)S_i \\ T_{i+1} = T_i^2 \end{cases}$

$\|A\| = \sup_{|x|=1} |Ax|, \|A^{2^k}\| \leq \|A\|^{2^k} \xrightarrow{k \rightarrow \infty} 0 \Rightarrow$  слагаемым  $A^{2^k}x$  можно пренебречь.

## 2.6. Вычисление обратной матрицы

**Задача:** дана  $A$  над полем, найти  $X: AX = E$ .

Из  $\det A \cdot \det X = 1$  следует, что  $A$  невырождена  $\Rightarrow$  решение единственно.

Каждый столбец матрицы  $X$  задаёт систему уравнений  $\Rightarrow$  нахождение  $X$  за  $\mathcal{O}(n^4)$  очевидно.

Чтобы получить время  $\mathcal{O}(n^3)$ , заметим, что у систем матрица  $A$  общая, различны лишь столбцы  $b \Rightarrow$  системы можно решать одновременно.

Также, как мы записывали  $b_i$  в  $a_{in}$ , если есть сразу  $b_{i0}, b_{i1}, \dots, b_{ik}$ , то  $\forall j$  запишем  $b_{ij}$  в  $a_{in+j}$ .

Далее будем оперировать со строками длины  $n+k$ . Время работы  $\mathcal{O}((n+k)n^2) = \mathcal{O}(n^3)$ .

**Короткое изложение:** записали  $AE$  как матрицу из  $2n$  столбцов, привели Гауссом  $A$  к диагональному, а затем даже единичному виду  $\Rightarrow$  на месте  $E$  у нас как раз  $A^{-1}$ .

**Следствие 2.6.1.** Над  $\mathbb{F}_2$  обратную матрицу мы научились искать за  $\mathcal{O}(n^3/w)$ .

## 2.7. Гаусс для евклидова кольца

Напомним, *евклидово кольцо* – область целостности с делением с остатком (есть  $+$ ,  $-$ ,  $*$  и  $/$  с остатком). Например,  $\mathbb{Z}$ . Или  $\mathbb{R}[x]$  – многочлены,  $\mathbb{Z}[i]$  – Гауссовы числа.

Сейчас у нас получится чуть изменить обычного Гаусса, приводящего матрицу  $A$  к треугольной. А вот к диагональному виду, увы, привести не получится.

Основная операция в Гауссе – имея столбец  $i$ , строки  $i$  и  $j$  при  $a_{ii} \neq 0$  занулить  $a_{ji}$ .

Раньше мы вычитали из строки  $a_j$  строку  $a_i$ , умноженную на  $\frac{a_{ji}}{a_{ii}}$ . Теперь у нас нет деления...

Зато у нас есть деление с остатком  $\Rightarrow$  есть алгоритм Евклида.

Давайте на элементах  $a_{ji}$  и  $a_{ii}$  запустим Евклида. Только по ходу Евклида вычитать будем строки целиком. Результат:  $\forall k < i$  все  $a_{ik}, a_{jk}$  как были нулями, так и остались, а один из  $a_{ji}, a_{ii}$  занулился. Пример:

$$\left[ \begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & \boxed{7} & 5 & 4 & 1 \\ 0 & \boxed{3} & 6 & 3 & 1 \end{array} \right] \xrightarrow{7-3 \cdot 2} \left[ \begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & \boxed{1} & -7 & -2 & -1 \\ 0 & \boxed{3} & 6 & 3 & 1 \end{array} \right] \xrightarrow{3-1 \cdot 3} \left[ \begin{array}{cccc|c} 2 & 8 & 2 & 0 & 1 \\ 0 & 1 & -7 & -2 & -1 \\ 0 & 0 & 27 & 9 & 4 \end{array} \right]$$

Как этим можно пользоваться?

1. Для подсчёта определителя квадратной матрицы.
2. Для решения системы  $Ax = b$ , если нет свободных переменных (например,  $\det A \neq 0$ )

Если есть свободные переменные, то во время восстановления ответа по треугольной матрице в формуле  $x_i = (b_i - \sum_{j=i+1..n-1} x_j \cdot a_{ij})/a_{ii}$  у нас может “не поделиться”.

В случае  $\det A \neq 0$  это значило бы “нет решений”, а тут это значит “мы неправильно задали значения свободным переменным”.



*Замечание 2.7.1.* Если стоит задача “проверки невырожденности матрицы над  $\mathbb{Z}$ ”, то разумно, чтобы избежать длинных чисел, вычисления проводить не над  $\mathbb{Z}$ , а по большому простому модулю. При подсчёте определителя матрицы над  $\mathbb{Z}$  для борьбы с длинными числами можно использовать приём из [разд. 1.12.1](#).

## 2.8. Разложение вектора в базисе

Вернёмся в мир полей и безопасного деления.

Если нам дают базис пространства  $\{v_1, \dots, v_n\}$  и вектор  $p$ , просят разложить  $p$  в базисе  $v$ , проще всего решить систему уравнений  $Ax = p$ , где  $v_i$  – столбцы матрицы  $A$ . Время  $\mathcal{O}(n^3)$ .

Если нам дают сразу много векторов  $p_1, p_2, \dots, p_k$ , то мы дописываем их к  $A$ :  $[A | p_1 p_2 \dots p_k]$ , как делали в [разд. 2.6](#), и получаем итоговое время  $\mathcal{O}(n^2(n+k) + nk) = \mathcal{O}(n^3 + n^2k)$ .

Теперь будем решать online задачу – нам нужно сделать некий предподсчёт от базиса, а вектора  $p_i$  будут выдавать по одному. Раскладывать каждый  $p_i$  хочется за  $\mathcal{O}(n^2)$ .

Заметим, что в Гауссе [разд. 2.2](#) мы как раз по сути разложили вектор... только не на исходные вектора, а на текущие строки матрицы. Ок, давайте для каждой строки матрицы хранить коэффициенты  $c_{ij}$ :  $a_i = \sum_j c_{ij} v_j$ , где  $a_i$  – строки матрицы, а  $v_j$  – исходные вектора. Тогда для нового вектора  $a_k = \sum_{i=0..k-1} \alpha_i a_i = \sum_j v_j (\sum_i \alpha_i c_{ij})$ . Новые коэффициенты  $c_{kj} = \sum_i \alpha_i c_{ij}$ . Итого мы за  $\mathcal{O}(n^2)$  нашли коэффициенты строки, которую собираемся добавить в базис.

### 2.8.1. Ортогонализация Грама-Шмидта

Другой способ сделать базис удобным для пользования – привести его к ортонормированному виду. Далее следует описание Ортогонализации Грама-Шмидта, знакомое вам с алгебры:

```

1 for i=0..k-1
2   for j=0..i-1
3     v[i] -= v[j] * scalarProduct(v[i], v[j])
4   v[i] /= sqrt(scalarProduct(v[i], v[i]))

```

Получить координаты вектора  $p$  в новом милом базисе проще простого:  $x_i = \langle p, v_i \rangle$ .

Время разложения вектора в базисе –  $\mathcal{O}(n^2)$  сложений и умножений.

Опять же, если мы хотим координаты в исходном базисе, то для каждого  $v_i$  нам нужно будет таскать вектор коэффициентов:  $v_i = \sum_j \alpha_j v_j^*$  (выражение через исходный базис), и вычитать  $v_i$ , вычитать и вектора коэффициентов. Время разложения  $p$  в исходном базисе – тоже  $\mathcal{O}(n^2)$ .

## 2.9. Вероятностные задачи (на практике)

Рассмотрим оргграф, на рёбрах которого написаны вероятности.

Для каждой вершины верно, что сумма исходящих вероятностей равна 1.

Если бы мы хотели с некоторой вероятностью оставаться в вершине, добавили бы петлю.

Что нас может интересовать?

1. Начав в вершине  $v$ , в какой вершине с какой вероятностей мы находимся на бесконечности?
2. Какова вероятность, что, начав в  $v$  мы дойдём до вершины  $A$  раньше, чем до вершины  $B$  ( $A$  – спасти принцессу,  $B$  – свалиться в болото)?
3. Какое матожидание числа шагов в пути из вершины  $v$  в вершину  $A$ ?

4. Какое условное матожидание числа шагов в пути из вершины  $v$  в вершину  $A$ , если попадание в  $B$  – смерть?

• **Орграф ацикличен  $\Rightarrow$  динамика**

Если исходный орграф ацикличен + в нём могут быть петли, все задачи решаются динамикой.

Пример для матожидания без петли:  $E[v] = 1 + \sum p[v \rightarrow x] \cdot E[x]$

Пример для матожидания с петлёй вероятности  $q$ :

$$E[v] = 1 + (1 - q)(\sum p[v \rightarrow x] \cdot E[x]) + q \cdot E[v] \Rightarrow E[v] = \frac{1}{1-q} + \sum p[v \rightarrow x] \cdot E[x].$$

• **Произвольный орграф  $\Rightarrow$  итерации или Гаусс**

Рассмотрим вторую задачу. Тогда  $p[A] = 1, p[B] = 0$ , а на каждую другую вершину есть линейное уравнение  $p[v] = \sum p[v \rightarrow x] \cdot p[x]$ . Давайте решать!

Гаусс работает за  $\mathcal{O}(V^3)$ , обладает скверной погрешностью.

Метод итераций в данном случае будет работать так:

1. Изначально все кроме  $p[A]$  нули.
2. Затем мы все  $p[v]$  пересчитываем по формуле:  $p[v] = \sum p[v \rightarrow x] \cdot p[x]$ .

Можно пересчитывать возведением матрицы в степень:

$k$  фаз за  $\mathcal{O}(V^3 \log k)$ , можно в лоб за  $\mathcal{O}(Ek)$ .

*Замечание 2.9.1.* Полезно ознакомиться с практиками, домашними заданиями, разборами.

*Замечание 2.9.2.* Пусть получившуюся систему уравнений задаёт матрица  $A$ .

Если бы было  $\|A\| < 1$ , мы бы уже сейчас сказали про сходимость. Но  $\|A\| \leq 1$ , поэтому анализ сходимости метода итераций ждёт вас в курсе “численных методов”.

# Лекция #3: Быстрое преобразование Фурье

10 сентября 2020

Перед тем, как начать говорить “Фурье” то, “Фурье” сё, нужно сразу заметить:

Есть **непрерывное преобразование Фурье**. С ним вы должны столкнуться на теорвере.

Есть **тригонометрический ряд Фурье**. И есть общий ряд Фурье в гильбертовом пространстве, который появляется в начале курса функционального анализа.

Мы же с вами будем заниматься исключительно **дискретным преобразованием Фурье**.

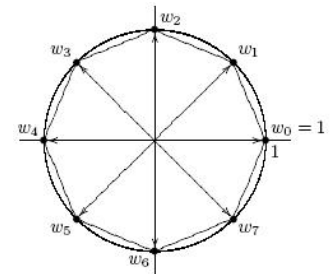
Коротко DFT (Discrete Fourier transform). FFT – по сути то же, первая буква означает “fast”.

**Задача:** даны два многочлена  $A, B$  суммарной длины  $\leq n$ , перемножить их за  $\mathcal{O}(n \log n)$ .

Длина многочлена –  $\gamma(A) = (\deg A) + 1$ . Вводим её, чтобы везде не писать “–1”.

Если даны  $n$  точек  $(x_i, y_i)$ , все  $x_i$  различны,  $\exists!$  интерполяционный многочлен длины  $n$ , построенный по этим точкам (из алгебры). Ещё заметим:  $\gamma(AB) = \gamma(A) + \gamma(B) - 1$ . Наш план:

1. Подобрать удачные  $k$  и точки  $w_0, w_1, \dots, w_{k-1}$ :  $k \geq \gamma(A) + \gamma(B) - 1 = n$ .
2. Посчитать значения  $A$  и  $B$  в  $w_i$ .
3.  $AB(x_i) = A(x_i)B(x_i)$ . Эта часть самая простая, работает за  $\mathcal{O}(n)$ .
4. Интерполировать  $AB$  длины  $k$  по полученным парам  $\langle w_i, AB(w_i) \rangle$ .



**Вспомним комплексные числа:**

$$e^{2\pi i \alpha} e^{2\pi i \beta} = e^{2\pi i (\alpha + \beta)} \quad e^{2\pi i \varphi} = (\cos \varphi, \sin \varphi), \quad \overline{(a, b)} = (a, -b) \Rightarrow \overline{e^{2\pi i \varphi}} = e^{2\pi i (-\varphi)}$$

$$\text{Извлечение корня } k\text{-й степени: } \sqrt[k]{z} = \sqrt[k]{e^{2\pi i \varphi}} = e^{2\pi i \varphi / k}$$

Если взять все корни из 1 степени  $2^t$ , возвести в квадрат, получатся ровно все корни степени  $2^{t-1}$ . Корни из 1 степени  $k$ :  $e^{2\pi i j / k}$ .

## 3.1. Прелюдия к FFT

Возьмём  $\min N = 2^t \geq n$  и  $w_j = e^{2\pi i j / N}$ . Тут мы предполагаем, что  $A, B \in \mathbb{C}[x]$  или  $A, B \in \mathbb{R}[x]$ .

Пусть есть многочлены  $A(x) = \sum a_i x^i$  и  $B(x) = \sum b_i x^i$ . Ищем  $C(x) = A(x)B(x)$ .

Обозначим их значения в точках  $w_0, w_1, \dots, w_{k-1}$ :  $A(w_i) = f a_i, B(w_i) = f b_i, C(w_i) = f c_i$ .

Схема быстрого умножения многочленов:

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

## 3.2. Собственно идея FFT

$$A(x) = \sum a_i x^i = (a_0 + x^2 a_2 + x^4 a_4 + \dots) + x(a_1 x + a_3 x^3 + a_5 x^5 + \dots) = P(x^2) + xQ(x^2)$$

Т.е. обозначили все чётные коэффициенты  $A$  многочленом  $P$ , а нечётные соответственно  $Q$ .

$\gamma(A) = n$ , все  $w_j^2 = w_{n/2+j}^2 \Rightarrow$  многочлены  $P$  и  $Q$  нужно считать не в  $n$ , а в  $\frac{n}{2}$  точках.

```

1 def FFT(a):
2     n = len(a)
3     if n == 1: return a # посчитать значение A(x) = a[0] в точке 1
4     a ---> p, q # разбили коэффициенты на чётные и нечётные
5     p, q = FFT(p), FFT(q)
6     w = exp(2pi*i/n) # корень из единицы n-й степени
7     for i=0..n-1: a[i] = p[i%(n/2)] + wi*q[i%(n/2)]
8     return a

```

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

### 3.3. Крутая реализация FFT

Чтобы преобразование работало быстро, нужно заранее предподсчитать все  $w_j = e^{2\pi i j/N}$ .

Заметим, что  $p$  и  $q$  можно хранить прямо в массиве  $a$ .

Тогда получается, что на прямом ходу рекурсии мы просто переставляем местами элементы  $a$ , и только на обратном 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 комплексные числа из STL: `complex<double>`.

```
1 const int K = 20, N = 1 << K;
2 complex<double> root[N];
3 int rev[N];
4 void init() {
5     for (int j = 0; j < N; j++) {
6         root[j] = exp(2*pi*i*j/N); // cos(2*pi*j/N), sin(2*pi*j/N)
7         rev[j] = (rev[j] >> 1) >> 1 + ((j & 1) << (K - 1));
8     }
9 }
```

Теперь, корни из единицы степени  $k$  хранятся в `root[j*N/k]`,  $j \in [0, k)$ . Две проблемы:

1. Доступ к памяти при этом не последовательный, проблемы с кешом.
2. Мы  $2N$  раз вычисляли тригонометрические функции.

⇒ можно лучше, вычисления корней #2:

```
1 for (int k = 1; k < N; k *= 2) {
2     num tmp = exp(pi/k);
3     root[k] = {1, 0}; // в root[k..2k) хранятся первые k корней степени 2k
4     for (int i = 1; i < k; i++)
5         root[k + i] = (i & 1) ? root[(k + i) >> 1] * tmp : root[(k + i) >> 1];
6 }
```

Теперь код собственно преобразования Фурье может выглядеть так:

```
1 void FFT(a, fa) { // a --> fa
2     for (int i = 0; i < N; i++)
3         fa[rev[i]] = a[i]; // можно иначе, но давайте считать массив «a» readonly
4     for (int k = 1; k < N; k *= 2) // уже посчитаны FFT от кусков длины k, база: k=1
5         for (int i = 0; i < N; i += 2 * k) // [i..i+k) [i+k..i+2k) --> [i..i+2k)
6             for (int j = 0; j < k; j++) { // оптимально написанный стандартный цикл FFT
7                 num tmp = root[k + j] * fa[i + j + k]; // вторая версия root[]
8                 fa[i + j + k] = fa[i + j] - tmp; // exp(2*pi*i*(j+n/2)/n) = -exp(2*pi*i*j/n)
9                 fa[i + j] = fa[i + j] + tmp;
10            }
11 }
```

### 3.4. Обратное преобразование

Теперь имея при  $w = e^{2\pi i/n}$ :

$$fa_0 = a_0 + a_1 + a_2 + a_3 + \dots$$

$$fa_1 = a_0 + a_1w + a_2w^2 + a_3w^3 + \dots$$

$$fa_2 = a_0 + a_1w^2 + a_2w^4 + a_3w^3 + \dots$$

...

Нам нужно научиться восстанавливать коэффициенты  $a_0, a_1, a_2, \dots$ , имея только  $fa_i$ .

Заметим, что  $\forall j \neq 0 \sum_{k=0}^{n-1} w^{jk} = 0$  (геометрическая прогрессия). А при  $j = 0$  получаем  $\sum_{k=0}^{n-1} w^{jk} = n$ .

$$\text{Поэтому } fa_0 + fa_1 + fa_2 + \dots = a_0n + a_1 \sum_k w^k + a_2 \sum_k w^{2k} + \dots = a_0n$$

$$\text{Аналогично } fa_0 + fa_1w^{-1} + fa_2w^{-2} + \dots = \sum_k a_0w^{-k} + a_1n + a_2 \sum_k w^k + \dots = a_1n$$

$$\text{И в общем случае } \sum_k fa_kw^{-jk} = a_jn.$$

Заметим, что это ровно значение многочлена с коэффициентами  $fa_k$  в точке  $w^{-j}$ .

Осталось заметить, что множества чисел  $\{w_j \mid j = 0..n-1\}$  и  $\{w_{-j} \mid j = 0..n-1\}$  совпадают  $\Rightarrow$

```
1 void FFT_inverse(fa, a) { // fa $\to$ a
2   FFT(fa, a)
3   reverse(a + 1, a + N) // $w^j \leftrightarrow w^{-j}$
4   for (int i = 0; i < N; i++) a[i] /= N;
5 }
```

### 3.5. Два в одном

Часто коэффициенты многочленов – вещественные числа.

Если у нас есть многочлены  $A(x), B(x) \in \mathbb{R}[x]$ , возьмём числа  $c_j = a_j + ib_j$  и посчитаем  $fc = FFT(c)$ . Тогда по  $fc$  за  $\mathcal{O}(n)$  можно легко восстановить  $fa$  и  $fb$ .

Для этого вспомним про сопряжения комплексных чисел:

$$\overline{x + iy} = x - iy, \overline{a \cdot b} = \overline{a} \cdot \overline{b}, w^{n-j} = w^{-j} = \overline{w^j} \Rightarrow \overline{fc_{n-j}} = \overline{C(w^{n-j})} = \overline{C(w^j)} \Rightarrow fc_j + \overline{fc_{n-j}} = 2 \cdot A(w^j) = 2 \cdot fa_j. \text{ Аналогично } fc_j - \overline{fc_{n-j}} = 2i \cdot B(w^j) = 2i \cdot fb_j.$$

Теперь, например, для умножения двух  $\mathbb{R}[x]$  можно использовать не 3 вызова FFT, а 2.

### 3.6. Умножение чисел, оценка погрешности

Общая схема умножения чисел:

цифра – коэффициент многочлена ( $x = 10$ ); умножим многочлены; сделаем переносы.

Число длины  $n$  в системе счисления 10 можно за  $\mathcal{O}(n)$  перевести в систему счисления  $10^k$ . Тогда многочлены будут длины  $n/k$ , умножение многочленов работать за  $\frac{n}{k} \log \frac{n}{k}$  (убывает от  $k$ ).

Возникает вопрос, какое максимальное  $k$  можно использовать?

Коэффициенты многочлена-произведения будут целыми числами до  $(10^k)^{2\frac{n}{k}}$ .

Чтобы в типе `double` целое число хранилось с погрешностью меньше 0.5 (тогда его можно правильно округлить к целому), оно должно быть не более  $10^{15}$ .

Получаем при  $n \leq 10^6$ , что  $(10^k)^2 10^6 / k \leq 10^{15} \Rightarrow k \leq 4$ .

Аналогично для типа `long double` имеем  $(10^k)^2 10^6 / k \leq 10^{18} \Rightarrow k \leq 6$ .

Это оценка сверху, предполагающая, что само FFT погрешность не накапливает... на самом деле эта оценка очень близка к точной.

# Лекция #4: Длинная арифметика

10 сентября 2020

Мы займёмся целыми беззнаковыми числами. Целые со знаком – ещё отдельно хранить знак. Вещественные – то же, но ещё хранить экспоненту:  $12.345 = 12345e-3$ , мы храним 12345 и  $-3$ .

Удобно хранить число в “массиве цифр”, младшие цифры в младших ячейках.

Во примерах ниже мы выбираем систему счисления  $\text{BASE} = 10^k$ ,  $k \rightarrow \max$ : нет переполнений.

Пусть есть длинное число  $a$ . При оценки времени работы будем использовать обозначения:

$|a| = n$  – битовая длина числа и  $\frac{n}{k}$  – длина числа, записанного в системе  $10^k$ . Помните,  $\max k \approx 9$ .

Если мы ленивы и уверены, что в процессе вычислений не появятся числа длиннее  $N$ , наш выбор – `int[N]`; , иначе обычно используют `vector<int>` и следят за длиной числа.

Примеры простейших операций:

```

1  const int N = 100, BASE = 1e9, BASE_LEN = 9;
2  void add( int *a, int *b ) { // сложение за  $O(n/k)$ 
3      for (int i = 0; i + 1 < N; i++) // +1, чтобы точно не было range check error
4          if ((a[i] += b[i]) >= BASE)
5              a[i] -= BASE, a[i + 1]++;
6  }
7  void subtract( int *a, int *b ) { // вычитание за  $O(n/k)$ ,  $a \geq b$ 
8      for (int i = 0; i + 1 < N; i++) // +1, чтобы точно не было range check error
9          if ((a[i] -= b[i]) < 0)
10             a[i] += BASE, a[i + 1]--;
11 }
12 int divide( int *a, int k ) { // деление на короткое за  $O(n/k)$ , делим со старших разрядов
13     long long carry = 0; // перенос с более старшего разряда, он же остаток
14     for (int i = N - 1; i >= 0; i--) {
15         carry = carry * BASE + a[i]; // максимальное значение carry <  $\text{BASE}^2$ 
16         a[i] = carry / k, carry %= k;
17     }
18     return carry; // как раз остаток
19 }
20 int mul_slow( int *a, int *b, int *c ) { // умножение за  $(n/k)^2$ 
21     fill(c, c + N, 0);
22     for (int i = 0; i < N; i++)
23         for (int j = 0; i + j < N; j++)
24             c[i + j] += a[i] * b[j]; // здесь почти наверняка произойдёт переполнение
25     for (int i = 0; i + 1 < N; i++) // сначала умножаем, затем делаем переносы
26         c[i + 1] += c[i] / BASE, c[i] %= BASE;
27 }
28 void out( int *a ) { // вывод числа за  $O(n/k)$ 
29     int i = N - 1;
30     while (i && !a[i]) i--;
31     printf("%d", a[i--]);
32     while (i >= 0) printf("%0*d", BASE_LEN, a[i--]); // воспользовались таки BASE_LEN!
33 }

```

Чтобы в строке 19 не было переполнения, нужно выбирать  $\text{BASE}$  так, что  $\text{BASE}^2 N$  помещалось в тип данных. Например, хорошо сочетаются  $\text{BASE} = 10^8, N = 10^3$ , тип – `unsigned long long`.

## 4.1. Бинарная арифметика

Пусть у нас реализованы простейшие процедуры: “+”, “-”, “\*2”, “/2”, “%2”, “>”, “≥”, “isZero”.

Давайте выразим через них “\*”, “\”, “gcd”. Обозначим  $|a| = n, |b| = m$ .

Умножение будет полностью изоморфно бинарному возведению в степень.

```

1 num mul(num a, num b) {
2   if (isZero(b)) return 0; // если храним число, как vector, то isZero за O(1)
3   num res = mul(mul2(a), div2(b));
4   if (mod2(b) == 1) add(res, a); // функция mod2 всегда за O(1)
5   return res;
6 }
```

Глубина рекурсии равна  $m$ . В процессе появляются числа не более  $(n+m)$  бит длины  $\Rightarrow$  каждая операция выполняется за  $O(\frac{n+m}{k}) \Rightarrow$  суммарное время работы  $O((n+m)\frac{m}{k})$ .

Если большее умножить на меньшее, то  $O(\max(n, m) \min(n, m)/k)$ .

Деление в чём-то похоже... деля  $a$  на  $b$ , мы будем пытаться вычесть из  $a$  числа  $b, 2b, 4b, \dots$

```

1 pair<num, num> div(num a, num b) { // найдём для удобства и частное, и остаток
2   num c = 1, res = 0;
3   while (b < a) // (n-m) раз
4     mul2(b), mul2(c);
5   while (!isZero(c)) { // Этот цикл сделает ≈ n-m итераций
6     if (a >= b) // O(n/k), так как длины a и b убывают от n до 1
7       sub(a, b), add(res, c); O(n/k)
8     div2(b), div2(c); O(n/k)
9   }
10  return {res, a};
11 }
```

Шагов главного цикла  $n-m$ . Все операции за  $O(\frac{n}{k}) \Rightarrow$  суммарное время  $O((n-m)\frac{n}{k})$ .

Наибольший общий делитель сделаем самым простым Евклидом “с вычитанием”.

Добавим только одну оптимизацию: если числа чётные, надо сразу их делить на два...

```

1 num gcd(num a, num b) {
2   int pow2 = 0;
3   while (mod2(a) == 0 && mod2(b) == 0)
4     div2(a), div2(b), pow2++;
5   while (!isZero(a) && !isZero(b)) {
6     while (mod2(a) == 0) div2(a);
7     while (mod2(b) == 0) div2(b);
8     if (a < b) swap(a, b);
9     a = sub(a, b); // одно из чисел станет чётным
10  }
11  if (isZero(a)) swap(a, b);
12  while (pow2-- > 0) mul2(a);
13  return a;
14 }
```

Шагов главного цикла не больше  $n+m$ . Все операции выполняются за  $\max(n, m)/k$ .

Отсюда суммарное время работы:  $O(\max(n, m)^2/k)$ .



## 4.2. Деление многочленов за $\mathcal{O}(n \log^2 n)$

Коэффициенты многочлена  $A(x)$ :  $A[0]$  – младший,  $A[\deg A]$  – старший.  $\gamma(A) = \deg A - 1$ .

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

Сперва простейшее решение за  $\mathcal{O}(\deg A \cdot \deg B)$ , призванное побороть страх перед делением:

```

1 pair<F*,F*> divide( int n, F *a, int m, F *b ) { // deg A = n, deg B = m, F - поле
2   F q[n-m+1];
3   for (int i = n - m; i >= 0; i--) { // выводим коэффициенты частного в порядке убывания
4     q[i] = a[i + m] / b[m]; // m - степень  $\Rightarrow b[m] \neq 0$ .
5     for (int j = 0; j <= m; j--) // конечно, вычитать имеет смысл, только если q[i]  $\neq 0$ 
6       a[i + j] -= b[j] * q[i]; // можно оптимизировать, перебирать только ненулевые b[j]
7   }
8   return {q, a}; // в a как раз остался остаток
9 }

```

Теперь перейдём к решению за  $\mathcal{O}(n \log^2 n)$ .

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  $B$ .

**Lm 4.2.1.**  $\deg A = m, \deg B = n \Rightarrow \deg Q = m - n$ , и  $Q$  зависит только от  $m - n + 1$  старших коэффициентов  $A$  и  $m - n + 1$  коэффициентов  $B$ .

*Доказательство.* Рассмотрим деление в столбик, шаг которого:  $A \leftarrow \alpha x^i B$ .  $\alpha = \frac{A[i+\deg B]}{B[\deg B]}$ . Поскольку  $i + \deg B \geq \deg B = n$ , младшие  $n$  коэффициентов  $A$  не будут использованы. ■

**Теперь будем решать задачу:**

Даны  $A, B \in \mathbb{R}[x]$ :  $\gamma(A) = \gamma(B) = n$ , найти  $C \in \mathbb{R}[x]$ :  $\gamma(C) = n$ , что у  $A$  и  $BC$  совпадают  $n$  старших коэффициентов.

```

1 int* Div( int n, int *A, int *B ) // n - степень двойки (для удобства)
2   C = Div(n/2, A + n/2, B + n/2) // нашли старших n/2 коэффициентов ответа
3   A' = Subtract(n, A, n + n/2 - 1, Multiply(C, B))
4   D = Div(n/2, A', B + n/2) // сейчас A' состоит из n/2 не нулей и n/2 нулей
5   return concatenate(D, C) // склеили массивы коэффициентов; вернули массив длины ровно n

```

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

**Время работы:**  $T(n) = 2T(n/2) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$ . Здесь  $\mathcal{O}(n \log n)$  – время умножения.

## 4.3. Деление чисел

Оптимально использовать метод Ньютона, внутри которого все умножения – FFT.

Тогда мы получим асимптотику  $\mathcal{O}(n \log n)$ . Об этом можно будет узнать на третьем курсе.

Сегодня лучшим результатом будет  $\mathcal{O}((n/k)^2)$ .

**Простейшие методы** (оценка времени деление числа битовой длины  $2n$  на число длины  $n$ ).

1. Бинпоиск по ответу:  $n^3/k^2$  при простейшем умножении,  $n^2 \log n$  при Фурье внутри.
2. Двоичное деление:  $n^2/k$  времени.
3. Деление в столбик:  $n^2/k, n^2/k^2$  времени. На нём остановимся подробнее.



#### 4.4. Деление чисел за $\mathcal{O}((n/k)^2)$

Делить будем в столбик. У нас уже было деление многочленов за квадрат. Нужно только научиться быстро искать старшую цифру частного.

• «Бинпоиск»:  $\mathcal{O}(n^2/k)$

Ищем старшую цифру частного бинарным поиском.  $\log \text{BASE} = k$  итераций бинпоиска, на каждой проверяем за  $\mathcal{O}(n/k)$ . Итоговое время деления  $\mathcal{O}((n/k) \times k \times (n/k)) = \mathcal{O}(n^2/k)$ .

• «Деление старших цифр»:  $\mathcal{O}(n^2/k^2)$

Найдём старшую цифру ещё быстрее. Для краткости обозначим  $S := \text{BASE}$ .

Пусть даны числа  $a = \sum_{i=0}^{l_a} a_i \cdot S^i$  ( $a_{l_a} > 0$ ),  $b = \sum_{i=0}^{l_b} b_i \cdot S^i$  ( $b_{l_b} > 0$ ).

(То есть  $|a| = n = l_a \cdot k$ ,  $|b| = m = l_b \cdot k$ .)

Положим  $\beta = b_{l_b} \cdot S + b_{l_b-1}$  — это число, образованное первыми двумя цифрами числа  $b$ . Мы хотим взять число, образованное первыми несколькими цифрами числа  $a$ , поделить его на  $\beta$  и получить хорошее приближение старшей цифры частного.

Одной цифры числа  $a$  точно не хватит: в результате деления всегда получится число, меньшее единицы. С другой стороны, трёх цифр не может быть мало: в целой части частного мы получим одну или две цифры.

Положим  $\alpha = \begin{cases} a_{l_a} \cdot S + a_{l_a-1}, & \text{если } a_{l_a} \cdot S + a_{l_a-1} \geq \beta; \\ a_{l_a} \cdot S^2 + a_{l_a-1} \cdot S + a_{l_a-2}, & \text{иначе.} \end{cases}$

*Замечание 4.4.1.*  $1 \leq \alpha/\beta < S$ , то есть целая часть частного состоит ровно из одной  $S$ -ичной цифры. Поскольку  $\alpha$ ,  $\beta$  и  $S$  целые, из этого получаем  $\beta \leq \alpha \leq \beta S - 1$ .

*Доказательство.*  $\alpha \geq \beta$  по определению  $\alpha$ , так что  $1 \leq \alpha/\beta$ .

$\alpha/\beta < S$ , поскольку иначе в  $\alpha$  можно было бы взять на одну цифру меньше. ■

Наконец, сформулируем центральное утверждение этого параграфа:

**Теорема 4.4.2.** Старшая цифра числа  $\frac{a}{b}$  лежит в отрезке  $\left[ \lfloor \frac{\alpha}{\beta} - 1 \rfloor, \lfloor \frac{\alpha}{\beta} + 1 \rfloor \right]$ .

**Алгоритм.** Отсюда мы сразу получаем алгоритм деления за  $\mathcal{O}(n^2/k^2)$ . Теперь вместо бинпоиска по старшей цифре мы вычисляем за  $\mathcal{O}(1)$  её приближение  $\frac{\alpha}{\beta}$ , отличающееся от правильной цифры не более чем на 2. Проверяя не более трёх кандидатов (каждая проверка за  $\mathcal{O}(n/k)$ ), узнаём правильную старшую цифру.

Чуть подробнее про реализацию.  $(n - m)/k$  раз (это длина частного), делаем следующее:

1. вычисляем приближение старшей цифры частного  $\tilde{c} = \lfloor \frac{\alpha}{\beta} - 1 \rfloor$  за  $\mathcal{O}(1)$ ;
2. вычитаем из  $a$  число  $\tilde{c} \cdot b \cdot S^i$  (умножение на цифру и вычитание за  $\mathcal{O}(m/k)$ );
3. делаем ещё не более *двух* вычитаний, пока  $a$  остаётся положительным.

Итого  $\mathcal{O}((n - m)t/k^2)$  времени.

**Доказательство теоремы.** Поставим в  $a$  запятую после префикса  $\alpha$ , а в  $b$  — после префикса  $\beta$ , полученные (нецелые) числа назовём  $a'$  и  $b'$ .

Формально,  $a' = a \cdot S^{-x}$  и  $b' = b \cdot S^{-y}$  для целых  $x$  и  $y$  таких, что  $\lfloor a' \rfloor = \alpha$  и  $\lfloor b' \rfloor = \beta$ . (На самом деле, мы знаем, что  $y = -l_b + 1$ .)

$\lfloor \frac{a'}{b'} \rfloor$  будет искомой старшей цифрой  $\frac{a}{b}$ . (Здесь есть небольшая тонкость. Какая?) Поэтому осталось показать, что расстояние между  $\frac{a'}{b'}$  и  $\frac{a}{b}$  не больше единицы.

Положим  $\ell := \frac{\alpha}{\beta+1}$  и  $r := \frac{\alpha+1}{\beta}$ .

**Lm 4.4.3.**  $\frac{a}{b} \in [\ell, r]$ .

*Доказательство.*  $\frac{a}{b} \geq \ell$ : числитель такой же, знаменатель меньше.

$\frac{a}{b} \leq r$ : числитель меньше, знаменатель такой же. ■

**Lm 4.4.4.**  $\frac{a'}{b'} \in [\ell, r]$ .

*Доказательство.*  $\frac{a'}{b'} \geq \ell$ :  $\lfloor a' \rfloor = \alpha \Rightarrow a' \geq \alpha$ ,  $\lfloor b' \rfloor = \beta \Rightarrow b' \leq \beta + 1$ . Таким образом, числитель больше, знаменатель меньше.

$\frac{a'}{b'} \leq r$ :  $\lfloor a' \rfloor = \alpha \Rightarrow a' \leq \alpha + 1$ ,  $\lfloor b' \rfloor = \beta \Rightarrow b' \geq \beta$ . Таким образом, числитель меньше, знаменатель больше. ■

Таким образом,  $\frac{a}{b}$  и  $\frac{a'}{b'}$  оба лежат в  $[\ell, r]$ . Поэтому расстояние между ними не больше  $r - \ell$ .

**Lm 4.4.5.**  $r - \ell \leq 1$ .

*Доказательство.*

$$r - \ell = \frac{\alpha + 1}{\beta} - \frac{\alpha}{\beta + 1} = \frac{(\alpha + 1)(\beta + 1) - \alpha\beta}{\beta(\beta + 1)} = \frac{\alpha + \beta + 1}{\beta(\beta + 1)} \stackrel{4.4.1}{\leq} \frac{(\beta S - 1) + \beta + 1}{\beta(\beta + 1)} = \frac{\beta(S + 1)}{\beta(\beta + 1)} = \frac{S + 1}{\beta + 1} \leq 1,$$

где последнее неравенство следует из  $\beta \geq S$ , поскольку это двухзначное число). ■

# Лекция #5: Умножение матриц и 4 русских

17 сентября 2020

## 5.1. Четыре русских: общие слова

Если задач какого-то вида мало, давайте предподсчитаем заранее ответы для всех возможных задач такого вида, а затем будем ими пользоваться в нужный момент за  $\mathcal{O}(1)$ .

## 5.2. Битовое сжатие в лоб

Пусть мы умножаем две матрицы над  $\mathbb{F}_2$ :  $C = A \times B$ .

$C_{ij} = \oplus_k (A_{ik} \& B_{kj})$ , разобьём эту сумму на части длины  $k$ .

Тогда, чтобы за  $\mathcal{O}(1)$  посчитать сумму сразу  $k$  слагаемых, достаточно

1. Так хранить строки  $A$  и столбцы  $B$ , чтобы за  $\mathcal{O}(1)$  получать целое число из нужных  $k$  бит.
2. Взять AND двух  $k$ -битных чисел.
3. Посчитать число бит в  $k$ -битном числе. Предподсчёт за  $\mathcal{O}(2^k)$ : `bn[i] = bn[i>>1] + (i&1)`.

Возьмём  $k = \log n$ , получим  $\mathcal{O}(n)$  на предподсчёт и  $\mathcal{O}(n^3 / \log n)$  на умножение.

На практике разумно делать предподсчёт, например, для  $k = 20$  при  $w = 32$  (размер машинного слова). Т.е. по сути алгоритм работает за  $n^3/w$ . А при  $n \leq k$  даже за  $\mathcal{O}(n^2)$ .

*Замечание 5.2.1.* Мы научились умножать не только над  $\mathbb{F}_2$ , но и над  $\mathbb{Z}$  (в предположении, что исходные матрицы содержат только нули и единицы).

## 5.3. Битовое сжатие строк над $\mathbb{F}_2$

Скажем, что строки матриц  $B$  и  $C$  – bitset-ы.

Тогда  $c_i$  ( $i$ -я строка  $C$ ) – сумма ровно тех строк  $B$ , которые помечены единицами в  $a_i$

```
1 for i in range(n):
2     for j in range(n):
3         if a[i][j]:
4             c[i] ^= b[j]
```

Итого простой и короткий код, асимптотика  $\mathcal{O}(n^3/w)$ .

## 5.4. Применяем идею 4-х русских

Сейчас будем умножать над  $\mathbb{Z}$  ( $\forall$  кольцо), предполагая, что  $A$  содержит только нули и единицы.

Разобьём  $A$  на части по  $k$  **столбцов**:  $A = A_1 A_2 \dots A_{n/k}$ .

Разобьём  $B$  на части по  $k$  **строк**:  $B = B_1 B_2 \dots B_{n/k}$ .

Заметим,  $A \times B = \sum_{i=1..n/k} (A_i \times B_i)$ . Проверьте размерности:  $(n \times k) \times (k \times n) = n \times n$ .

Теперь сосредоточимся на умножении  $A_1 \times B_1$ .

Каждая строка  $i$  матрицы  $A_1$  задаёт  $i$ -ю строку произведения, которая является суммой тех строк  $B_1$ , которые помечены единицами в  $i$ -й строке  $A_1$ .

Вспомним, что  $A_1$  состоит из  $\{0, 1\}$ . Рассмотрим все  $2^k$  строк из  $\{0, 1\}$  длины  $k$  и для всех них за  $2^k n$  предподсчитаем результат (сумму, являющуюся строкой длины  $n$ ):

`sum[p] = add(sum[p ^ (1 << bit)], b[bit])`, где

`bit` – любой единичный бит  $p$ , а функция `add` за  $\mathcal{O}(n)$  складывает строки.

Получили, что время на  $A_1 \times B_1$  это  $2^k n + n^2$ , чтобы это осталось  $\Theta(n^2)$ , максимальное  $k$  мы

можем взять  $\Theta(\log n)$ . Итого  $\frac{n}{k}$  умножение по  $\Theta(n^2)$  каждое  $\Rightarrow \mathcal{O}(n^3/\log n)$  в сумме.

### 5.5. Умножение матриц над $\mathbb{F}_2$ за $\mathcal{O}(n^3/(w \log n))$

Если в явном виде применить идеи из двух последних параграфов, то получится ровно  $\mathcal{O}(n^2/\log n)$  операций со строками, каждая за  $\mathcal{O}(n/w) \Rightarrow \mathcal{O}(n^3/(w \log n))$ .

### 5.6. НОП за $\mathcal{O}(n^2/\log^2 n)$ (на практике)

**Задача:** даны две строки над алфавитом  $\{0, 1\}$ , найти длину НОП.

Рассмотрим обычную динамику: 
$$f[i, j] = \begin{cases} f[i-1, j-1] + 1 & \text{если } s[i] = t[j] \\ \max(f[i-1, j], f[i, j-1]) & \text{иначе} \end{cases}$$

Идея: зафиксируем  $k = \frac{1}{4} \log n$  и будем за  $\mathcal{O}(1)$  сразу насчитывать кусок матрицы  $k \times k$ .

Заметим  $\forall i, j \ 0 \leq f[i+1, j] - f[i, j] \leq 1 \wedge 0 \leq f[i, j+1] - f[i, j] \leq 1$ . Также  $\forall i \ f[i, 0] = f[0, i] = 0$ . Давайте хранить только битовые матрицы  $x[i, j] = f[i, j+1] - f[i, j]$ ,  $y[i, j] = f[i+1, j] - f[i, j]$ . Зафиксируем  $k$  и любую клетку  $[i, j]$  пусть мы знаем “угол квадрата”:  $y[i..i+k, j]$  и  $x[i, j..j+k]$ . Противоположный “угол квадрата” ( $y[i..i+k, j]$  и  $x[i, j..j+k]$ ) зависит только от  $4k$  бит:  $y[i..i+k, j], x[i, j..j+k], s[i..i+k], t[j..j+k] \Rightarrow$  используем четырёх русских и за  $\mathcal{O}^*(2^{4k})$  всё предподсчитываем. Важно, что  $y[i..i+k, j], x[i, j..j+k], y[i..i+k, j], x[i, j..j+k], s[i..i+k], t[j..j+k]$  – целые  $k$ -битные числа  $\Rightarrow$  операции с ними происходят за  $\mathcal{O}(1)$ .

Заметьте, для каждого квадрата  $k \times k$  мы получали только значения на границы.

Другие нам и не нужны. Также мы ни в какой момент времени не пытались считать  $f$ , нам хватает  $x$  и  $y$ . Реальные значения  $f$  возникают внутри предподсчёта.

Ещё нам в самом конце нужна собственно длина НОП – это сумма последней строки  $x$ .

# Лекция #6: Базовые алгоритмы на строках

17 сентября 2020

## 6.1. Обозначения, определения

$s, t$  – строки,  $|s|$  – длина строки,  $\bar{s}$  – перевёрнутая  $s$ ,  
 $s[l:r)$  и  $s[l:r]$  – подстроки,  
 $s[0:i)$  – префикс,  $s[i:|s|-1] = s[i:]$  – суффикс.  
 $\Sigma$  – алфавит,  $|\Sigma|$  – размер алфавита.  
 Говорят, что  $s$  – подстрока  $t$ , если  $\exists l, r: s = t[l:r)$ .

## 6.2. Поиск подстроки в строке

Даны текст  $t$  и строка  $s$ . Ещё иногда говорят “строка (string)  $t$  и образец (pattern)  $s$ ”.

Вхождением  $s$  в  $t$  назовём позицию  $i: s = t[i:i+|s|)$ .

Возможны различные формулировки задачи поиска подстроки в строке:

- Проверить, есть ли хотя бы одно вхождение  $s$  в  $t$ .
- Найти количество вхождений  $s$  в  $t$ .
- Найти позицию любого вхождения  $s$  в  $t$ , или вернуть  $-1$ , если таких нет.
- Вернуть множества всех вхождений  $s$  в  $t$ .

### 6.2.1. C++

В языке C++ у строк типа `string` есть стандартный метод `find`. Работает за  $\mathcal{O}(|s| \cdot |t|)$ , возвращает целое число – номер позиции в исходной строке, начиная с которого начинается первое вхождение подстроки или `string::npos`.

Функция из `<cstring>` `strstr(t, s)` ищет  $s$  в  $t$ . Работает **за линию** в Unix, **за квадрат** в Windows.

В обоих случаях квадрат имеет очень маленькую константу (AVX-регистры).

Все вхождения можно перечислить таким циклом:

```
1 for (size_t pos = t.find(s); pos != string::npos; pos = st.find(s, pos + 1))
2   ; // pos - позиция вхождения
```

или таким

```
1 for (char *p = t; (p = strstr(p, s)) != 0; p++)
2   ; // p - указатель на позицию вхождения в t
```

### 6.2.2. Префикс функция и алгоритм КМП

**Def 6.2.1.**  $\pi_0(s)$  – длина *максимального собственного префикса*  $s$ , совпадающего с суффиксом  $s$ .

**Def 6.2.2.** Префикс-функция строки  $s$  – массив  $\pi(s): \pi(s)[i] = \pi_0(s[0:i))$ .

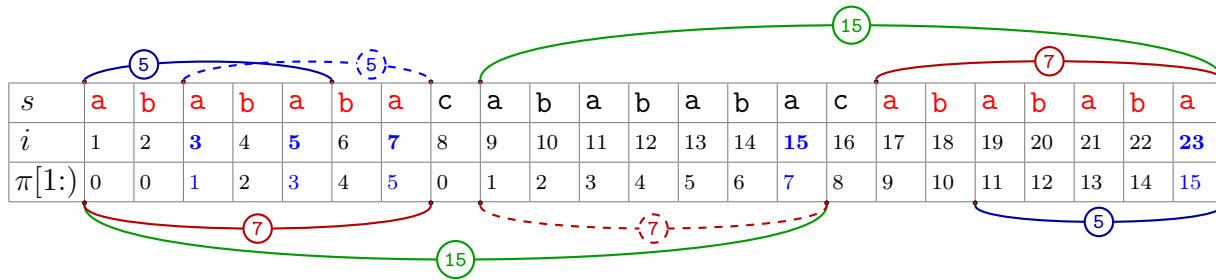
Когда из контекста понятно, о префикс-функции какой строки идёт речь, пишут просто  $\pi[i]$ .

#### • Алгоритм Кнута-Мориса-Пратта.

Пусть ‘#’ – любой символ, который не встречается ни в  $t$ , ни в  $s$ . Создадим новую строку  $w = s\#t$  и найдем её префикс-функцию. Благодаря символу ‘#’  $\forall i \pi(w)[i] \leq |s|$ . Такие  $i$ , что  $\pi(w)[i] = |s|$ , задают позиции окончания вхождений  $s$  в  $w \Rightarrow (j = i - 2|s| - 1)$  – начало вхождения  $s$  в  $t$ .

- Вычисление префикс-функции за  $\mathcal{O}(n)$ .

Все префиксы равные суффиксам для строки  $s[0:i)$  это  $X = \{i, \pi[i], \pi[\pi[i]], \pi[\pi[\pi[i]]], \dots\}$



Пример (см. картинку выше): для всей строки  $s$  имеем  $X = \{23, 15, 7, 5, 3, 1\}$ .

Заметим, что или  $\pi[i+1] = 0$ , или  $\pi[i+1]$  получается, как  $x + 1$  для некоторого  $x \in X$ .

Будем перебирать  $x \in X$  в порядке убывания, получается следующий код:

```

1 p[1] = 0, n = |s|
2 for (i = 2; i <= n; i++)
3     int k = p[i - 1];
4     while (k > 0 && s[k] != s[i - 1])
5         k = p[k];
6     if (s[k] == s[i - 1])
7         k++;
8     p[i] = k;

```

Заметим, что с учётом последней строки цикла первая не нужна, получаем:

```

1 p[1] = k = 0, n = |s|
2 for (i = 2; i <= n; i++)
3     while (k > 0 && s[k] != s[i - 1])
4         k = p[k];
5     if (s[k] == s[i - 1])
6         k++;
7     p[i] = k;

```

Вычисление префикс функции работает за  $\mathcal{O}(n)$ , так как  $k$  увеличится  $\leq n$  раз  $\Rightarrow$  суммарное число шагов цикла **while** не более  $n$ .

Собственно КМП работает за  $\mathcal{O}(|s| + |t|)$  и требует  $\mathcal{O}(|s| + |t|)$  дополнительной памяти.

*Упражнение:* придумайте, как уменьшить количество допамяти до  $\mathcal{O}(|s|)$ .

### 6.2.3. LCP

**Def 6.2.3.**  $\text{lcp}[i, j]$  (*largest common prefix*) для строки  $s$  – длина наибольшего общего префикса суффиксов  $s[i:]$  и  $s[j:]$ .

Вычислить массив  $\text{lcp}$  можно за  $\mathcal{O}(n^2)$ , так как  $\text{lcp}[i, j] = \begin{cases} 1 + \text{lcp}[i+1, j+1] & s[i] = s[j] \\ 0 & \text{иначе} \end{cases}$

Аналогично можно определить и вычислить массив  $\text{lcp}$  для двух разных строк.

### 6.2.4. Z-функция

**Def 6.2.4.**  $Z$ -функция – массив  $z$  такой, что  $z[0] = 0, \forall i > 0 \ z[i] = \text{lcp}[0, i]$ .

Для поиска подстроки снова введем  $w = s\#t$  и посчитаем  $Z(w)$ .

Найдем все позиции  $i$ :  $Z(w)[i] = |s|$ . Это позиции всех вхождений строки  $s$  в строку  $t$ .

Осталось научиться вычислять  $Z$ -функцию за линейное время.

```

1 z[0] = 0;
2 for (i = 1; i < n; i++)
3     int k = 0;
4     while (s[i + k] == s[k])
5         k++;
6     z[i] = k;
```

Приведенный алгоритм работает за  $\mathcal{O}(n^2)$ . На строке  $aaa...a$  оценка  $n^2$  достигается.

Ключом к ускорению является следующая лемма:

**Lm 6.2.5.**  $\forall l < i < l + z[l] = r$  имеем  $s[0:z[l]] = s[l:r]$  и  $s[i-l:z[l]] = s[i:r]$ .

Следствие леммы:  $z[i] \geq \min(r - i, z[i - l])$ . Логично взять  $l: r = l + z[l] = \max$ .

Немного модифицируем код, чтобы получить асимптотику  $\mathcal{O}(n)$ .

```

1 z[0] = 0, l = r = 0;
2 for (i = 1; i < n; i++)
3     int k = max(0, min(r - i, z[i - l]))
4     while (s[i + k] == s[k])
5         k++
6     z[i] = k
7     if (i + z[i] > r) l = i, r = i + z[i]
```

**Теорема 6.2.6.** Приведенный выше алгоритм работает за  $\mathcal{O}(n)$ .

*Доказательство.*  $k++ \Rightarrow r++$ , а  $r$  может увеличиваться  $\leq n$  раз. ■

## 6.3. Полиномиальные хеши строк

Основная идея этой секции – научиться с предподсчетом за  $\mathcal{O}$ (суммарной длины строк) вероятно сравнить на равенство любые их подстроки за  $\mathcal{O}(1)$ .

Например, мы уже умеем считать частичные суммы за  $\mathcal{O}(1) \Rightarrow$  можем за  $\mathcal{O}(1)$  проверить, равны ли суммы символов в подстроках. Если не равны  $\Rightarrow$  строки точно не равны...

**Def 6.3.1.** Хеш-функция объектов из мн-ва  $A$  в диапазон  $[0, m)$  – любая функция  $A \rightarrow \mathbb{Z}/m\mathbb{Z}$ .

Например, сумма символов строки, посчитанная по модулю 256 – пример хеш-функции из множества строк в диапазон  $[0, 256)$ . Задача – придумать более удачную хеш-функцию.

### • Полиномиальная хеш-функция

**Def 6.3.2.** Пусть  $s = s_0s_1\dots s_{n-1} \Rightarrow h_{p,m}(s) = (s_0p^{n-1} + s_1p^{n-2} + \dots + s_{n-1}) \bmod m$

$h_{p,m}(s)$  – полиномиальный хеш для строки  $s$  посчитанный в точке  $p$  по модулю  $m$ .

По сути мы взяли многочлен (полином) с коэффициентами “символы строки” и посчитали его значение в точке  $p$  по модулю  $m$ . Можно было бы определить  $h_{p,m}(s) = \sum_i s_i p^i$ , но при реализации нам будет удобен порядок суммирования из 6.3.2.

```

1 typedef long long T;
2 T *h;
3 void initialize( int n, char* s ) {
4     h = new T[n + 1]; // h[i] - хеш префикса s[0:i]
5     h[0] = 0; // хеш пустой строки действительно 0...
6     for (int i = 0; i < n; i++)
7         h[i + 1] = ((__int128)h[i] * p + s[i]) % m; // 0 < m < 2^63
8 }
9 T getHash( int l, int r, char* s ) { // [l,r)
10    // deg[r - l] = p^{r-l}, никогда не пишите здесь лишний if
11    T res = (h[r] - (__int128)h[l] * deg[r - l]) % m;
12    return res < 0 ? res + m : res;
13 }
```

**Def 6.3.3.** Коллизия хешей – ситуация вида  $s \neq t$ ,  $h_{p,m}(s) = h_{p,m}(t)$ .

Займёмся точными оценками чуть позже, пока предположим, что  $\forall$  простого  $m$ , если мы выбираем  $p$  равномерно в  $[0, m)$ , вероятность коллизии при одном сравнении равна  $\frac{1}{m}$ .

Из умения сравнивать строки на равенство за  $\mathcal{O}(1)$  следует алгоритм поиска строки в тексте:

### 6.3.1. Алгоритм Рабина-Карпа

Можно искать  $s$  в  $t$ , предподсчитав полиномиальные хеши для  $t$  и для каждого потенциального вхождения  $[i, i+|s|)$  сравнить за  $\mathcal{O}(1)$  хеш подстроки  $t[i, i+|s|)$  с хешом  $s$ .

Если хеши совпали, то возможно два развития событий: мы можем

или проверить за линию равенство строк, или, не проверяя, выдать вхождение  $i$ .

Для задачи поиска одного вхождения в первом случае мы получили RP, во втором ZPP.

На практике используют оба подхода.

Для задачи поиска всех вхождений проверять каждое вхождение за линию – слишком долго.

### • Оптимизируем память.

Преимущество Рабина-Карпа над  $\pi$ -функцией и  $Z$ -функцией – реализация с  $\mathcal{O}(1)$  доппамяти. Обозначим  $n = |s|$  и  $ht_i = h_{p,m}(t[i : i + n])$ . Посчитаем  $h_{p,m}(s)$ ,  $ht_0$  и  $p^n$ .

Осталось, зная  $ht_i$ , научиться считать  $ht_{i+1} = ht_i \cdot p - t_i \cdot p^n$ .

### 6.3.2. Наибольшая общая подстрока за $\mathcal{O}(n \log n)$

**Задача:** даны строки  $s$  и  $t$ , найти  $w$ :  $w$  – подстрока  $s$ , подстрока  $t$ ,  $|w| \rightarrow \max$ .

Заметим, что если  $w$  – общая подстрока  $s$  и  $t$ , то любой её префикс тоже  $\Rightarrow$

функция  $f(k) = \text{“есть ли у } s \text{ и } t \text{ общая подстрока длины } k\text{”}$  – монотонный предикат  $\Rightarrow$



максимальное  $k: f(k) = 1$  можно найти бинарным поиском ( $\mathcal{O}(\log \min(|s|, |t|))$  итераций).

Осталось для фиксированного  $k$  за  $\mathcal{O}(|s| + |t|)$  проверить, есть ли у  $s$  и  $t$  общая подстрока длины  $k$ . Сложим хеши всех подстрок  $s$  длины  $k$  в хеш-таблицу. Для каждой подстроки  $t$  длины  $k$  поищем её хеш в хеш-таблице. Если нашли совпадение, как и в Рабине-Карпе есть два пути – проверить за линию или сразу поверить в совпадение. Оба метода работают.

### 6.3.3. Оценки вероятностей

#### • Многочлены

Пусть  $m$  – простое число.

**Lm 6.3.4.** Число корней многочлена степени  $n$  над  $\mathbb{Z}/m\mathbb{Z}$  не более  $n$ .

**Lm 6.3.5.** Пусть  $t \in \mathbb{Z}/m\mathbb{Z} \Rightarrow \Pr[P(t) = 0] = \frac{1}{m}$ , где  $P$  – случайный многочлен.

**Lm 6.3.6.** Матожидание числа корней случайного многочлена степени  $n$  над  $\mathbb{Z}/m\mathbb{Z}$  равно 1.

Первую лемму вы знаете из курса алгебры, третья сразу следует из второй ( $m$  корней, каждый с вероятностью  $1/m$ ), вторая получается из того, что  $P(x) = Q(x)(x-t) + r$ : у случайного  $P(x)$  все  $r$  равновероятны.

Нам важна простота модуля  $m$ , для непростого оценки неверны.

Например, у многочлена  $x^{64}$  при  $m = 2^{64}$  любое чётное число является корнем.

#### • Связь со строками

$h_{p,m}(S) = S(p) \bmod m$ , где  $S$  – и строка, и многочлен с коэффициентами  $S_i$ .

Тогда  $h_{p,m}(S) = h_{p,m}(T) \Leftrightarrow (S - T)(p) \equiv 0 \bmod m$ .

Запишем несколько следствий из лемм про многочлены.

*Следствие 6.3.7.*  $\forall$  пары  $\langle p, m \rangle$  вероятность совпадения хешей случайных строк  $s$  и  $t - \frac{1}{m}$ .

*Доказательство.* Разность многочленов  $s$  и  $t$  – случайный многочлен  $(s - t)$ . Далее 6.3.5. ■

*Следствие 6.3.8.*  $\forall m, \forall s, t \Pr_p[h_{p,m}(s) = h_{p,m}(t)] \leq \frac{\max(|s|, |t|)}{m}$ .

По-русски: даны фиксированные строки, выбираем случайное  $p$ , оцениваем  $\Pr$  коллизии.

*Доказательство.* Подставим многочлен  $(s - t)$  в лемму 6.3.4. ■

Теперь пусть сравнений строк было много.

**Теорема 6.3.9.** Пусть дано множество **случайных** различных строк, и сделано  $k$  сравнений  $\langle p, m \rangle$  хешей каких-то из этих строк  $\Rightarrow$  вероятность существования коллизии не более  $\frac{k}{m}$ .

*Доказательство.*  $\Pr[\text{коллизии}] \leq E[\text{коллизий}] = k \cdot \Pr[\text{коллизии при 1 сравнении}] = k/m$ . ■

**Теорема 6.3.10.** Пусть дано множество **произвольных** различных строк длины  $\leq n$ , выбрано случайное  $p$  и сделано  $k$  сравнений  $\langle p, m \rangle$  хешей каких-то из этих строк  $\Rightarrow$  вероятность существования коллизии  $\leq \frac{nk}{m}$ .

*Доказательство.* Суммарное число корней у  $k$  многочленов степени  $\leq n$  не более  $nk$ . ■

*Замечание 6.3.11.* На самом деле оценка  $\frac{nk}{m}$  из 6.3.10 не достигается. В практических расчётах можно смело пользоваться оценкой  $\frac{k}{m}$  из 6.3.9.

### 6.3.4. Число различных подстрок (на практике)

Рассмотрим два решения, оценим их вероятности ошибок.

Решение #1: сложить хеши всех  $n(n-1)/2$  подстрок в хеш-таблицу.

Решение #2: отдельно решаем для каждой длины, внутри сложим хеши всех  $\leq n$  подстрок в хеш-таблицу, просуммируем размеры хеш-таблиц.

В первом случае, у нас неявно происходит  $\approx n^4/8$  сравнений подстрок  $\Rightarrow$  вероятность наличия коллизии  $\approx \frac{n^4/8}{m} \Rightarrow$  при  $n = 1000$  нам точно не хватит 32-битного модуля, при  $n = 10\,000$  для  $m \approx 10^{18}$  вероятность коллизии  $\approx \frac{1}{800}$ .

Во втором случае  $\approx \sum_{i=1..n} i^2/2 \approx n^3/6$  сравнений подстрок  $\Rightarrow$  вероятность наличия коллизии  $\approx \frac{n^3/6}{m} \Rightarrow$  при  $n = 10\,000$  для  $m \approx 10^{18}$  вероятность коллизии  $\approx \frac{1}{6 \cdot 10^6}$ .

### 6.3.5. Алгоритм Бойера-Мура

Даны текст  $t$  и шаблон  $s$ . Требуется найти хотя бы одно вхождение  $s$  в  $t$  или сказать, что их нет. БМ – алгоритм, решающий эту задачу за время  $\mathcal{O}(\frac{|t|}{|s|})$  в среднем и  $\mathcal{O}(|t| \cdot |s|)$  в худшем.

#### • Наивная версия алгоритма

```

1 for (p = 0; p <= |t| - |s|; p++)
2   for (k = |s| - 1; k >= 0; k--)
3     if (t[p + k] != s[k])
4       break;
5   if (k < 0)
6     return 1;
```

То есть, мы прикладываем шаблон  $s$  ко всем позициям  $t$ , сравниваем символы с конца.

#### • Оптимизации

Каждый раз мы сдвигаем шаблон на 1 вправо.

Сдвинем лучше сразу так, чтобы несовпавший символ текста  $t[p+k]$  совпал с каким-либо символом шаблона. Эта оптимизация называется «правилом плохого символа».

```

1 for (i = 0; i < |s|; i++)
2   pos[s[i]].push_back(i); // для каждого символа список позиций
3 for (p = 0; p <= |t| - |s|; p += dp)
4   for (k = |s| - 1; k >= 0; k--)
5     if (t[p + k] != s[k])
6       break;
7   if (k < 0)
8     return 1
9   auto &v = pos[t[p + k]]; // нужно в v найти последний элемент меньший k
10  for (i = v.size() - 1; i >= 0 && v[i] >= k; i--)
11    ;
12  dp = (k - (i < 0 ? -1 : v[i])); // сдвигаем так, чтобы вместо s[k] оказался s[v[i]]
```

Вторая оптимизация «правило хорошего суффикса» – использовать информацию, что суффикс  $u = s[k+1:]$  уже совпал с текстом  $\Rightarrow$  нужно сдвинуть  $s$  до следующего вхождения  $u$ . Тут нам поможет  $Z$ -функция от  $\bar{s}$ :  $|u| = |s| - k - 1$ , мы ищем  $\text{shift}[|u|] = \min j: z(\bar{s})[j] \geq |u|$ , и сдвигаем на  $j$ . На самом деле мы даже знаем, что следующий символ не совпадает, поэтому ищем  $\min j: z(\bar{s})[j] = |u|$ .

```

1 z <-- z_function(reverse(s))
2 for (j = |s| - 1; j >= 0; j--)
3   shift[z[j]] = r;
```

В итоге алгоритм Бойера-Мура сдвигает шаблон на  $\max(x, y)$ , где  $x = dp, y = \text{shift}[|s| - k - 1]$ .

Время и память, требуемые на предподсчёт –  $\Theta(|s|)$ . Предподсчёт зависит только от  $s$ .

### Пример выполнения Бойера-Мура:

$t = \text{"abcabcabbababa"} , \quad s = \text{"baba"}$

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |              |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--------------|
| a | b | c | a | b | c | a | b | b | a | a | a | b | a | b | a |              |
| b | a | b | a | a | b | a | a | b | a | b | a | b | a | b | a | x = 3, y = 2 |
|   |   |   | b |   | b |   |   |   |   |   |   |   |   |   |   | x = 3, y = 2 |
|   |   |   |   |   |   | b | a | b | a | b | a |   |   |   |   | x = 1, y = 2 |
|   |   |   |   |   |   |   |   | b | a | b | a |   |   |   |   | x = 1, y = 2 |
|   |   |   |   |   |   |   |   |   | b | a | b | a | b | a | b | x = 1, y = 2 |
|   |   |   |   |   |   |   |   |   |   | b | a | b | a | b | a | ok           |

Алгоритм можно продолжить модифицировать, например искать  $\min j$ : после сдвига на  $j$ , совпадут с шаблоном все уже открытые символы в  $t$ . Тогда в первом же шаге примера сдвиг будет на 4 вместо трёх.

Другой пример:  $s = \underbrace{aa \dots a}_n$  в строке  $t = \underbrace{bb \dots b}_m$ . Тогда каждый раз мы сравниваем лишь один символ, а сдвигаемся на  $n$  позиций  $\Rightarrow$  время  $\Theta(|m|/|n|)$ .

## Лекция #7: Суффиксный массив

24 сентября 2020

**Def 7.0.1.** Суффиксный массив  $s$  – отсортированный массив суффиксов  $s$ .

Суффиксы сортируем в лексикографическом порядке. Каждый суффикс однозначно задается позицией начала в  $s \Rightarrow$  на выходе мы хотим получить перестановку чисел от 0 до  $n-1$ .

• **Тривиальное решение:** `std::sort` отработает за  $\mathcal{O}(n \log n)$  операций ' $<$ '  $\Rightarrow$  за  $\mathcal{O}(n^2 \log n)$ .

### 7.1. Построение за $\mathcal{O}(n \log^2 n)$ хешами

Мы уже умеем сравнивать хешами строки на равенство, научимся сравнивать их на " $>/<$ ".

Бинпоиском за  $\mathcal{O}(\log(\min(|s|, |t|)))$  проверок на равенство найдём  $x = lcp(s, t)$ .

Теперь  $less(s, t) = (s[x] < t[x])$ . Кстати, в C/C++ после строки всегда идёт символ с кодом 0.

Получили оператор меньше, работающий за  $\mathcal{O}(\log n)$  и требующий  $\mathcal{O}(n)$  предподсчёта.

Итого: суффмассив за  $\mathcal{O}(n + (n \log n) \cdot \log n) = \mathcal{O}(n \log^2 n)$ .

При написании сортировки нам нужно теперь минимизировать в первую очередь именно число сравнений  $\Rightarrow$  с точки зрения C++: STL быстрее будет работать `stable_sort` (MergeSort внутри).

*Замечание 7.1.1.* Заодно научились за  $\mathcal{O}(\log n)$  сравнивать на больше/меньше любые подстроки.

### 7.2. Применение суффиксного массива: поиск строки в тексте

**Задача:** дана строка  $t$ , приходят строки-запросы  $s_i$ : “является ли  $s_i$  подстрокой  $t$ ”.

Предподсчёт: построим суффиксный массив  $p$  строки  $t$ .

В суффиксном массиве сначала лежат все суффиксы  $< s_i$ , затем  $\geq s_i \Rightarrow$  бинпоиском можно найти  $\min k: t[p_k:] \geq s_i$ . Осталось заметить, что  $(s_i - \text{префикс } t[p_k:]) \Leftrightarrow (s_i - \text{подстрока } t)$ .

Внутри бинпоиска можно сравнивать строки за линию, получим время  $\mathcal{O}(|s_i| \log |t|)$  на запрос.

Можно за  $\mathcal{O}(\log |t|)$  с помощью хешей, для этого нужно один раз предподсчитать хеши для  $t$ , а при ответе на запрос насчитать хеши  $s_i$ . Получили время  $\mathcal{O}(|s_i| + \log |t| \cdot \log |s_i|)$  на запрос.

В разд. 7.5 мы улучшим время обработки запроса до  $\mathcal{O}(|s_i| + \log |t|)$ .

### 7.3. Построение за $\mathcal{O}(n^2)$ и $\mathcal{O}(n \log n)$ цифровой сортировкой

Заменим строку  $s$  на строку  $s\#$ , где  $\#$  – символ, лексикографически меньший всех в  $s$ .

Будем сортировать циклические сдвиги  $s\#$ , порядок совпадёт с порядком суффиксом.

Длину  $s\#$  обозначим  $n$ .

**Решение за  $\mathcal{O}(n^2)$ :** цифровая сортировка.

Сперва подсчётом по последнему символу, затем по предпоследнему и т.д.

Всего  $n$  фаз сортировок подсчётом. В предположении  $|\Sigma| \leq n$  получаем время  $\mathcal{O}(n^2)$ .

Суффмассив, как и раньше задаётся перестановкой начал... теперь циклических сдвигов.

**Решение за  $\mathcal{O}(n \log n)$ :** цифровая сортировка с удвоением длины.

Пусть у нас уже отсортированы все подстроки длины  $k$  циклической строки  $s\#$ .

Научимся за  $\mathcal{O}(n)$  переходить к подстрокам длины  $2k$ .

Давайте требовать не только отсортированности но и знания “равны ли соседние в отсортированном порядке”. Тогда линейным проходом можно для каждого  $i$  считать тип (цвет) циклического сдвига  $c[i]: (0 \leq c[i] < n) \wedge (s[i:i+k] < s[j:j+k] \Leftrightarrow c[i] \leq c[j])$ .

Любая подстрока длины  $2k$  состоит из двух половин длины  $k \Rightarrow$  переход  $k \rightarrow 2k$  – цифровая сортировка пар  $\langle c[i], c[i+k] \rangle$ .

Прекратим удвоение  $k$ , когда  $k \geq n$ . Порядки подстрок длины  $k$  и  $n$  совпадут.

*Замечание 7.3.1.* В обоих решениях в случае  $|\Sigma| > n$  нужно первым шагом отсортировать и перенумеровать символы строки. Это можно сделать за  $\mathcal{O}(n \log n)$  или за  $\mathcal{O}(n + |\Sigma|)$  подсчётом.

**Реализация решения за  $\mathcal{O}(n \log n)$ .**

$p[i]$  – перестановка, задающая порядок подстрок длины  $s[i:i+k]$  циклической строки  $s\#$ .

$c[i]$  – тип подстроки  $s[i:i+k]$ .

За базу возьмём  $k = 1$

```
1 bool sless( int i, int j ) { return s[i] < s[j]; }
2 sort(p, p + n, sless);
3 cc = 0; // текущий тип подстроки
4 for (i = 0; i < n; i++) // тот самый линейный проход, насчитываем типы строк длины 1
5   cc += (i && s[p[i]] != s[p[i-1]]), c[p[i]] = cc;
```

Переход: (у нас уже отсортированы строки длины  $k$ )  $\Rightarrow$  (уже отсортированы строки длины  $2k$  по второй половине)  $\Rightarrow$  (осталось сделать сортировку подсчётом по первой половине).

```
1 // pos - массив из n нулей
2 for (i = 0; i < n; i++)
3   pos[c[i] + 1]++; // обойдёмся без лишнего массива cnt
4 for (i = 1; i < n; i++)
5   pos[i] += pos[i - 1];
6 for (i = 0; i < n; i++) { // p[i] - позиция начала второй половины
7   int j = (p[i] - k) mod n; // j - позиция начала первой половины
8   p2[pos[c[j]]++] = j; // поставили подстроку s[j,j+2k) на правильное место в p2
9 }
10 cc = 0; // текущий тип подстроки
11 for (i = 0; i < n; i++) // линейным проходом насчитываем типы строк длины 2k
12   cc += (i && pair_of_c(p2[i]) != pair_of_c(p2[i-1]])), c2[p2[i]] = cc;
13 c2.swap(c), p2.swap(p); // не забудем перейти к новой паре (p,c)
```

Здесь  $\text{pair\_of\_c}(i)$  – пара  $\langle c[i], c[(i + k) \bmod n] \rangle$  (мы сортировали как раз эти пары!).

*Замечание 7.3.2.* При написании суффмассива в констесте рекомендуется, прочтя конспект, написать код самостоятельно, без подглядывания в конспект.

## 7.4. LCP за $\mathcal{O}(n)$ : алгоритм Касаи

Алгоритм Касаи считает LCP соседних суффиксов в суффиксном массиве. Обозначения:

- $p[i]$  – элемент суффмассива,
- $p^{-1}[i]$  – позиция суффикса  $s[i:]$  в суффмассиве,
- $\text{next}_i = p[p^{-1}[i] + 1]$ ,  $\text{lcp}_i = \text{LCP}(i, \text{next}_i)$ . Наша задача – насчитать массив  $\text{lcp}_i$ .

*Утверждение 7.4.1.* Если у  $i$ -го и  $j$ -го по порядку суффикса в суффмассиве совпадают первые  $k$  символов, то на всём отрезке  $[i, j]$  суффмассива совпадают первые  $k$  символов.

**Lm 7.4.2.** Основная идея алгоритма Касаи:  $lcp_i > 0 \Rightarrow lcp_{i+1} \geq lcp_i - 1$ .

*Доказательство.* Отрежем у  $s[i:]$  и  $s[next_i:]$  по первому символу.

Получили суффиксы  $s[i+1:]$  и какой-нибудь  $r$ .

$(s[i:] \neq s[next_i:]) \wedge (\text{первый символ у них совпадал}) \Rightarrow$

$(r \text{ в суффмассиве идёт после } s[i+1:]) \wedge (\text{у них совпадают первых } lcp_i - 1 \text{ символов}) \xRightarrow{7.4.1}$

у  $s[i+1:]$  и  $s[next_{i+1}]$  совпадает хотя бы  $lcp_i - 1$  символ  $\Rightarrow lcp_{i+1} \geq lcp_i - 1$ . ■

Собственно алгоритм заключается в переборе  $i \nearrow$  и подсчёте  $lcp_i$  начиная с  $\max(0, lcp_{i+1} - 1)$ .

**Задача:** уметь выдавать за  $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$  LCP любых двух суффиксов строки  $s$ .

**Решение:** используем Касаи для соседних, а для подсчёта LCP любых других считаем RMQ. RMQ мы решили в прошлом семестре. Например, Фарах-Колтоном-Бендером за  $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ .

## 7.5. Быстрый поиск строки в тексте

Представим себе простой бинпоиск за  $\mathcal{O}(|s| \log(|text|))$ . Будем стараться максимально переиспользовать информацию, полученную из уже сделанных сравнений.

Для краткости  $\forall k$  обозначим  $k$ -й суффикс ( $text[p_k:]$ ) как просто  $k$ .

**Инвариант:** бинпоиск в состоянии  $[l, r]$  уже знает  $lcp(s, l)$  и  $lcp(s, r)$ .

Сейчас мы хотим найти  $lcp(s, m)$  и перейти к  $[l, m]$  или  $[m, r]$ .

Заметим,  $lcp(s, m) \geq \max\{\min\{lcp(s, l), lcp(l, m)\}, \min\{lcp(s, r), lcp(r, m)\}\} = x$ .

Мы умеем искать  $lcp(l, m)$  и  $lcp(r, m)$  за  $\mathcal{O}(1) \Rightarrow \text{for } (lcp(s, m) = x; \text{ можем}; lcp(s, m)++)$ .

Кстати,  $lcp(l, m)$  и  $lcp(r, m)$  не обязательно считать Фарах-Колтоном-Бендером, так как, аргументы  $lcp$  – не произвольный отрезок, а вершина дерева отрезков (состояние бинпоиска). Предподсчитаем  $lcp$  для всех  $\leq 2|text|$  вершин и по ходу бинпоиска будем спускаться по Д.О.

**Теорема 7.5.1.** Суммарное число увеличений на один  $lcp(s, ?)$  не более  $|x|$

*Доказательство.* Сейчас бинпоиск в состоянии  $l_i, m_i, r_i$ . Следующее состояние:  $l_{i+1}, r_{i+1}$ .

Предположим,  $lcp(s, l_i) \geq lcp(s, r_i)$ . Будем следить за величиной  $z_i = \max\{lcp(s, l_i), lcp(s, r_i)\}$ .

Пусть  $lcp(s, m_i) < z_i \Rightarrow lcp(s, m) = x \wedge l_{i+1} = l_i \Rightarrow z_{i+1} = z_i$ . Иначе  $x = z_i \wedge z_{i+1} = lcp(s, m_i)$ . ■

# Лекция #8: Ахо-Корасик и Укконен

1 октября 2020

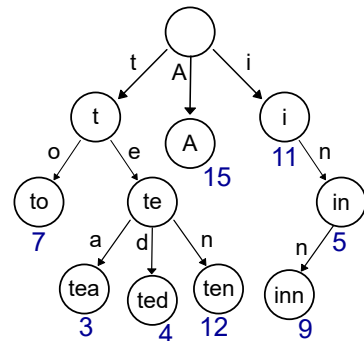
## 8.1. Бор

Бор – корневое дерево. Рёбра направлены от корня и подписаны буквами. Некоторые вершины бора подписаны, как конечные.

Базовое применение бора – хранение словаря `map<string, T>`.

Пример из [wiki](#) бора, содержащего словарь  $\{A:15, to:7, tea:3, ted:4, ten:12, i:11, in:5, inn:9\}$ .

Для строки  $s$  операции `add(s)`, `delete(s)`, `getValue(s)` работают, как спуск вниз от корня.



Самый простой способ хранить бор: `vector<Vertex> t;`, где `struct Vertex { int id[|Σ|]; }`. Сейчас рёбра из вершины  $t$  хранятся в массиве `t.id[]`. Есть другие структуры данных:

| Способ хранения | Время спуска по строке                 | Память на ребро         |
|-----------------|----------------------------------------|-------------------------|
| array           | $\mathcal{O}( s )$                     | $\mathcal{O}( \Sigma )$ |
| list            | $\mathcal{O}( s  \cdot  \Sigma )$      | $\mathcal{O}(1)$        |
| map (TreeMap)   | $\mathcal{O}( s  \cdot \log  \Sigma )$ | $\mathcal{O}(1)$        |
| HashMap         | $\mathcal{O}( s )$ с большой const     | $\mathcal{O}(1)$        |
| SplayMap        | $\mathcal{O}( s  + \log S)$            | $\mathcal{O}(1)$        |

Иногда для краткости мы будем хранить бор массивом `int next[N][|Σ|];`, где `next[v][c] == 0`  $\Leftrightarrow$  ребра нет.

### • Сортировка строк

Если мы храним рёбра в структуре, способной перебирать рёбра в лексикографическом порядке (не хеш-таблица, не список), можно легко отсортировать массив строк:

(1) добавить их все в бор, (2) обойти бор слева направо.

Для `SplayMap` и  $n$  и строк суммарной длины  $S$ , получаем время  $\mathcal{O}(S + n \log S)$ .

Для `TreeMap` получаем  $\mathcal{O}(S \log |\Sigma|)$ .

*Замечание 8.1.1.* Если бы мы научились сортировать строки над произвольным алфавитом за  $\mathcal{O}(|S|)$ , то для  $\Sigma = \mathbb{Z}$ , получилась бы сортировка целых чисел за  $\mathcal{O}(|S|)$ .

Часто размер алфавита считают  $\mathcal{O}(1)$ .

Например строчные латинские буквы – 26, или любимый для биологов  $|\{A, C, G, T\}| = 4$ .

## 8.2. Алгоритм Ахо-Корасик

Даны текст  $t$  и словарь  $s_1, s_2, \dots, s_m$ , нужно научиться искать словарные слова в тексте.

Простейший алгоритм, отлично работающий для коротких слов, – сложить словарные слова в бор и от каждой позиции текста  $i$  попытаться пройти вперёд, откладывая суффикс  $t_i$  вниз по бору, и отмечая все концы слов, которые мы проходим. Время работы –  $\mathcal{O}(|t| \cdot \max |s_i|)$ .

Ту же асимптотику можно получить, сложив все хеши всех словарных слов в хеш-таблицу, и



проверив, есть ли в хеш-таблице какие-нибудь подстроки  $t$  длины не более  $\max |s_i|$ .

Давайте теперь оптимизируем первое решение также, как префикс-функция, позволяет простейший алгоритм поиска подстроки в строке улучшить до линейного времени. Обобщение префикс-функции на бор – суффиксные ссылки:

**Def 8.2.1.**  $\forall$  вершины бора  $v$ :

$str[v]$  – строка, написанная на пути от корня бора до  $v$ .

$suf[v]$  – вершина бора, соответствующая самому длинному суффиксу  $str[v]$  в боре.

$\forall$  позиции текста  $i$  насчитаем вершину бора  $v_i$ :  $str[v_i]$  – суффикс  $t[0:i]$ ,  $|str[v_i]| \rightarrow \max$ .

**Пересчёт  $v_i$ :**

```
1 v[0] = root, p = root
2 for (i = 0; i < |t|; i++)
3     while (next[p][t[i]] == 0) // нет ребра
4         p = suf[p]
5     v[i + 1] = p = next[p][t[i]]
```

Чтобы цикл `while` всегда останавливался введём фиктивную вершину `f` и сделаем `suf[root] = f`,  $\forall c \text{ next}[f][c] = \text{root}$ .

**Поиск словарных слов.** Пометим все вершины бора, посещённые в процессе: `used[vi] = 1`. В конце алгоритма поднимем пометки вверх по суффиксным ссылкам: `used[v]  $\Rightarrow$  used[suf[v]]`. Для  $i$ -го словарного слова при добавлении мы запомнили вершину `end[i]`, тогда наличия этого слова в тексте лежит в `used[end[i]]`. Также можно насчитывать число вхождений.

**Суфсссылки.** Чтобы всё это счастье работало осталось насчитать суфсссылки.

Способ #1: полный автомат.

```
1 suf[v] = go[suf[parent[v]]][parent_char[v]];
2 go[v][c] = (next[v][c] ? next[v][c] : go[suf[v]][c]);
```

Естественно, чтобы от `parent[v]` и `suf[v]` всё было уже посчитано, поэтому нужно или перебирать вершины в порядке bfs от корня, или считать эту динамику рекурсивно-лениво.

Способ #2: пишем bfs от корня и пытаемся продолжить какой-нибудь суффикс отца.

```
1 q <-- root
2 while q --> v:
3     z = suf[parent[v]]
4     while next[z][parent_char[v]] == 0:
5         z = suf[z]
6     suf[v] = next[z][parent_char[v]]
```

Этот способ экономнее по памяти, если `next` – не массив, а, например, `map<int, int>`.

**Теорема 8.2.2.** Время построения линейно от длины суммарной строк, но не от размера бора.

*Доказательство.* Линейность от размера бора ломается на примере «бамбук длины  $n$  из букв  $a$ , из листа которого торчат рёбра по  $n$  разным символам». Линейность от суммарной длины строк следует из того, что если рассмотреть путь, соответствующий  $\forall$  словарному слову  $s_i$ , то при вычислении суфсссылок от вершин именно этого пути, указатель  $z$  в `while` всё время поднимался, а затем опускался не более чем на 1  $\Rightarrow$  сделал не более  $2|s_i|$  шагов. ■

### 8.3. Суффиксное дерево

**Def 8.3.1.** *Сжатый бор: разрешим на ребре писать не только букву, но и строку. При этом из каждой вершины по каждой букве выходит всё ещё не более одного ребра.*

**Def 8.3.2.** *Суффиксное дерево – сжатый бор построенный из суффиксов строки.*

**Lm 8.3.3.** Сжатое суффиксное дерево содержит не более  $2n$  вершин.

*Доказательство.* Индукция: база один суффикс, 2 вершины, добавляем суффиксы по одному, каждый порождает максимум +1 развилку и +1 лист. ■

### 8.4. Суффиксное дерево, решение задач (на практике)

- **Число различных подстрок.**

Это ровно суммарная длина всех рёбер. Так как любая подстрока есть префикс суффикса  $\Rightarrow$  откладывается от корня дерева вниз до «середины» ребра.

- **Поиск подстрок в тексте.**

Строим суффдерево от текста.  $\forall$  строку  $s$  можно за  $\mathcal{O}(|s|)$  искать в тексте спуском по дереву.

- **Общая подстрока  $k$  строк.**

Построим дерево от  $s_1\#_1s_2\#_2\dots s_k\#_k$ , найдём самую глубокую вершину, в поддереве которой содержатся суффиксы  $k$  различных типов. Время работы  $\mathcal{O}(\sum |s_i|)$ , оптимально по асимптотике. Константу времени работы можно улучшать за счёт уменьшения памяти – строить суффдерево не от конкатенации, а лишь от одной из строк.

- **Построение суффдерева из суффмассива+LCP**

Пусть мы уже построили дерево из первых  $i$  суффиксов в порядке суффмассива. Храним путь от корня до конца  $i$ -го. Чтобы добавить  $(i+1)$ -й, поднимаемся до высоты  $LCP(i, i+1)$  и делаем новую развилку, новый лист. Это несколько **pop**-ов и не более одного **push**-а. Итого  $\mathcal{O}(n)$ .

- **Построение суффмассива+LCP из суффдерева**

Считаем, что дерево построено от строки  $s\$ \Rightarrow$  (листья = суффиксы).

Обходим дерево слева направо. Если в вершине используется неупорядоченный **map** для хранения рёбер, сперва отсортируем их. При обходе выписываем номера листьев-суффиксов.

$LCP(i, i+1)$  – максимально высокая вершина, из пройденных по пути из  $i$  в  $i+1$ .

Время работы  $\mathcal{O}(n)$  или  $\mathcal{O}(n \log |\Sigma|)$ .

## 8.5. Алгоритм Укконена

Альтернативное описание можно прочитать на [neerc-wiki](https://neerc.wiki).

Обозначение:  $ST(s)$  – суффиксное дерево строки  $s$ .

Алгоритм Укконена – онлайн алгоритм построения суффиксного дерева. Нам поступают по одной буквы  $c_i$ , мы хотим за амортизированное  $\mathcal{O}(1)$  из  $ST(s)$  получать  $ST(sc_i)$ .

За квадрат это делать просто: храним позиции концов всех суффиксов, каждый из них продлеваем вниз на  $c_i$ , если нужно, создаём при этом новые рёбра/вершины.

Ускорение #1: суффиксы, ставшие листьями, растут весьма однообразно – рассмотрим ребро  $[l, r)$ , за которое подвешен лист, тогда всегда происходит  $r++$ . Давайте сразу присвоим  $[l, \infty)$ .

Теперь опишем жизненный цикл любого суффикса:

рождается в корне, ползёт вниз по дереву, разветвляется, становится саморастущим листом.

Нам интересно обработать только момент разветвления.

**Lm 8.5.1.**  $\neg$  Суффикс длины  $k$  не разветвился  $\Rightarrow$  все более короткие тоже не разветвились.

*Доказательство.* Суффикс длины  $k$  не разветвился  $\Rightarrow$  он встречался в  $s$  как подстрока.

Все более короткие являются его суффиксами  $\Rightarrow$  тоже встречаются в  $s \Rightarrow$  не разветвятся. ■

Ускорение #2: давайте хранить только позицию самого длинного неразветвившегося суффикса. Пока он спускается по дереву, ничего не нужно делать. Как только он разветвится, нужно научиться быстро переходить к следующему по длине (отрезать первую букву).

Ускорение #3: отрезать первую букву = перейти по суффикссылке, давайте от всех вершин поддерживать суффикссылки. Если мы были в вершине, когда не смогли пойти вниз, теперь всё просто, перейдём по её суффикссылке. Если же мы стояли посередине ребра и создали новую вершину  $v$ , от неё следует посчитать суффикссылку. Для этого возьмём суффикссылку её отца  $p[v]$  и из  $\text{suf}[p[v]]$  спустимся вниз на строку, соединяющую  $p[v]$  и  $v$ .

```

1 void build( char *s ) {
2     int N = strlen(s), VN = 2 * Ns;
3     int vn = 2, v = 1, pos = 0; // идём по ребру из p[v] в v, сейчас стоим в pos
4     int suf[VN], l[VN], r[VN], p[VN]; // «ребро p[v] → v» = s[l[v]:r[v]]
5     map<char,int> t[VN]; // собственно рёбра нашего бора
6     for (int i = 0; i < |Σ|; i++) t[0][i] = 1; // 0 = фиктивная, 1 = корень
7     l[1] = -1, r[1] = 0, suf[1] = 0;
8     for (int n = 0; n < N; n++) {
9         char c = s[n];
10        auto new_leaf = [&]( int v ) {
11            p[vn] = v, l[vn] = n, r[vn] = ∞, t[v][c] = vn++;
12        };
13        go::
14        if (r[v] <= pos) { // дошли до вершины, конца ребра
15            if (!t[v].count(c)) { // по символу c нет ребра вперёд, создаём
16                new_leaf(v), v = suf[v], pos = r[v];
17                goto go;
18            }
19            v = t[v][c], pos = l[v] + 1; // начинаем идти по новому ребру
20        } else if (c == s[pos]) {
21            pos++; // спускаемся по ребру
22        } else {
23            int x = vn++; // создаём развилку
24            l[x] = l[v], r[x] = pos, l[v] = pos;
25            p[x] = p[v], p[v] = x;
26            t[p[x]][s[l[x]]] = x, t[x][s[pos]] = v;
27            new_leaf(x);
28            v = suf[p[x]], pos = l[x]; // вычисляем позицию следующего суффикса
29            while (pos < r[x])
30                v = t[v][s[pos]], pos += r[v] - l[v];
31            suf[x] = (pos == r[x] ? v : vn);
32            pos = r[v] - (pos - r[x]);
33            goto go;
34        }
35    }
36 }

```

**Теорема 8.5.2.** Суммарное время работы  $n$  первых шагов равно  $\mathcal{O}(n)$ .

*Доказательство.* Понаблюдаем за величиной  $\varphi$  “число вершин на пути от корня до нас”.

Пока мы идём вниз,  $\varphi$  растёт, когда переходим по суффикссылке,  $\varphi$  уменьшается максимум на 1  $\Rightarrow$  суммарное число шагов вниз не больше  $n$ . ■

## 8.6. LZSS

Решим ещё одну задачу – сжатие текста алгоритмом LZSS.

В отличие от использования массива, дерево даёт чисто линейную асимптотику и простейшую реализацию – насчитаем для каждой вершины  $l[v]$  = самый левый суффикс в поддереве и при попытке найти  $j < i$ :  $LCP(j, i) = \max$  будем спускаться из корня, пока  $l[v] < i$ .

# Лекция #9: Потоки (база)

10 октября

## 9.1. Основные определения

Дан орграф  $G$ , у каждого ребра  $e$  есть пропускная способность  $c_e \in \mathbb{R}$ .

**Def 9.1.1.** Поток в орграфе из  $s$  в  $t$  – сопоставленные рёбрам числа  $f_e \in \mathbb{R}$ :

$$(\forall \text{ ребра } e \quad 0 \leq f_e \leq c_e) \wedge (\forall \text{ вершины } v \neq s, t \quad \sum_{e \in \text{in}(v)} f_e = \sum_{e \in \text{out}(v)} f_e)$$

Вершина  $s$  называется *исток*ом, вершина  $t$  *сток*ом.

Говорят, что по ребру  $e$  течёт  $f_e$  единиц потока.

Определение говорит “поток течёт из истока в сток и ни в какой вершине не задерживается”.

**Def 9.1.2.** Величина потока  $|f| = \sum_{e \in \text{out}(s)} f_e - \sum_{e \in \text{in}(s)} f_e$  (сколько вытекает из истока).

Утверждение 9.1.3. В сток втекает ровно столько, сколько вытекает из истока.

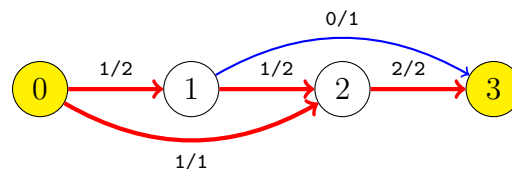
Замечание 9.1.4.  $|f|$  может быть отрицательной: пусть по ребру  $t \rightarrow s$  единицу потока.

**Def 9.1.5.** Циркуляцией называется поток величины 0.

### • Примеры потока

Рассмотрим пока граф с единичными пропускными способностями.

1.  $\forall$  цикл – циркуляция.
2.  $\forall$  путь из  $s$  в  $t$  – поток величины 1.
3.  $\forall k$  не пересекающихся по рёбрам путей из  $s$  в  $t$  – поток величины  $k$ .
4. На картинке поток величины 2, подписи  $f_e/c_e$ .



**Def 9.1.6.** Остаточная сеть потока  $f - G_f$ , граф с пропускными способностями  $c_e - f_e$ .

**Def 9.1.7.** Дополняющий путь – путь из  $s$  в  $t$  в остаточной сети  $G_f$ .

**Lm 9.1.8.** Если по всем рёбрам дополняющего пути  $p$  увеличить величину потока на  $x = \min_{e \in p} (c_e - f_e)$ , получится корректный поток величины  $|f| + x$ .

## 9.2. Обратные рёбра

**Def 9.2.1.** Для каждого ребра сети  $G$  с пропускной способностью  $c_e$  создадим обратное ребро  $e'$  пропускной способностью 0. При этом по определению  $f_{e'} = -f_e$ .

Добавим в граф обратные рёбра, упростим определения потока и величины потока:

Теперь  $\forall$  потока  $f$  должно выполняться  $\forall v \neq s, t \quad \sum_{e \in \text{out}(v)} f_e = 0$ , величина потока  $|f| = \sum_{e \in \text{out}(s)} f_e$ .

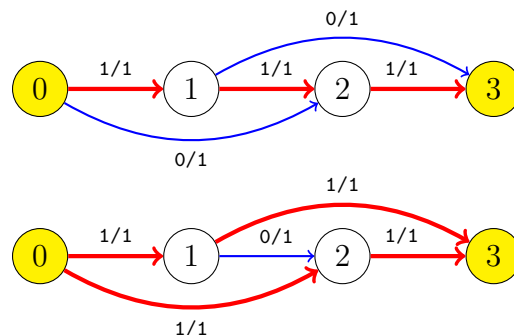
Здесь  $\text{out}(v)$  – множество прямых и обратных рёбер, выходящих из  $v$ .

**Def 9.2.2.** Ребро называется насыщенным, если  $f_e = c_e$ , иначе оно ненасыщено.

Утверждение 9.2.3. Если по прямому ребру течёт поток, обратное ненасыщено.

После добавления обратных рёбер в  $G$ , они появились и в  $G_f$ . Поэтому для такого потока из 0 в 3 величины 1 в  $G_f$  есть дополняющий путь  $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ .

Увеличим поток по пути  $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ , получим новый поток. Заметьте, добавляя +1 потока к ребру  $2 \rightarrow 1$ , мы уменьшаем поток по ребру  $1 \rightarrow 2$ .



*Замечание 9.2.4.* Сейчас мы научимся разбивать поток величины 2 на 2 непересекающихся по рёбрам пути. Если бы мы действовали жадно (*найдем какой-нибудь первый путь, удалим его рёбра из графа, на оставшихся рёбрах найдем второй путь*), нас постигла бы неудача. Поток же благодаря обратным рёбрам получается даже при неверном первом пути найти дополняющий путь и получить поток размера два.

### 9.3. Альтернативное определение

Нередко удобнее работать с альтернативными определениями:

*Транспортная сеть.* У нас есть:

- граф  $G(V, E)$  (ориентированный или нет);
- исток  $s \in V$  и сток  $t \in V$  ( $s \neq t$ );
- пропускные способности  $c: V^2 \rightarrow \mathbb{R}_+$ ;
- поток  $f: V^2 \rightarrow \mathbb{R}$ .

И должны быть выполнены следующие свойства:

1.  $\forall v, u \in V \quad f(v, u) = -f(u, v)$ ;
2.  $\forall v, u \in V \quad f(v, u) \leq c(v, u)$ ;
3.  $\forall v \in V \setminus \{s, t\} \quad \sum_{u \in V} f(v, u) = 0$ ;
4. если  $e = (vu) \in E$ , то  $c(v, u) = c_e \geq 0$ ; иначе  $c(v, u) = 0$ .

*Величина потока* (обозначаем  $|f|$ ) — это число  $\sum_{u \in V} f(s, u)$ .

### 9.4. Декомпозиция потока (на практике)

**Def 9.4.1.** *Элементарный поток* — путь из  $s$  в  $t$ , по которому течёт  $x$  единиц потока.

**Def 9.4.2.** *Декомпозиция потока  $f$*  — представление  $f$  в виде суммы элементарных потоков (путей) и циркуляций.

**Lm 9.4.3.**  $|f| > 0 \Rightarrow \exists$  путь из  $s$  в  $t$  по рёбрам  $e: f_e > 0$ .

• **Алгоритм декомпозиции за  $\mathcal{O}(E^2)$**

Пока  $|f| > 0$  найдём путь  $p$  из  $s$  в  $t$  по рёбрам  $e: f_e > 0$ , по всем рёбрам пути  $p$  уменьшим поток на  $x = \min_{e \in p} f_e$ .

**Lm 9.4.4.** Время работы  $\mathcal{O}(E^2)$

*Доказательство.* По рёбрам  $e: f_e > 0$  поток только убывает. После отщепления одного пути, как минимум у одного из рёбер  $f_e$  обнулится  $\Rightarrow$  не более  $E$  поисков пути. ■

## 9.5. Теорема и алгоритм Форда-Фалкерсона

**Def 9.5.1.** Для любых множеств  $S, T \subseteq V$  определим

$$F(S, T) = \sum_{a \in S, b \in T} f_{a \rightarrow b}, \quad C(S, T) = \sum_{a \in S, b \in T} c_{a \rightarrow b}$$

Сумма включает обратные рёбра  $\Rightarrow$  на графе из одного ребра  $e: t \rightarrow s, f_e = 1 \quad F(\{s\}, \{t\}) = -1$ .

**Lm 9.5.2.**  $\forall v \in V \begin{cases} F(\{v\}, V) = 0 & v \neq s, t \\ F(\{v\}, V) = |f| & v = s \end{cases}$

**Lm 9.5.3.**  $\forall S \quad F(S, S) = 0$

*Доказательство.* В вместе с каждым ребром в сумму войдёт и обратное ему. ■

**Lm 9.5.4.**  $\forall S, T \quad F(S, T) \leq C(S, T)$

*Доказательство.* Сложили неравенства  $f_e \leq c_e$  по всем рёбрам  $e: S \rightarrow T$ . ■

**Def 9.5.5.** Разрез – дизъюнктное разбиение вершин  $(S, T): V = S \sqcup T, s \in S, t \in T$ .

**Def 9.5.6.** Величина разреза  $(S, T) = C(S, T)$ .

**Lm 9.5.7.**  $\forall$  разреза  $(S, T) \quad |f| = F(S, T)$

*Доказательство.* Интуитивно: поток вытекает из  $s$ , нигде не задерживается  $\Rightarrow$  он весь протечёт через разрез. Строго:  $F(S, T) = F(S, T) + F(S, S) = F(S, V) = F(\{s\}, V) + 0 + \dots + 0 = |f|$ . ■

**Lm 9.5.8.**  $\forall$  разреза  $(S, T)$  и потока  $f \quad |f| \leq C(S, T)$

*Доказательство.*  $|f| = F(S, T) \leq C(S, T)$  (пользуемся леммами 9.5.4 и 9.5.7). ■

**Теорема 9.5.9.** Форда-Фалкерсона

(1)  $|f| = \max \Leftrightarrow \nexists$  дополняющий путь

(2)  $\max |f| = \min C(S, T)$  (максимальный поток равен минимальному разрезу)

*Доказательство.*  $\exists$  дополняющий путь  $\Rightarrow$  можно увеличить по нему  $f \Rightarrow |f| \neq \max$ .

Пусть нет дополняющего пути  $\Rightarrow$  dfs из  $s$  по ненасыщенным рёбрам не посетит  $t$ . Множество посещённых вершин обозначим  $S$ , обозначим  $T = V \setminus S$ . Из  $S$  в  $T$  ведут только  $e: f_e = c_e$ .

Значит,  $|f| = F(S, T) = C(S, T)$ . Из леммы 9.5.8 следует, что  $|f| = \max, C(S, T) = \min$ . ■

### • Поиск минимального разреза

Из доказательства теоремы 9.5.9 мы заодно получили алгоритм за  $\mathcal{O}(E)$  поиска min разреза по max потоку.

### • Алгоритм Форда-Фалкерсона

Из теоремы следует простейший алгоритм поиска максимального потока: пока есть дополняющий путь  $p$ , найдём его, толкнём по нему  $x = \min_{e \in p} (c_e - f_e)$  единиц потока.

*Утверждение 9.5.10.* Если все  $c_e \in \mathbb{Z}$ , алгоритм конечен.

Время работы алгоритма мы умеем оценивать сверху только как  $\mathcal{O}(|f| \cdot E)$ .

При  $c_e \leq \text{polynom}(|V|, |E|)$  получаем  $|f| \leq \text{polynom}(|V|, |E|) \Rightarrow \Phi. \Phi.$  работает за полином.

При экспоненциально больших  $c_e$  на практике мы построим тест: время работы  $\Omega(2^{V/2})$ .

## 9.6. Реализация, хранение графа

Первый способ хранения графа более естественный:

```
1 struct Edge {
2     int a, b, f, c, rev; // a ---> b
3 };
4 vector<Edge> c[n]; // c[c[v][i].b][c[v][i].rev] - обратное ребро
5 for (Edge e : c[v]) // перебор рёбер, смежных с v
6     ;
```

Второй часто работает быстрее, и позволяет проще обращаться к обратному ребру.

Поэтому про него поговорим подробнее.

```
1 struct Edge {
2     int a, b, f, c; // собственно ребро
3     int next; // интрузивный список, список на массиве
4 };
5 vector<Edge> edges;
6 vector<int> head(n, -1); // для каждой вершин храним начало списка
7 for (int i = head[v]; i != -1; i = edges[i].next)
8     Edge e = edges[i]; // перебор рёбер, смежных с v
```

Добавить орребро можно так:

```
1 void add(a, b, c):
2     edges.push_back({a, b, 0, c}); // прямое
3     edges.push_back({b, a, 0, 0}); // обратное
```

Заметим, что взаимобратные рёбра добавляются парами  $\Rightarrow$

$\forall i$  к  $\text{edges}[i]$  обратным является  $\text{edges}[i^1]$ .

Теперь реализуем алгоритм  $\Phi. \Phi.$  Также, как и Кун, dfs, ищущий путь, сразу на обратном ходу рекурсии будет изменять поток по пути.

```
1 bool dfs(int v):
2     u[v] = 1;
3     for (int i = head[v]; i != -1; i = edges[i].next):
4         Edge &e = edges[i];
5         if (e.f < e.c && !u[e.b] && (e.b == t || dfs(e.b))):
6             e.f++, edges[i ^ 1].f--; // не забудьте пересчитать обратное ребро
7         return 1;
8     return 0;
```

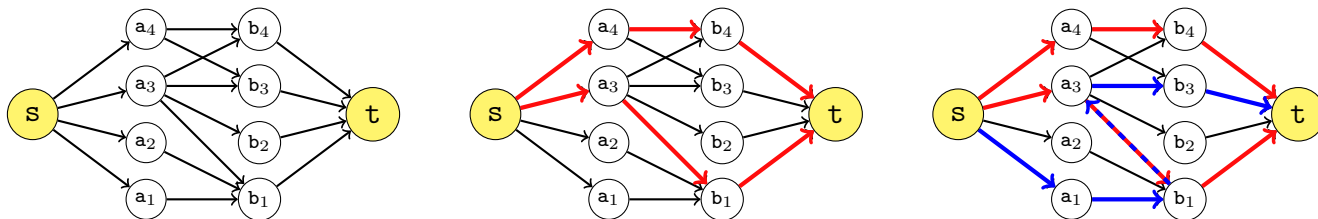
По сути мы лишь нашли путь из  $s$  в  $t$  в остаточной сети  $G_f$ .

Если мы хотим толкать не единицу потока, а  $\min_e(c_e - f_e)$ , нужно, чтобы dfs на прямом ходу рекурсии насчитывал минимум и возвращал из рекурсии полученное значение.



## 9.7. Двудольные графы (на практике)

### 9.7.1. Паросочетание



Картинки: собственно сеть  $\rightarrow$  какой-то поток в сети  $\rightarrow$  дополняющий путь.

Чтобы с помощью потоков искать паросочетание, добавляем исток и сток, ориентируем рёбра графа из первой доли во вторую. Пропускные способности “из истока” и “в сток” – единицы (ограничение на суммарный поток через вершину). Между долями можно  $+\infty$ , можно 1.

Алгоритм Форда-Фалекрсона работает за  $\mathcal{O}(|f|E) = \mathcal{O}(|M|E) \leq \mathcal{O}(VE)$ .

**Корректность.** Следует из двух биекций: (1) между потоками в построенной нами сети и паросочетаниями, (2) между дополняющими путями в нашей цепи и Ч.Д.П. в исходном графе.

#### • Что нового мы научились делать?

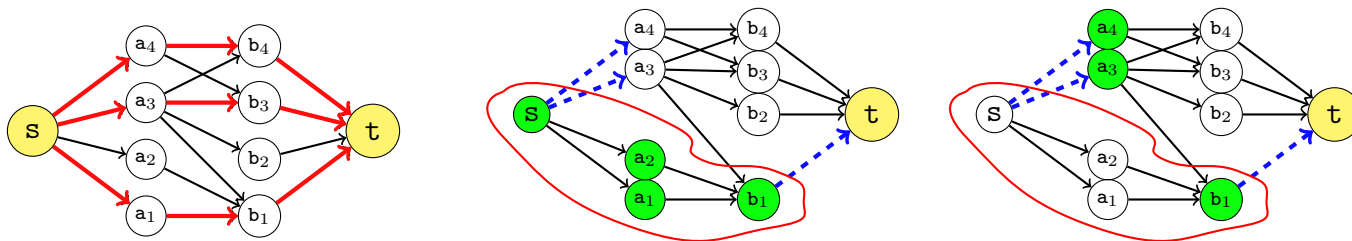
Алгоритм Диница поиска потока (см. дальше) сработает на такой сети быстрее: за  $\mathcal{O}(EV^{1/2})$ .

**Def 9.7.1.** *Мультисочетание – обобщение паросочетания, подмножество рёбер графа такое, что для каждой вершины  $v$  верно ограничение на максимальную степень  $w_v$ .*

Заменяем пропускные способности “из истока”, “в сток” на  $w_v$ . Максимальный поток даст нам максимальное мультисочетание. А алгоритм Диница найдёт его за  $\mathcal{O}(E^{3/2})$ .

### 9.7.2. Вершинное покрытие

Существует ещё третья биекция:  $\min$  разрез  $\longleftrightarrow \min$  вершинное покрытие.



Картинки:  $\max$  поток  $\rightarrow$  построенный по нему  $\min$  разрез  $\rightarrow \min$  вершинное покрытие.

Рёбрам между долями сделаем  $c_e = +\infty \Rightarrow \min$  разрезе их, конечно, не будет.

Мы уже умеем строить  $\min$  cover, используя dfs от Куна:  $C = B^+ \cup A^-$ .

Осталось заметить, что в построенном разрезе  $S \sqcup T$  по построению  $S = \{s\} \cup A^+ \cup B^+$ .

**Новое:** научились искать взвешенное минимальное вершинное покрытие в двудольном графе. Для этого поменяем пропускные способности рёбрам “из истока” и “в сток” на веса вершин.

## 9.8. Леммы, позволяющие работать с потоками

**Lm 9.8.1.** Условие  $0 \leq f_e \leq c_e$  можно заменить на  $(f_e \leq c_e) \wedge (-f_e \leq 0)$ .

То есть, и прямые, и обратные рёбра обладают пропускными способностями, ограничениями сверху на поток, по ним текущий. А про ограничения снизу можно не думать.

**Lm 9.8.2.**  $f_1$  и  $f_2$  – потоки в  $G \Rightarrow f_2 - f_1$  – поток в  $G_{f_1}$ .

*Доказательство.*  $\forall$  ребра  $e$  имеем  $f_{2e} \leq c_e \Rightarrow f_{2e} - f_{1e} \leq c_e - f_{1e}$ .

Это верно и для прямых, и для обратных. Теперь проверим сумму в вершине:

$$\forall v \sum_{e \in \text{out}(v)} (f_{2e} - f_{1e}) = \forall v \neq s, t \sum_{e \in \text{out}(v)} f_{2e} - \forall v \sum_{e \in \text{out}(v)} f_{1e} = 0 - 0 = 0. \quad \blacksquare$$

*Следствие 9.8.3.*  $\forall f_1, f_2: |f_1| = |f_2| \Rightarrow f_2$  можно получить из  $f_1$  добавлением циркуляции из  $G_{f_1}$ .

**Lm 9.8.4.**  $|f_2 - f_1| = |f_2| - |f_1|$

*Доказательство.* Также, как в 9.8.2, распишем сумму для вершины  $s$ .  $\blacksquare$

**Lm 9.8.5.** Если  $f_2$  – поток в  $G_{f_1}$ ,  $f_1 + f_2$  – поток в  $G$

**Lm 9.8.6.**  $|f_1 + f_2| = |f_1| + |f_2|$

## 9.9. Алгоритмы поиска потока

### 9.9.1. Эдмондс-Карп за $\mathcal{O}(VE^2)$

Алгоритм прост: путь ищем bfs-ом, проталкиваем по пути  $\min_e(c_e - f_e)$ . Конец.

**Lm 9.9.1.** После увеличения потока по пути, найденному bfs-ом, для любой вершины  $v$  расстояние от истока не уменьшится:  $\forall v \ d[s, v] = d[v] \nearrow$ .

*Доказательство.* От противного.

Расстояния до увеличения потока по пути  $p$  обозначим  $d_0$ , после –  $d_1$ .

Возьмём  $v$ :  $d_1[v] < d_0[v]$ , а из таких  $d_1[v] = \min$ .

Рассмотрим кратчайший путь  $q$  из  $s$  в  $v$ :  $s \rightsquigarrow \dots \rightsquigarrow x \rightarrow v \Rightarrow d_1[x] < d_1[v] \Rightarrow d_1[x] \geq d_0[x]$ .

Но  $d_1[v] \geq d_0[x] + 1 \Rightarrow$  ребра  $(x \rightarrow v)$  не было до увеличения потока по  $p \Rightarrow$  ребро  $(v \rightarrow x) \in p \xRightarrow{(*)} d_0[x] = d_0[v] + 1 \Rightarrow d_1[v] = d_1[x] + 1 \geq d_0[x] + 1 = d_0[v] + 2$ . Противоречие.  $\blacksquare$

(\*) – здесь и только здесь мы пользовались тем, что  $p$  – кратчайший.

#### • Время работы Эдмондса-Карпа

Толкаем по пути мы всё ещё  $\min_e(c_e - f_e) \Rightarrow$  после каждого bfs-а хотя бы одно ребро насытится. Чтобы ещё раз пройти по насыщенному ребру  $e$ , нужно сперва уменьшить по нему потока  $\Rightarrow$  пройти по обратному к  $e$ . Рассмотрим  $e: a \rightarrow b$ , кратчайший путь прошёл через  $e \Rightarrow d[b] = d[a] + 1$ . Когда кратчайший путь пройдёт через обратное к  $e$  имеем

$$d'[a] = d'[b] + 1 \stackrel{9.9.1}{\geq} d[b] + 1 = d[a] + 2$$

Расстояние до  $a$  между двумя насыщениями  $e$  увеличится хотя бы на 2  $\Rightarrow$  каждое ребро  $e$  насытится не более  $\frac{V}{2}$  раз  $\Rightarrow$  суммарное число насыщений  $\leq \frac{VE}{2} \Rightarrow$  Э.К. работает за  $\mathcal{O}(VE^2)$

*Следствие 9.9.2.* В графах с  $\mathbb{R}$  пропускными способностями  $\exists$  max поток.

*Доказательство.* В оценке времени работы Э.К. мы не пользовались целочисленностью. По завершении Э.К. нет дополняющих путей  $\Rightarrow$  по 9.5.9 поток максимален.  $\blacksquare$

### 9.9.2. Масштабирование за $\mathcal{O}(E^2 \log U)$

Будем пытаться сперва найти толстые пути:

перебирать  $k \downarrow$ , искать пути в остаточной сети, по которым можно толкнуть хотя бы  $2^k$ .

Для этого dfs-у разрешим ходить только по рёбрам:  $f_e + 2^k \leq c_e$ .

```

1 for k = logU .. 0:
2   u <-- 0
3   while dfs(s, 2^k):
4     u <-- 0
5     flow += 2^k

```

Ф.Ф. искал любой путь в  $G_f$ , мы ищем в  $G_f$  пути толщиной  $2^k$ . Время работы  $\mathcal{O}(E^2 \log U)$ .

Алгоритм можно оптимизировать, толкая по пути не  $2^k$ , а  $\min_e(c_e - f_e) \geq 2^k$ .

Асимптотика в худшем случае не улучшится. На практике алгоритм ведёт себя как  $\approx E^2$ .

#### • Доказательство времени работы

Поток после фазы, на которой мы искали  $2^k$ -пути, обозначим  $F_k$ . В остаточной сети  $G_{F_k}$  нет пути толщины  $2^k \Rightarrow$  есть разрез, для всех рёбер которого верно  $c_e - f_e < 2^k$ .

Далее два способа завершить доказательство:

1. Рассмотрим декомпозицию  $F_{k-1} - F_k$  на пути. Все пути имеют толщину  $2^{k-1}$ , все проходят через разрез  $\Rightarrow$  все по разным рёбрам разреза  $\Rightarrow$  их не больше, чем рёбер в разрезе  $\leq E$ .

Доказали, что путей при переходе от  $F_k$  к  $F_{k-1}$  не более  $E$ .

2. Величина этого разреза в  $G_{F_k}$  меньше  $2^k |E|$ , поэтому максимальный поток в  $G_{F_k}$  меньше  $2^k |E|$ , поэтому на очередной фазе будет меньше  $\frac{2^k E}{2^{k-1}} = 2E$  итераций.

# Лекция #10: Потоки (быстрые)

5 ноября

## 10.1. Алгоритм Диница

У нас уже есть алгоритм Эдмондса-Карпа, ищущий  $\mathcal{O}(VE)$  путей за  $\mathcal{O}(E)$  каждый.

Построим сеть кратчайших путей, расстояние  $s \rightsquigarrow t$  обозначим  $d$ .

Слоем  $A_i$  будем называть вершины на расстоянии  $i$  от  $s$ .

Э.К. сперва найдёт сколько-то путей длины  $d$ , затем расстояние  $s \rightsquigarrow t$  увеличится.

**Lm 10.1.1.** Пока  $\exists$  путь длины  $d$ , он имеет вид  $s = v_0 \in A_0, v_1 \in A_1, v_2 \in A_2, \dots, t = v_d \in A_d$

*Доказательство.* В первый момент это очевидно. Затем в  $G_f$  будут появляться новые рёбра – обратные к  $v_i \rightarrow v_{i+1}$ . Все такие рёбра идут назад по слоям  $\Rightarrow$  новых рёбер идущих вперёд по слоям не образуется  $\Rightarrow$  каждый раз при поиске пути единственный способ за  $d$  шагов из  $s$  попасть в  $t$  –  $d$  раз по одному из старых рёбер идти ровно в следующий слой. ■

Научимся искать все пути длины  $d$  за  $\mathcal{O}(E + dk_d)$ , где  $k_d$  – количество путей.

Выделим множество  $E'$  – рёбра  $A_i \rightarrow A_{i+1}$  в  $G_f$ . Будем запускать dfs по  $E'$ .

Модифицируем dfs: если пройдя по рёбру  $e$ , dfs не нашёл путь до  $t$ , он удалит  $e$  из  $E'$ .

Каждый из  $k_d$  dfs-ов сделал  $d$  успешных шагов и  $x_i$  неуспешных, но  $\sum x_i \leq E$ , так как после каждого неуспешного шага мы удаляем ребро из  $E'$ .

```

1 void dfs(int v) {
2     while (head'[v] != -1) {
3         Edge &e = edges[head'[v]];
4         if (e.f < e.c && (e.b == t || dfs(e.b))) {
5             // нашли путь
6         }
7         head'[v] = e.next;
8     }
9 }

```

Заметьте, что массив пометок вершин, обычный для любого dfs, тут можно не использовать.

### • Алгоритм Диница

Состоит из фаз вида:

- (1) запустить bfs, который построил слоистую сеть и нашёл  $d$ .
- (2) пока в слоистой сети есть путь длины  $d$ ,  
найдем его dfs-ом и толкнем по нему  $\min_e(c_e - f_e)$  единиц потока.

**Теорема 10.1.2.** Время работы алгоритма Диница  $\mathcal{O}(V^2E)$ .

*Доказательство.* Фаз всего не более  $V$  штук, так как после каждой  $d \uparrow$

Фаза с расстоянием  $d$  работает за  $\mathcal{O}(E + d \cdot k_d)$ .

$$\sum (E + d \cdot k_d) \leq VE + V \sum k_d \leq VE + V(VE)$$

Последнее известно из алгоритма Эдмондса-Карпа. ■

### • Алгоритм Диница с масштабированием потока

Масштабирование потока – не только конкретный алгоритм, но и общая идея:

```
1 for (int k = log U; k >= 0; k--)
2   пускаем поток на графе с пропускными способностями  $(c_e - f_e)/2^k$ 
```

Давайте искать поток именно алгоритмом Диница.

**Теорема 10.1.3.** Время работы алгоритма Диница с масштабированием  $\mathcal{O}(VE \log U)$ .

*Доказательство.* Каждая из фаз масштабирования – алгоритм Диница, который найдёт не более  $E$  путей и работает за время (делаем то же, что и в теореме 10.1.2):

$$\sum (E + d \cdot k_d) \leq VE + V \sum k_d \leq VE + VE = \mathcal{O}(VE)$$

Значит, суммарно все  $\log U$  фаз отработают за  $\mathcal{O}(VE \log U)$ . ■

## 10.2. Алгоритм Хопкрофта-Карпа

**Lm 10.2.1.** В единичной сети ( $c_e \equiv 1$ ) фаза Диница работает за  $\mathcal{O}(E)$

*Доказательство.* Если dfs пройдёт по ребру  $e$ , он его в любом случае удалит – и если не найдёт по нему путь, и если найдёт по нему путь: ( $c_e = 1$ )  $\Rightarrow e$  насытится. ■

Мы уже умеем искать паросочетание за  $\mathcal{O}(VE)$  через потоки.

Давайте в том же графе запустим алгоритм Диница.

**Теорема 10.2.2.** Число фаз Диница на сети для поиска паросочетания не более  $2\sqrt{V}$ .

*Доказательство.* Сделаем первые  $\sqrt{V}$  фаз, получили поток  $f$ , посмотрим на  $G_f$ .

В остаточной сети все пути имеют длину хотя бы  $\sqrt{V}$ . Вспомним биекцию между допущами в  $G_f$  и ДЧП для паросочетания  $\Rightarrow$  все ДЧП тоже имеют длину хотя бы  $\sqrt{V}$ .

Пусть поток  $f$  задаёт паросочетание  $P$ , рассмотрим максимальное  $M$ .

$M \nabla P$  содержит  $k = |M| - |P|$  непересекающихся ДЧП, каждый длины  $\geq \sqrt{V} \Rightarrow$

$k \leq \sqrt{V} \Rightarrow$  Диницу осталось найти  $\leq \sqrt{V}$  путей.

Осталось заметить, что за каждую фазу Диница находит хотя бы один путь. ■

### • Алгоритм Хопкрофта-Карпа

```
1 void dfs(int v):
2   for (x ∈ N(v)): // x - сосед во второй доле
3     if (used2[x]++ == 0) // проверили, что в x попадаем впервые
4       if (pair2[x] == -1 || (dist[pair2[x]] == dist[v]+1 && dfs(pair2[x]))):
5         pair1[v] = x, pair2[x] = v
6         return 1
7   return 0
8 while (bfs нашёл путь свободной в свободную): // цикл по фазам
9   used2 <-- 0 // пометки для вершин второй доли
10  for (v ∈ A): // вершины первой доли
11    if (pair1[v] == -1): // вершина свободна
12      dfs(v)
```

$\forall$  вершины  $v$  второй доли в  $G_f$  из  $v$  исходит не более одного ребра.

Для свободной вершины это ребро в сток  $t$ , для несвободной в её пару в первой доле.

Значит, заходить в  $v$  dfs-ам одной фазы Диница имеет смысл только один раз.

Давайте вместо “удаления рёбер” помечать вершины второй доли. Посмотрим на происходящее, как на поиск ДЧП для паросочетания. Поймём, что сток с истоком нам особо не нужны...

### 10.3. Теоремы Карзанова

Определим пропускную способность вершины:

$$c[v] = \min(c_{in}[v], c_{out}[v]), \text{ где } c_{in}[v] = \sum_{e \in in[v]} c_e, c_{out}[v] = \sum_{e \in out[v]} c_e$$

**Теорема 10.3.1.** Число фаз алгоритма Диницы не больше  $2\sqrt{C}$ , где  $C = \sum_v c[v]$ .

*Доказательство.* Заметим, что  $\forall$  потока  $f$  и вершины  $v \neq s, t$  величины  $c_{in}[v], c_{out}[v], c[v]$  равны значениям в исходном графе. Запустим первые  $\sqrt{C}$  фаз, получим поток  $f_0$ , пусть  $|f^*| = \max$ , рассмотрим декомпозицию  $f^* - f_0$ . Она состоит из  $k = |f^*| - |f_0|$  единичных путей длины хотя бы  $\sqrt{C}$  (не считая  $s$  и  $t$ ). Обозначим  $\alpha_v$  – сколько путей проходят через  $v$ . Тогда:

$$k\sqrt{C} \leq \sum_{v \neq s, t} \alpha_v \leq \sum_{v \neq s, t} c[v] = C \Rightarrow k \leq \sqrt{C}$$

Получили, что число фаз  $\leq \sqrt{C} + k \leq 2\sqrt{C}$ . ■

*Следствие 10.3.2.* Из теоремы следует время работы Хопкрофта-Карпа.

*Утверждение 10.3.3.* В единичных сетях  $C \leq E \Rightarrow$  алгоритм Диница работает за  $\mathcal{O}(E^{3/2})$ .

**Теорема 10.3.4.** Число фаз алгоритма Диницы не больше  $2U^{1/3}V^{2/3}$ , где  $U = \max_e c_e$ .

*Доказательство.* Запустим первые  $k$  фаз (оптимальное  $k$  выберем позже), на  $(k+1)$ -й получим слоистую сеть из  $\geq k+1$  слоёв. Обозначим размеры слоёв  $a_0, a_1, a_2, \dots, a_k$ .

Тогда величина  $\min$  разреза не более  $\min_{i=1..k} (a_{i-1}a_iU)$ .

Максимум такого минимума достигается при  $a_1 = a_2 = \dots a_k = \frac{V}{k}$ .

Получили разрез размера  $U(\frac{V}{k})^2 \Rightarrow$  осталось не более чем столько фаз  $\Rightarrow$

$$\text{всего фаз не более } f(k) = k + U(\frac{V}{k})^2. \quad k \uparrow, U(\frac{V}{k})^2 \downarrow.$$

Асимптотический минимум  $f$  достигается при  $k = U(\frac{V}{k})^2 \Rightarrow k^3 = UV^2$ , число фаз  $\leq 2(UV^2)^{1/3}$ . ■

### 10.4. Диниц с link-cut tree (на практике)

Улучшим время одной фазы алгоритма Диница с  $\mathcal{O}(VE)$  до  $\mathcal{O}(E \log V)$ .

Построим остовное дерево с корнем в  $t$  по входящим не насыщенным рёбрам.

Теперь  $E$  раз пускаем поток, по пути дерева  $s \rightsquigarrow t$  и перестраиваем дерево.

Для этого находим на пути  $s \rightsquigarrow t$  любое одно насыщенное ребро  $a \rightarrow b$ , разрезаем его, и для вершины  $a$  добавляем в дерево следующее ребро из  $out[a]$ . Цикла появиться не может: рёбра идут вперёд по слоям. Зато у  $a$  могли просто кончиться рёбра, тогда  $a$  объявляем тупиковой веткой развития, и рекурсивно разрезаем ребро, входящее в  $a$ .

Заметим, что *link-cut-tree* со *splay-tree* умеет делать все описанные операции за  $\mathcal{O}(\log V)$ :

- Поиск минимума и позиции минимума величины  $c_e - f_e$  на пути.
- Уменьшение величины  $c_e - f_e$  на пути.
- Разрезание ребра (cut), проведение нового ребра (link).

Один cut может рекурсивно удалить много рёбер, сильно перестроить дерево. Несмотря на это каждое ребро удалится не более одного раза  $\Rightarrow$  суммарное время всех cut –  $\mathcal{O}(E \log V)$ .

## 10.5. Глобальный разрез

**Задача:** найти разбиение  $V = A \sqcup B: A, B \neq \emptyset, C(A, B) \rightarrow \min$ .

Простейшее решение: переберём  $s, t$  и найдём разрез между ними. Конечно, можно взять  $s = 1$ . Время работы  $\mathcal{O}(V \cdot \text{Flow})$ .

На практике покажем, что на единичных сетях эта идея работает уже за  $\mathcal{O}(E^2)$ .

### 10.5.1. Алгоритм Штор-Вагнера (на практике)

Выберем  $a_1 = 1$ . Пусть  $A_i = \{a_1, a_2, \dots, a_i\}$ . Определим  $a_{i+1} = v \in V \setminus A_i: C(A_i, \{v\}) = \max$ .

*Утверждение 10.5.1.* Минимальный разрез между  $a_n$  и  $a_{n-1} - S = \{a_n\}, T = V \setminus S$ , где  $n = |V|$ .

*Доказательство.* Можно прочесть на **e-maxx**. ■

**Алгоритм:** или  $a_n$  и  $a_{n-1}$  по разные стороны оптимального глобального разреза, и ответ равен  $C(\{a_n\}, V \setminus \{a_n\})$ , или  $a_n$  и  $a_{n-1}$  можно стянуть в одну вершину.

**Время работы:**  $V$  фаз, каждая за  $\mathcal{O}(\text{Dijkstra})$ , получаем  $\mathcal{O}(V^3)$  либо  $\mathcal{O}(V(E + V \log V))$ .

### 10.5.2. Алгоритм Каргера-Штейна

Пусть  $c_e \equiv 1$ . Минимальную степень обозначим  $k$ .

$\exists v: \deg_v = k \Rightarrow C(\{v\}, V \setminus \{v\}) = k \Rightarrow$  в min разрезе не более  $k$  рёбер.  $E = \frac{1}{2} \sum \deg_v \geq \frac{1}{2} kV$ .

Возьмём случайное ребро  $e$ , вероятность того, что оно попало в разрез  $Pr[e \in \text{cut}] \leq \frac{k}{E} = \frac{2}{V}$ .

#### • Алгоритм Каргера.

Пока в графе  $> 2$  вершин, выбираем случайное ребро, не являющееся петлёй, стягиваем его концы в одну вершину. В конце  $V' = \{A, B\}$ , объявляем минимальным разрезом  $V = A \sqcup B$ .

**Время работы:**  $T(V) = V + T(V - 1) = \Theta(V^2)$ . **Здесь  $V$  – время стягивания двух вершин.**

**Вероятность успеха:** ни разу не ошиблись с  $Pr \geq \frac{V-2}{V} \cdot \frac{V-3}{V-1} \cdot \frac{V-4}{V-2} \cdot \dots \cdot \frac{1}{3} = \frac{2 \cdot 1}{V \cdot (V-1)} \geq \frac{2}{V^2}$ .

Чтобы алгоритм имел константную вероятность ошибки, достаточно запустить его  $V^2 \Rightarrow$  получили RP-алгоритм за  $\mathcal{O}(V^4)$ .

#### • Алгоритм Каргера-Штейна.

В оценке  $\frac{V-3}{V-1} \cdot \frac{V-4}{V-2} \cdot \dots \cdot \frac{1}{3}$  большинство первых сомножителей близки к 1. Последние сомножители –  $\frac{2}{4}, \frac{1}{3}$  напротив весьма малы  $\Rightarrow$  остановимся, когда в графе останется  $\frac{V}{\sqrt{2}}$  вершин.

Вероятность ни разу не ошибиться при этом будет  $\frac{V/\sqrt{2}(V/\sqrt{2}-1)}{V(V-1)} \approx \frac{1}{2}$ .

**Время, потраченное на  $(V - V/\sqrt{2})$  сжатий –  $\Theta(V^2)$ .**

После этого сделаем два рекурсивных вызова от получившегося графа с  $V/\sqrt{2}$  вершинами. Алгоритм рандомизированный  $\Rightarrow$  ветки рекурсии могут дать разные разрезы  $\Rightarrow$  вернём мини-

мальный из полученных.

**Время работы:**  $T(V) = V^2 + 2T(\frac{V}{\sqrt{2}}) = V^2 + 2(\frac{V}{\sqrt{2}})^2 + 4T(\frac{V}{\sqrt{2}^2}) = \dots = V^2 \log V$ .

**Вероятность ошибки.** Ищем  $p(V)$ , вероятность успеха на графе из  $V$ .

Обозначим  $p(V) = q_k, p(V/\sqrt{2}) = q_{k-1}, \dots, \Rightarrow$

$$q_i = \frac{1}{2}(1 - (1 - q_{i-1})^2) = q_{i-1} - \frac{1}{2}q_{i-1}^2$$

Теперь можно просто доказать по индукции утверждение  $q_k \geq \frac{1}{k+2}$ .

Или можно поступить так:  $q_i - q_{i-1} = -q_{i-1}^2$ . Левая часть похожа на производную  $\Rightarrow$  решим диффур  $q'(x) = -q(x)^2 \Leftrightarrow \frac{-q'(x)}{q(x)^2} = 1 \Leftrightarrow q(x)^{-1} = x + C \Rightarrow q_k = \Theta(\frac{1}{k})$ .

Таким образом,  $p(V) = \Theta(\frac{1}{\log V})$ .

Короткая и быстрая реализация получается через `random_shuffle` исходных рёбер.

Подробнее в разборе **практики**.

## 10.6. (\*) Алгоритм Push-Relabel

Конспект по этой теме лежит в отдельном **файле**.



# Лекция #11: Mincost

12 ноября 2020

## 11.1. Mincost k-flow в графе без отрицательных циклов

Сопоставим всем прямым рёбрам вес (стоимость)  $w_e \in \mathbb{R}$ .

**Def 11.1.1.** *Стоимость потока  $W(f) = \sum_e w_e f_e$ . Сумма по прямым рёбрам.*

Обратному к  $e$  рёбру  $\bar{e}$  сопоставим  $w_{\bar{e}} = -w_e$ .

Если толкнуть поток сперва по прямому, затем по обратному к нему ребру, стоимость не изменится. Когда мы толкаем единицу потока по пути **path**, изменение потока и стоимости потока теперь выглядят так:

```
1 for (int e : path):
2     edges[e].f++
3     edges[e ^ 1].f--
4     W += edges[e].w;
```

**Задача mincost k-flow:** найти поток  $f: |f| = k, W(f) \rightarrow \min$

При решении задачи мы будем говорить про веса путей, циклов, “отрицательные циклы”, кратчайшие пути... Везде вес пути/цикла – сумма весов рёбер ( $w_e$ ).

**Решение #1.** Пусть в графе нет отрицательных циклов, а также все  $c_e \in \mathbb{Z}$ .

Тогда по аналогии с алгоритмом Ф.Ф., который за  $\mathcal{O}(k \cdot \text{dfs})$  искал поток размера  $k$ , мы можем за  $\mathcal{O}(k \cdot \text{FordBellman})$  найти mincost поток размера  $k$ . Обозначим  $f_k$  оптимальный поток размера  $k \Rightarrow f_0 \equiv 0, f_{k+1} = f_k + \text{path}$ , где  $\text{path}$  – кратчайший в  $G_{f_k}$ .

**Lm 11.1.2.**  $\forall k, |f| = k \quad (W(f) = \min) \Leftrightarrow (\nexists \text{ отрицательного цикла в } G_f)$

*Доказательство.* Если отрицательный цикл есть, увеличим по нему поток,  $|f|$  не изменится,  $W(f)$  уменьшится. Пусть  $\exists f^*: |f^*| = |f|, W(f^*) < W(f)$ , рассмотрим поток  $f^* - f$  в  $G_f$ .

Это циркуляция, мы можем декомпозировать её на циклы  $c_1, c_2, \dots, c_k$ .

Поскольку  $0 > W(f^* - f) = W(c_1) + \dots + W(c_k)$ , среди циклов  $c_i$  есть отрицательный. ■

**Теорема 11.1.3.** Алгоритм поиска mincost потока размера  $k$  корректен.

*Доказательство.* База: по условию нет отрицательных циклов  $\Rightarrow f_0$  корректен.

Переход: обозначим  $f_{k+1}^*$  mincost поток размера  $k+1$ , смотрим на декомпозицию  $\Delta f = f_{k+1}^* - f_k$ .  $|\Delta f| = 1 \Rightarrow$  декомпозиция = путь  $p$  + набор циклов. Все циклы по 11.1.2 неотрицательны  $\Rightarrow W(f_k + p) \leq W(f_{k+1}^*) \Rightarrow$ , добавив, кратчайший путь мы получим решение не хуже  $f_{k+1}^*$ . ■

**Lm 11.1.4.** Если толкнуть сразу  $0 \leq x \leq \min_{e \in p} (c_e - f_e)$  потока по пути  $p$ , то получим оптимальный поток размера  $|f| + x$ .

*Доказательство.* Обозначим  $f^*$  оптимальный поток размера  $|f| + x$ , посмотрим на декомпозицию  $f^* - f$ , заметим, что все пути в ней имеют вес  $\geq W(p)$ , а циклы вес  $\geq 0$ . ■

## 11.2. Потенциалы и Дейкстра

Для ускорения хотим Форда-Беллмана заменить на Дейкстру.

Для корректности Дейкстры нужна неотрицательность весов.

В прошлом семестре мы уже сталкивались с такой задачей, когда изучали **алгоритм Джонсона**.

### • Решение задачи mincost k-flow.

Запустим один раз Форда-Беллмана из  $s$ , получим массив расстояний  $d_v$ , применим потенциалы  $d_v$  к весам рёбер:

$$e: a \rightarrow b \Rightarrow w_e \rightarrow w_e + d_a - d_b$$

Напомним, что из корректности  $d$  имеем  $\forall e \ d_a + w_e \geq d_b \Rightarrow w'_e \geq 0$ .

Более того: для всех рёбер  $e$  кратчайших путей из  $s$  верно  $d_a + w_e = d_b \Rightarrow w'_e = 0$ .

В  $G_f$  найдём Дейкстрой из  $s$  кратчайший путь  $p$  и расстояния  $d'_v$ .

Пустим по пути  $p$  поток, получим новый поток  $f' = f + p$ .

В сети  $G'_f$  могли появиться новые рёбра (обратные к  $p$ ). Они могут быть отрицательными.

Пересчитаем веса:

$$e: a \rightarrow b \Rightarrow w_e \rightarrow w_e + d'_a - d'_b$$

Поскольку  $d'$  – расстояния, посчитанные в  $G_f$ , все рёбра из  $G_f$  останутся неотрицательными.

$p$  – кратчайший путь, все рёбра  $p$  станут нулевыми  $\Rightarrow$  рёбра обратные  $p$  тоже будут нулевыми.

### • Псевдокод

```

1 def applyPotentials(d):
2     for e in Edges:
3         e.w = e.w + d[e.a] - d[e.b]
4 d <-- FordBellman(s)
5 applyPotentials(d)
6 for i = 1..k:
7     d, path <-- Dijkstra(s)
8     for e in path: e.f += 1, e.rev.f -= 1
9     applyPotentials(d)

```

## 11.3. Графы с отрицательными циклами

**Задача:** найти mincost циркуляцию.

**Алгоритм Клейна:** пока в  $G_f$  есть отрицательный цикл, пустим по нему  $\min_e (c_e - f_e)$  потока.

Пусть  $\forall e \ c_e, w_e \in \mathbb{Z} \Rightarrow W(f)$  каждый раз уменьшается хотя бы на 1  $\Rightarrow$  алгоритм конечен.

**Задача:** найти mincost  $k$ -flow в графе с отрицательными циклами.

**Решение #1:** найти за  $|W(f)|$  итераций mincost циркуляцию, перейти от  $f_0$  за  $k$  итераций к  $f_k$ .

**Решение #2:** найти любой поток  $f: |f| = k$ , в  $G_f$  найти mincost циркуляцию, сложить с  $f$ .

## 11.4. Mincost flow (на практике)

**Задача:** найти  $f: W(f) = \min$ , размер  $f$  не важен.

Обозначим  $f_k$  – оптимальный поток размера  $k$ ,  $p_k$  кратчайший путь в  $G_{f_k}$ .

**Lm 11.4.1.**  $W(p_k) \nearrow$ , как функция от  $k$ .

*Доказательство.* Аналогично доказательству леммы для Эдмондса-Карпа 9.9.1.

От противного. Был поток  $f$ , мы увеличили его по кратчайшему пути  $p$ .

Расстояния в  $G_f$  обозначим  $d_0$ , в  $G_{f+p}$  —  $d_1$ .

Возьмём  $v$ :  $d_1[v] < d_0[v]$ , а из таких ближайшую к  $s$  в дереве кратчайших путей.

Рассмотрим кратчайший путь  $q$  в  $G_{f+p}$  из  $s$  в  $v$ :  $s \rightsquigarrow \dots \rightsquigarrow x \rightarrow v$ .

$e = (v \rightarrow x)$ ,  $d_1[v] = d_1[x] + w_e$ ,  $d_1[x] \geq d_0[x] \Rightarrow d_1[v] \geq d_0[x] + w_e \Rightarrow$  ребра  $(x \rightarrow v)$  нет в  $G_f \Rightarrow$

ребро  $(v \rightarrow x) \in p \Rightarrow d_0[x] = d_0[v] + w_e = d_0[v] - w_e \Rightarrow$

$d_1[v] = d_1[x] + w_e \geq d_0[x] + w_e = (d_0[v] - w_e) + w_e = d_0[v]$ . Противоречие. ■

*Следствие 11.4.2.*  $(W(f_k) = \min) \Leftrightarrow (W(p_{k-1}) \leq 0 \wedge W(p_k) \geq 0)$ .

Осталось найти такое  $k$  бинарным или линейным поиском. На текущий момент мы умеем искать  $f_k$  или за  $\mathcal{O}(k \cdot VE)$  с нуля, или за  $\mathcal{O}(VE)$  из  $f_{k-1} \Rightarrow$  линейный поиск будет быстрее.

## 11.5. Полиномиальные решения

Mincost flow мы можем бинарным поиском свести к mincost k-flow.

Mincost k-flow мы можем поиском любого потока размера  $k$  свести к mincost циркуляции.

Осталось научиться за полином искать mincost циркуляцию.

• **Решение #1:** модифицируем алгоритм Клейна, будем толкать  $\min_e (c_e - f_e)$  потока по циклу  $\min$  среднего веса. Заметим, что  $(\exists \text{ отрицательный цикл}) \Leftrightarrow (\min \text{ средний вес} < 0)$ .

Решение работает за  $\mathcal{O}(VE \log(V \max w))$  поисков цикла. Цикл ищется алгоритмом Карпа за  $\mathcal{O}(VE)$ .

Доказано будет на практике.

• **Решение #2:** Capacity Scaling.

Начнём с графа  $c'_e \equiv 0$ , в нём mincost циркуляция тривиальна.

Будем понемногу наращивать  $c'_e$  и поддерживать mincost циркуляцию. В итоге хотим  $c'_e \equiv c_e$ .

```

1 for k = logU..0:
2   for e in Edges:
3     if c_e содержит бит 2^k:
4       c'_e += 2^k // e: ребро из a_e в b_e
5       Найдём p - кратчайший путь b_e → a_e
6       if W(p) + w_e ≥ 0:
7         нет отрицательных циклов ⇒ циркуляция f оптимальна
8       else:
9         пустим 2^k потока по циклу p + e (изменим f)
10        пересчитаем потенциалы, используя расстояния, найденные Дейкстрой

```

Время работы алгоритма  $E \log U$  запусков Дейкстры =  $E(E + V \log V) \log U$ .

**Lm 11.5.1.** После 9-й строки циркуляция  $f$  снова минимальна.

*Доказательство.*  $f$  — минимальная циркуляция до 4-й строки,  $f'$  — после.

Как обычно, рассмотрим  $f' - f$ . Это тоже циркуляция. Декомпозируем её на единичные циклы.

Любой цикл проходит через  $e$  (иначе  $f$  не оптимальна). Через  $e$  проходит не более  $2^k$  циклов.

Каждый из этих циклов имеет вес не меньше веса  $p + e \Rightarrow W(f') \geq W(f + 2^k(p + e))$ . ■

## Лекция #12: Паросочетания

19 ноября

### 12.1. Определения

**Def 12.1.1.** Паросочетание (*matching*) – множество попарно не смежных рёбер  $M$ .

**Def 12.1.2.** Вершинное покрытие (*vertex cover*) – множество вершин  $C$ , что у любого ребра хотя бы один конец лежит в  $C$ .

**Def 12.1.3.** Независимое множество (*independent set*) – множество попарно несмежных вершин  $I$ .

**Def 12.1.4.** Клика (*clique*) – множество попарно смежных вершин.

**Def 12.1.5.** Совершенное паросочетание – паросочетание, покрывающее все вершины графа. В двудольном графе совершенным является паросочетание, покрывающее все вершины меньшей доли.

**Def 12.1.6.** Относительно любого паросочетания все вершины можно поделить на

- покрытые паросочетанием (принадлежащие паросочетанию),
- не покрытые паросочетанием (свободные).

Обозначения: Matching (M), Vertex Cover (VC или C), Independent Set (IS или I).

### 12.2. Сведения

Пусть дан граф  $G$ , заданный матрицей смежности  $g_{ij}$ . Инвертацией  $G$  назовём граф  $G'$ , заданный матрицей смежности  $g'_{ij} = 1 - g_{ij}$ . тогда независимое множество в  $G$  задаёт клику в  $G'$ , а клика в  $G$  задаёт независимое множество в  $G'$ .

*Следствие 12.2.1.* Задачи поиска max клики и max IS сводятся друг к другу.

**Lm 12.2.2.** Дополнение любого VC – IS. Дополнение любого IS – VC.

*Следствие 12.2.3.* Все три задачи поиска min VC, max IS, max clique сводятся друг к другу.

*Утверждение 12.2.4.* Задачи поиска min VC, max IS, max clique NP-трудны.

Утверждение было доказано в прошлом семестре в разделе про сложность.

### 12.3. Дополняющие чередующиеся пути

**Def 12.3.1.** Чередующийся путь – простой путь, в котором рёбра чередуются в смысле принадлежности паросочетанию.

**Def 12.3.2.** Дополняющий чередующийся путь (ДЧП) – чередующийся путь, первая и последняя вершина которого не покрыты паросочетанием.

**Lm 12.3.3.** Паросочетание  $P$  максимально  $\Leftrightarrow \nexists$  ДЧП (лемма о дополняющем пути).

*Доказательство.* Пусть  $\exists$  ДЧП  $\Rightarrow$  инвертируем все рёбра на нём, получим паросочетание размера  $|P + 1|$ . Докажем теперь, что если  $\exists$  паросочетание  $M: |M| > |P|$ , то  $\exists$  ДЧП. Для этого рассмотрим  $S = M \nabla P$ . Степень каждой вершины в  $S$  не более двух (одно ребро из  $M$ , одно из  $P$ )  $\Rightarrow S$  является объединением циклов и путей. Каждому пути сопоставим число  $a_i$  – разность количеств рёбер из  $M$  и  $P$ . Тогда  $|M| = |P| + \sum_i a_i \Rightarrow \exists a_i > 0 \Rightarrow$  один из путей – ДЧП. ■

Лемма доказана для произвольного графа, но с лёгкостью найти ДЧП мы сможем только для двудольного графа.

## 12.4. Поиск паросочетания в двудольном графе

**Lm 12.4.1.** Пусть  $G$  – двудольный граф, а  $P$  паросочетание в нём. Построим оргграф  $G'(G, P)$ , в котором из первой доли во вторую есть все рёбра  $G$ , а из второй в первую долю только рёбра из  $P$ . Тогда есть биекция между путями в  $G'$  и чередующимися путями в  $G$ .

**Lm 12.4.2.** Поиск ДЧП в  $G \Leftrightarrow$  поиску пути в  $G'$  из свободной вершины в свободную.

*Следствие 12.4.3.* Мы получили алгоритм поиска максимального паросочетания  $M$  за  $\mathcal{O}(|M| \cdot E)$ :

0.  $P \leftarrow \emptyset$
1. Попробуем найти путь dfs-ом в  $G'(G, P)$
2. if не нашли  $\Rightarrow M$  максимально
3. else goto 1

## 12.5. Реализация

Важной идеей является применение ДЧП к паросочетания на обратном ходу рекурсии.

```

1 def dfs(v):
2     used[v] = 1 # массив пометок для вершин первой доли
3     for x in graph[v]: # рёбра из 1-й доли во вторую
4         if (pair[x] == -1) or (used[pair[x]] == 0 and dfs(pair[x])):
5             pair[x] = v # массив пар для вершин второй доли
6             return True
7     return False

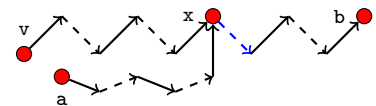
```

Граф  $G'$  в явном виде мы нигде не строим. Вместо этого, когда идём из 1-й доли во вторую, перебираем все рёбра ( $v \rightarrow x$  in  $\text{graph}[v]$ ), а когда из 2-й в первую, идём только по ребру паросочетания ( $x \rightarrow \text{pair}[x]$ ).

## 12.6. Алгоритм Куна

**Lm 12.6.1.** В процессе поиска максимального паросочетания  $\nexists$  ДЧП из  $v \Rightarrow$  ДЧП из  $v$  уже никогда не появится.

Пусть появился ДЧП  $p: v \rightsquigarrow u$  после применения ДЧП  $q: a \rightsquigarrow b$ . Тогда пойдём по  $p$  из  $v$ , найдём  $x \in p$  – первую вершину в  $q$ . Из  $x$  пойдём по пути  $a \leftrightarrow b$  по ребру паросочетания, дойдём до конца, получим пути  $v \rightsquigarrow (a \vee b)$ , который существовал до применения  $q$ .



Получили алгоритм Куна:

```
1 for v in range(n):
2     used = [0] * n
3     dfs(v)
```

## 12.7. Оптимизации Куна

Обозначим за  $k$  размер максимального паросочетания. Сейчас время работы алгоритма Куна  $\mathcal{O}(VE)$  даже для  $k = \mathcal{O}(1)$ . Будем обнулять пометки `used[]` только, если мы нашли ДЧП.

```
1 used = [0] * n
2 for v in range(n):
3     if dfs(v):
4         used = [0] * n
```

Алгоритм остался корректным, так как, между успешными запусками `dfs` граф  $G'$  не меняется. И теперь работает за  $\mathcal{O}(kE)$ .

Следующая оптимизация – “жадная инициализация”. До запуска Куна переберём все рёбра и те из них, что можем, добавим в паросочетание.

**Lm 12.7.1.** Жадная инициализация даст паросочетание размера  $\geq \frac{k}{2}$ .

*Доказательство.* Если мы взяли ребро, которое на самом деле не должно лежать в максимальном паросочетании  $M$ , мы заблокировали возможность взять  $\leq 2$  рёбер из  $M$ . ■

При использовании жадной инициализации у нас появляется необходимость поддерживать массив `covered[v]`, хранящий для вершины первой доли, покрыта ли она паросочетанием.

Попробуем теперь сделать следующее:

```
1 used = [0] * n
2 for v in range(n):
3     if not covered[v]:
4         dfs(v)
```

Код работает за  $\mathcal{O}(V + E)$ . И, если паросочетание не максимально, найдёт хотя бы один ДЧП. А может найти больше чем один... в этом и заключается последняя оптимизация “вообще не обнулять пометки”: пока данный код находит хотя бы один путь, запускать его.

*Замечание 12.7.2.* Докажите, что если мы используем последнюю оптимизацию, “жадная инициализация” является полностью бесполезной.

Напоминание: мы умеем обнулять пометки за  $\mathcal{O}(1)$ . Для этого помеченной считаем вершины  $v$ : “`used[v] == cc`”, тогда операция “`cc++`” сделает все вершины не помеченными.

## 12.8. Поиск минимального вершинного покрытия (на практике)

**Lm 12.8.1.**  $\forall M, VC$  верно, что  $|VC| \geq |M|$

*Доказательство.* Для каждого ребра  $e \in M$ , нужно взять в  $VC$  хотя бы один из концов  $e$ . ■

Пусть у нас уже построено максимальное паросочетание  $M$ . Запустим `dfs` на  $G'$  из всех свободных вершин первой доли. Обозначим первую долю  $A$ , вторую  $B$ . Посещённые `dfs`-ом вершины соответственно  $A^+$  и  $B^+$ , а непосещённые  $A^-$  и  $B^-$ .

**Теорема 12.8.2.**  $X = A^- \cup B^+$  – минимальное вершинное покрытие.

*Доказательство.* Если бы из  $a \in A^+$  было бы ребро в  $b \in B^-$ , мы бы по нему прошли, и  $b$  лежала бы в  $B^+ \Rightarrow$  в  $A^+ \cup B^-$  нет рёбер  $\Rightarrow X$  – вершинное покрытие. Оценим размер  $X$ : все вершины из  $A^- \cup B^+$  – концы рёбер паросочетания  $M$ , т.к. **dfs** не нашёл дополняющего пути. Более того это концы обязательно разных рёбер паросочетания, т.к. если один конец ребра паросочетания лежит в  $B^+$ , то **dfs** пойдёт по нему, и второй конец окажется в  $A^+$ . Итого  $|X| \leq |M|$ . Из этого и 12.8.1 следует  $|X| = |M|$  и  $|X| = \max$ . ■

*Следствие 12.8.3.*  $A^+ \cup B^-$  – максимальное независимое множество.

*Замечание 12.8.4.* Мы умеем строить  $\min VC$  и  $\max IS$  за  $\mathcal{O}(V + E)$  при наличии максимального паросочетания.

*Следствие 12.8.5.*  $\max M = \min VC$  (теорема Кёнига).

## 12.9. Обзор решений

Мы изучили алгоритм Куна со всеми оптимизациями.

Асимптотически время его работы  $\mathcal{O}(VE)$ , на практике же он жутко шустрый.

На графах  $V, E \leq 10^5$  решение укладывается в секунду.

∃ также алгоритм Хопкрофта-Карпа за  $\mathcal{O}(EV^{1/2})$ . Его мы изучим в контексте потоков.

В регулярном двудольном графе можно найти совершенное паросочетание за время  $\mathcal{O}(V \log V)$ .

[Статья Михаила Капралова и ко.](#)

## 12.10. Решения для произвольного графа

### 12.10.1. Обзор

Наиболее стандартным считается решение через сжатие “соцветий” (видим нечётный цикл – сожмём его, найдём паросочетание в новом цикле, разожмём цикл, перестроим паросочетание). Этот подход используют реализации Габова за  $\mathcal{O}(V^3)$  и Тарьяна за  $\mathcal{O}(VE\alpha)$ . Подробно эту тему мы будем изучать на 3-м курсе.

Оптимальный по времени – алгоритм Вазирани,  $\mathcal{O}(EV^{1/2})$ .

Кроме этого есть два подхода, которые мы обсудим подробнее.

### 12.10.2. Матрица Татта

Пусть дан *двудольный* граф  $G(L, R, E)$ ,  $|L| = |R| = n$ . Определим для него матрицу Эдмондса  $D$  размера  $n \times n$ :  $d_{i,j} = x_{i,j}$ , если есть ребро из  $i \in L$  в  $j \in R$ . В противном случае  $d_{i,j} = 0$ .

Здесь  $x_{i,j}$  – это  $n^2$  различных переменных.

$\det D$  – это многочлен от  $n^2$  переменных. Легко видеть, что он тождественно равен нулю тогда и только тогда, когда в графе нет совершенного паросочетания (слагаемые в определителе не взаимоуничтожаются, а любое такое слагаемое соответствует совершенному паросочетанию).

Пусть теперь граф произвольный. Определим для него **матрицу Татта**  $T$ . Для каждого ребра  $(i, j)$  элементы  $t_{ij} = x_{ij}, t_{ji} = -x_{ij}$ .

Остальные элементы равны нулю. Здесь  $x_{ij}$  – переменные.

Получается, для каждого ребра неорграфа мы ввели ровно одну переменную.



$\det T$  – многочлен от  $n(n-1)/2$  переменных.

**Теорема Татта:**  $\det T \neq 0 \Leftrightarrow \exists$  совершенное паросочетание.

Доказательство чуть сложнее случая доказательства для матрицы Эдмондса и будет в курсе дискретной математики.

Чтобы проверить  $\det T \equiv 0$ , используем лемму Шварца-Зиппеля: в каждую переменную  $x_{ij}$  подставим случайное значение, посчитаем определитель матрицы над полем  $\mathbb{F}_p$ , где  $p = 10^9 + 7$ , получим вероятность ошибки не более  $\deg(\det T)/p = n/p$ .

Время работы  $\mathcal{O}(n^3)$  (нахождение определителя матрицы по модулю  $p$ ), алгоритм умеет лишь проверять наличие совершенного паросочетания.

Алгоритм можно модифицировать сперва для определения размера максимального паросочетания, а затем для его нахождения.

Пример:  $n = 3$ ,  $E = \{(1, 2), (2, 3)\}$ ,  $T = \begin{bmatrix} 0 & x_{12} & 0 \\ -x_{12} & 0 & x_{23} \\ 0 & -x_{23} & 0 \end{bmatrix}$ , подставляем  $x_{12} = 9$ ,  $x_{23} = 7$ , считаем

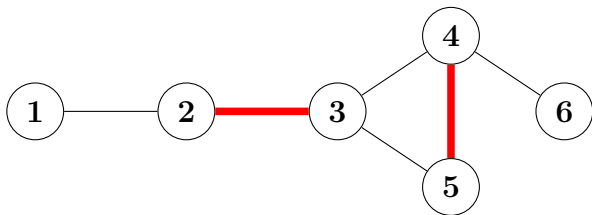
$$\det \begin{bmatrix} 0 & 9 & 0 \\ -9 & 0 & 7 \\ 0 & -7 & 0 \end{bmatrix} = 0 \Rightarrow \text{с большой вероятностью нет совершенного паросочетания.}$$

### 12.10.3. Читерский подход

Давайте на недвудольном графе запустим dfs из Куна для поиска дополняющей цепи...

При этом пометать, как посещённые, будем вершины обеих долей в одном массиве `used`.

С некоторой вероятностью алгоритм успешно найдёт дополняющий путь.



Если мы запускаем dfs из вершины 1, то у неё есть шанс найти дополняющий путь  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 6$ . Но если из вершины 3 dfs сперва попытается идти в 4, то он пометит 4 и 5, как посещённые, больше в них заходить не будет, путь не найдёт.

Т.е. на данном примере в зависимости от порядка соседей вершины 3 dfs или найдёт, или не найдёт путь. Если выбрать случайный порядок, то найдёт с вероятностью  $1/2$ .

Это лучше, чем “при некотором порядке рёбер вообще не иметь возможности найти путь”, поэтому первой строкой dfs добавляем `random_shuffle(c[v].begin(), c[v].end())`.

На большинстве графов алгоритм сможет найти максимальное паросочетание.  $\exists$  графы, на которых вероятность нахождения дополняющей цепи экспоненциально от числа вершин мала.



## Лекция #13: Паросочетания (продолжение)

26 ноября

### 13.1. Классификация рёбер двудольного графа

Дан двудольный граф  $G = \langle V, E \rangle$ . Задача – определить  $\forall e \in E$ ,  $\exists$  ли максимальное паросочетание  $M_1: e \in M_1$ , а также  $\exists$  ли максимальное паросочетание  $M_2: e \notin M_2$ .  
Иначе говоря, мы хотим разбить рёбра на три класса:

1. Должно лежать в максимальном паросочетании. **MUST**.
2. Может лежать в максимальном паросочетании, а может не лежать. **MAY**.
3. Не лежит ни в каком максимальном паросочетании. **NO**.

**Решение:** для начала найдём любое максимальное паросочетание  $M$ .

Если мы найдём класс **MAY**, то  $\text{MUST} = M \setminus \text{MAY}$ ,  $\text{NO} = \bar{M} \setminus \text{MAY}$ .

**Lm 13.1.1.**  $e \in \text{MAY} \Leftrightarrow \exists P$  – чередующийся относительно  $M$  путь чётной длины или чётный цикл, при этом  $e \in P$ .

*Доказательство.* Если ребро  $e \in \text{MAY}$ , то  $\exists$  другое максимальное паросочетание  $M': e \in M \nabla M'$ . Симметрическая разность, как мы уже знаем, состоит из чередующихся путей и циклов.

Посмотрим с другой стороны: если относительно  $M$  есть  $P$  – чередующийся путь чётной длины или чередующийся цикл, то  $P \subseteq \text{MAY}$ , так как и  $M$ , и  $M \nabla P$  являются максимальными, а каждое ребро  $P$  лежит ровно в одном из двух. ■

Осталось научиться находить циклы и пути алгоритмически. Для этого рассмотрим тот же граф  $G'$ , на котором работает dfs из Куна. С чётными путями всё просто: все они начинаются в свободных вершинах, dfs из Куна, запущенный от всех свободных вершин *обеих долей* пройдёт ровно по всем рёбрам, которые можно покрыть чётными путями, и пометит их.

**Lm 13.1.2.** Ребро  $e$  лежит на чётном цикле  $\Leftrightarrow$  концы  $e$  лежат в одной компоненте сильной связности графа  $G'$ .

### 13.2. Stable matching (marriage problem)

Сформулируем задачу на языке мальчиков/девочек. Есть  $n$  мальчиков, у каждого из них есть список девочек  $bs[a]$ , которые ему нравятся в порядке от наиболее приоритетных к менее. Есть  $m$  девочек, у каждой есть список мальчиков  $as[b]$ , которые ей нравятся в таком же порядке. Мальчики и девочки хотят образовать пары.

Никто не готов образовывать пару с тем, кто вообще отсутствует в его списке.

И для мальчиков, и для девочек наименее приоритетный вариант – остаться вообще без пары.

Будем обозначать  $p_a$  – пара мальчика  $a$  или  $-1$ ,  $q_b$  – пара девочки  $b$  или  $-1$ .

**Def 13.2.1.** Паросочетание называется **не стабильным**, если  $\exists$  мальчик  $a$  и девочка  $b$ : мальчику  $a$  нравится  $b$  больше чем  $p_a$  и девочке  $b$  нравится  $a$  больше чем  $q_b$ .

**Def 13.2.2.** Иначе паросочетание называется **стабильным**

### • Алгоритм поиска: “мальчики предлагают, девочки отказываются”

Изначально проинициализируем  $p_a = bs[a].first$ , далее, пока есть

два мальчика  $i, j: b = p_i = p_j \neq -1$ , Девочка  $b$  откажет тому из них, кто ей меньше нравится.

Пусть она отказала мальчику  $i$ , тогда делаем  $bs[i].remove\_first()$ ,  $p_i = bs[i].first$ .

**Теорема 13.2.3.** Алгоритм всегда завершится и найдёт стабильное паросочетание

*Доказательство.* Длины списков  $bs[i]$  убывают  $\Rightarrow$  завершится. Пусть мальчик  $a$  и девочка  $b$  образуют не стабильность. Но это значит, что  $a$  перед тем, как образовать пару с  $p_a$ , предлагал себя  $b$ , а она ему отказала. Но зачем?! Ведь он ей нравился больше. Противоречие. ■

Аккуратная реализация даёт время  $\mathcal{O}(\sum_i |bs[i]| + \sum_j |as[j]|)$ : для каждой девочки  $b$  поддерживаем  $s_b = \{a: p_a = b\}$ , при присваивании  $p_a \leftarrow b$  добавляем  $a$  в  $s_b$  и, если  $|s_b| \geq 2$ , вызываем рекурсивную процедуру `откажиВсемКромеЛучшего(b)`.

## 13.3. Венгерский алгоритм

Дан взвешенный двудольный граф, заданный матрицей весов  $a: n \times n$ , где  $a_{ij}$  – вес ребра из  $i$ -й вершины первой доли в  $j$ -ю вершину второй доли. Задача – найти совершенное паросочетание минимального веса.

Формально: найти  $\pi \in S_n: \sum_i a_{i\pi_i} \rightarrow \min$ .

Иногда задачу называют *задачей о назначениях*, тогда  $a_{ij}$  – стоимость выполнения  $i$ -м работником  $j$ -й работы, нужно каждому работнику сопоставить одну работу.

**Lm 13.3.1.** Если  $a_{ij} \geq 0$  и  $\exists$  совершенное паросочетание на нулях, оно оптимально.

**Lm 13.3.2.** Рассмотрим матрицу  $a'_{ij} = a_{ij} + row_i + col_j$ , где  $row_i, col_j \in \mathbb{R}$ .

Оптимальные паросочетания для  $a'$  и для  $a$  совпадают.

*Доказательство.* Достаточно заметить, что в  $(f = \sum_i a_{i\pi_i})$  войдёт ровно по одному элементу каждой строки, каждого столбца  $\Rightarrow$  если все элементы строки (столбца) увеличить на константу  $C$ , в не зависимости от  $\pi$  величина  $f$  увеличится на  $C \Rightarrow$  оптимум перейдёт в оптимум. ■

Венгерский алгоритм, как Кун, перебирает вершины первой доли и от каждой пытается строить ДЧП, но использует при этом только нулевые рёбра. Если нет нулевого ребра, то  $x = \min$  на  $A^+ \times B^- > 0$ . Давайте все столбцы из  $B^-$  уменьшим на  $x$ , а все строки из  $A^-$  увеличим на  $x$ .

|       | $B^+$ | $B^-$ |
|-------|-------|-------|
| $A^+$ |       | $-x$  |
| $A^-$ | $+x$  | 0     |

В итоге в подматрице  $A^+ \times B^-$  на месте минимального элемента появится 0, в матрице  $A^- \times B^+$  элементы увеличатся, остальные не изменятся. При этом все элементы матрицы остались неотрицательными. Рёбра из  $A^- \times B^+$  могли перестать быть нулевыми, но они не лежат ни в текущем паросочетании, ни в дереве дополняющих цепей:  $M \subseteq (A^- \times B^-) \cup (A^+ \times B^+)$ , рёбра дополняющих цепей идут из  $A^+$ .

### 13.3.1. Реализация за $\mathcal{O}(V^3)$

Венгерский алгоритм =  $V$  поисков ДЧП.

Поиск ДЧП = инициализировать  $A^+ = B^+ = \emptyset$  и не более  $V$  раз найти минимум  $x = a_{ij}$  в  $A^+ \times B^-$ . Если  $x > 0$ , то пересчитать матрицу весов. Посетить столбец  $j$  и строку  $pair_j$ .

Чтобы быстро увеличивать столбец/строку на константу, поддерживаем  $row_i, col_j$ .

Реальное значение элемента матрицы:  $a'_{ij} = a_{ij} + row_i + col_j$ . Увеличение строки на  $x$ :  $row_j += x$ .

Чтобы найти минимум  $x$ , а также строку  $i$ , столбец  $j$ , на которых минимум достигается, воспользуемся идеей из алгоритма Прима:  $w_j = \min_{i \in A^+} \langle a'_{ij}, i \rangle$ . Тогда  $\langle \langle x, i \rangle, j \rangle = \min_{j \in B^-} \langle w_j, j \rangle$ .

Научились находить  $(x, i, j)$  за  $\mathcal{O}(n)$ , осталось при изменении  $row_i, col_j$  пересчитать  $w_j: j \in B^-$ .  $col_j += y \Rightarrow w_k += y$ . А  $row_i$  будет меняться только у  $i \in A^- \Rightarrow$  на  $\min_{i \in A^+}$  не повлияет.

*Замечание 13.3.3.* Можно выбирать  $\min$  в множестве  $w_j: j \in B^-$  не за линию, а используя кучи.

### 13.3.2. Псевдокод

Обозначим, как обычно, первую долю  $A$ , вторую  $B$ , посещённые вершины –  $A^+, B^+$ .

Также, как в Куне, если  $x \in B$ , то  $pair[x] \in A$  – её пара в первой доли.

Строки – вершины первой доли ( $A$ ). В нашем коде строки –  $i, v, u$ .

Столбцы – вершины второй доли ( $B$ ). В нашем коде столбцы –  $j$ .

$pair[b \in B]$  – её пара в  $A$ ,  $pair2[a \in A]$  – её пара в  $B$ .

```

1. row  $\leftarrow$  0, col  $\leftarrow$  0
2. for v  $\in$  A
3.    $A^+ = \{v\}, B^+ = \emptyset$  // (остальное в  $A^-, B^-$ ).
4.   for j  $\in B^-$ : w[j] = (a[v][j] + row[v] + col[j], v) // (минимум и номер строки)
5.   while (True) // (пока не нашли путь из v в свободную вершину B)
6.     ((x,i),j) = min{(w[j],j): j  $\in B^-$ } // (минимум и позиция минимума в  $A^+ \times B^-$ )
7.     // (i - номер строки, j - номер столбца  $\Rightarrow$  a[i,j] + row[i] + col[j] == x)
8.     for i  $\in A^-$ : row[i] += x;
9.     for j  $\in B^-$ : col[j] -= x, w[j].value -= x;
10.    // (в итоге мы уменьшили  $A^+ \times B^-$ , увеличили  $A^- \times B^+$ , пересчитали w[j]).
11.    j перемещаем из  $B^-$  в  $B^+$ ; запоминаем prev[j] = i;
12.    if (u=pair[j]) == -1
13.      break // дополняющий путь: j,prev[j],pair2[prev[j]],prev[pair2[prev[j]]],...
14.    u перемещаем из  $A^-$  в  $A^+$ ;
15.    пересчитываем все w[j] = min(w[j], pair(a[u][j] + row[u] + col[j], u));
16.    применим дополняющий путь  $v \rightsquigarrow j$ , пересчитаем pair[], pair2[]

```

Текущая реализация даёт время  $\mathcal{O}(V^3)$ .

Внутри цикла while строки 6, 8, 9, 15 работают за  $\mathcal{O}(V)$  каждая.

Из них 8 и 9 улучшить до  $\mathcal{O}(1)$ , храня специальные величины addToAMinus, addToBMinus.

Строки 6 и 15 можно улучшить до  $\langle \mathcal{O}(\log V), deg[x] \cdot \mathcal{O}(1) \rangle$ , применив кучу Фибоначчи.

Итого получится  $\mathcal{O}(V(E + V \log V))$ .

## Лекция #14: (\*) Суффиксный массив за $\mathcal{O}(n)$

2 декабря 2020

---

Построим суффиксный массив за время  $\mathcal{O}(n)$  – алгоритм Каркайнена-Сандерса.

На вход получаем строку  $s$  длины  $n$ , при этом  $0 \leq s_i \leq \frac{3}{2}n$ .

Выход – суффиксный массив. Сортируем именно суффиксы, а не циклические сдвиги.

Допишем к строке 3 нулевых символа. Теперь сделаем новый алфавит:  $w_i = (s_i, s_{i+1}, s_{i+2})$ .

Отсортируем  $w_i$  цифровой сортировкой за  $\mathcal{O}(n)$ , перенумеруем их от 0 до  $n-1$ .

Запишем все суффиксы строки  $s$  над новым алфавитом:

$$t_0 = w_0 w_3 w_6 \dots$$

$$t_1 = w_1 w_4 w_7 \dots$$

$$t_2 = w_2 w_5 w_8 \dots$$

...

$$t_{n-1} = w_{n-1}$$

Про суффиксы  $t_{3k+i}$ , где  $i \in \{0, 1, 2\}$ , будем говорить “суффикс  $i$ -типа”.

Запустимся рекурсивно от строки  $t_0 t_1$ . Длина  $t_0 t_1$  не более  $2 \lceil \frac{n}{3} \rceil$ .

Теперь мы умеем сравнивать между собой все суффиксы 0-типа и 1-типа.

Суффикс 2-типа = один символ + суффикс 0-типа  $\Rightarrow$

их можно рассматривать как пары и отсортировать за  $\mathcal{O}(n)$  цифровой сортировкой.

Осталось сделать merge двух суффиксных массивов.

Операция merge работает за линейку, если есть “operator  $<$ ”, работающий за  $\mathcal{O}(1)$ .

Нужно научиться сравнивать суффиксы 2-типа с остальными за  $\mathcal{O}(1)$ .

$\forall i, j: t_{3i+2} = s_{3i+2} t_{3i+3}, t_{3j} = s_{3j} t_{3j+1} \Rightarrow$  чтобы сравнить суффиксы 2-типа и 0-типа, достаточно уметь сравнивать суффиксы 0-типа и 1-типа. Умеем.

$\forall i, j: t_{3i+2} = s_{3i+2} t_{3i+3}, t_{3j+1} = s_{3j+1} t_{3j+2} \Rightarrow$  чтобы сравнить суффиксы 2-типа и 1-типа, достаточно уметь сравнивать суффиксы 0-типа и 2-типа. Только что научились.

## • Псевдокод.

Пусть у нас уже есть `radixSort(a)`, возвращающий перестановку.

```

1 def getIndex(a): # новая нумерация,  $\mathcal{O}(|a| + \max_i a[i])$ 
2   p = radixSort(a)
3   cc = 0
4   ind = [0] * n
5   for i in range(n):
6     cc += (i > 0 and a[p[i]] != a[p[i-1]])
7     ind[p[i]] = cc
8   return ind
9
10 def sufArray(s): #  $0 \leq s_i \leq \frac{3}{2}n$ 
11   n = len(s)
12   if n < 3: return slowSlowSort(s)
13   s += [0, 0, 0]
14   w = getIndex( [(s[i], s[i+1], s[i+2]) for i in range(n)] )
15   index01 = range(0, n, 3) + range(1, n, 3) # с шагом 3
16   p01 = sufArray( [w[i] for i in index01] )
17   pos = [0] * n
18   for i in range(len(p01)): pos[index01[p01[i]]] = i # позиция 01-суффикса в p01
19   index2 = range(2, n, 3)
20   p2 = getIndex( [(w[i], pos[i+1]) for i in index2] )
21   def less(i, j): #  $i \bmod 3 = 0/1, j \bmod 3 = 2$ 
22     if i mod 3 == 1: return (s[i], pos[i+1]) < (s[j], pos[j+1])
23     else: return (s[i], s[i+1], pos[i+2]) < (s[j], s[j+1], pos[j+2])
24   return merge(p01 o index01, p2 o index2, less) # o - композиция: index01[p01[i]], ...

```

Для  $n \geq 3$  рекурсивный вызов делается от строго меньшей строки:

$3 \rightarrow 1+1, 4 \rightarrow 2+1, 5 \rightarrow 2+2, \dots$

Неравенством  $s_i \leq \frac{3}{2}n$  мы в явном виде в коде нигде не пользуемся.

Оно нужно, чтобы гарантировать, что `radixSort` работает за  $\mathcal{O}(n)$ .

# Лекция #15: Игры на графах

3 декабря 2020

Тема подробно раскрыта в обзоре А. С. Станкевича «Игры на графах». В частности, там описана невошедшая в лекцию теория Смита для суммы игр с циклами.

## 15.1. Основные определения

**Def 15.1.1.** *Игра на орграфе: по вершинам графа перемещается фишка, за ход игрок должен сдвинуть фишку по одному из рёбер.*

**Симметричная игра** – оба игрока могут ходить по всем рёбрам

**Несимметричная игра** – каждому игроку задано собственное множество рёбер

*Проигрывает игрок, который не может ходить.*

**Упражнение 15.1.2.** Пусть по условию игры “некоторые вершины являются выигрышными или проигрышными для некоторых игроков”. Такую игру можно вложить в определение выше.

*Замечание 15.1.3.* Чтобы не придумывать ничего отдельно для несимметричных игр, обычно просто вводят новый граф, вершины которого – пары  $\langle v, who \rangle$  (вершина и, кто ходит).

Результат симметричной игры определяется графом  $G$  и начальной вершиной  $v \in G$ .

Игру будем обозначать  $(G, v)$ , результат игры  $r(G, v)$ , или для краткости  $r(v)$ .

Классифицируем  $v \in G$ : в зависимости от  $r(G, v)$  назовём вершину  $v$  проигрышной (L), выигрышной (W) или ничейной (D).

Также введём множества:  $WIN = \{v \mid r(v) = W\}$ ,  $LOSE = \{v \mid r(v) = L\}$ ,  $DRAW = \{v \mid r(v) = D\}$ .

*Замечание 15.1.4.* В несимметричных играх, если вершина, например, проигрышна, ещё важно добавлять, какой игрок ходил первым: “проигрышная для 1-го игрока”.

**Lm 15.1.5.** Если для вершины в условии задачи не указан явно её тип, то:

$$W \Leftrightarrow \exists \text{ ход в } L, \quad L \Leftrightarrow \text{все ходы в } W$$

### 15.1.1. Решение для ациклического орграфа

Граф ациклический  $\Rightarrow$  вспомним про динамическое программирование.

$dp[v] = r(G, v)$ ; изначально “-1”, т.е. “не посчитано”.

База: пометим все вершины, информация о которых дана по условию.

Далее ленивая динамика:

```

1 int result( int v ) {
2     int &r = dp[v];
3     if ( r != -1 ) return r;
4     r = L; // например, если исходящих рёбер нет, результат уже верен
5     for (int x : edges[v])
6         if (result(x) == L) // ищем ребро в проигрышную
7             r = W; // добавь break, будь оптимальнее!
8     return r;
9 }
```

### 15.1.2. Решение для графа с циклами (ретроанализ)

Будем пользоваться 15.1.5. Цель: как только есть вершина, которая по лемме должна стать W/L, делаем её такой и делаем из этого некие выводы. Процесс можно реализовать через dfs/bfs. Мы выберем именно bfs, чтобы в будущем вычислить *длину игры*. Итак, ретроанализ:

```

1 queue <-- все вершины, которые по условию W/L.
2 while !queue.empty()
3     v = queue.pop()
4     for x in inner_edges[v]: // входящие в v рёбра
5         if lose[v]:
6             make_win(x, d[v]+1)
7         else:
8             if ++count[x] == deg[x]: // в count считается число проигрышных рёбер
9                 make_lose(x, d[v]+1)

```

Функции `make_win` и `make_lose` проверяют, что вершину помечают первый раз, если так, добавляют её в очередь. Второй параметр – обычное для `bfs` расстояние до вершины.

Все помеченные вершины по 15.1.5 помечены правильно.

Непомеченные вершины, чтобы им было не обидно, пометим D.

**Теорема 15.1.6.** Ничейные – ровно вершины с пометкой D.

*Доказательство.* Из вершины  $v$  типа D есть рёбра только в D и W (в L нет, т.к. тогда бы наш алгоритм пометил  $v$ , как W). Также из любой D есть хотя бы одно ребро в D. Мы игрок, мы находимся в D. Каков у нас выбор? Если пойдём в W, проиграем. Проигрывать не хотим  $\Rightarrow$  пойдём в D  $\Rightarrow$  вечно будем оставаться в D  $\Rightarrow$  ничья. ■

**Def 15.1.7.**  $len(G, v)$  – длина игры, сколько ходов продлится игра, если выигрывающий игрок хочет выиграть максимально быстро, а проигрывающий максимально затянуть игру.

*Замечание 15.1.8.* После ретроанализа  $d[v] = len(G, v)$ , так как ретроанализ:

1. Перебирал вершины в порядке возрастания расстояния.
2. Для выигрышной вершины брал наименьшую проигрышную.
3. Для проигрышной вершины брал наибольшую выигрышную.

Строго доказать можно по индукции. Инварианты: при обработке  $v$  все вершины со строго меньшей длиной игры уже обработаны; если посчитано  $r(v)$ , то посчитано верно.

*Замечание 15.1.9.* На практике разбиралась садистская версия той же задачи: проигрывающий хочет побыстрее выиграть и начать новую партию, а выигрывающий подольше наслаждаться превосходством (т.е. оставлять себе возможность выиграть). Описание решения: после первого ретроанализа поменять местами смысл L/W, запустить второй ретроанализ.

## 15.2. Ним и Гранди, прямая сумма

**Def 15.2.1.** Прямая сумма графов  $G_1 = \langle V_1, E_1 \rangle$  и  $G_2 = \langle V_2, E_2 \rangle$  – граф с вершинами  $\langle v_1, v_2 \rangle \mid v_1 \in V_1, v_2 \in V_2$  и рёбрами  $(a, b) \rightarrow (a, c) \mid (b, c) \in E_2$  и  $(a, b) \rightarrow (c, b) \mid (a, c) \in E_1$ .

**Def 15.2.2.** Прямая сумма игр:  $\langle G_1, v_1 \rangle \times \langle G_2, v_2 \rangle = \langle G_1 \times G_2, (v_1, v_2) \rangle$

По сути “у нас есть два графа, можно делать ход в любом одном из них”.



**Def 15.2.3.**  $tex(A) = \min x \mid x \geq 0, x \notin A$ .

**Пример 15.2.4.**  $A = \{0, 1, 7, 10\} \Rightarrow tex(A) = 2$ ;  $A = \{1, 2, 3, 4\} \Rightarrow tex(A) = 0$

**Def 15.2.5.** На ациклическом орграфе можно задать Функцию Гранди  $f[v]$ :

Пусть из  $v$  ведут рёбра в  $Out(v) = \{x_1, x_2, \dots, x_k\} \stackrel{def}{\Rightarrow} f[v] = tex\{f[x_1], \dots, f[x_k]\}$ .

**Lm 15.2.6.**  $f[v] = 0 \Leftrightarrow v \in LOSE$  (доказывается элементарной индукцией)

**Пример 15.2.7.** Игра “Ним”. На столе  $n$  камней, за ход можно брать любое положительное число камней. Заметим  $f[n] = n$ , выигрышная стратегия – взять всё.

Пока теория не выглядит полезной. Чтобы осознать полезность, рассмотрим прямые суммы тех же игр – есть  $n$  кучек спичек или  $n$  кучек камней, и брать можно, соответственно, только из одной из кучек. На вопрос “кто выиграет в таком случае” отвечают следующие теоремы:

**Теорема 15.2.8.**  $f[\langle v_1, v_2 \rangle] = f[v_1] \oplus f[v_2]$

*Доказательство.* Для доказательства воспользуемся индукцией по размеру графа.

Из вершины  $v$  в процессе игры сможем прийти только в  $C(v)$  – достижимые из  $v$  вершины.

Для любого ребра  $\langle v_1, v_2 \rangle \rightarrow \langle x_1, x_2 \rangle$  верно, что  $|C(\langle v_1, v_2 \rangle)| > |C(\langle x_1, x_2 \rangle)|$ .

База индукции: или из  $v_1$ , или из  $v_2$  нет рёбер.

Переход индукции: для всех  $\langle G_1, x_1, G_2, x_2 \rangle$  с  $|C(\langle x_1, x_2 \rangle)| < |C(\langle v_1, v_2 \rangle)|$  уже доказали

$$f[\langle x_1, x_2 \rangle] = f[x_1] \oplus f[x_2]$$

Осталось честно перечислить все рёбра из  $\langle v_1, v_2 \rangle$  и посчитать  $tex$  вершин, в которые они ведут.

$$A = \underbrace{\{f[v_1] \oplus f[x_{21}], f[v_1] \oplus f[x_{22}], \dots, f[x_{11}] \oplus f[v_2], f[x_{12}] \oplus f[v_2], \dots\}}_{\text{игрок сделал ход из } v_2}$$

Здесь  $Out(v_1) = \{x_{11}, x_{12}, \dots\}$ ,  $Out(v_2) = \{x_{21}, x_{22}, \dots\}$ . Доказываем, что  $tex A = f[v_1] \oplus f[v_2]$ .

Во-первых, поскольку  $\forall i f[x_{1i}] \neq f[v_1] \wedge \forall i f[x_{2i}] \neq f[v_2]$ , имеем  $f[v_1] \oplus f[v_2] \notin A$ .

Докажем, что все меньшие числа в  $A$  есть. Обозначим  $x = f[v_1]$ ,  $y = f[v_2]$ ,  $M = f[v_1] \oplus f[v_2]$ .

Будем пользоваться тем, что из  $v_1$  есть ходы во все числа из  $[0, x)$ , аналогично  $v_2 \rightarrow [0, y)$ .

Пусть  $k$  – старший бит  $M$  и пришёл он из  $x \Rightarrow$

ходами  $x \rightarrow [0, 2^k)$  мы получим  $2^k$  в  $x$  различных  $k$ -битных чисел, т.е. все числа из  $[0, 2^k)$ .

Чтобы получить числа  $[2^k, M)$  перейдём от задачи  $\langle x, y, M \rangle$  к  $\langle x - 2^k, y, M - 2^k \rangle$ .

Воспользуемся индукцией. База:  $M = 0$ . ■

**Следствие 15.2.9.** Для суммы большего числа игр аналогично:  $f[v_1] \oplus f[v_2] \oplus \dots \oplus f[v_k]$ .

**Замечание 15.2.10.** Рассмотрим раскраску части плоскости  $[0, +\infty) \times [0, +\infty)$ .

В клетку  $[i, j]$  ставится минимальное неотрицательное целое число, которого нет левее и ниже.

Получится ровно  $i \oplus j$ , так как мы ставим ровно  $tex$  в игре Ним на кучках  $\{i, j\}$ .

## 15.3. Вычисление функции Гранди

Ациклический граф  $\Rightarrow$  динамика. Осталось быстро научиться считать  $tex$ :

используем “обнуление” массива за  $\mathcal{O}(1)$  и получим  $\mathcal{O}(deg_v)$  на вычисление  $tex$  вершины  $v$ :



```
1 int cc, used[MAX_MEX]; // изначально нули
2
3 cc++;
4 for (int x : out_edges[v])
5     used[f[x]] = cc; // функция Гранди x уже посчитана
6 for (f[v] = 0; used[f[v]] == cc; f[v]++)
7     ;
```

Итого суммарное время работы  $\mathcal{O}(V+E)$ .

Нужно ещё оценить MAX\_MEX. Тривиальная оценка даёт  $\mathcal{O}(\max_v \deg_v)$ , можно точнее:

**Lm 15.3.1.**  $\forall G = \langle V, E \rangle, \forall v f[v] \leq \sqrt{2E}$

*Доказательство.* Пусть в графе есть вершина с функцией Гранди  $k \Rightarrow$  из неё есть рёбра в вершины с функцией Гранди  $0, 1, \dots, k-1$ . А из вершины с функцией Гранди  $k-1$  есть ещё  $k-1$  рёбер и т.д. Итого:  $k + (k-1) + (k-2) + \dots + 1 = \frac{k(k+1)}{2}$  рёбер  $\Rightarrow k(k+1) \leq 2E \Rightarrow k \leq \sqrt{2E}$ . ■

Лекция #16: (\*) Число простых на  $[1, n]$  за  $n^{2/3}$ 

9 декабря 2020

Минимальный простой делитель  $x$  обозначаем  $d[x]$ . Массив  $d$  на  $[1, m]$  мы умеем насчитывать решетом Эратосфена за  $\mathcal{O}(m)$ .  $d[1] := +\infty$ . Заодно мы нашли за  $\mathcal{O}(m)$  все простые на  $[1, m]$ .

**Def 16.0.1.**  $\pi(n)$  – количество простых чисел от 1 до  $n$

**Def 16.0.2.**  $f(n, k) = |\{x \in [1, n] \mid d[x] \geq p_k\}|$ , где  $p_k$  –  $k$ -е простое.

**Lm 16.0.3.**  $\pi(n) = \pi(\sqrt{n}) + f(n, k(\sqrt{n}))$ , где  $k(x)$  – номер первого простого, большего  $x$ .

Теперь  $\pi(\sqrt{n})$  и  $k(\sqrt{n})$  найдём за линию решетом, а  $f(n, k(\sqrt{n}))$  рекурсивно по рекурренте:

$$f(n, k) = f(n, k-1) - f(\lfloor n/p_{k-1} \rfloor, k-1)$$

Поясним формулу:  $(f(n, k-1) - f(n, k))$  – количество чисел вида  $p_{k-1} \cdot x$ , где  $d[x] \geq p_{k-1}$ .

Количество таких  $x$  на  $[1, n]$  есть  $f(\lfloor n/p_{k-1} \rfloor, k-1)$ .

База:  $f(n, 0) = n$ ,  $f(0, k) = 0$ .

• **Самое важное отсечение.**

$f(n, k)$  есть количество пар  $\langle i, d[i] \rangle$ :  $i \leq n \wedge d[i] \geq p_k$ .

Зафиксируем некое  $m$ , предподсчитаем  $d[1..m]$ , теперь  $\forall n \leq m$   $f(n, k)$  – запрос на плоскости.

Более того, мы можем вычислять  $f$  процедурой вида:

```

1 int result = 0;
2 void calc(int n, int k, int sign) {
3     if (k == 0)
4         result += sign * n;
5     else if (n <= m)
6         queries[n].push_back({k, sign});
7     else
8         calc(n, k - 1, sign), calc(n / p[k - 1], k - 1, -sign);
9 }
```

Тогда в итоге мы получим пачку из  $q$  запросов на плоскости, уже отсортированных по  $n$ . Обработаем их одним проходом сканирующей прямой с деревом Фенвика за  $\mathcal{O}((m + q) \log m)$ .

• **Оценка времени работы, выбор  $m$ .**

Если мы считаем  $\pi(n)$ , пришли рекурсией в состояние  $(x, k)$ , то  $x = \lfloor n/y \rfloor$ .

Посчитаем число состояний рекурсии  $(x, k)$ , что в  $f(x, k)$  отсечение  $x \leq m$  ещё не сработало, а в  $f(x/p_{k-1}, k-1)$  уже сработало.

1. Есть не более  $\sqrt{n}$  таких состояний с  $x = n$ .

2. Если же  $x \neq n, x = \lfloor n/y \rfloor \Rightarrow p_{k-1} \leq y \leq n/m$ , т.к. простые мы перебираем по убыванию.

Осталось посчитать “число пар  $\langle x, k \rangle$ ” = “число пар  $\langle y, p_{k-1} \rangle$ ”, их  $\mathcal{O}((n/m)^2)$ .

Вспомнив, что простых до  $t$  всего  $\Theta(t/\log t)$ , можно дать более точную оценку:  $\mathcal{O}((n/m)^2/\log \frac{n}{m})$ .

Каждая пара даст 1 запрос, увеличит время scanline на  $\log m$ .

В предположении  $m = \Theta(n^\alpha)$ ,  $\log m = \Theta(\log \frac{n}{m}) \Rightarrow$  общее время работы  $\mathcal{O}(m \log m + (n/m)^2)$ .

Асимптотический минимум достигается, как обычно при  $m \log m = (n/m)^2 \Rightarrow m \log^{1/3} m = n^{2/3}$ .

**Упражнение 16.0.4.** Итоговое время работы –  $m \log m = \Theta((n \log n)^{2/3})$ .

• **Другие оптимизации.**

Предлагают предподсчёт для малых  $k$ :  $k \leq 8$  на практике,  $k = \Theta(\log n / \log \log n)$  в теории. Для  $n < p_{k-1}$  можно отвечать не за  $\log$ , а за  $\mathcal{O}(1)$ .

# Лекция #17: Хеширование

12 декабря 2020

## 17.1. Универсальное семейство хеш функций

[wiki] [итмо-конспект] [Carter, Wegman'1977]

**Def 17.1.1.** *Хеш-функция.*

Сжимающее отображение  $h : U \rightarrow M$ ,  $|U| > |M|$ ,  $|M| = m$ .

**Def 17.1.2.** *Универсальная система хеш-функций (1-я версия определения).*

Множество хеш-функций  $\mathcal{H}$  – универсальная система, если

$$\forall x, y \ x \neq y: \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

**Def 17.1.3.** *Универсальная система хеш-функций (2-я версия определения  $\equiv$  1-й).*

$$\forall x, y \ x \neq y: \sum_{h \in \mathcal{H}} [h(x) = h(y)] \leq \frac{|\mathcal{H}|}{m}$$

**Теорема 17.1.4.**  $\mathcal{H}_{p,m} = \{(a, b) : x \rightarrow ((ax + b) \bmod p) \bmod m\}$ ,  $a \in [1, p)$ ,  $b \in [0, p)$ .

Утверждение:  $\mathcal{H}_{p,m}$  универсально для  $|U| = [0, p)$  и  $M = [0, m)$  ( $m \leq p$ ).

*Доказательство.* Зафиксируем пару  $x, y$ . Проверим для неё определение универсальности.

$$f(x) = ax + b \bmod p, f(y) = ay + b \bmod p$$

$[f(x) = f(y) \Leftrightarrow a(x-y) \equiv 0 \bmod p] \Rightarrow (a \neq 0, x \neq y)$  по модулю  $p$  коллизий нет и у нас есть биекция  $\langle a, b : a \neq 0 \rangle \leftrightarrow \langle f(x), f(y) : f(x) \neq f(y) \rangle$ .

Осталось понять, для сколько хеш-функций  $f(x) \equiv f(y) \bmod m$ ?

$$\sum_{(a,b), a \neq 0} [f(x) \equiv f(y) \bmod m] \stackrel{\text{биекция}}{=} \sum_{f(x) \neq f(y)} [f(x) \equiv f(y) \bmod m] \stackrel{(*)}{\leq} \frac{p(p-1)}{m} = \frac{|\mathcal{H}|}{m}$$

(\*) Зафиксируем  $f(x)$ , будем перебирать  $f(y) = f(x)+1, f(x)+2, \dots$ . В каждом блоке из  $m$  вариантов, только последний даёт  $f(x) \equiv f(y) \bmod m \Rightarrow$  таких  $f(y)$  ровно  $\lfloor \frac{p-1}{m} \rfloor \leq \frac{p-1}{m}$ . ■

## 17.2. Оценки для хеш-таблицы с закрытой адресацией

[wiki] [wiki-probability] [2-choice-hashing]

Пусть у нас есть универсальное семейство хеш-функций  $\mathcal{H}$ .

Хеш-таблица с закрытой адресацией (на списках) – вероятностный алгоритм.

Вся вероятностная часть заключается в том, что мы выбираем случайную  $h \in \mathcal{H}$ .

Операции **Find(x)**, **Del(x)** работают за длину списка.

Оценим для них матожидание времени работы.

$$E[\text{time}(\text{find}(z))] = E[\text{длины списка}] = \sum_{x \in \text{table}} \Pr[h(x) = h(z)] = 1 + (n-1) \cdot \frac{1}{m} = \mathcal{O}(1 + \frac{n}{m}), \text{ где}$$

$n$  – количество элементов в таблице, а  $m$  – размер таблицы (и диапазон хеш-функции).

*Замечание 17.2.1.* Мы оценили именно матожидание времени работы. Матожидание средней длины списка всегда  $\frac{m}{n}$ , даже если все элементы всегда класть в один список.

Теперь несколько утверждений без доказательства.

*Утверждение 17.2.2.*  $E[\text{максимальной длины списка}] = \mathcal{O}(\log n)$

*Утверждение 17.2.3.* 2-choice hashing. Модифицируем хеш-таблицу: будем использовать две хеш-функции  $h_1, h_2$  и при добавлении элемента будем выбирать из списков  $a[h_1(x)], a[h_2(x)]$  список меньшей длины. Тогда  $E[\text{максимальной длины списка}] = \Theta(\log \log n)$ .

### 17.3. Фильтр Блюма

[\[wiki\]](#) [\[итмо-конспект\]](#)

#### Прелюдия.

Мы хотим вероятностную структуру данных, которая умеет делать всего две операции – добавлять  $x$ , и проверять, добавлен ли уже в структуру  $x$ .

Главная фишка нашей структуры по сравнению с более простыми аналогами (массив, хеш-таблица) — очень мало памяти,  $\mathcal{O}(n)$  бит.

#### Собственно структура.

Хотим хранить не более  $n$   $x$ -ов. Для этого у нас есть  $m$  бит и  $k$  хеш-функций  $h_1, \dots, h_k$ .

```
1 bitset<m> a; // m нулей
2 Add(x): a[h1] = a[h2] = ... = a[hk] = 1;
3 Find(x): return a[h1] & a[h2] & ... & a[hk];
```

Собственно алгоритм окончен, осталось разобраться, с какой вероятностью это работает.

Во-первых, если Find вернул 0, элемент точно ещё не был добавлен.

Оценим ошибку Find-а, который вернул 1. При оценке для простоты предположим, что все  $nk$  значений, которые вернули  $k$  хеш-функций данных  $n$  элементов, равномерно распределены.

$$Pr[\text{в } i\text{-й ячейке } 0] = (1 - 1/m)^{kn} \approx \exp\left(\frac{-kn}{m}\right) \Rightarrow Pr[\text{ложного срабатывания Find}] = \left(1 - \exp\left(\frac{-kn}{m}\right)\right)^k$$

Посчитаем, какое при фиксированных  $\langle n, m \rangle$  оптимально выбрать  $k$ ?

Дифференцируем по  $k$ , решаем  $Pr'(k) = 0$ , получаем  $k = (\ln 2) \cdot \frac{m}{n}$  и  $Pr[\text{error}] = 2^{-k} = 0.6185^{m/n}$ .

Например, если нам дали лишь  $5n$  бит на всё про всё, мы достигли вероятности ошибки  $\approx 9\%$ .

### 17.4. (\*) Совершенное хеширование

[\[wiki\]](#) [\[итмо-конспект\]](#) [\[cs.cmu.edu\]](#) [\[practice:bbhash\]](#) [\[Fredman'1984\]](#)

Вспомним, как мы ищем младший бит 64-битного целого  $x$ .

```
1 uint64_t i2(uint64_t x) { return x & (~x + 1); }
```

Таким образом мы получили младший бит  $i$  в форме  $2^i$ .

Осталось сделать последнее действие,  $2^i \rightarrow i$ .

```
1 int table[67];
2 void init() { forn(i, 64) table[2i % 67] = i; }
3 int get(uint64_t i2) { return table[i2 % 67]; }
```

Здесь все остатки по модулю 67 различны  $\Rightarrow$

мы получили детерминированный алгоритм поиска номера младшего бита за  $\mathcal{O}(1)$ .

$x \rightarrow x \bmod 67$  – пример совершенной хеш-функции для множества

фиксированных ключей  $\{x_0 = 2^0, x_1 = 2^1, \dots, x_{63} = 2^{63}\}$ .

**Def 17.4.1.** *Совершенная хеш-функция – любая инъективная функция.*

То есть, хеш-функция, у которой по определению не возникает коллизий.

Чтобы построить такую хеш-функцию, нам, конечно, нужно заранее знать набор ключей  $x_1, x_2, \dots, x_k$  (см. пример с младшим битом).

### Задача построения совершенной хеш-функции.

Дан набор ключей  $x_1, x_2, \dots, x_n$ , найти такую функцию

$h: X = \{x_1, x_2, \dots, x_n\} \rightarrow [0, m) \cap \mathbb{Z}$ , что все  $h(x_i)$  различны и функция вычислима за  $\mathcal{O}(1)$ .

$m$  будем называть размером хеш-функции ( $m$  в итоге станет размером массива хеш-таблицы).

### 17.4.1. Одноуровневая схема

Возьмём  $m = n^2$  и любое универсальное семейство хеш-функций  $\mathcal{H}: X \rightarrow [0, m)$ .

Например,  $\mathcal{H} = \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$ ,  $m \leq p$ ,  $p$  простое.

$E(\text{количества коллизий}) = \frac{1}{2}n(n-1) \cdot \frac{1}{m} < \frac{1}{2} \Rightarrow$  с вероятностью хотя бы  $\frac{1}{2}$  коллизий нет  $\Rightarrow$

Алгоритм: берём случайную  $h \in \mathcal{H}$ , пока не повезёт.  $E(\text{времени работы}) = \mathcal{O}(n)$ , размер  $n^2$ .

### 17.4.2. Двухуровневая схема

Возьмём  $m = n$  и любое универсальное семейство хеш-функций  $\mathcal{H}: X \rightarrow [0, m)$ .

Выберем случайную  $h \in \mathcal{H}$  и посмотрим на списки  $A_y = \{x \mid h(x) = y\}$ .

Для каждого  $A_y$  построим одноуровневую совершенную хеш-функцию  $f_y$ .

Оценим суммарный диапазон всех внутренних уровней:

$$E\left(\sum_y |A_y|^2\right) = \sum_{ij} Pr[h(x_i) = h(x_j)] = n \cdot 1 + n(n-1) \cdot \frac{1}{n} = 2n-1$$

Осталось внутренние уровни выписать в один массив.

Для этого берём префиксные суммы:  $\text{pref\_sum}[y+1] = \text{pref\_sum}[y] + |A_y|^2$ .

**Итого, наша хеш-функция:** `int hash(x) { y = h(x); return pref_sum[y] + f[y](x); }`

$E(\text{времени работы}) = \mathcal{O}(n)$ ,  $E(\text{размера}) = 2n-1$ .

### 17.4.3. Графовый подход

Начнём с забавного факта

*Утверждение 17.4.2.* Рассмотрим случайный неорграф из  $3n$  вершин и  $n$  рёбер.

$Pr[\text{отсутствия циклов}] \geq \frac{1}{2}$ .

Возьмём  $m = 3n$  и любое универсальное семейство хеш-функций  $\mathcal{H}: X \rightarrow [0, m)$ .

Выберем  $h_1, h_2 \in \mathcal{H}$ . Каждому  $x_i$  сопоставим ребро  $\langle h_1(x_i), h_2(x_i) \rangle$ .

Если граф не ациклический, выберем другие хеш-функции, повторим. Пусть ациклический.

Тогда запишем значения  $f$  в вершинах и определим  $\text{hash}(x_i) = (f[h_1(x_i)] + f[h_2(x_i)]) \bmod n$ .

Как записать значения  $f$ , чтобы было верно  $\text{hash}(x_i) = i$ ? Граф – лес. В каждом дереве в корне пишем ноль (любое число), остальные числа расставит `dfs` по дереву.

## 17.5. Хеширование кукушки

[wiki] [Pagh,Rodler'2001]

За сколько работает хеш-таблица на списках?

Add за  $\mathcal{O}(1)$  в худшем, Find и Del за  $\mathcal{O}(1)$  в среднем.

Сейчас мы придумаем, как сделать наоборот:

Add за  $\mathcal{O}(1)$  в среднем, Find и Del за  $\mathcal{O}(1)$  в худшем.

Заведём массив  $a$  размера  $m = 3n$  и две хеш-функции:  $h_1, h_2 \in \mathcal{H}_m$ .

$\forall$  элемент  $x$  всегда живёт в одной из двух ячеек —  $h_1(x)$  или  $h_2(x)$ .

Find и Add, очевидно, обращаются не более чем к двум ячейкам —  $h_1(x)$  и  $h_2(x)$ .

Add сперва ищет свободную среди  $h_1(x), h_2(x)$ . Если заняты обе, начнём выталкивать элементы:

```
1 AddIfBothAreOccupied(x)
2   i = h1(x)
3   while (a[i] != -1)
4       y = a[h1(x)], a[h1(x)] = x, x = y
5       i = h1[y] + h2[y] - i // вторая из двух ячеек, где может жить y
```

Почему и за сколько работает Add?

Представим себе граф с рёбрами между  $h_1(x)$  и  $h_2(x)$ . Как мы уже знаем, он с большой вероятностью ацикличен (значит Add хотя бы не циклится). Теперь осталось дожидаться, пока голос Белы Боллобаша (автор книги Random Graphs) не нашепчет нам недостающую мудрость — глубина случайного дерева  $\mathcal{O}(\log n)$ .