Дмитрий Павлов

Лекции по информатике

Санкт-Петербург, 2003 год

Содержание

Введение

Этот конспект был создан по материалам лекций, которые автор проводил в рамках подготовки школьников Санкт-Петербурга ко всероссийской командной олимпиаде по программированию и всероссийской олимпиаде по информатике. В нём я сделал попытку изложить те темы информатики, которые имеют применение в олимпиадах и которые недостаточно хорошо освещены в доступной литературе по информатике; конспект содержит далеко не полный материал по каждой из тем. Многие идеи из глав про геометрию были высказаны Николаем Дуровым на зимних студенческих сборах по программированию.

Предполагается, что читатель уже читал следующие книги:

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (The MIT Press, 1990).

Русский перевод К. Белова, Ю. Боравлёва, Д. Ботина, В. Горелика, Д. Дерягина, Ю. Калнишкана, А. Катановой, С. Львовского, А. Ромащенко, К. Сонина, К. Трушкина, М. Ушакова, А. Шеня, В. Шувалова, и М. Юдашкина, под редакцией А. Акимова, М. Вьюгина, Д. Дерягина, А. Евфимьевского, Ю. Калнишкана, А. Ромащенко, А. Чернова, М. Ушакова, А. Шеня, и В. Ященко, *Алгоритмы: построение и анализ* (Москва: МЦНМО, 2000), 955 с.

Donald E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms* (Reading, Mass.: Addison-Wesley, 1968), xxii + 634 pp. Second printing, revised, July 1969.

Second edition, completely revised, December 1973. Second printing, revised, February 1975.

Third edition, completely revised, May 1997, xx + 650 pp.

Русский перевод Г. П. Бабенко и Ю. М. Баяковского, под редакцией К. И. Бабенко и В. С. Штаркмана, *Искусство программирования для ЭВМ*, том 1: *Основные алгоритмы* (Москва: Мир, 1976), 735 с.

Русский перевод С. Г. Тригуба, Ю. Г. Гордиенко, и И. В. Красикова, под редакицей С. Н. Тригуба, *Искусство программирования*, том 1: *Основные алгоритмы* (Москва: Издательский дом Вильямс, 2000), 720 с.

Donald E. Knuth, *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms* (Reading, Mass.: Addison-Wesley, 1969), xii + 624 pp. Second printing, revised, November 1971.

Second edition, completely revised, January 1981, xiv + 689 pp.

Third edition, completely revised, September 1997, xiv + 762 pp.

Русский перевод Г. П. Бабенко, Э. Г. Белаги, и Л. В. Майорова, под редакцией К. И. Бабенко, *Искусство программирования для ЭВМ*, том 2: *Получисленные алгоритмы* (Москва: Мир, 1977), 724 с.

Русский перевод Л. Ф. Козаченко, В. Т. Тертышного, и И. В. Красикова, под редакицей С. Н. Тригуба, *Искусство программирования*, том 2: *Получисленные алгоритмы* (Москва: Издательский дом Вильямс, 2000), 832 с.

Donald E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching* (Reading, Mass.: Addison-Wesley, 1973), xii + 722 pp. + foldout illustration. Second printing, revised, March 1975, xii + 725 pp.

Second edition, completely revised, February 1998, xiv + 780 pp.

Русский перевод Н. И. Вьюковой, В. А. Галатенко, и А. Б. Ходулёва, под редакцией Ю. М. Баяковского и В. С. Штаркмана, *Искусство программирования для ЭВМ*, том 3: *Сортировка и поиск* (Москва: Мир, 1978), 844 с.

Русский перевод В. Т. Тертышного и И. В. Красикова, под редакцей С. Н. Тригуба, *Искусство программирования*, том 3: *Сортировка и поиск* (Москва: Издательский дом Вильямс, 2000), 832 с.

Обозначения

Математические обозначения

$\mathbf{N}$	множество всех неотрицательных целых чисел;
$\mathbf{N}^*$	множество всех положительных целых чисел;
$[a\,..\,b]$	множество всех целых чисел от a до b включительно;
$[a\,..\,b)$	аналогично, b исключается;
$(a\,..\,b]$	аналогично, a исключается;
$(a\,..\,b)$	аналогично, a и b исключаются;
$\lfloor a \rfloor$	округление вниз числа a;
$\lceil a \rceil$	округление вверх числа a;
$f^{[k]}$	композиционная степень функции f;
$a \ll b$	сдвиг a влево на b бит;
$a \gg b$	сдвиг a вправо на b бит;
$a \wedge b$	побитовое И чисел a и b;
$a \vee b$	побитовое ИЛИ чисел a и b;
$a \oplus b$	побитовое исключающее ИЛИ чисел a и b;
$a \leftarrow b$	присвоить переменной a значение b
$a \leftrightarrow b$	обменять значения переменных a и b местами.

Псевдокод

Фрагменты кода будут приводиться на псевдокоде — упрощённом варианте паскаля и C. Указание вложенности команд обозначается отступами. Выражение a/b обозначает $\lfloor a/b \rfloor$. Для упрощения написания будут использоваться следующие сокращения:

сокращение	длинный вариант
$a{+}{+}$	$a \leftarrow a + 1$
$a{-}{-}$	$a \leftarrow a - 1$
$a \mathrel{+}= b$	$a \leftarrow a + b$
$a \mathrel{-}= b$	$a \leftarrow a - b$
$a \mathrel{*}= b$	$a \leftarrow a * b$
$a \mathrel{/}= b$	$a \leftarrow a/b$

Математическое введение

Асимптотика

В этом разделе все числа будем считать вещественными.

Определим проколотую окрестность радиуса ε следующим образом:
$S(x+0,\varepsilon) = (x, x+\varepsilon)$,
$S(x-0,\varepsilon) = (x-\varepsilon, x)$,
$S(x,\varepsilon) = (x-\varepsilon, x) \cup (x, x+\varepsilon)$,
$S(+\infty,\varepsilon) = (\varepsilon, +\infty)$,
$S(-\infty,\varepsilon) = (-\infty, -\varepsilon)$,
$S(\infty,\varepsilon) = (-\infty, -\varepsilon) \cup (\varepsilon, +\infty)$.

Первый аргумент функции S будем называть точкой.

При помощи понятия проколотой окрестности можно определить понятие предела функции в точке a следующим образом: $\lim_{x\to a} f(x) = c$, где c — точка, в том и только в том случае, если для любого положительного ε найдется положительное δ, такое, что если $x \in S(a,\delta)$, то $f(x) \in S(c,\varepsilon) \cup \{c\}$.

Для произвольной точки a будем писать $f(x) = O(g(x))$ при $x \to a$, если существует некоторое c и некоторое положительное ε, такие, что $|f(x)| \le c \cdot |g(x)|$ как только $x \in S(a,\varepsilon)$.

Исходя из этого определения, введём также следующие два обозначения:
$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$ (при $x \to a$);
$f(x) = \Theta(g(x)) \Leftrightarrow f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$ (при $x \to a$).

Если существует функция $h(x)$, такая, что $\lim_{x\to a} h(x) = 0$, и при этом $f(x) = h(x) \cdot g(x)$ в некоторой ε-окрестности точки a, то будем обозначать это как $f(x) = o(g(x))$ при $x \to a$. С помощью этого определения определим также $f(x) = \omega(g(x)) \Leftrightarrow g(x) = o(f(x))$ (при $x \to a$).

Используются также обозначения вида $f(x)+o(g(x))$, $f(x)/\Theta(g(x))$, и тому подобные. Здесь под $O(f(x))$, $\Omega(f(x))$, $\Theta(f(x))$, $o(f(x))$, $\omega(f(x))$ подразумеваются множества функций $g(x)$, таких, что $g(x) = O(f(x))$, …, $g(x) = \omega(f(x))$. В этих обозначениях записи вида $A+B$, A^B, и т. п., означают множества всех функций $f(x)$, которые представимы в виде $g(x)+h(x)$, $g(x)^{h(x)}$, и т. д., где $g(x) \in A$, $h(x) \in B$. Наконец, особо стоит отметить, что знак равенства на самом деле обозначает знак включения ($\subset$) при указанных соглашениях. Например, запись $x^2 + o(\sqrt{x}) = O(x^{1/3}) \cdot x^2 + O(\sqrt{x})$ при $x \to +\infty$ означает, что любую функцию, имеющую вид $x^2+f(x)$, где $f(x) = o(\sqrt{x})$ при $x \to +\infty$, можно представить в виде $f(x) \cdot x^2 + g(x)$, где $f(x) = O(x^{1/3})$ и $g(x) = O(\sqrt{x})$ при $x \to +\infty$.

Поскольку в информатике, как правило, интересен только рост функции на бесконечности, то пояснения вида $x \to +\infty$ обычно опускают. Также часто используют запись вида $O(A)$ вместо $O(|A|)$, где A — множество.

Кроме того, иногда используют следующие обозначения:
$f(x) \prec g(x) \Leftrightarrow f(x) = o(g(x))$,
$f(x) \succ g(x) \Leftrightarrow f(x) = \omega(g(x))$,
$f(x) \asymp g(x) \Leftrightarrow f(x) = \Theta(g(x))$,
$f(x) \sim g(x) \Leftrightarrow f(x) = g(x) + o(g(x))$.

Отношения и соответствия

Бинарные отношения

Если $R \subset A \times A$, то говорят, что R является бинарным отношением на множестве A. Принадлежность (x,y) отношению R обозначается xRy. Будем считать, что у обозначения xRy более высокий приоритет, чем у всех логических операций. Разность $(A \times A) \setminus R$ обозначается $\overline{R}$ и называется дополнением отношения R.

Отношение R
- рефлексивно, если $\forall x\,(xRx)$;
- антирефлексивно, если $\forall x\,(\neg xRx)$;
- симметрично, если $\forall x \forall y\,(xRy \Rightarrow yRx)$;
- асимметрично, если $\forall x \forall y\,(xRy \Rightarrow \neg yRx)$;

- антисимметрично, если $\forall x \forall y \, (xRy \wedge yRx \Rightarrow x = y)$;
- транзитивно, если $\forall x \forall y \forall z \, (xRy \wedge yRz \Rightarrow xRz)$;
- связно, если $\forall x \forall y \, (xRy \vee yRx \vee x = y)$.

Упражнение. *Докажите, что асимметричное отношение антирефлексивно и антисимметрично, а транзитивное и антирефлексивное отношение асимметрично.*

Рефлексивное, симметричное и транзитивное отношение называется отношением эквивалентности. Транзитивное и антирефлексивное отношение называется отношением строгого порядка. Транзитивное, рефлексивное и антисимметричное отношение называется отношением нестрогого порядка. Отношением порядка называется отношение строгого или нестрогого порядка. Связное отношение порядка называется отношением линейного порядка.

Далее считается, что операция $=$ является отношением эквивалентности, операция $<$ является отношением строгого линейного порядка, операция $\leq$ является отношением нестрогого линейного порядка и операции $\neq$, $\geq$ и $>$ являются дополнениями соответствующих отношений.

Соответствия

Если $F \subset X \times Y$, то пишут $F \colon X \to Y$ и говорят, что F является соответствием из X в Y. Принадлежность (x, y) соответствию F обозначается $y = F(x)$. Далее считаем, что $x \in X$, $y \in Y$.

Соответствие F

- всюду определено, если $\forall x \exists y \, (y = F(x))$;
- однозначно, если $\forall x \forall y_1 \forall y_2 \, (y_1 = F(x) \wedge y_2 = F(x) \Rightarrow y_1 = y_2)$;
- сюръективно, если $\forall y \exists x \, (y = F(x))$;
- инъективно, если $\forall y \forall x_1 \forall x_2 \, (y = F(x_1) \wedge y = F(x_2) \Rightarrow x_1 = x_2)$.

Для каждого соответствия F из X в Y определено обратное соответствие F^{-1} из Y в X, такое, что $F^{-1} = \{(y, x) \mid (x, y) \in F\}$.

Если $A \subset X$, то множество $F(A) = \{y \mid \exists x \, (x \in A \wedge y = F(x))\}$ называется образом множества A. Если $B \subset Y$, то множество $F^{-1}(B)$ называется прообразом множества B.

Однозначное всюду определенное соответствие называется функцией. Для функции F можно говорить о ее значении в точке x и писать $F(x)$, подразумевая при этом тот единственный y, для которого $y = F(x)$. Множество всех функций из A в B обозначается B^A.

Сюръективная и инъективная функция называется биекцией. Биекция множества целых чисел от 1 до n на себя называется перестановкой n чисел.

Функция из множества $S \times S$ во множество S называется бинарной операцией на множестве S. Если F — операция, то вместо $F(x, y)$ пишут xFy.

Операция $*$ на множестве S

- ассоциативна, если $\forall a \forall b \forall c \, ((a * b) * c = a * (b * c))$;
- коммутативна, если $\forall a \forall b \, (a * b = b * a)$;
- идемпотентна, если $\forall a \, (a * a = a)$.

Если операция $*$ ассоциативна, то мы вправе писать произведения вида $a_1 * a_2 * \cdots * a_{n-1} * a_n$, не указывая при этом порядок расстановки скобок.

Алгебраические структуры

Пусть на множестве S задана операция $*$. Тогда пара $\langle S, * \rangle$ называется группоидом. Группоид, у которого операция $*$ ассоциативна, называется полугруппой. Очевидно, что в полугруппе можно определить операцию возведения элемента в любую положительную целую степень ($a^n = a$ для $n = 1$, $a^n = a * a^{n-1} = a^{n-1} * a$ для $n > 1$), поскольку операция $*$ ассоциативна, то определение корректно. Если во множестве S есть элемент e, такой, что $a \in S \Rightarrow a * e = e * a = a$, то элемент e называется единицей.

Упражнение. *Докажите, что в группоиде может быть не больше одной единицы. Докажите, что в любой группоид можно добавить новый элемент, являющийся единицей, не изменив при этом значения операции на первоначальных элементах группоида.*

Полугруппа, в которой есть единица, называется моноидом. Понятно, что любую полугруппу можно преобразовать в моноид, добавив в нее единицу. Поэтому везде, где используется моноид,

можно использовать и полугруппу. В моноиде элементы можно возводить в любую неотрицательную целую степень ($a^0 = e$).

Пример. *Полугруппа* $\langle \mathbf{Z}, \min \rangle$ *не является моноидом. Однако, если мы добавим элемент* $+\infty$*, такой, что* $x \in \mathbf{Z} \Rightarrow \min(+\infty, x) = \min(x, +\infty) = x$*, то* $\langle \mathbf{Z} \cup \{+\infty\}, \min \rangle$ *будет моноидом.*

Если в моноиде верно равенство $a * b = e$, то a называется левым обратным элементом к b, а b называется правым обратным к элементу a.

Упражнение. *Докажите, что если в моноиде у элемента есть левый и правый обратные элементы, то все его левые и правые обратные элементы совпадают (в этом случае они называются просто обратными элементами).*

Моноид, в котором для каждого элемента есть обратный, называется группой. В группе обратный к a элемент обозначается a^{-1}. Для групп очевидным образом можно определить операцию возведения в любую целую степень ($a^{-n} = (a^{-1})^n$). Группа, у которой операция $*$ коммутативна, называется абелевой. Для абелевых групп используется аддитивная символика, то есть бинарная операция обозначается $+$, обратный элемент обозначается $-a$, единица обозначается 0 и возведение в n-ую степень обозначается na.

Тройка $\langle S, +, \cdot \rangle$ называется полем, если $\langle S, + \rangle$ является абелевой группой, $\langle S, \cdot \rangle$ является коммутативным моноидом, $\langle S \setminus \{0\}, \cdot \rangle$ является группой, $0 \neq 1$, и для любых a, b, и c будет верно, что $a(b + c) = ab + ac$.

Если в поле существует такое целое положительное n, что $n \cdot 1 = 0$, то минимальное из таких n называется характеристикой поля. Если такого n нет, то характеристика поля считается равным нулю.

Некоторые аспекты аналитической геометрии на плоскости

Точки и вектора

В этой главе мы отождествим точки и вектора на плоскости с комплексными числами. Как известно, существует два основных способа записи комплексных чисел. Первый из них использует для представления числа z его действительную и мнимую части, которые обозначаются как $\Re z$ и $\Im z$, в такой форме z записывается как $z = (x, y) = x+iy$, где $x = \Re z$ и $y = \Im z$. В геометрии вещественная и мнимая части называются абсциссой и ординатой соответственно, а такой способ представления называется декартовой системой координат. Второй способ использует для представления числа z его модуль $|z|$ и аргумент $\arg z$, в такой форме z записывается как $z = re^{i\phi}$, где $r = |z|$ и $\phi = \arg z$. В геометрии модуль и аргумент называются полярным радиусом и полярным углом, а сам способ представления называется полярной системой координат.

Следует отметить, что в случае $z = 0$ аргумент оставляется неопределённым. Если же $z \neq 0$, то аргумент не может быть определён однозначно, ибо путём прибавления чисел, кратных 2π мы можем получать другие значение аргумента. Поэтому часто пишут функцию аргумента с большой буквы, подразумевая при этом всё множество значений аргмента:

$$\operatorname{Arg} z = \arg z + 2\pi k,$$

где k принимает всевозможные целые значения. Под записью $\arg z$ мы будем понимать одно фиксированное значение аргумента, способ определения которого может варьироваться.

Введём теперь специальное обозначение для тригонометрической формы записи числа: будем писать, что $z = \langle r, \phi\rangle$, если $z = re^{i\phi}$.

Пусть теперь нам дано число z, такое, что $z = (x, y) = \langle r, \phi\rangle$. Мы хотим перейти от одного представления к другому. В одну сторону это делается тривиальным образом: $x = r\cos\phi$, $y = r\sin\phi$ (в случае, когда $r = 0$, в качестве ϕ подходит любое значение). В другую сторону переход выполняется немного сложнее. Очевидно, что $r = \sqrt{x^2 + y^2}$. Для вычисления аргумента от нас требуется решить систему из двух уравнений:

$$\cos\phi = x/r;$$

$$\sin\phi = y/r.$$

Эту систему можно решать многими способами:

$$\phi = \begin{cases} \arccos(x/r), & \text{если } y \geq 0, \\ -\arccos(x/r), & \text{если } y < 0; \end{cases}$$

$$\phi = \begin{cases} \arcsin(y/r), & \text{если } x \geq 0, \\ \pi - \arcsin(y/r), & \text{если } x < 0; \end{cases}$$

$$\phi = \begin{cases} \arctan(y/x), & \text{если } x > 0, \\ \pi + \arctan(y/x), & \text{если } x < 0, \\ \pi/2, & \text{если } x = 0 \text{ и } y > 0, \\ -\pi/2, & \text{если } x = 0 \text{ и } y < 0. \end{cases}$$

На практике преобразование из декартовой системы координат в полярную можно выполнить гораздо проще.*

---

* Действительно, команда сопроцессора `FPATAN` вычисляет полярный угол (результат принадлежит отрезку $[-\pi, \pi]$); для вычисления полярного угла в языке C с использованием типа **long double** можно написать $\operatorname{atan2l}(y, x)$, эта функция объявлена в файле `math.h`; для компилятора Borland Delphi можно использовать аналогичную функцию arctan2, объявленную в модуле Math (эта функция делает не что иное, как вызывает команду сопроцессора `FPATAN`, несмотря на те неверные утверждения, которые написаны в документации компилятора и в комментариях к реализации этой функции). Для вычисления полярного радиуса можно использовать функцию hypotl, объявленную в файле `math.h` и функцию hypot, объявленную в модуле Math. Напоминаю, что в языке C можно избавиться от суффикса l, включив файл `tgmath.h`.

Преимущество использования комплексных чисел состоит в том, что многие геометрические операции имеют простую запись в виде операций над комплексными числами. Так, например, если $z = (x, y) = \langle r, \phi \rangle$, то $-z = (-x, -y) = \langle r, \phi + \pi \rangle$, $\bar{z} = (x, -y) = \langle r, -\phi \rangle$; таким образом, операция изменения знака является центральной симметрией относительно начала координат, а операция сопряжения является осевой симметрией относительно оси абсцисс.

Далее, если $z_1 = (x_1, y_1) = \langle r_1, \phi_1 \rangle$ и $z_2 = (x_2, y_2) = \langle r_2, \phi_2 \rangle$, то

$$z_1 + z_2 = (r_1 \cos \phi_1 + r_2 \cos \phi_2, r_1 \sin \phi_1 + r_2 \sin \phi_2);$$

$$|z_1 + z_2| = r_1^2 + r_2^2 + 2 r_1 r_2 \cos(\phi_1 - \phi_2).$$

Таким образом, мы получили теорему косинусов как следствие формулы для суммы комплексных чисел.

Произведение z_1 и z_2 есть

$$z_1 z_2 = \langle r_1 r_2, \phi_2 + \phi_1 \rangle = (x_1 x_2 - y_1 y_2, x_1 y_2 + x_2 y_1).$$

Таким образом, мы получили формулу для поворота данной точки на данный угол относительно начала координат.

Если теперь рассмотреть произведение $\bar{z}_1$ и z_2, то мы получим следующее:

$$\bar{z}_1 z_2 = \langle r_1 r_2, \phi_2 - \phi_1 \rangle = (x_1 x_2 + y_1 y_2, x_1 y_2 - x_2 y_1).$$

Абсцисса полученной точки называется скалярным произведением z_1 и z_2, а ордината называется псевдоскалярным произведением или векторным произведением на плоскости. Угол $\phi = \phi_2 - \phi_1$ есть не что иное, как ориентированный угол между векторами z_1 и z_2, то есть такой угол, что при повороте вектора z_1 против часовой стрелки на этот угол он станет сонаправлен с вектором z_2. Из приведённой выше формулы следует, что ориентированный угол можно вычислить как полярный угол точки $\bar{z}_1 z_2$. В частности, получаем, что $r_1 r_2 \cos \phi = x_1 x_2 + y_1 y_2$ и $r_1 r_2 \sin \phi = x_1 y_2 - x_2 y_1$. Из второй формулы сразу следует, что векторное произведение является ориентированной площадью параллелограмма, натянутого на вектора z_1 и z_2. Эта площадь положительна, если кратчайший поворот от z_1 к z_2 происходит против часовой стрелки, отрицательна, если кратчайший поворот происходит по часовой стрелке, и равна нулю, если вектора z_1 и z_2 коллинеарны.

Прямые и окружности

Как известно из линейной алгебры, единственными нетривиальными объектами на плоскости, обладающими в некотором смысле полностью линейной структурой, являются линейные многообразия размерности 0 и 1, которые называются соответственно точками и прямыми.

В этом параграфе мы будем задавать точки двумя координатами (x, y). Для задания прямых есть несколько способов. Как следует из определения линейного многообразия, прямая есть множество точек, получаемых прибавлением ко всем точкам линейного пространства размерности 1 некоторого фиксированного вектора (x_0, y_0). Линейное пространство размерности 1 можно в свою очередь задать базисом, состоящим из одного элемента, являющегося точкой (a, b). Из определения базиса следует, что $(a, b) \neq (0, 0)$.

Таким образом, мы получили параметрическое задание прямой: все точки на заданной прямой можно представить в виде

$$(x_0 + at, y_0 + bt) \quad (1),$$

где параметр t принимает произвольные вещественные значения. Это также можно записать в виде системы двух уравнений:

$$x - x_0 = at; \quad (2)$$

$$y - y_0 = bt. \quad (3)$$

Выполним следующее равносильное преобразование этой системы: умножим первое уравнение на b и вычтем из него второе уравнение, умноженное на a, это будет первым уравнением; умножим

первое уравнение на a и прибавим к нему второе уравнение, умноженное на b, это будет вторым уравнением. Таким образом, мы получим следующие два уравнения:

$$b(x-x_0)-a(y-y_0)=0; \tag{4}$$

$$a(x-x_0)+b(y-y_0)=(a^2+b^2)t. \tag{5}$$

В силу того, что детерминант матрицы

$$\begin{pmatrix} b & -a \\ a & b \end{pmatrix}$$

равен a^2+b^2, что отлично от нуля, система уравнений (2) и (3) равносильна системе уравнений (4) и (5). Заметим также, что уравнение (5) равносильно следующему:

$$t=\frac{a(x-x_0)+b(y-y_0)}{a^2+b^2}. \tag{6}$$

Таким образом, мы фактически исключили t из наших уравнений, и получили, что прямая описывается единственным уравнением

$$ay_0-bx_0+bx-ay=0,$$

которое является частным случаем уравнения вида $a_0+a_1x+a_2y=0$. Изучением уравнений такого вида мы сейчас и займёмся.

Прежде всего заметим, что случай $a_1=a_2=0$ является бессмысленным, так как уравнение $a_0=0$ имеет своим множеством решений либо пустое множество, либо всю плоскость. Поэтому в дальнейшем мы предполагаем, что хотя бы одно из чисел a_1 и a_2 отлично от нуля.

Заметим, что можно классифицировать точки в зависимости от знака, который они дают при подстановке в формулу $a_0+a_1x+a_2y$. На прямой лежат все те и только те точки, которые обращают это выражение в 0. Рассмотрим теперь множество всех точек, которые дают положительное значение этого выражения. Очевидно, что это множество является выпуклым, так как если $a_0+a_1x_1+a_2y_1>0$ и $a_0+a_1x_2+a_2y_2>0$, и при этом даны числа λ_1 и λ_2, такие, что $\lambda_1\geq 0$, $\lambda_2\geq 0$, и $\lambda_1+\lambda_2=1$, то

$$a_0+a_1(\lambda_1x_1+\lambda_2x_2)+a_2(\lambda_1y_1+\lambda_2y_2)=\lambda_1(a_0+a_1x_1+a_2y_1)+\lambda_2(a_0+a_1x_2+a_2y_2)>0,$$

так как оба слагаемых не меньше нуля, и хотя бы одно из них больше нуля. Таким же образом доказывается выпуклость множества всех точек, для которых $a_0+a_1x+a_2y<0$. Тем самым мы доказали, что две полуплоскости, на которые плоскость разбивает прямая $a_0+a_1x+a_2y=0$, определяются неравенствами $a_0+a_1x+a_2y>0$ и $a_0+a_1x+a_2y<0$.

Мы уже можем преобразовывать параметрическое представление в неявное, посмотрим теперь, как можно решать обратную задачу. Как известно из линейной алгебры, в случае, когда $(a_1,a_2)\neq(0,0)$, уравнение $a_0+a_1x+a_2y=0$ представляет собой систему линейных уравнений, ранг которой равен 1. Таким образом, все решения этого уравнения могут быть получены прибавлением к одному фиксированному решению этого уравнения (x_0,y_0) всевозможных решений соответствующей однородной системы $a_1x+a_2y=0$.

В силу того, что существует бесконечное число точек (x_0,y_0), удовлетворяющих поставленным в задаче условиям, необходимо зафиксировать одну из них. Для этого мы возьмём в качестве искомой точку, лежащую на минимальном расстоянии от начала координат. Фактически, нам требуется минимизировать значение функции x^2+y^2 при условии, что $a_0+a_1x+a_2y=0$. Применяя метод неопределённых множителей Лагранжа, получаем систему уравнений:

$$a_0+a_1x+a_2y=0;$$

$$2x=\lambda a_1;$$

$$2y=\lambda a_2.$$

Отсюда получаем, что $x = \lambda a_1/2$, $y = \lambda a_2/2$, подставляя эти выражения в первое уравнение, мы получаем, что

$$a_0 + a_1^2\lambda/2 + a_2^2\lambda/2 = 0,$$

откуда получаем, что

$$\lambda = \frac{-2a_0}{a_1^2 + a_2^2}.$$

Таким образом,

$$x = \frac{-a_0 a_1}{a_1^2 + a_2^2};$$

$$y = \frac{-a_0 a_2}{a_1^2 + a_2^2}.$$

В силу того, что все функции в условии задачи являются непрерывно дифференцируемыми на всей плоскости и ранг матрицы $(a_1 \quad a_2)$ равен 1, данная точка будет единственной точкой на всей плоскости, которая может дать условный экстремум. В любом случае нетрудно проверить, что эта точка лежит на данной прямой, а независимое доказательство того, что она является ближайшей к началу координат будет дано ниже.

Нам осталось выразить в общем виде решение однородной системы, состоящей из одного уравнения $a_1 x + a_2 y = 0$. Нетрудно увидеть, что

$$(x, y) = \frac{t}{a_1^2 + a_2^2}(a_2, -a_1).$$

Сложив два получившихся выражения, мы получим общий вид всех точек, лежащих на данной прямой:

$$\left(\frac{-a_0 a_1 + a_2 t}{a_1^2 + a_2^2}, \frac{-a_0 a_2 - a_1 t}{a_1^2 + a_2^2}\right). \tag{7}$$

Рассмотрим теперь следующую задачу: пусть на прямой, заданной уравнением вида (1) или (7), дана точка (x, y). Требуется определить значение переменной t. Нетрудно увидеть, что в обоих случаях мы получаем систему из двух уравнений вида

$$c_1 t = d_1,$$

$$c_2 t = d_2,$$

где хотя бы одно из чисел c_1 и c_2 отлично от нуля. Один из способов решения этой системы состоит в том, чтобы взять уравнение с ненулевым коэффициентом при t и решить его.

Однако этот метод обладает тем недостатком, что небольшие погрешности при округлении могут привести к полностью неверному результату. Более того, так как все вычисления проводятся приближённо, данная система очень часто может быть несовместной в силу небольших погрешностей округления, однако наш метод решения должен уметь справляться с такими вещами. Поставим себе задачу найти значение t, минимизирующее сумму квадратов отклонений левых частей от правых, то есть величину $(d_1 - c_1 t)^2 + (d_2 - c_2 t)^2$. В силу того, что коэффициент при t^2 в этом выражении положителен, искомое значение t будет равно

$$t = \frac{c_1 d_1 + c_2 d_2}{c_1^2 + c_2^2}.$$

Метод решения подобных систем называется методом наименьших квадратов. В применении к уравнению (1) этот метод даёт формулу (6), а если применить его к уравнению (7), то мы получим следующую простую формулу:

$$t = a_2 x - a_1 y.$$

Эта формула оказывается даже проще той, которую мы получили бы, использовав метод, приведённый в начале. Важное свойство двух полученных формул заключается также в том, что они дают одинаковый результат для всех точек, лежащих на прямой, перпендикулярной данной.

Именно поэтому небольшие погрешности при округлении координат дают небольшие погрешности в значениях t.

Рассмотрим теперь снова формулу (7) и найдём выражение для квадрата расстояния от начала координат до точки с данным значением t. Подставляя в формулу $x^2 + y^2$ соответствующие значения, получаем:

$$x^2+y^2 = \frac{a_0^2a_1^2 - 2a_0a_1a_2t + a_2^2t^2 + a_0^2a_2^2 + 2a_0a_1a_2t + a_1^2t^2}{(a_1^2+a_2^2)^2} = \frac{(a_1^2+a_2^2)a_0^2 + (a_1^2+a_2^2)t^2}{(a_1^2+a_2^2)^2} = \frac{a_0^2+t^2}{a_1^2+a_2^2}. \quad (8)$$

Эта формула имеет много важных применений. В частности, сразу видно, что единственной точкой, в которой достигается минимум расстояния до начала координат будет точка $t = 0$, то есть как раз та точка, которая была найдена при помощи метода неопределённых множителей.

Далее, расстояние до ближайшей точки будет равно $a_0^2/(a_1^2 + a_2^2)$. Нетрудно также найти с помощью этой формулы расстояние от произвольной точки (x_0, y_0) до данной прямой. Для этого достаточно выполнить замену координат $x' = x - x_0$ и $y' = y - y_0$, откуда $x = x' + x_0$ и $y = y' + y_0$, подставляя эти выражения в уравнение $a_0 + a_1x + a_2y = 0$, получаем уравнение прямой в новых координатах: $(a_0 + a_1x_0 + a_2y_0) + a_1x' + a_2y' = 0$, таким образом мы получаем хорошо известную формулу для расстояния от точки (x_0, y_0) до прямой $a_0 + a_1x + a_2y = 0$:

$$d^2 = \frac{(a_0 + a_1x_0 + a_2y_0)^2}{a_1^2 + a_2^2}.$$

Рассмотрим теперь задачу пересечения прямой $a_0 + a_1x + a_2y = 0$ и окружности $x^2 + y^2 = r^2$. (Случай окружности с центром в произвольной точке приводится к данному при помощи тривиального преобразования системы координат так, как было показано выше.)

Для решения этой системы уравнений воспользуемся формулой (8) и получим следующее:

$$\frac{t^2 + a_0^2}{a_1^2 + a_2^2} = r^2,$$

что равносильно тому, что

$$t^2 = r^2(a_1^2 + a_2^2) - a_0^2 = d.$$

Дальнейший ход решения зависит от знака d. Если $d < 0$, то очевидно, что решений нет (это соответствует случаю, когда прямая и окружность не пересекаются). Если $d = 0$, то в этом случае есть ровно одна точка пересечения, соответствующая значению $t = 0$ (в данном случае прямая и окружность касаются). Если $d > 0$, то есть ровно две точки пересечения, соответствующие двум решениям уравнения $t^2 = d$.

Пусть теперь мы хотим пересечь окружности $x^2 + y^2 = r^2$ и $(x - x_0)^2 + (y - y_0)^2 = R^2$. (Здесь применимо аналогичное замечание про перенос координат.) Для того, чтобы решить эту систему из двух уравнений, первое уравнение мы оставим неизменным, а второе из них заменим на разность первого и второго, в результате чего получится следующее уравнение:

$$R^2 - r^2 - x_0^2 - y_0^2 + 2x_0x + 2y_0y = 0. \quad (9)$$

Разберём теперь два случая. Если $x_0 = y_0 = 0$, то уравнение (9) превращается в уравнение $R^2 - r^2 = 0$, которое не зависит от x и y, и либо истинно при всех x и y, либо ложно. Тем самым случай совпадающих окружностей и случай различных концентрических окружностей разобран. Если же $(x_0, y_0) \neq (0, 0)$, то уравнение (9) задаёт прямую, которую можно пересечь с окружностью $x^2 + y^2 = r^2$ описанным выше методом.

Основы проективной геометрии на плоскости

Однородные наборы

Зафиксируем число n. Будем рассматривать наборы из n чисел, где под числом подразумевается элемент некоторого поля. Набор, все компоненты которого равны 0, будем называть нулевым. Будем называть набор однородным, если он отличен от нулевого. Назовем однородные наборы u и v эквивалентными и будем писать $u \sim v$, если для любых i и j, таких, что $0 \leq i < n$ и $0 \leq j < n$ будет верно $u_i v_j = u_j v_i$. Нетрудно заметить, что $u \sim v$ в том и только в том случае, когда $u_i v_j = u_j v_i$ для всех i и j, таких, что $0 \leq i < j < n$.

Теорема. *Введённое отношение является отношением эквивалентности.*

Очевидно, что $u \sim u$ и $u \sim v \Rightarrow v \sim u$, поэтому отношение $\sim$ рефлексивно и симметрично. Осталось доказать его транзитивность

Пусть $u \sim v$ и $v \sim w$. Докажем, что $u \sim w$. Нам требуется проверить выполнение равенства $u_i w_j = u_j w_i$ для всех i и j. Если $i = j$, то равенство верно. Пусть теперь $i \neq j$. Так как $u \sim v$ и $v \sim w$, то $u_i v_j = u_j v_i$ и $v_i w_j = v_j w_i$. Если $v_i \neq 0$ и $v_j \neq 0$ то перемножив два предыдущих равенства получим $u_i v_i v_j w_j = u_j v_i v_j w_i$, так как $v_i v_j \neq 0$, то мы можем разделить на $v_i v_j$ и получить $u_i w_j = u_j w_i$. Если $v_i = 0$ и $v_j \neq 0$, то исходя из равенств $u_i v_j = u_j v_i = 0$ и $v_j \neq 0$ получаем $u_i = 0$. Аналогично $w_i = 0$. Отсюда $u_i w_j = u_j w_i = 0$. Случай $v_i \neq 0$ и $v_j = 0$ разбирается аналогично. Пусть теперь $v_i = v_j = 0$. По определению однородного набора мы можем взять такое k, что $v_k \neq 0$. Так как $u_i v_k = u_k v_i = 0$, то $u_i = 0$. Аналогично получаем, что $u_j = 0$, а также $w_i = w_j = 0$. Равенство $u_i w_j = u_j w_i$ становится очевидным.

Таким образом, отношение $\sim$ является отношением эквивалентности. Соответственно, все однородные наборы разбиваются на классы эквивалентности. Следует отметить, что объявление нулевого набора запрещённым было вполне естественно, так как нулевой набор был бы эквивалентен любому, и у нас бы нарушилось свойство транзитивности, при доказательство которого мы как раз и использовали то, что однородный набор не является нулевым. Класс эквивалентности однородного набора $w = (w_0, \ldots, w_{n-1})$ будем обозначать $(w_0 : \cdots : w_{n-1})$.

Отметим теперь следующее свойство: если $u \sim v$, то $u_i = 0 \Leftrightarrow v_i = 0$. В самом деле, пусть $u_i = 0$. Возьмём такое j, что $u_j \neq 0$. Теперь $u_i v_j = u_j v_i = 0$, отсюда получаем, что $v_i = 0$. Таким образом, либо $u_i = v_i = 0$, либо $u_i \neq 0$ и $v_i \neq 0$. Из равенства $u_i v_j = u_j v_i$ следует, что отношение u_i / v_i для всех v_i, не равных нулю, будет одним и тем же ненулевым числом. Таким образом, мы получаем, что эквивалентные наборы пропорциональны, а именно, существует такое ненулевое число, что если мы домножим на него первый набор, то получим второй. Это число определяется единственным образом.

На самом деле, определение отношения эквивалентности как свойства наборов быть пропорциональными является даже более общим подходом, чем тот, который был рассмотрен в начале. Мы можем определить для любого векторного пространства V соответствующее ему проективное пространство $P(V)$ следующим образом: $P(V) = (V \setminus \{0\})/\sim$, где $\sim$ является отношением эквивалентности на $V \setminus \{0\}$, при этом $u \sim v$ в том и только в том случае, если существует такое ненулевое λ, что $u = \lambda v$.

Осталось заметить, что понятие однородных и эквивалентных наборов очевидным образом распространяется на матрицы.

Проективная плоскость

В этом и последующих параграфах везде считаем, что $n = 3$, если не указано обратное.

Введем теперь следующие определения:

- Точка — класс эквивалентности однородных наборов.
- Прямая — класс эквивалентности однородных наборов.
- Точка x инцидентна прямой y, если $x_0 y_0 + x_1 y_1 + x_2 y_2 = 0$.
- Точка x двойственна прямой y, если $x \sim y$.

Следует отметить, что когда мы рассматриваем набор координат точки или прямой, мы берём одного из представителей класса эквивалентности однородных наборов, соответствующих данной

точке или прямой, при этом определения, использующие эти координаты, не должны зависеть от того, какой набор мы выбрали. Определения инцидентности и двойственности очевидно обладают этим свойством. Заметим, что определение двойственности можно также понимать как равенство классов эквивалентности.

Сопоставим каждой точке x вектор-столбец, состоящий из её координат:

$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}.$$

Аналогично, сопоставим прямой y вектор-строку

$$y = \begin{pmatrix} y_0 & y_1 & y_2 \end{pmatrix}.$$

Условие инцидентности прямой x точке y теперь записывается как $xy = 0$; условие двойственности записывается как $x = y^T$, таким образом, точка x переходит в прямую x^T и прямая y переходит в точку y^T. (Мы отождествляем матрицы размера 1×1 и элементы поля.)

Из определений следует, что при замене всех объектов на двойственные отношение инцидентности сохраняется. В частности, в любом утверждении о точках и прямых можно заменить все прямые на точки и все точки на прямые и снова получить верное утверждение. Поэтому когда мы формулируем какой-то результат относительно точек и прямых, мы автоматически подразумеваем ему двойственный.

Построим две различные точки, инцидентные данной прямой x. Если $x_0 = 0$, то точки $(1 : 0 : 0)$ и $(0 : -x_2 : x_1)$ будут искомыми. Если $x_0 \neq 0$, то точки $(-x_1 : x_0 : 0)$ и $(-x_2 : 0 : x_0)$ будут искомыми.

Найдём общий вид точек, инцидентных данной прямой x, если нам даны две различные точки a и b, инцидентные этой прямой. Каждая такая точка y является решением уравнения $xy = x_0y_0 + x_1y_1 + x_2y_2 = 0$, и наоборот, каждое решение этого уравнения, за исключением нулевого, даёт точку, инцидентную прямой x. Поскольку набор x является однородным, одна из его компонент отлична от 0, и, следовательно, ранг матрицы системы равен 1, а размерность пространства её решений будет равна 2. Поскольку у нас уже есть два линейно независимых решения a и b, то общий вид решения будет такой:

$$\alpha a + \beta b = (\alpha a_0 + \beta b_0, \alpha a_1 + \beta b_1, \alpha a_2 + \beta b_2).$$

Здесь при $\alpha = \beta = 0$ мы получаем нулевой набор, поэтому α и β не могут одновременно обращаться в 0. Однако, любые другие пары α и β дают корректные точки, инцидентные прямой x. Различные пары значений α и β дают различные наборы, однако точки, соответствующие этим наборам, могут совпадать. Точки, задаваемые парами (α_1, β_1) и (α_2, β_2) будут совпадать в том и только в том случае, если $(\alpha_1, \beta_1) \sim (\alpha_2, \beta_2)$. Таким образом, каждая точка на прямой однозначно задается классом эквивалентности $(\alpha : \beta)$, и наоборот, каждый такой класс эквивалентности задаёт точку на прямой. Таким образом, мы установили биективное соответствие между точками, инцидентными данной прямой и однородными наборами из двух компонент.

Если нам даны две различные точки a и b, то существует ровно одна прямая, которая инцидентна обеим этим точкам. В самом деле, прямая, инцидентная двум точкам, должна удовлетворять системе из двух линейных уравнений, а поскольку точки a и b различны, то ранг матрицы системы будет равен 2 и размерность пространства решений будет равна 1, очевидно, что этому пространству соответствует единственная прямая.

Нетрудно увидеть, что эта прямая имеет вид $a \times b = (a_1b_2 - a_2b_1 : a_2b_0 - a_0b_2 : a_0b_1 - a_1b_0)$. Если точки a и b совпадают, то эта формула дает запрещенный набор $(0 : 0 : 0)$, что вполне естественно; если же точки различны, то и набор будет ненулевым.

Отсюда можно сразу получить условие того, что три точки лежат на одной прямой (коллинеарны), в самом деле, для этого необходимо и достаточно, чтобы детерминант матрицы, состоящей из координат этих точек, был равен нулю. Свойство, двойственное коллинеарности, называется конкуррентностью.

Проективная замена координат

Назовём проективным преобразованием класс эквивалентности невырожденных матриц размера 3×3. Аналогично точкам и прямым, когда мы говорим о матрице проективного преобразования, мы говорим о конкретном представителе этого класса, при этом определения, использующие матрицу преобразования, не должны зависеть от выбора этого представителя.

Мы будем считать, что проективное преобразование задаёт биективное соответствие на множестве всех однородных наборов; мы будем говорить о старых и новых координатах объекта, подразумевая под старыми координатами координаты объекта до преобразования, и под новыми координатами координаты объекта после преобразования.

Действие проективного преобразования на точки и прямые определяется следующим образом: точке x в новых координатах соответствует точка Vx в старых координатах, а прямой y в новых координатах соответствует прямая yV^{-1} в старых координатах. Здесь V — матрица проективного преобразования.

Выведем формулы, выражающие новые координаты объектов через их старые координаты. Для координат точки получаем, что новые координаты точки имеют вид $V^{-1}x$, где x — её старые координаты, а для новых координат прямой получаем аналогичное выражение yV, где y — старые координаты прямой. Нетрудно заметить, что эти две формулы означают, что преобразование, имеющее матрицу V^{-1}, является обратным к исходному.

Для того, чтобы наше определения были корректны, необходимо и достаточно показать, что определения точки, прямой, и отношения инцидентности не изменяются при проективном преобразовании.

Поскольку мы показали, что обратное преобразование также является проективным преобразованием, достаточно доказывать все свойства в одну сторону.

Для определений точки и прямой это выражается в доказательстве того, что эквивалентные однородные наборы переходят в эквивалентные однородные наборы. (Из этого утверждения сразу следует, что классы эквивалентности переходят в классы эквивалентности, поскольку образ класса эквивалентности в старых координатах будет являться подмножеством класса эквивалентности в новых координатах, и наоборот.) Следующие два вывода показывают, что однородные наборы переходят в однородные:

$$Vx = 0 \Leftrightarrow x = V^{-1}0 = 0,$$

$$yV^{-1} = 0 \Leftrightarrow y = V0 = 0.$$

Следующие два вывода показывают, что эквивалентные наборы переходят в эквивалентные:

$$x = \lambda y \Rightarrow Vx = V(\lambda y) = \lambda(Vy),$$

$$x = \lambda y \Rightarrow xV^{-1} = (\lambda y)V^{-1} = \lambda(yV^{-1}).$$

Докажем теперь, что инцидентные точка и прямая переходят в инцидентные точку и прямую:

$$0 = yx = yV^{-1}Vx = (yV^{-1})(Vx) = 0.$$

Тем самым мы доказали, что определения базовых свойств инвариантны относительно проективного преобразования.

То, что для преобразования точек используется матрица V, а для преобразования прямых используется матрица V^{-1}, является прямым следствием требования инвариантности свойства инцидентности, на самом деле, из формулы преобразования координат точки $x' = Vx$ и требования инвариантности инцидентности сразу следует формула преобразования координат прямой: $y' = yV^{-1}$, для её вывода достаточно подставить в уравнение инцидентности выражение старых координат точки через новые.

Следует заметить, что свойство двойственности не всегда является инвариантным относительно проективного преобразования. Однако мы можем использовать понятие проективного преобразования, чтобы обобщить понятие двойственности. Сопоставим точке x двойственную прямую $(Vx)^T$, а прямой y двойственную точку $(yV^{-1})^T$. Таким образом, мы сначала применяем

преобразование V, переходя от новых координат к старым, а затем используем обычную двойственность. Нетрудно заметить, что обычная двойственность является частным случаем данного соответствия между точками и прямыми, если V является тождественным преобразованием. Очевидно, что определённое таким образом соответствие сохраняет отношение инцидентности, поскольку $(Vx)^T(yV^{-1})^T = (yV^{-1}Vx)^T = (yx)^T = yx$. Осталось выяснить, когда данное соответствие является двойственностью, то есть при двукратном применении переводит объект сам в себя. Для этого необходимо и достаточно выполнение следующих двух тождеств при любых x и y соответственно:

$$((Vx)^TV^{-1})^T = (V^TV^{-1})^Tx = x;$$

$$(V(yV^{-1})^T)^T = yV^{-1}V^T = y.$$

Нетрудно видеть, что это верно в том и только в том случае, когда обе матрицы $(V^TV^{-1})^T$ и $V^{-1}V^T$ являются единичными, что равносильно тому, что $V = V^T$, то есть матрица преобразования должна быть симметричной.

На самом деле, можно было задать только правило преобразования точек, а затем вывести из требований двойственности и инвариантности свойств инцидентности правило преобразования прямых.

Конические сечения

Назовём коническим сечением класс эквивалентности однородных симметричных матриц 3×3. Будем говорить, что точка x инцидентна коническому сечению W, если $x^TWx = 0$. Будем говорить, что прямая y инцидентна коническому сечению W, если $yW^{-1}y^T = 0$. Очевидно, что эти определения не зависят от того, какие из эквивалентных векторов и матриц мы выбираем.

При проективном преобразовании с матрицей преобразования V новая матрица конического сечения имеет вид V^TWV, где W — старая матрица.

Для доказательства корректности этого определения заметим сначала, что матрица конического сечения переходит в матрицу конического сечения, то есть остаётся невырожденной и симметричной при проективном преобразовании. В самом деле, $\det(V^TWV) = \det V^T \det W \det V \neq 0$, так как $\det V^T = \det V \neq 0$ и $\det W \neq 0$ по определению V и W. Кроме того, очевидно, что эквивалентные матрицы снова переходят в эквивалентные матрицы. (Свойство однородности следует из невырожденности.)

Осталось доказать, что свойства инцидентности точки или прямой коническому сечению остаются инвариантными при проективном преобразовании. Это делается непосредственной подстановкой выражений новых координат прямых и точек через старые в уравнения инцидентности:

$$x'^TW'x' = (V^{-1}x)^T(V^TWV)(V^{-1}x) = x^TWx;$$

$$y'W'^{-1}y'^T = (yV)(V^TWV)^{-1}(yV)^T = yW^{-1}y^T.$$

Здесь применимо аналогичное замечание о том, что формула преобразования матрицы является прямым следствием требования инвариантности свойства инцидентности; из формулы преобразования координат точки $x' = Vx$ и требования инвариантности инцидентности сразу следует формула преобразования матрицы; для её вывода достаточно подставить в уравнение инцидентности точки коническому сечению выражение старых координат точки через новые.

Нетрудно заметить, что уравнение инцидентности точки коническому сечению $x^TWx = 0$ можно проинтерпретировать как инцидентность точки x прямой x^TW, а это не что иное, как определённая раньше двойственная прямая относительно преобразования с матрицей W, так как $x^TW = (W^Tx)^T = (Wx)^T$, поскольку $W = W^T$. Аналогичное уравнение для прямой $yW^{-1}y^T = 0$ интерпретируется как инцидентность прямой y двойственной ей точки $W^{-1}y^T = (yW^{-1})^T$. Таким образом, понятие двойственности относительно проективного преобразования оказывается очень плодотворным в применении к коническим сечениям (на самом деле, поскольку двойственность определена только относительно преобразований с симметричными матрицами, любому такому преобразованию соответствует коническое сечение). Двойственность определённая таким образом

называется полярной двойственностью, прямая, двойственная точке, называется её полярой, точка, двойственная прямой, называется её полюсом. Таким образом, точка инцидентна коническому сечению в том и только в том случае, когда она инцидентна свой поляре относительно этого конического сечения, аналогично, прямая инцидентна коническому сечению в том и только в том случае, если она инцидентна своему полюсу относительно этого конического сечения.

Пересечение прямой и конического сечения

Пусть задано коническое сечение W и прямая x. Требуется найти все точки, которые инцидентны как x, так и W.

Так как все точки, инцидентные данной прямой, можно записать в виде $\alpha a+\beta b$, то мы можем подставить это выражение в уравнение инцидентности точки коническому сечению: $x^TWx = (\alpha a+\beta b)^TW(\alpha a+\beta b) = (\alpha a^T+\beta b^T)W(\alpha a+\beta b) = a^TWa\alpha^2+(a^TWb+b^TWa)\alpha\beta+b^TWb\beta^2 = A\alpha^2+2B\alpha\beta+C\beta^2 = 0$, где $A = a^TWa$, $B = a^TWb = b^TWa$, $C = b^TWb$. Это условие можно переписать также как равенство нулю квадратичной формы:

$$\gamma^T\begin{pmatrix}A & B\\ B & C\end{pmatrix}\gamma = 0, \quad \text{где } \gamma = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}.$$

Итак, нам осталось теперь решить вышеприведенное однородное квадратное уравнение. Как уже упоминалось, нас интересуют только классы эквивалентности $(\alpha:\beta)$; понятно, что домножив решение уравнения на константу мы снова получим решение этого уравнения. При $A = B = C = 0$ решениями будут являться все пары $(\alpha:\beta)$. Пусть теперь хотя бы одно из чисел A, B, и C не равно нулю. Если $A = 0$, то уравнение принимает вид $2Bxy+Cy^2 = 0$. Очевидно, что при $y = 0$ подходит любой x, отсюда получаем решение $(1:0)$. Если теперь предположить, что $y \neq 0$, то мы получаем уравнение $2Bx = -Cy$, у которого решением будет $(-C:2B)$ (очевидно, что этот набор будет корректным). Заметим что эти решения совпадают в том и только в том случае, если $B = 0$. Итак, случай $A = 0$ выглядит следующим образом: если $B = 0$, то будет одно решение $(1:0)$; а если $B \neq 0$, то будет два решения $(1:0)$ и $(-C:2B)$. Если же $A \neq 0$, то уравнению можно придать вид $x^2+2(B/A)xy+(C/A)y^2 = 0$; заметим, что $y = 0$ влечёт за собой $x = 0$, поэтому мы можем обозначить $z = x/y$ и получить следующее уравнение: $z^2+2(B/A)z+C/A = 0$; его можно переписать как $(z+B/A)^2 = (B^2-AC)/A^2$, или, что то же самое, $(Az+B)^2 = B^2-AC$, теперь понятно, что множество его решений имеет вид $z = (-B+z_0)/A$, где z_0 пробегает все решения уравнения $z^2 = B^2-AC$. Таким образом, получаем, что в случае $A \neq 0$ общий вид решений будет следующим: $(-B+z:A)$, где z является решением уравнения $z^2 = B^2-AC$. Поскольку извлечение квадратного корня является процедурой, зависящей от конкретного поля, мы не можем произвести дальнейшие упрощения.

Теорема. *Если любая точка, инцидентная данной прямой, инцидентна также и некоторому коническому сечению, то характеристика поля, над которым мы работаем, равна 2. Обратно, для любого поля характеристики 2 существует такое коническое сечение и такая прямая, что все точки, инцидентные этой прямой, инцидентны коническому сечению.*

Для доказательства первой части выполним проективную замену координат, и переведём нашу прямую в прямую $(1:0:0)$, а матрицу нашего конического сечения переведём в матрицу W. Поскольку точка $(0:1:0)$ инцидентна прямой, то она также должна быть инцидентна и коническому сечению, отсюда $W_{1,1} = 0$. Аналогично получаем, что $W_{2,2} = 0$. Подставляя теперь точку $(0:1:1)$ в уравнение инцидентности, получаем, что $W_{1,2}+W_{2,1} = W_{1,2}+W_{1,2} = 0$, если характеристика поля отлична от 2, то данное уравнение равносильно тому, что $W_{1,2} = W_{2,1} = 0$, а отсюда получаем, что $\det W = 0$, что невозможно по определению конического сечения.

Для доказательства второй части достаточно рассмотреть коническое сечение со следующей матрицей:

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0\end{pmatrix}.$$

Очевидно, что точка инцидентна такому коническому сечению в том и только в том случае, если $x_0^2+x_1x_2+x_2x_1 = x_0^2 = 0$. Таким образом, получаем, что все точки, инцидентные прямой $(1:0:0)$ инцидентны также коническому сечению.

Касательные прямые к коническому сечению

Теорема. *Если характеристика поля, над которым мы работаем, отлична от 2, то прямая y инцидентна коническому сечению W в том и только в том случае, если у неё с ним ровно одна точка пересечения.*

Рассмотрим сначала случай, когда матрица W диагональная. Пусть a и b — две различные точки, инцидентные прямой y. Тогда условия единственности точки пересечения выглядят так: $B^2 - AC = 0$ и A и C не равны одновременно нулю, где $A = a^T W a$, $B = a^T W b$, $C = b^T W b$, обозначая за λ_i значение $W_{i,i}$, получаем, что $A = \sum \lambda_i a_i^2$, $B = \sum \lambda_i a_i b_i$, $C = \sum \lambda_i b_i^2$. Здесь и далее суммирование ведётся по всем i, j, и k, равным 0, 1, и 2. Заметим сначала, что на любой прямой есть хотя бы одна точка, не инцидентная коническому сечению, поэтому можно выбрать a и b так, чтобы A и C не были одновременно равны нулю. Подставляя выражения для A, B, и C в условие $B^2 - AC = 0$, получаем:

$$\left(\sum_i \lambda_i a_i^2\right)\left(\sum_i \lambda_i b_i^2\right) - \left(\sum_i \lambda_i a_i b_i\right)^2 = \sum_{i,j} (\lambda_i a_i^2 \lambda_j b_j^2 - \lambda_i a_i b_i \lambda_j a_j b_j) = \sum_{i,j} \lambda_i \lambda_j a_i b_j (a_i b_j - a_j b_i);$$

поскольку при $i = j$ слагаемое суммирование становится равным 0, а при обмене значений i и j множитель $a_i b_j$ заменяется на $-a_j b_i$, то получаем, что искомая сумма равна

$$\sum_{i<j} \lambda_i \lambda_j (a_i b_j - a_j b_i)^2 = \lambda_0 \lambda_1 \lambda_2 \sum_k \lambda_k^{-1} (a_{k'} b_{k''} - a_{k''} b_{k'})^2,$$

где $k' = (k+1) \bmod 3$, $k'' = (k+2) \bmod 3$; поскольку $\det W \neq 0$, то все λ_i отличны от 0. Поскольку выражения $a_{k'} b_{k''} - a_{k''} b_{k'}$ являются коэффициентами искомой прямой y (которая инцидентна точкам a и b), то получаем, что исходная сумма пропорциональна (с ненулевым коэффициентом) выражению

$$\sum_k \lambda_k^{-1} y_k^2 = y W^{-1} y^T.$$

Таким образом, теорема доказана для случая диагональной матрицы.

Как известно, любую квадратичную форму можно привести к сумме квадратов соответствующей заменой координат. Пусть V — матрица преобразования, при котором матрица конического сечения W имеет диагональный вид, то есть матрица $V^T W V$ будет диагональной. Всё, что нам осталось показать — это то, что оба свойства, равносильность которых мы доказываем, остаются инвариантными относительно замены координат. Для второго свойства это было показано ранее, а для первого свойства это утверждение очевидно, поскольку свойство прямой иметь одну общую точку с коническим сечением выражается непосредственно через свойство инцидентности точки коническому сечению, которое инвариантно относительно замены координат.

Общие касательные к двум коническим сечениям

Рассмотрим обычную двойственность, при которой точке x сопоставляется прямая x^T и прямой y сопоставляется точка y^T. Как известно, двойственность сохраняет отношение инцидентности между точками и прямыми. Однако очевидно, что коническое сечение также должно преобразовываться под действием двойственности для того, чтобы свойства инцидентности точек и прямых коническому сечению оставались инвариантными относительно двойственности. Для этого двойственным коническому сечению с матрицей W будем считать коническое сечение с матрицей W^{-1}. Нетрудно заметить, что все три свойства инцидентности действительно будут инвариантны по отношению к такой двойственности.

Если при полярной двойственности неизменными оставались координаты конического сечения, что позволяет, например удобным образом сводить задачу поиска прямых, инцидентных данной точке и данному коническому сечению к задаче пересечения поляры этой точки и конического сечения, то при такой двойственности неизменными остаются координаты прямых и точек, что позволяет, например, сводить задачу поиска прямых, инцидентных двум заданным коническим сечениям к задаче поиска точек, инцидентных двум двойственным коническим сечениям.

Сводка формул проективной геометрии на плоскости

- Основные формулы

условие эквивалентности	$u_i v_j = u_j v_i,\ 0 \le i < j < 3$
уравнение инцидентности	$yx = 0$
условие двойственности	$x = y^T$
две различные точки на прямой x	
$x_0 = 0$:	$(1:0:0)$ и $(0:-x_2:x_1)$
$x_0 \ne 0$:	$(-x_1:x_0:0)$ и $(-x_2:0:x_0)$
общий вид точек на прямой	$\alpha a + \beta b$
прямая, проходящая через две точки	$a \times b = (a_1 b_2 - a_2 b_1 : a_2 b_0 - a_0 b_2 : a_0 b_1 - a_1 b_0)$

- Проективные преобразования

объект	старые координаты	новые координаты
точка	x	$V^{-1}x$
	Vx	x
прямая	y	yV
	yV^{-1}	y
коническое сечение	W	$V^T W V$
	${V^{-1}}^T W V^{-1}$	W

- Конические сечения

уравнение инцидентности для точки	$x^T W x = 0$
уравнение инцидентности для прямой	$yW^{-1}y^T = 0$
поляра точки	$x^T W$
полюс прямой	$W^{-1}y^T$

Соотношение евклидовой и проективной плоскости

Сопоставим точке $(x_0 : x_1 : x_2)$ точку евклидовой плоскости с координатами $(x_1/x_0, x_2/x_0)$. Обратное сопоставление осуществим следующим образом: точке (x, y) евклидовой плоскости сопоставим точку $(1 : x : y)$. Указанное сопоставление не является взаимно однозначным, поскольку точкам с $x_0 = 0$ не соответствует ни одна точка евклидовой плоскости. Такие точки называются бесконечно удалёнными. Нетрудно теперь заметить, что прямой a будет соответствовать прямая $a_0 + a_1 x + a_2 y = 0$ на евклидовой плоскости, и наоборот, прямой $a_0 + a_1 x + a_2 y = 0$ на евклидовой плоскости будет соответствовать прямая $(a_0 : a_1 : a_2)$. При указанном соответствии прямой $(1 : 0 : 0)$ не соответствует ни одна прямая на евклидовой плоскости. Нетрудно заметить, что эта прямая состоит в точности из бесконечно удалённых точек, поэтому она называется бесконечно удалённой.

При помощи указанного соответствия можно перенести на проективную плоскость все метрические функции евклидовой плоскости (скалярное, псевдоскалярное и комплексное произведения, расстояние и длину, а также углы и понятия параллельности, перпендикулярности, и полуплоскости). Следует отметить, что все эти понятия будут иметь смысл лишь для конечных точек и прямых.

Практические замечания по написанию программ

Поскольку в настоящее время на олимпиадах используются 32-разрядные компиляторы, то следует учитывать их специфику. В частности, наиболее быстрыми типами данных являются 32-разрядные типы. Не следует забывать о существовании 64-битных типов (int64 в Delphi, long long в C, __int64 в Visual C++).

В качестве вещественного типа данных рекомендуется использовать 10-байтовый формат расширенной двойной точности (double-extended в терминологии IEEE 754, extended в Delphi, long double в C). При наличии особых требований к скорости можно использовать 8-байтовый формат двойной точности (double), либо заставить компилятор выравнивать 10-байтовые значения по границе 16 байт.

Во всех компиляторах теперь нет 64K-барьера на размер массива. Однако размер стека во многих компиляторах изначально фиксирован и возможно, что его потребуется увеличить.

C

GNU C

Рекомендуемый шаблон:

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main( void )
{
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
  return 0;
}
```

Pascal

Borland Delphi

Рекомендуемый шаблон:

```
{$APPTYPE CONSOLE}
{$O-,Q+,R+}
{$MINSTACKSIZE 1000000}
uses Math, SysUtils;
begin
  assign(input, 'input.txt');
  reset(input);
  assign(output, 'output.txt');
  rewrite(output);

  close(input);
  close(output);
end.
```


Free Pascal

Если можете не писать под этим компилятором, не пишите под ним.

Рекомендуемый шаблон:

```
{$GOTO ON}
{$H+,I+,Q+,R+,S+}
uses math, sysutils;
begin
  assign(input, 'input.txt');
  reset(input);
  assign(output, 'output.txt');
  rewrite(output);

  close(input);
  close(output);
end.
```


Работа с вещественными числами

При работе с вещественными числами часто возникают некоторые трудности. А именно, при большинстве операций числа округляются и возникают погрешности. Если мы вычисляем значение некоторой функции в точке, значение которой получено в результате вычислений, при этом наша функция не является непрерывной в окрестности этой точки, то может случиться так, что небольшая погрешность приведет к значительному отличию настоящего значения функции от вычисленного. Например, если в результате вычислений получилось число $1 - 10^{-10}$, которое с математической точки зрения должно быть равно 1, то беря целую часть от этого числа мы в одном случае получим 0, а в другом — 1. Поэтому при вычислении таких функций необходимо проявлять осторожность.

Простейшая модель

Наиболее простой подход основывается на введении некоторого параметра ε. Определим сначала операции сравнений чисел. Для каждого числа x мы рассматриваем его ε-окрестность $[x-\varepsilon, x+\varepsilon]$. Все числа, попадающие в эту окрестность мы считаем равными x. Определенное таким образом отношение равенства будет рефлексивным и симметричным, но не будет транзитивным, поэтому не может считаться отношением эквивалентности. Однако в большинстве случаев это не играет роли, поскольку, например, обычная операция сложения вещественных чисел не является ассоциативной. Остальные операции определяются так, чтобы соответствовать операции равенства.

$$\begin{array}{ll} a = b & |a - b| \le \varepsilon \\ a \ne b & |a - b| > \varepsilon \\ a < b & a + \varepsilon < b \\ a \le b & a \le b + \varepsilon \\ a > b & a > b + \varepsilon \\ a \ge b & a + \varepsilon \ge b \end{array}$$

Следует отметить, что операции $<$ и $\le$ будут по-прежнему являться отношениями строгого и нестрогого порядка соответственно.

Используя эти операции, мы можем определить операции принадлежности четырем видам интервалов ($[a,b]$, $[a,b)$, $(a,b]$, (a,b)) как принадлежность интервалам такого же типа, но с модифицированными границами ($[a-\varepsilon, b+\varepsilon]$, $[a-\varepsilon, b-\varepsilon)$, $(a+\varepsilon, b+\varepsilon]$, $(a+\varepsilon, b-\varepsilon)$ соответственно). Теперь вычисление любой функции осуществляется следующим образом: разобьем область определения функции на максимальные по включению промежутки, на каждом из которых функция определена, а также непрерывна во всех его внутренних точках (маловероятно, что нам придется вычислять функцию, для которой это сделать нельзя), и модифицируем эти промежутки в соответствии с нашим ε. Теперь, когда мы вычисляем значение функции для заданного аргумента, мы сначала определяем, какому из модифицированных промежутков он принадлежит. Если аргумент лежит вне соответствующего исходного промежутка, то заменяем его на ближайшую к нему точку в этом промежутке. Затем вычисляем значение функции как обычно. Например, для функции пола (целой части) промежутки имеют вид $[k, k+1)$, где k — целое число. Модифицированные промежутки имеют вид $[k-\varepsilon, k+1-\varepsilon)$, и вычислить значение точки x мы можем как $\lfloor x + \varepsilon \rfloor$, поскольку при прибавлении ε каждое число попадает в свой промежуток, а значение функции на промежутке постоянно. Следует отметить, что для устойчивости этого способа может потребоваться брать разные значения ε для разных промежутков, кроме того, значение ε для конкретной границы промежутка (левой или правой) должно не превосходить половины расстояния от этой границы до ближайшего конца промежутка с соответствующей стороны. Если предполагается вычисление функции только в тех точках, в которых она определена, то следует брать ε равным половине соответствующего минимального расстояния, а в случае, если искомая граница является нижней или верхней гранью области определения функции, ε следует полагать равным ∞.

Стандартные функции преобразуются следующим образом:

$\lfloor x \rfloor$	$\lfloor x + \varepsilon \rfloor$
$\lceil x \rceil$	$\lceil x - \varepsilon \rceil$

$\operatorname{sign}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \\ 1, & \text{if } x > 0 \end{cases}$	$\begin{cases} -1, & \text{if } x < -\varepsilon \\ 0, & \text{if } -\varepsilon \leq x \leq \varepsilon \\ 1, & \text{if } x > \varepsilon \end{cases}$
$\tan(x)$	$\tan(x)$, if $\pi k - \pi/2 + \varepsilon < x < \pi k + \pi/2 - \varepsilon$ for some $k \in \mathbf{Z}$
$\log(x)$	$\log(x)$, if $x > \varepsilon$
$\sqrt{x}$	$\begin{cases} 0, & \text{if } -\varepsilon \leq x < 0 \\ \sqrt{x}, & \text{if } x \geq 0 \end{cases}$

Функции от нескольких аргументов обрабатываются аналогичным образом. Например, функция полярного угла $\arg(x, y)$ будет принимать те же значения, но иметь другую область определения: $|x| > \varepsilon$ или $|y| > \varepsilon$. Следует отметить, что для непрерывных функций, таких, как $\min$ и $\max$ использование эпсилона не только не требуется, но и не желательно, хотя при вычислении этих функций используется операция $<$, которая не является непрерывной.

Структуры данных

Хип

Хип (heap) — структура данных, хранящая некоторый набор элементов, над которыми задано отношение линейного порядка, и поддерживающая следующие операции с этим набором:

- получить минимальный элемент за $O(1)$,
- извлечь минимальный элемент,
- добавить элемент,
- извлечь элемент в заданной позиции,
- уменьшить значение элемента в заданной позиции,
- увеличить значение элемента в заданной позиции,
- изменить значение элемента в заданной позиции,
  (все за $O(\log(n))$, где n — размер хипа),
- превратить массив из m элементов в хип за $O(m)$,
- превратить хип из n элементов в отсортированный массив за $O(n \log(n))$.

Для реализации хипа используют двоичное дерево, упакованное в массив следующим образом: корень имеет номер 1, левый сын элемента с номером i имеет номер $2i$, правый сын имеет номер $2i + 1$, отец имеет номер $\lfloor i/2 \rfloor$. В массиве используются элементы с номерами от 1 до n, где n — размер хипа.

```
# const max — максимальный размер хипа
# A[1 .. max] — хип
# size — размер хипа

init()
  size ← 0

parent(i)
  return i/2

left(i)
  return 2 * i

right(i)
  return 2 * i + 1
```

Каждому элементу в хипе сопоставлено некоторое значение. Для того, чтобы массив мог считаться хипом, необходимо, чтобы каждый элемент был не меньше своего родителя ($A[i] \geq A[parent(i)]$).

Для реализации всех операций с хипом используются две служебные процедуры просеивания вверх и вниз. Пусть у нас был хип, в котором увеличили значение i-го элемента. Мы хотим получившийся массив превратить обратно в хип. Для этого используется процедура *sift_down*:

```
sift_down(i)
  while true
    smallest ← i
    l ← left(i)
    if (l ≤ size) ∧ (A[smallest] > A[l])
      smallest ← l
    r ← right(i)
    if (r ≤ size) ∧ (A[smallest] > A[r])
      smallest ← r
    if i = smallest
      break
    A[i] ↔ A[smallest]
    i ← smallest
```

Эта процедура действует следующим образом: если i-й элемент меньше либо равен каждому из своих сыновей, то делать ничего не надо. В противном случае меняем его с большим сыном, и повторяем процедуру для элемента, на месте которого мы теперь находимся.

Если же мы уменьшили значение i-го элемента, то для превращения получившегося массива в хип используется процедура *sift_up*:

```
sift_up(i)
  x ← A[i]
  while (i > 1) ∧ (A[parent(i)] > x)
    A[i] ← A[parent(i)]
    i ← parent(i)
  A[i] ← x
```

Эта процедура меняет нас со своим родителем, пока мы меньше его.

С помощью этих процедур реализуются все остальные:

```
insert(x)
  size++
  A[size] ← x
  sift_up(size)

get_min()
  if size = 0
    underflow()
  return A[1]

extract_min()
  x ← get_min()
  A[1] ← A[size]
  size--
  sift_down(1)
  return x

increase_key(i, x)
  A[i] ← x
  sift_down(i)

decrease_key(i, x)
  A[i] ← x
  sift_up(i)

change_key(i, x)
  if A[i] > x
    decrease_key(i, x)
  else
    increase_key(i, x)

delete(i)
  x ← A[size]
  size--
  change_key(i, x)

build()
  for i ← size .. 1
    sift_down(i)
```

```
sort()
  j ← size
  for i ← size .. 1
    A[i] ← extract_min()
  size ← j
  # теперь массив A отсортирован по убыванию
  for i ← 1 .. size/2
    A[i] ↔ A[size + 1 − i]
```

Следует отметить, что хип не поддерживает операцию эффективного поиска элемента по значению. Однако, если структура элементов и объем доступной памяти таковы, что для каждого элемента мы можем хранить его позицию в хипе, то достаточно просто изменить все процедуры так, чтобы эффективно поддерживать массив позиций элементов: после всех строк вида $heap[i] = j$ вставить $pos[heap[i]] = i$ (здесь $pos[j]$ — позиция элемента со значением j в хипе).

Структуры данных для упорядоченных множеств

Пусть у нас есть множество X, на котором задано отношение линейного порядка. Мы хотим получить структуру данных, которая хранит некоторое множество S, являющееся подмножеством множества X, и поддерживающую с этим множеством следующие операции (здесь $S \subset X$, $x \in X$, $k \in \mathbf{Z}$):

- $add(S, x)$: $S \leftarrow S \cup \{x\}$;
- $find(S, x)$: возвратить $[x \in S]$;
- $remove(S, x)$: $S \leftarrow S \setminus \{x\}$;
- $select(S, k)$: возвратить k-ый в порядке неубывания элемент из S;
- $index(S, x)$: возвратить минимальное k, такое, что $select(S, k) = x$;
- $split(S, x)$: разделить множество S на два множества: в одном будут находиться все элементы меньшие x, в другом — все остальные;
- $merge(S_1, S_2)$: объединить множества S_1 и S_2, такие, что все элементы в S_1 меньше всех элементов в S_2, в одно множество $S = S_1 \cup S_2$.
- $build(A)$: создать множество S, содержащее все элементы из массива A, элементы которого даны в неубывающем порядке;
- $sort(S)$: записать элементы множества S в массив в неубывающем порядке;

Возможна также ситуация, когда S является мультимножеством. В этом случае операция *find* возвращает количество вхождений данного элемента, а определения операций *add* и *remove* модифицируются естественным образом.

Существуют структуры данных, поддерживающие операции операции *build* и *sort* за $O(S)$ и остальные операции за $O(\log(S))$, которые не зависят от структуры множества X и при работе с элементами используют только операции сравнения. Такие структуры используют тот или иной вид сбалансированных деревьев поиска (balanced search trees). Наиболее известные разновидности сбалансированных деревьев: АВЛ-деревья (Г. М. Адельсон-Вельский, Е. М. Ландис); 2-3-trees (J. E. Hopcroft); B-trees (R. Bayer, E. M. McCreight); Red-Black trees (R. Bayer); Splay trees (D. D. Sleator, R. E. Tarjan).

Однако эти структуры достаточно сложны в реализации и не всегда оптимальны для конкретного множества X.

Бор

Бор (trie) — позволяет эффективно реализовать указанные операции в случае, когда X — множество строк над заданным алфавитом Σ. Бор — это ориентированное дерево, каждому ребру которого сопоставлен символ нашего алфавита. При этом из одной вершины может выходить не более одного ребра, помеченного данным символом. Если мы занумеруем узлы бора последовательными натуральными числами, то наиболее просто представить бор в памяти можно в виде двумерного массива *next* следующим образом: если из вершины i в вершину j ведет ребро, помеченное символом c, то $next[i][c] = j$. Если же из вершины i не выходит ребро, помеченное символом c, то $next[i][c] = 0$. Таким образом, на каждую вершину используется $O(\Sigma)$ ячеек памяти. Определим

на множестве вершин функцию *path* следующим образом: значение *path* от корня нашего дерева равняется пустой строке. Если из вершины i в вершину j ведет ребро, помеченное символом c, то $path(j)$ будет равен $path(i)$, к которому приписали символ c. Теперь мы можем хранить наше множество S в виде бора следующим образом: заведем массив *count*, такой, что $count[i] = find(path(i))$, то есть $count[i]$ — количество строк в множестве S, равных $path(i)$ (для обычного множества это 0 или 1, для мультимножества — любое неотрицательное число). Кроме того, мы потребуем, чтобы для любой строки s из нашего множества существовало такое i, что $path(i) = s$. Теперь массив *count* однозначно определяет наше множество S. Теперь уже можно реализовать операции *add*, *find* и *remove*, однако для того, чтобы эффективно реализовать операции *index* и *select* нам потребуется еще один дополнительный массив *size*, такой, что $size[i]$ равен количеству строк в множестве S, начинающихся на $path(i)$. Для удобства реализации полагаем также $size[0] = 0$.

При реализации всех операций кроме *merge* мы предполагаем, что номер корневого узла хранится в переменной *root*.

```
# const max — максимальное количество узлов в боре
# const nil = 0 — используется для указания отсутствия связи
# root — номер корневого узла
# last — номер последнего использованного узла в боре
# next[1 .. max][Σ]
# count[1 .. max]
# size[0 .. max]

alloc()                                  # выделение и инициализация нового узла
  last++
  count[last] ← 0
  size[last] ← 0
  for c ∈ Σ
    next[last][c] ← nil
  return last

init()
  size[nil] ← 0
  last ← 0
  root ← alloc()

add(s)
  cur ← root
  size[cur]++
  for i ← 1 .. |s|
    if next[cur][s[i]] = nil
      next[cur][s[i]] ← alloc()
    cur ← next[cur][s[i]]
    size[cur]++
  count[cur]++

find(s)
  cur ← root
  for i ← 1 .. |s|
    if next[cur][s[i]] = nil
      return 0
    else
      cur ← next[cur][s[i]]
  return count[cur]
```



```
remove(s)
  if find(s) = 0
    error()
  cur ← root
  size[cur]--
  for i ← 1 .. |s|
    cur ← next[cur][s[i]]
    size[cur]--
  count[cur]--

index(s)
  cur ← root
  k ← 1
  for i ← 1 .. |s|
    k += count[cur]
    for c < s[i]
      k += size[next[cur][c]]
    if next[cur][s[i]] = nil
      return k
    cur ← next[cur][s[i]]
  return k

select(k)
  if (k < 1) ∨ (k > size[root])
    error()
  cur ← root
  |s| ← 0
  while true
    if k ≤ count[cur]
      return s
    else
      k -= count[cur]
      for c ∈ Σ
        if k ≤ size[next[cur][c]]
          s += c
          cur ← next[cur][c]
          break
        else
          k -= size[next[cur][c]]
```

Операции *add*, *select*, *index*, *split* и *merge* работают за время $O(s \cdot \Sigma)$, где s — строка, с которой мы работаем в функциях *add*, *index* и *split*, которую находим в функции *select* и которая является наибольшим общим префиксом для всех пар строк из множеств S_1 и S_2 в функции *merge*. Операции *find* и *remove* работают за время $O(s)$.

На самом деле, можно обойтись без массива *count* и упростить все процедуры, если дополнить каждую строку справа символом #, где $\# \notin \Sigma$ и для любого c из Σ верно, что $\# < c$.

Числовой бор

После того, как мы научились строить эффективную структуру данных для поддерживания множества строк, мы можем довольно тривиальным образом получить аналогичную структуру для множества чисел. А именно, мы сейчас установим отображение s из множества неотрицательных целых чисел в множество строк, сохраняющее операцию меньше, то есть для любых целых x и y выполняется условие $s(x) < s(y)$. Если мы обладаем такой операцией, то все операции для множества чисел можно реализовать как операции для множества строк. В качестве отображения s можно взять двоичное представление заданного числа, дополненное нулями слева так, чтобы все

числа имели одинаковую длину. Если все числа, с которыми мы работаем, меньше M, то минимальное количество бит, необходимое для их представления, есть $L = \lceil \log(M) \rceil$. Таким образом, мы просто заменяем все фрагменты кода в программе вида

for $i \leftarrow 1 \mathinner{..} |s|$

на следующую строку:

for $i \leftarrow L - 1 \mathinner{..} 0$,

и вместо $s[i]$ везде используем $(s \gg i) \wedge 1$. Соответственно, обработка одного числа занимает время $O(L)$. Кроме того, поскольку теперь у нас все элементы находятся на последнем уровне дерева, то массив *count* нам больше не нужен, так как если $count[i] > 0$, то $count[i] = size[i]$.

Структуры данных для поддержания статистик

Пусть у нас есть множество A и моноид $\langle B, * \rangle$, при этом задана функция $f \colon A \to B$. Мы хотим эффективно изменять значение функции f в заданной точке, и для заданного множества C, такого, что $C \subset A$, быстро вычислять произведение $\prod_{c \in C} f(c)$. Мы не накладываем требования коммутативности на операцию $*$, однако следует понимать, что в таком случае необходимо чётко задать порядок, в котором перебираются элементы множества C.

Заметим, что с помощью такой структуры мы можем организовать структуру данных для поддержки упорядоченного мультимножества, в самом деле, сопоставим каждому элементу в качестве значения функции f число его вхождений в мультимножество, а в качестве операции моноида возьмём сложение; тогда операции *add* и *remove* сведутся к изменению значения элемента, операция *find* сведётся к вычислению суммы элементов множества, состоящего из того элемента, число вхождений которого мы хотим узнать, операция *index* сводится к подсчёту суммы значений всех элементов, не превосходящих нашего, операция *select* также может быть эффективно реализована для тех структур, которые мы будем рассматривать.

Если на структуру множеств A и C не накладывать никаких дополнительных ограничений, то эффективное вычисление произведения (быстрее, чем за $O(|C|)$) невозможно. Однако во многих частных случаях существуют соответствующие эффективные структуры данных.

Мы приведём описания двух структур данных для случая, когда множество A является некоторым отрезком множества целых чисел. Первая такая структура, дерево отрезков, работает с произвольным моноидом, и использует память, не более чем в три раза превышающую объём памяти, необходимый для того, чтобы хранить значения функции f для элементов искомого множества; другая — двоичное индексированное дерево Фенвика, требует, чтобы $\langle B, * \rangle$ был группой, но при этом требует памяти столько же, сколько требовалось для хранения исходных значений функции; в случае обеих структур сами исходные значения хранить не надо.

Обе структуры организованы следующим образом: каждому целому числу i из некоторого множества мы сопоставляем отрезок $segment(i)$; сама структура данных будет представлять собой массив, в котором для каждого такого i будет храниться произведение всех элементов, лежащих внутри этого отрезка, при этом произведение вычисляется слева направо. Мы будем организовывать структуру отрезков $segment(i)$ таким образом, что произведение элементов на произвольном отрезке можно выразить через небольшое количество значений, хранящихся в структуре данных. Число элементов в нашей структуре данных, зависящих от данного элемента, также будет невелико, что позволяет быстро обновить значение любого элемента.

В описаниях структур мы будем обозначать произведение значений функции f на отрезке $[i \mathinner{..} j]$ (элементы перебираются слева направо) за $product(i, j)$; операцию обновления элемента с номером i будем обозначать за $update(i, x)$. Здесь за x обозначено новое значение элемента с номером i, или то значение, на которое этот элемент надо умножить. Первый подход является наиболее общим, а второй удобен для дерева Фенвика.

Дерево отрезков

Рассмотрим двоичное дерево, вершины которого проиндексированы также, как и вершины хипа. Будем считать, что в листьях дерева хранятся искомые значения функции f, а значение элемента, хранящегося во внутренней вершине будет равно произведению значения его левого сына на значение его правого сына; таким образом, искомое значение будет состоять из произведения зна-

чений всех листов, которые в качестве предка имеют заданную вершину. Например, для хранения 22 значений будет использоваться следующее дерево:

<table>
<tr><td colspan="32">1</td></tr>
<tr><td colspan="16">2</td><td colspan="16">3</td></tr>
<tr><td colspan="8">4</td><td colspan="8">5</td><td colspan="8">6</td><td colspan="8">7</td></tr>
<tr><td colspan="4">8</td><td colspan="4">9</td><td colspan="4">10</td><td colspan="4">11</td><td colspan="4">12</td><td colspan="4">13</td><td colspan="4">14</td><td colspan="4">15</td></tr>
<tr><td colspan="2">16</td><td colspan="2">17</td><td colspan="2">18</td><td colspan="2">19</td><td colspan="2">20</td><td colspan="2">21</td><td colspan="2">22</td><td colspan="2">23</td><td colspan="2">24</td><td colspan="2">25</td><td colspan="2">26</td><td colspan="2">27</td><td colspan="2">28</td><td colspan="2">29</td><td colspan="2">30</td><td colspan="2">31</td></tr>
<tr><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td><td>38</td><td>39</td><td>40</td><td>41</td><td>42</td><td>43</td><td>44</td><td>45</td><td>46</td><td>47</td><td>48</td><td>49</td><td>50</td><td>51</td><td>52</td><td>53</td></tr>
</table>

Дадим теперь формальное описание дерева отрезков и операций на нём. Пусть множество A представляет собой отрезок из M идущих подряд целых чисел; положим $L = \lceil \log M \rceil$ и $P = 2^L$. Будем считать, что минимальный элемент множества A равен P; если это не так, то можно просто прибавлять соответствующее значение к номерам всех элементов. Сопоставим каждому узлу с номером i отрезок $segment(i) = [i \cdot 2^{L-\lfloor \log(i) \rfloor} \mathinner{..} (i+1) \cdot 2^{L-\lfloor \log(i) \rfloor} - 1]$. Заведём массив, состоящий из $2P-1$ элемента, которые нумеруются начиная с 1. В i-ом элементе этого массива мы будем хранить произведение всех значений функции f на отрезке $segment(i)$.

Теперь уже нетрудно написать функции для работы с деревом отрезков. Не следует забывать, что множество A есть не что иное, как отрезок $[P \mathinner{..} P+M)$; в типичных случаях применения элементы нумеруются с 0, и поэтому перед каждой операцией к номерам элементов добавляют P. В процедуре инициализации мы считаем, что элементы массива *value* с номерами в отрезке $[P \mathinner{..} P+M)$ уже заполнены соответствующими значениями. Также мы считаем, что несуществующим элементам с номерами из отрезка $[P+M \mathinner{..} 2P)$ присвоено значение e (e является нейтральным элементом моноида). При такой реализации $P \le 2M-1$ и $2P-1 \le 4M-3$, соответственно используемое число ячеек памяти меньше четырёх M. Константу 4 можно уменьшить до 3 с сохранением оценок эффективности, если отбросить конечный отрезок неиспользуемых едничиных элементов массива, при этом необходимо внести соответствующие исправления в процедуры.

```
init()
  for i ← P − 1 .. 1
    value[i] ← value[2 ∗ i] ∗ value[2 ∗ i + 1]

update(i, x)
  value[i] ← x
  while i > 1
    i /= 2
    value[i] ← value[2 ∗ i] ∗ value[2 ∗ i + 1]

product(i, j)
  lp ← e
  rp ← e
  while i ≤ j
    if (i ∧ 1) = 1
      lp ← lp ∗ value[i]
      i++
    if (j ∧ 1) = 0
      rp ← value[j] ∗ rp
      j−−
    i /= 2
    j /= 2
  return lp ∗ rp
```

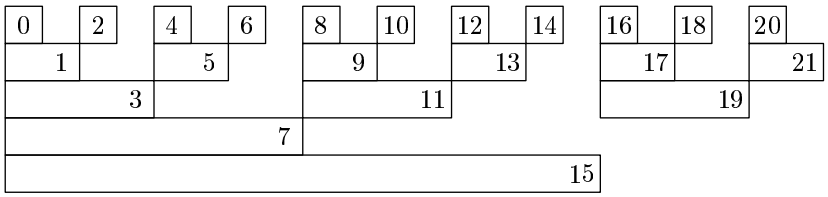
Двоичное индексированное дерево Фенвика

Пусть теперь множество A является отрезком $[0 \mathinner{..} M)$. Определим две вспомогательные функции:

$$low(i) = i \wedge (i+1),$$

$$high(i) = i \vee (i + 1).$$

Нетрудно заметить, что функция *low* обнуляет последний блок единиц, примыкающий к правому краю двоичного представления числа i, а функция *high* инвертирует младший нулевой бит числа i. Заведём теперь массив, индексируемый элементами множества A; в i-ом элементе этого массива мы будем хранить произведение всех значений функции на отрезке $segment(i) = [low(i) \mathinner{..} i]$. Полученная структура данных называется деревом Фенвика. Например, для хранения 22 значений будет использоваться следующее дерево:



Поскольку наш моноид является группой, в нём верно следующее тождество: $product(i, j) = product(0, i - 1)^{-1} product(0, j)$. Соответственно, достаточно написать частный случай функции *product*, у которой первый аргумент нулевой.

```
init()
  for i ← 0 .. n − 1
    value[i] ← e

low(i)
  return i ∧ (i + 1)

high(i)
  return i ∨ (i + 1)

lp(i)
  p ← e
  while i ≥ 0
    p ← value[i] ∗ p
    i ← low(i) − 1
  return p

product(i, j)
  return lp(i − 1)^{−1} ∗ lp(j)

update(i, x)
  while i < n
    value[i] ← value[i] ∗ y
    i ← high(i)
```

Заметим, что функция *update* требует, чтобы группа была абелевой, в то время как остальные функции работают с произвольными группами.

Упражнение. *Модифицируйте функцию update так, чтобы она работала с произвольными группами с сохранением асимптотики времени работы. Ваша функция должна допускать произвольное изменение данного элемента, а не умножение его на заданное число (на самом деле любую операцию можно проэмулировать другой, но при реализации удобен только один из способов).*

Функция *lp* вычисляет произведение элементов с номерами не больше i. Она разбивает отрезок $[0 \mathinner{..} i]$ на отрезки вида $[low(i) \mathinner{..} i]$, произведение значений в каждом из которых хранится в одном из элементов массива *value*.

Отметим важное отличие в функциональности процедуры *update*: для дерева отрезков в качестве обновления элемента мы подразумевали изменение его значения на заданное, а здесь в качестве обновления мы подразумеваем умножение значения элемента на заданное. Если мы хотим,

чтобы процедура обновления имела такую же функциональность как в дереве отрезков, необходимо в начало процедуры поместить присваивание вида $y \leftarrow x/product(i,i)$. Очевидно, что элемент с номером j при процедуре обновления надо менять тогда и только тогда, когда $i \in [low(j)\,..\,j]$. Все такие j мы можем получить, последовательно устанавливая младший нулевой бит i в единицу.

Упражнение. *Разработайте процедуру инициализации дерева Фенвика произвольными значениями за линейное время.*

Упражнение. *При использовании формулы $product(i,j) = product(0,i-1)^{-1}product(0,j)$ нетрудно увидеть, что начальные элементы и в том, и в другом произведении совпадают. Модифицируйте процедуру product так, чтобы она не перемножала пары взаимно обратных элементов. Оцените среднее время работы функции $product(i,i)$ (оно есть $o(\log M)$).*

Обобщение дерева отрезков и дерева Фенвика на случай многомерных структур

Пусть теперь множества A и C имеют вид $[a\,..\,b] \times [c\,..\,d] = \{(x,y) \mid x \in [a\,..\,b] \wedge y \in [c\,..\,d]\}$. Для такой постановки задачи мы можем обобщить дерево отрезков и дерево Фенвика на случай двух размерностей. Массив нашей структуры данных станет двумерным, и в элементе (i,j) мы будем хранить произведение всех элементов (k,l), таких, что $k \in segment(i)$ и $l \in segment(j)$.

Очевидно, что все функции для работы с деревом отрезков и деревом Фенвика тривиальным образом обобщаются на этот двумерный случай.

Вариант реализации процедуры обновления для дерева Фенвика:

```
update(i, j, y)
  while i < N
    xupdate(i, j, y)
    i ← high(i)

xupdate(i, j, y)
  while j < N
    value[i][j] ← value[i][j] * y
    j ← high(j)
```

Не составляет труда обобщить этот метод на случай более высоких размерностей.

Двоичный поиск

Двоичный поиск — это общий метод решения задач на минимакс. Задача на минимакс — это задача, в которой требуется минимизировать какой-либо параметр, который равен максимуму по какому-то множеству значений.

Для того, чтобы мы могли применить двоичный поиск, мы должны обладать возможностью произвести сравнение значения оптимального параметра и некоторого заданного, то есть определить, меньше или больше наше значение чем оптимальное.

Типичной задачей на минимакс является задача решения уравнения: имеется неубывающая функция f, задано некоторое y, требуется найти такой x, что $f(x) = y$, либо сказать, что такого нет.

Более формально модель двоичного поиска можно описать следующим образом: пусть заданы два множества A и B, на каждом из которых задан линейный порядок. Для произвольных a и b из множества A обозначим за $dist(a, b)$ количество таких x, что $a \le x \le b$. Мы должны уметь находить такой элемент $mid(a, b)$, что $dist(a, mid(a, b)) - 1 < \alpha \cdot dist(a, b)$ и $dist(mid(a, b), b) - 1 < \beta \cdot dist(a, b)$ для некоторых констант α и β, меньших 1. Пусть задана монотонно неубывающая функция $f : A \to B$ (то есть $a < b \Rightarrow f(a) \le f(b)$). Мы хотим найти минимальное a, такое, что $f(a) \ge b$ для заданного b. Для этого мы на каждом шаге рассматриваем интервал $[b, c]$, в котором может находиться наше a. В зависимости от того, будет ли значение функции в точке $mid(b, c)$ меньше или больше требуемого значения, мы изменяем левую или правую границу интервала. Время работы алгоритма будет составлять $O(\log(dist(a, b)))$, где $[a, b]$ — интервал, в котром находится искомое значение. Константа в обозначении $O()$ будет тем меньше, чем ближе будут α и β к $1/2$.

При описании алгоритмов двоичного поиска мы будем использовать сравнения вида $x < opt$ для сравнения значения x с оптимальным.

Вещественный двоичный поиск

В случае, когда мы оптимизируем вещественный параметр, довольно просто сделать двоичный поиск следующим образом:

$l \leftarrow lower$
$u \leftarrow upper$
while $l + \varepsilon < u$
  $m \leftarrow (l + u)/2$
  if $m < opt$ # можно также $m \le opt$
    $l \leftarrow m$
  else
    $u \leftarrow m$
return l

Время работы нашего алгоритма составляет $\Theta(\log((upper - lower)/\varepsilon))$, где ε — точность, с которой мы должны найти ответ, а интервал, в котором он находится — это $[lower, upper]$. Последняя строка алгоритма предполагает, что ответ всегда существует, в противном случае нам надо проверить, подходит ли он.

Дискретный двоичный поиск

Когда мы оптимизируем целочисленный параметр, двоичный поиск делается аналогично, однако здесь есть некоторые тонкости с граничными условиями. Есть два варианта дискретного двоичного поиска, в зависимости от того, доступна ли нам проверка на строгое или на нестрогое равенство.


```
# Вариант 1
l ← lower
u ← upper
while l < u
   m ← (l + u)/2                    # Внимание: l + u, не l + u + 1
   if m < opt                       # Внимание: <, не ≤
      l ← m + 1                     # Внимание: m + 1, не m
   else
      u ← m                         # Внимание: m, не m + 1
result ← l
# Результат — первое a, такое, что a ≥ opt

# Вариант 2
l ← lower
u ← upper
while l < u
   m ← (l + u + 1)/2                # Внимание: l + u + 1, не l + u
   if m ≤ opt                       # Внимание: ≤, не <
      l ← m                         # Внимание: m, не m + 1
   else
      u ← m − 1                     # Внимание: m − 1, не m
return l
# Результат — последнее a, такое, что a ≤ opt
```


Meet-in-the-Middle

Meet-in-the-Middle — это метод решения уравнений вида $f(x) = g(y)$, где $x \in X$ и $y \in Y$, который работает за время $(X + Y)\log(X)$.

Он работает следующим образом: переберем все возможные значения x и запишем пару значений $(x, f(x))$ в массив, который мы отсортируем по второму элементу (значению функции). Затем будем перебирать всевозможные значения y, для каждого из них будем вычислять $f(y)$, которое мы будем искать в нашем отсортированном массиве. Таким образом, время работы нашего алгоритма составляет $O(X\log(X))$ на сортировку, и $O(Y\log(X))$ на двоичный поиск, что дает в сумме $O((X + Y)\log(X))$.

Классическая задача на метод Meet-in-the-Middle — это задача дискретного логарифмирования, которая состоит в том, что для заданных целых a, b и m требуется найти минимальное неотрицательное x, такое, что $a^x - b$ делится на m. Фактически, мы решаем уравнение $a^x \bmod m = b \bmod m$. Так как левая часть принимает не больше m различных значений, то значения x начиная с m не имеет смысла рассматривать, так как a^x для них совпадает с более ранними значениями. Поскольку у нас здесь только одно неизвестное, то нам необходимо преобразовать наше уравнение, чтобы в нем было два неизвестных. Делается это при помощи приема, называемого baby-step, giant-step. А именно, мы выбираем целую положительную константу n, зависящую от m и представляем x в виде $x = ni - j$, где i изменяется от 1 до $\left\lceil \frac{m}{n} \right\rceil$, а j — от 1 до n. Очевидно, что любое x можно представить в таком виде. Если a и m взаимно просты, то уравнение $a^x \bmod m = b \bmod m$ переписывается как $a^{ni} \bmod m = ba^j \bmod m$, и здесь мы уже можем применить Meet-in-the-Middle. Время работы нашего алгоритма составляет $O((n+m/n)\log(n))$, очевидно, что минимум достигается тогда, когда $n = \Theta(\sqrt{m})$ и составляет $\Theta(\sqrt{m}\log(m))$.

Упражнение. *Модифицируйте этот алгоритм, чтобы он работал для любых значений a и m (не только взаимно простых).*

Динамическое программирование

Игры на ориентированных графах

Игры на ацикличных ориентированных графах

Задан ацикличный ориентированный граф. На графе производится игра по следующим правилам. Игроки ходят по очереди. На каждом ходу есть некоторая текущая вершина. Ход состоит в том, что игрок, делающий ход, выбирает любое ребро, ведущее из текущей вершины, и заменяет текущую вершину на конечную вершину этого ребра. Если из текущей вершины не ведет ни одного ребра, то игрок проигрывает (а его противник выигрывает). Начальная вершина задана. Требуется определить, может ли тот игрок, который ходит первым, выиграть как бы ни играл противник, и если да, то указать для него любой ход, ведущий к выигрышу.

Задача решается динамическим программированием сверху вниз. Подзадачами у нас будут являться решения для заданной вершины. При рассмотрении очередной, ранее не рассматривавшейся вершины, мы определяем для всех вершин, в которые из нашей ведут ребра, являются ли они выигрышными или проигрышными (для того игрока, который из них ходит). После этого, если мы можем сделать ход в проигрышную позицию, то наша позиция является выигрышной, а иначе она является проигрышной (поскольку все ходы из нашей позиции ведут в выигрышные позиции для нашего противника, то есть проигрышные для нас позиции). Поскольку граф ацикличный, то наш алгоритм не зациклится. Время его работы составить $\Theta(V + E)$, где E — количество ребер, достижимых из нашей начальной вершины, а V — количество вершин в нашем графе.

Непосредственно из описания алгоритма получается следующая теорема.

Теорема. *Ровно один игрок имеет выигрышную стратегию в игре на ацикличном ориентированном графе. Статус каждой вершины (является ли она выигрышной или проигрышной для того игрока, который из нее ходит) определяется с помощью следующих трех правил.*

- *Вершина, из которой не выходят ребра, является проигрышной.*
- *Вершина, из которой ведет ребро в проигрышную вершину, является выигрышной.*
- *Вершина, из которой все ребра ведут в выигрышные вершины, является проигрышной.*

Нетрудно также адаптировать алгоритм решения этой задачи для многих других вариаций.

Упражнение. *Пусть каждому ребру графа сопоставлено некоторое число. Результатом игры теперь будет не выигрыш одного из игроков, а сумма чисел, записанных на всех ребрах, которые были пройдены, перед тем, как игра закончилась. Первому игроку требуется максимизировать эту сумма, а второму — минимизировать. Разработайте алгоритм, который определяет максимальный возможный выигрыш первого игрока, как бы ни ходил второй.*

Игры на произвольных ориентированных графах

Игра на произвольном ориентированном графе определяется аналогично игре на ацикличном ориентированном графе. В случае, когда ни один из игроков не может выиграть, игра считается ничейной.

Для начала докажем теорему, аналогичную теореме из предыдущего раздела.

Теорема. *Игра на ориентированном графе либо имеет выигрышную стратегию для одного из игроков, либо является ничейной. После того, как мы узнали статус всех вершин для которых можно применить три правила предыдущей теоремы, все вершины с неизвестным статусом являются ничейными.*

Для доказательства будем применять, пока это возможно, три правила из предыдущей теоремы для определения статуса некоторых вершин. Если теперь ни одно правило применить нельзя, то все вершины с неизвестным статусом будут ничейными. В самом деле, если для какой-то вершины статус определен, то он определен правильно. Докажем, что для каждой вершины с неизвестным статусом у нас есть стратегия, при которой мы не можем проиграть, и у нашего противника есть стратегия, при которой мы не можем выиграть. Если для вершины неизвестен ее статус, то из нее нет ребер, ведущих в проигрышные вершины, и при этом есть хотя бы одно ребро, ведущее в

вершину с неизвестным статусом (в противном случае все ребра из нашей вершины вели бы в выигрышные вершины и наша вершины была бы проигрышной). Когда одному из игроков надо сделать ход из вершины с неизвестным статусом, ему нет смысла идти в выигрышные вершины, поскольку в этом случае его противник выиграет и он проиграет. Значит, все, что ему остается сделать — идти в вершину с неизвестным статусом, но поскольку его противнику придется руководствоваться теми же соображениями, то оба игрока так и будут ходить по вершинам с неизвестным статусом бесконечно долго. Следовательно, все такие вершины являются ничейными.

В данном случае задача решается динамическим программированием снизу вверх. Так же, как и в предыдущем случае, подзадачами у нас будут определение статуса заданной вершины. Для каждой вершины v, статус которой неизвестен, мы храним число $od(v)$, равное количеству ребер, выходящих из этой вершины и ведущих в вершины с неизвестным статусом. Общая схема алгоритма следующая: для каждой вершины, из которой не выходит ни одно ребро, мы запускаем поиск в глубину по ребрам в обратную сторону, который находит статус некоторых вершин. После этого все вершины с неизвестным статусом помечаем как ничейные. Для данной вершины поиск в глубину состоит в следующем: мы помечаем ее как проигрышную, и смотрим, какие ребра, у которых статус начальной вершины не известен, входят в нашу вершину. Для каждой начальной вершины такого ребра мы помечаем ее как выигрышную, а затем смотрим на все ребра ведущие в нее, и начинающиеся в вершине с неизвестным статусом. Для всех таких ребер мы уменьшаем значение od для их начальных вершин. Если теперь это значение для какой-то вершины стало равно нулю, это означает, что все ребра, ведущие из нее ведут в выигрышные вершины, и, следовательно, она является проигрышной, и от нее мы тоже запускаем поиск в глубину.

Время работы алгоритма составляет $O(V+E)$, где E — количество ребер в нашем графе, а V — количество его вершин.

Исчисление путей в ориентированном графе

Пусть задан ориентированный мультиграф с петлями. Мы хотим узнать, сколькими способами из заданной вершины можно добраться до другой заданной вершины при помощи пути заданной длины. Следует отметить, что существует алгебраическое решение этой задачи, основанное на степенях матрицы смежности, которое в этом разделе обсуждаться не будет. Пусть задана наша начальная вершина s. Обозначим за $a_{t,k}$ количество способов добраться из вершины s в вершину t при помощи пути длины k. Очевидно, что $a_{t,0} = [s = t]$. Если $k > 0$, то для $a_{t,k}$ верна следующая формула:

$$a_{t,k} = \sum_{end(e)=t}^{e \in E} a_{beg(e),k-1}.$$

Понятно, что переход от одного столбца матрицы к другому можно организовать за время $O(E)$.

Очевидно, что при помощи матрицы a мы можем пронумеровать все пути от s до t в антилексикографическом порядке (лексикографическом порядке для перевернутых строк). Для нумерации путей от s до t длины k (для $k = 0$ путь ровно один – пустой, будем считать, что $k > 0$) переберем все ребра, входящие в вершину t. Пусть v_1, v_2, ..., v_m — начальные вершины этих ребер, записанные в порядке возрастания. Тогда путям от s до t длины k, проходящим через вершину v_1 в качестве предпоследней вершины, мы присвоим номера от 1 до $a_{v_1,k-1}$, через вершину v_2 — номера от $a_{v_1,k-1}+1$ до $a_{v_1,k-1}+a_{v_2,k-1}$, через вершину v_k — номера от $\sum_{1 \le i < k} a_{v_i,k-1}+1$ до $\sum_{1 \le i \le k} a_{v_i,k-1}$. Теперь нетрудно сгенерировать по пути его номер и по номеру его путь за время $O(Vk)$, где V — количество вершин нашего графа.

Упражнение. *Придумайте, как пронумеровать пути в обычном лексикографическом порядке.*

Упражнение. *Придумайте, как пронумеровать все пути от s к t, содержащие не более k ребер в лексикографическом порядке.*

Упражнение. *Пусть каждому ребру графа сопоставлен целый положительный вес. Придумайте, как пронумеровать все пути заданной стоимости в лексикографическом порядке.*

Динамика по профилю

Динамика по профилю основывается на исчислении путей в графе, каждая вершина которого представляет собой некоторое состояние (профиль). В некоторых случаях граф может быть нестационарным (то есть меняться после продвижения по очередному ребру).

Замощения

Пусть у нас есть поле $m \times n$ клеток, некоторые клетки которого удалены. Пусть у нас есть набор фигур, каждая из которых представляет собой множество клеток. Количество экземпляров каждой фигуры не ограничено. Рассматриваются замощения поля, то есть расположение определенного количества фигур на поле так, чтобы никакие две фигуры не перекрывались и каждая фигура лежала целиком на неудаленных клетках поля. Наиболее типичными задачами на замощения являются:

- Найти количество способов замощения поля;
- Найти замощение с максимальным количеством фигур;
- Найти замощение с минимальным количеством неиспользованных клеток;
- Найти замощение с минимальным количеством фигур, такое, что в него нельзя добавить еще одну фигуру;
- А также многое другое.

В большинстве случаев конфигурации (профили) будут составлять те и только те фигуры, которые имеет клетки как с абсциссами большими или равными некоторого фиксированного в данный момент значения, так и с абсциссами меньшими этого значения (то есть это будут фигуры, пересекающие некоторую вертикальную линию между столбцами).

Пример. *Найти количество способов замостить поле $m \times n$ доминошками 1×2 и 2×1, так, чтобы не было непокрытых клеток.*

В данном случае профилем будет множество доминошек, пересекающих данную вертикальную прямую, то есть профиль будет двоичным числом из m битов. Таким образом, $a_{i,j}$ будет равно количеству способов замостить поле $m \times (i+1)$ так, чтобы первые i столбцов были заполнены полностью, а в последнем столбце были заняты только те клетки, которые соответствуют единичкам в профиле j, при этом в последнем столбце не должно быть вертикальных доминошек. Понятно, что ответом будет являться $a_{n,0}$. Для $i = 0$ значения a определяются очевидным образом: $a_{0,j} = [j = 0]$. Для $i > 0$ мы можем записать для $a_{i,j}$ рекуррентное соотношение:

$$a_{i,j} = \sum_{0 \le k < 2^m} a_{i-1,k} w_{k,j},$$

где $w_{i,j}$ — количество способов перейти от профиля i к профилю j. Понятно, что в одних и тех же позициях у i и j не могут стоять единицы. Поэтому если $i \wedge j \neq 0$, то $w_{i,j} = 0$. Иначе, рассмотрим столбец, лежащий между двумя вертикальными прямыми, одна из которых соответствует профилю i, а другая — профилю j. Очевидно, что все свободные клетки в этом столбце должны быть замощены вертикальными доминошками, поскольку горизонтальная доминошка должна относиться либо к первому, либо ко второму профилю. Понятно, что такого способа не существует, если у нас есть свободный участок нечетной длины, не имеющий сверху и снизу свободных клеток. Если у нас нет таких участков, то очевидно, что способ единственный. Итак,

$$w_{i,j} = \begin{cases} f(2^m \vee i \vee j), & \text{if } i \wedge j = 0; \\ 0, & \text{otherwise.} \end{cases}$$

Здесь $f(0) = 1$, $f(2k+1) = f(k)$, $f(4k+2) = 0$, $f(4k) = f(k)$. Время работы алгоритма составляет $O(4^m n)$.

Упражнение. *Решите данную задачу, если часть клеток удалена из поля.*

Упражнение. *Решите данную задачу за время $O((1+\sqrt{2})^m n)$.*

Пример. *Найти количество способов замостить поле $m \times n$ доминошками $1 \times k$ и $k \times 1$, так, чтобы не было непокрытых клеток.*

Здесь профиль уже будет не двоичный, а k-ичный. Если i-ая цифра профиля равна j, то это означает, что доминошка пересекает вертикальную линию так, что слева от линии находится j клеток. Если $j = 0$, то в i-й строке вертикальная линия не пересекается доминошкой. Время работы алгоритма составляет $O(k^{2m}n)$.

Упражнение. *Оцените время работы алгоритма при использовании приема, аналогичного использованному в предыдущем упражнении.*

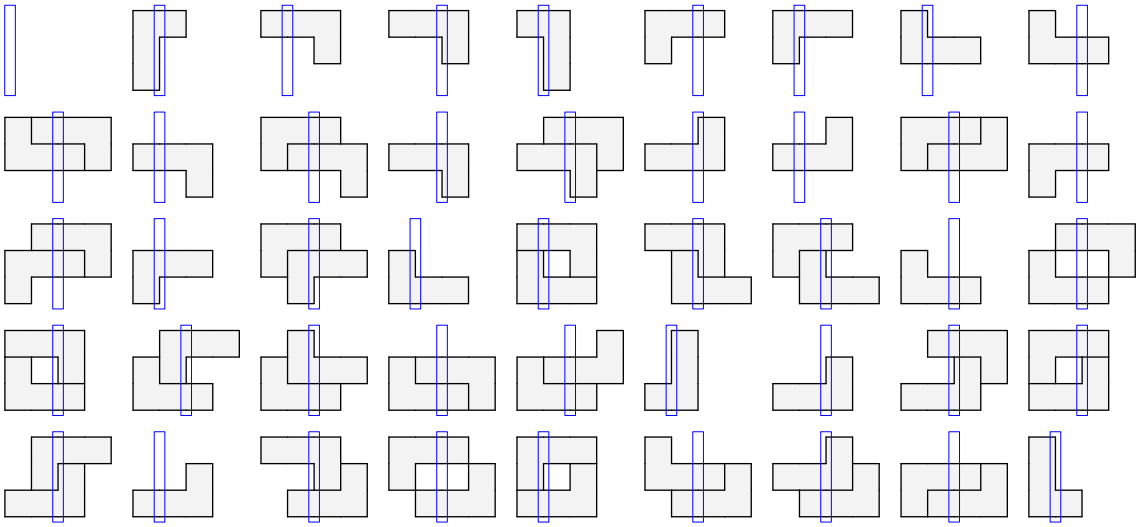
Пример. *В заданном поле $m \times n$, из которого удалены некоторые клетки, разместить минимальное количество доминошек так, чтобы нельзя было добавить еще одной доминошки.*

Здесь профилем будет троичное число из m цифр. Профиль будет обозначать конфигурацию некоторого столбца. Если i-ая цифра профиля равна нулю, то это означает, что соответствующая клетка столбца свободна. Если она равна единице, то это означает, что она занята. И, наконец, если она равна двойке, это означает, что она свободна, но точно должна быть покрыта доминошкой. Отсюда нетрудно получить алгоритм решения этой задачи, работающий за время $O(9^m n)$.

Упражнение. *Модифицируйте алгоритм, чтобы он работал за время $O(6^m n)$.*

Пример. *Найти количество способов замостить поле $m \times n$ тетрамино в виде буквы 'Г'.*

Здесь профилями будут всевозможные конфигурации фигур, пересекающих данную вертикальную линию. Например, в случае $m = 3$ будет 45 профилей:



Некоторые профили никогда не могут появиться ни в одном заполнении, однако это не влияет на решение. При переходе от одного профиля к другому столбец между соответствующими вертикальными линиями должен быть полностью заполнен, так как любое тетрамино, пересекающее этот столбец, должно пересекать хотя бы одну из вертикальных линий. Кроме того, любая фигура первого профиля либо должна совпадать с какой-либо фигурой второго профиля, либо не должна пересекаться ни с одной из них. Подсчет количества заполнений производится аналогично случаю для доминошек.

Упражнение. *Оцените время работы соответствующего алгоритма решения этой задачи.*

Теория языков и автоматов

Пусть задано конечное непустое множество Σ, называемое алфавитом. Пара $\langle n, f\rangle$, где $n \in \mathbf{N}$ и $f\colon [1\,..\,n] \to \Sigma$, называется строкой над алфавитом Σ. Строки $\langle n_1, f_1\rangle$ и $\langle n_2, f_2\rangle$ называются равными, если $n_1 = n_2$ и $f_1(k) = f_2(k)$ для всех k. Если $p = \langle n, f\rangle$, то длина строки p равна n и обозначается $|p|$. Кроме того, i-й символ строки p равен $f(i)$ и обозначается p_i. Конкатенацией строк $\langle n_1, f_1\rangle$ и $\langle n_2, f_2\rangle$ будет строка $\langle n, f\rangle$, такая, что $n = n_1 + n_2$, и

$$f(k) = \begin{cases} f_1(k), & \text{if } 1 \le k \le n_1; \\ f_2(k - n_1), & \text{if } n_1 < k \le n_1 + n_2. \end{cases}$$

Конкатенация строк p и q обозначается pq. Если $pqr = s$, то p является префиксом s, q является фактором (подстрокой) s, и r является суффиксом s.

Множество всех строк над алфавитом Σ обозначается Σ^*. Пара $\langle \Sigma^*, \cdot\rangle$, где $\cdot$ — операция конкатенации строк, будет моноидом. Этот моноид называется свободным моноидом над алфавитом Σ. Единицу этого моноида обозначают ε и называют пустой строкой.

Языком над алфавитом Σ называют подмножество Σ^*. Если L_1 и L_2 — языки над алфавитом Σ, то их конкатенация определяется как $L_1 L_2 = \{p \in \Sigma^* \mid p = qr \wedge q \in L_1 \wedge r \in L_2\}$.

Упражнение. *Докажите, что $\langle 2^{\Sigma^*}, \cdot\rangle$ является моноидом, где $\cdot$ — это операция конкатенации языков.*

Поскольку множество всех языков является моноидом относительно операции конкатенации, мы можем определить возведение языка в степень обычным для моноида образом. Вводятся специальные обозначения: $L^* = \bigcup_{i\ge 0} L^i$ (замыкание языка), $L^+ = \bigcup_{i>0} L^i$ (положительное замыкание языка).

Множество Σ можно рассматривать как язык, состоящий из односимвольных строк. Замыкание такого языка будет равно Σ^*, то есть наше обозначение для множества строк не вступает в конфликт с обозначением замыкания.

Определим теперь несколько языков, которые мы будем использовать. Пусть P — язык над алфавитом Σ. Определим язык $L(P)$ следующим образом: $L(P) = \Sigma^* P \Sigma^*$. Положим также $L_k(P) = \Sigma^k P \Sigma^*$. Если $p \in \Sigma^*$, то для краткости полагаем $L(p) = L(\{p\})$, $L_k(p) = L_k(\{p\})$.

Строка p является подстрокой строки t, если $p \in L(t)$. Строка p входит в строку t в позиции k, если $p \in L_k(t)$. Определим $S(p, t)$ как множество позиций, в которых строка p входит в строку t.

Алгоритм Кнута-Морриса-Пратта

Рассмотрим задачу о поиске подстроки в строке. Она состоит в определении истинности высказывания $t \in L(p)$ и вычислении $S(p, t)$. Алгоритм Кнута-Морриса-Пратта (Knuth-Morris-Pratt) позволяет сделать это за время $O(|p| + |t|)$.

Если $p = \varepsilon$, то обработаем этот случай отдельно. Далее считаем, что $p \ne \varepsilon$.

Будем писать $s \preceq t$, если s является и префиксом, и суффиксом t. Очевидно, что это отношение транзитивно, рефлексивно и антисимметрично, поэтому оно является отношением нестрогого порядка. Кроме того, если $a \preceq c$ и $b \preceq c$, то будет верно $a \preceq b$ или $b \preceq a$. Будем писать $s \prec t$, если $s \preceq t$ и $s \ne t$. Определим на множестве Σ^+ суффикс-функцию $\sigma(s)$ как строку максимальной длины из множества всех строк t, таких, что $t \prec s$.

Теорема. *Высказывание $t \preceq s$ верно в том и только в том случае, если существует такое $k \ge 0$, что $\sigma^{[k]}(s) = t$.*

Используя транзитивность, сразу получаем, что $\sigma^{[k]}(s) \preceq s$. Если $t = s$, то можно взять $k = 0$. Если $t \prec s$, то рассмотрим такое k, что $|\sigma^{[k]}(s)| \le |t| < |\sigma^{[k-1]}(s)|$. Отсюда $\sigma^{[k]}(s) \preceq t$ и $t \prec \sigma^{[k-1]}(s)$, и, следовательно, $t = \sigma^{[k]}(s)$.

Определим теперь для заданной строки s префикс-функцию $\pi_s\colon [1\,..\,|s|] \to [0\,..\,|s|)$, такую, что $\pi_s(i) = |\sigma(s_i)|$, где s_i — префикс длины i строки s.

Теорема. *Если $\pi_p(i) > 0$, то существует k, такое, что $\pi_p(i) = \pi_p^{[k]}(i-1)+1$.*

Если $j = \pi_p(i) > 0$, то первые j символов строки p являются суффиксом префикса p длины i. Если отбросить у префикса последний символ, то получим, что первые $j-1$ символов строки p являются суффиксом префикса p длины $i-1$. Из предыдущей теоремы получаем, что $j-1 = \pi_p^{[k]}(i-1)$.

Из этой теоремы мы сразу получаем алгоритм вычисления префикс-функции:

```
π[1] ← 0
k ← 0
for i ← 2 .. |p|
  while true
    if p[k + 1] = p[i]
      k ← k + 1
      break
    if k = 0
      break
    k ← π[k]
  π[i] ← k
```

Теперь уже нетрудно вывести сам алгоритм Кнута-Морриса-Пратта. В самом деле, нетрудно доказать, что строка p входит в строку t в позиции i в том и только в том случае, если существует такое k, что $\pi_{pt}^{[k]}(i+2|p|) = |p|$. При реализации алгоритма сначала вычисляется префикс-функция для строки p как указано выше, а затем вычисление продолжается для строки t таким образом, чтобы нам не пришлось итерировать префикс функцию для выяснения факта вхождения строки в данной позиции:

```
k ← 0
for i ← 1 .. |t|
  while true
    if p[k + 1] = t[i]
      k ← k + 1
      break
    if k = 0
      break
    k ← π[k]
  if k = |p|
    yield i − |p|
    k ← π[k]
```

Алгоритм выдает все позиции в которых строка p входит в строку t в возрастающем порядке.

Оценим время работы алгоритма Кнута-Морриса-Пратта. Поскольку каждое увеличение k на единицу соответствует не более, чем одной итерации цикла **for**, и значение k всегда неотрицательно, а на каждой итерации цикла **while** значение k уменьшается, либо цикл заканчивает свою работу, то суммарное количество итераций цикла **while** есть $O()$ от длины обрабатываемой строки. Таким образом, алгоритм Кнута-Морриса-Пратта работает за время $O(|p|+|t|)$.

Алгоритм Ахо-Корасик

Нетрудно обобщить задачу поиска подстроки в строке следующим образом. Пусть дан язык P. Мы хотим уметь вычислять $t \in L(P)$ и каждого p из множества P находить множество $S(p,t)$. Алгоритм Ахо-Корасик (Aho-Corasick) позволяет сделать это за время

$$O\left(\left(\sum_{p\in P}|p|\right)\cdot|\Sigma| + |t| + \sum_{p\in P}|S(p)|\right).$$

Если в P есть пустая строка, то обработаем её и удалим из множества P. Построим бор для множества P. Пусть *root* — номер корневого узла бора. Кроме массива *next* будем хранить массивы

parent и *char*, такие, что если $next[i][c] = j$, то $parent[j] = i$, $char[j] = c$; $parent[root]$ полагаем равным **nil**. Заведем функцию *fin* на множестве узлов бора, такую, что $fin(i) = path(i) \in P$. Если $fin(i)$ истинно, то будем хранить в $str(i)$ строку $path(i)$ (или её идентификатор во множестве P), а в $len(i)$ — её длину. Для каждой вершины i, отличной от корня, определим суффиксную ссылку $suflink(i)$ как такое j, отличное от i, что $path(j)$ является максимальным возможным суффиксом $path(i)$; $suflink(root)$ полагается равным **nil**.

Основываясь на алгоритме Кнута-Морриса-Пратта, нетрудно вычислить суффиксные ссылки для данного бора. Общая схема алгоритма вычисления суффиксных ссылок следующая: сначала полагаем, что *suflink* для всех узлов равен **nil**. Затем для всех узлов, кроме корня, вызываем рекурсивную процедуру *calc*, которая вычисляет значение суффиксной ссылки для данного узла. Процедура *calc* вычисляет также функцию *lift*, такую, что $lift(root) = \mathbf{nil}$ и для $i \neq root$

$$lift(i) = \begin{cases} i, & \text{if } fin(i); \\ lift(suflink(i)), & \text{otherwise.} \end{cases}$$

```
calc(i)
  if suflink[i] = nil
    if parent[i] = root
      suflink[i] ← root
    else
      c ← calc(parent[i])
      while true
        if next[c][char[i]] ≠ nil
          c ← next[c][char[i]]
          break
        if c = root
          break
        c ← calc(c)
      suflink[i] ← c
    if fin[i]
      lift[i] ← i
    else
      lift[i] ← lift[suflink[i]]
  return suflink[i]
```

Теперь мы можем написать основную процедуру поиска.

```
c ← root
for i ← 1 .. |t|
  while true
    if next[c][t[i]] ≠ nil
      c ← next[c][t[i]]
      break
    if c = root
      break
    c ← suflink[c]
  d ← c
  while true
    d ← lift[d]
    if d = nil
      break
    yield (str[d], i − len[d])
    d ← suflink[d]
```

Алгоритм выдает все такие пары (p, i), что строка p из множества P входит в строку t в позиции i. Пары с одинаковыми p идут в порядке возрастания i. Оценка времени работы алгоритма Ахо-Корасик доказывается аналогично оценке времени работы алгоритма Кнута-Морриса-Пратта.

Множитель $|\Sigma|$ в оценке времени работы присутствует лишь потому, что нам надо инициализировать массив *next*. Если массив *next* уже инициализирован, то множитель $|\Sigma|$ можно убрать.

Автоматные языки

Регулярные языки и выражения

Язык L является регулярным в том и только в том случае, если выполняется хотя бы одно из следующих условий:

- L — конечный язык
- L — конкатенация двух регулярных языков
- L — объединение двух регулярных языков
- L — замыкание регулярного языка

Первое условие часто заменяют на следующее условие:

- $L = \emptyset \vee L = \{\varepsilon\} \vee L = \{c\}$, где $c \in \Sigma$.

Все регулярные языки можно записать при помощи регулярных выражений следующим образом. Обозначим за $L(R)$ язык, соответствующий регулярному выражению R. Тогда:

- $L(\emptyset) = \emptyset$
- $L(\varepsilon) = \{\varepsilon\}$
- $L(c) = \{c\}$, где $c \in \Sigma$
- $L(ab) = L(a)L(b)$, где a и b — регулярные выражения
- $L(a \mid b) = L(a) \cup L(b)$, где a и b — регулярные выражения
- $L(a*) = L(a)^*$, где a — регулярное выражение

Операция объединения имеет более низкий приоритет, чем операция конкатенации, которая имеет более низкий приоритет, чем операция замыкания.

Конечные автоматы

Конечный автомат — это пятерка $\langle \Sigma, S, s, T, f \rangle$, где Σ — алфавит автомата, S — множество состояний автомата, s — начальное состояние автомата, T — множество конечных состояний автомата, $f : S \times \Sigma \cup \{\varepsilon\} \times S \to \mathbf{B}$ — функция переходов автомата, такая, что $f(a, c, b)$ истинно в том и только в том случае, если из состояния a можно перейти в состояние b по символу c. Если $c = \varepsilon$, то из состояния a можно перейти в состояние b без использования символа. Такой переход называется ε-переходом. Конечный автомат обрабатывает цепочки входных символов. В каждый момент времени автомат находится в определенном состоянии. Начальное состояние — это состояние s. После того, как автомат прочитал очередной символ c, находясь при этом в состоянии a, он может перейти в любое из состояний b, таких, что $f(a, c, b)$ истинно. Также в любой момент времени автомат может перейти по ε-переходу. Если после прочтения всей цепочки автомат может находится в одном из конечных состояний (то есть в одном из состояний множества T), то он допускает эту цепочку, в противном случае он её не допускает. Для автомата очевидным способом можно определить язык всех слов, которые он допускает. Конечный автомат называют детерминированным, если его функция переходов представима в виде $f : S \times \Sigma \to S$, то есть у него нет ε-переходов и из каждого состояния по каждому символу у него ровно один переход. Обычный конечный автомат называют также недетерминированным.

Удаление ε-переходов

Для любого автомата можно построить автомат без ε-переходов, распознающий тот же язык. Для этого сначала сделаем следующее: для всех троек a, b, c, таких, что из a в b и из b в c можно перейти по ε-переходу добавим ε-переход из a в c. Фактически, мы строим транзитивное замыкание графа ε-переходов. Теперь для всех пар a и b, между которыми есть ε-переход, перебираем все состояния c и все символы d, по которым можно перейти из b в c. Для всех таких четверок добавляем переход по символу d из a в c. Аналогично, перебираем все b и c, между которыми есть ε-переход и все состояния a и символы d, по которым можно перейти из a в b, и добавляем для каждой такой четверки переход из a в c по символу d. Кроме того, если из начального состояния можно перейти в конечное по ε-переходу, мы делаем начальное состояние конечным. После этого мы удаляем все ε-переходы.

Автоматные версии алгоритмов Кнута-Морриса-Пратта и Ахо-Корасик

Построим детерминированный автомат для языка $L(p)$, где $p \in \Sigma^*$. Множеством состояний у нас будут числа от 0 до $|p|$. Начальным состоянием будет 0, конечным — $|p|$. Функцию переходов можно вычислить при помощи алгоритма, аналогичного алгоритму Кнута-Морриса-Пратта.

$k \leftarrow 0$
for $c \in \Sigma$
  $next[0][c] \leftarrow [c = p[1]]$
for $i \leftarrow 1 \mathinner{..} |p| - 1$
  for $c \in \Sigma$
    if $c = p[i + 1]$
      $next[i][c] \leftarrow i + 1$
    else
      $next[i][c] \leftarrow next[k][c]$
  $k \leftarrow next[k][p[i + 1]]$
for $c \in \Sigma$
  $next[|p|][c] \leftarrow |p|$

Если правую часть присваивания в последней строке заменить на $next[k][c]$, то получится автомат для языка $\Sigma^*\{p\}$. При помощи такого автомата мы можем делать то же самое, что умеет делать алгоритм Кнута-Морриса-Пратта.

Построим теперь детерминированный автомат для языка $L(P)$. Так же, как и в алгоритме Ахо-Корасик, построим бор для множества P и найдем функцию *fin*. Множеством состояний у нас будет множество узлов бора V. Начальным состоянием будет корень бора *root*. Множеством конечных состояний будет множество тех и только тех узлов, для которых функция *fin* истинна. Теперь можно написать рекурсивную процедуру вычисления значения функции перехода для заданных аргументов, а также функцию вычисления суффиксных ссылок.

$calc(c)$
  if $suflink[c] =$ **nil**
    if $parent[c] = root$
      $suflink[c] \leftarrow root$
    else
      $suflink[c] \leftarrow comp(calc(parent[c]), char[c])$
  return $suflink[c]$

$comp(c, d)$
  if $next[c][d] =$ **nil**
    if $c = root$
      $next[c][d] \leftarrow root$
    else
      $next[c][d] \leftarrow comp(calc(c), d)$
  return $next[c][d]$

Главная процедура оформляется следующим образом:

for $c \in V$
  for $d \in \Sigma$
    if $fin[c]$
      $next[c][d] \leftarrow c$
    else
      $comp(c, d)$

Можно модифицировать этот автомат, чтобы он распознавал язык $\Sigma^* P$, если заменить последний **if** на вызов функции $comp(c, d)$, и положить множество конечных состояний равным множеству тех и только тех узлов, для которых функция *lift* отлична от **nil**.

Автоматная динамика

Пусть у нас задан регулярный язык L. Мы хотим узнать, сколько слов длины l принадлежат

этому языку.

Задача решается при помощи динамического программирования. Построим детерминированный автомат для языка L. Обозначим за $a_{i,j}$ количество строк длины j, при обработке которых автомат перейдет из начального состояния s в состояние i. Тогда $a_{i,0} = [s = i]$; для $j > 0$:

$$a_{i,j} = \sum_{\substack{k\in S\\ c\in\Sigma\\ f(k,c)=i}} a_{k,j-1}.$$

Ответом будет являться

$$\sum_{k\in T} a_{k,l}.$$

Отсюда нетрудно получить процедуру для подсчета количества строк языка длины не больше l, а также аналогичные процедуры для вариаций этой задачи.

Контекстно-свободные языки и грамматики

Контекстно-свободная грамматика G — это четверка $\langle T, N, S, P\rangle$, где T — множество терминалов, N — множество нетерминалов, S — стартовый нетерминал, P — множество продукций. При этом необходимо, чтобы $T \cap N = \emptyset$ и $S \in N$. Продукция — это пара $\langle A, \alpha\rangle$, где $A \in N$ и $\alpha \in V^*$, где $V = T \cup N$. Если $\alpha B\gamma \in V^*$, и у нас есть продукция $B \to \beta$, то говорят, что строка $\alpha\beta\gamma$ порождается из строки $\alpha B\gamma$ ($\alpha B\gamma \to \alpha\beta\gamma$). Как правило, грамматика G подразумевается, поэтому она часто опускается в обозначениях. Определим отношение выводимости $\to^*$ следующим образом: $\alpha \to^* \beta \Leftrightarrow (\alpha = \beta) \vee (\exists\gamma\colon \alpha \to^* \gamma \wedge \gamma \to \beta)$.

Контекстно-свободная грамматика G задает соответствующий ей контекстно-свободный язык $L(G)$, который состоит из всех тех и только тех строк терминалов, которые выводятся из стартового символа.

Нормальная форма Хомского

Нетерминал A называется полезным, если существует вывод вида $S \to^* \alpha A\beta \to^* w$, где $w \in \Sigma^*$. Грамматика имеет нормальную форму Хомского (Chomsky) если у нее все нетерминалы полезные и все продукции имеют одну из двух следующих форм:

- $A \to BC$
- $A \to a$

Здесь A, B и C — нетерминалы, a — терминал.

Если пустое слово принадлежит грамматике, то очевидно, что ее нельзя записать в нормальной форме Хомского. Однако любую грамматике, не содержащую пустого слова, можно преобразовать в нормальную форму Хомского. Для этого надо сначала все продукции, у которых в правой части больше двух символов, преобразовать в продукции первого вида, затем удалить продукции вида $A \to \varepsilon$, затем удалить продукции вида $A \to B$, затем удалить бесполезные нетерминалы. Все эти преобразования можно выполнить за $O(n^2)$ действий (n — длина грамматики). Полученная грамматика будет иметь длину $O(n^2)$.

Алгоритм Кока-Янгера-Касами

Пусть у нас задана контекстно-свободная грамматика G и строка терминалов w этой грамматики. Задача распознавания состоит в определении истинности утверждения $w \in L(G)$. Задача разбора состоит в построении дерева разбора данного слова. Алгоритм Кока-Янгера-Касами (Cocke-Younger-Kasami) позволяет решить эти две задачи за время $O(|w|^3)$.

Приведем нашу грамматику к нормальной форме Хомского. Проверка принадлежности пустой строки грамматике делается тривиально. Будем считать, что $w \neq \varepsilon$. Обозначим за $a_{i,j,k}$ верность утверждения о выводимости строки $w_i \dots w_j$ из нетерминала k. Тогда $a_{1,|w|,S}$ будет являться ответом. Если $i = j$, то $a_{i,j,k}$ будет истинно в том и только в том случае, если есть продукция $k \to w_i$. Если $i < j$, то

$$a_{i,j,k} = \bigvee_{\substack{i\le l<j\\ p\in\Pi}} p = \langle k, k_1k_2\rangle \wedge a_{i,l,k_1} \wedge a_{l+1,j,k_2}.$$

При помощи массива $a_{i,j,k}$ тривиально получается разбор строки w.

Грамматика называется однозначной, если для любого слова из ее языка существует единственный вариант его разбора. Если грамматика G однозначна, то мы можем подсчитать количество слов длины l в этой грамматике при помощи модификации алгоритма Кока-Янгера-Касами. Однозначность грамматики гарантирует, что каждое слово будет подсчитано ровно один раз. Более того, мы можем получить процедуру перечисления всех слов грамматики заданной длины в лексикографическом порядке (то есть сгенерировать по слову его номер среди всех слов грамматики, упорядоченных в лексикографическом порядке). Для этого достаточно написать процедуру подсчета слов грамматики, имеющих заданный префикс. Делается это аналогично алгоритму Кока-Янгера-Касами.

Теория строк

Базовые определения

Зафиксируем множество A, называемое алфавитом. С этого момента мы предполагаем, что все символы берутся из этого алфавита. Определим множество строк длины k следующим образом:

$$A^0 = \{\emptyset\};$$

$$A^k = A \times A^{k-1} \quad \text{для } k > 0.$$

Соответственно, множество всех строк определяется следующим образом:

$$A^* = \bigcup_{i \in \mathbf{N}} A^i.$$

Единственный элемент множества A^0 называется пустой строкой.

Начиная с этого момента маленькими латинскими буквами будут обозначаться элементы множества A, а маленькими греческими буквами будут обозначаться элементы множества A^*.

Для любых a и α определена операция атомарной конкатенации a и α, результатом которой будет строка $a\alpha = (a, \alpha)$. Очевидно, что любая строка β является либо пустой ($\beta = \emptyset$), либо представима в виде $\beta = a\alpha$. Это позволяет вводить определения на строках при помощи индукции.

Например, длина строки α определяется следующим образом: $|\emptyset| = 0$, $|a\alpha| = 1 + |\alpha|$.

Аналогично можно определить конкатенацию строк: $\emptyset\beta = \beta$, $(a\alpha)\beta = a(\alpha\beta)$. Очевидно, что $\alpha\emptyset = \emptyset\alpha = \alpha$ и $\alpha(\beta\gamma) = (\alpha\beta)\gamma$; таким образом, множество всех строк над алфавитом A образует моноид относительно операции конкатенации, при этом нейтральным элементом будет пустая строка.

Будем говорить, что строка α является префиксом строки β, если существует такая строка γ, что $\beta = \alpha\gamma$. Очевидно, что строка γ определяется единственным образом.

Аналогичным образом определяется понятие суффикса: строка α является суффиксом строки β, если существует такая строка γ, что $\beta = \gamma\alpha$.

Наконец, α является фактором строки β, если существуют такие γ и δ, что $\beta = \gamma\alpha\delta$.

Префикс, суффикс и фактор строки α называются собственными, если они отличны от α.

Определим теперь на множестве строк лексикографический порядок. А именно, мы будем писать, что $\alpha < \beta$ в том и только в том случае, если α является собственным префиксом β или если существуют такие γ, a и b, что γa является префиксом α, γb является префиксом β и $a < b$.

Введём теперь функцию префикса и функцию суффикса. За $\mathrm{pref}_i(\alpha)$ обозначим префикс длины i строки α если такой существует, в противном случае значением будет сама строка α:

$$\mathrm{pref}_0(\alpha) = \emptyset,$$

$$\mathrm{pref}_i(\emptyset) = \emptyset,$$

$$\mathrm{pref}_i(a\alpha) = a\,\mathrm{pref}_{i-1}(\alpha) \quad \text{для } i > 0.$$

Аналогичным образом вводится функция суффикса:

$$\mathrm{suff}_i(\alpha) = \alpha, \quad \text{если } i \geq |\alpha|;$$

$$\mathrm{suff}_i(a\alpha) = \mathrm{suff}_i(\alpha), \quad \text{в противном случае.}$$

Строки α и β называются сопряженными, если существуют такие строки γ и δ, что $\alpha = \gamma\delta$ и $\beta = \delta\gamma$.

Строка α называется нетривиальным циклическим сдвигом строки β, если $\alpha = \gamma\delta$ и $\beta = \delta\gamma$ для некоторых непустых строк γ и δ.

Строка α называется примитивной, если для любых непустых строк β и γ, таких, что $\alpha = \beta\gamma$, будет верно, что $\alpha \neq \gamma\beta$.

Простые строки

Строка λ называется простой, если она непустая и она меньше любого своего собственного непустого суффикса, то есть для любых α и β таких, что $\alpha \neq \emptyset$, $\beta \neq \emptyset$, и $\lambda = \alpha\beta$ будет верно, что $\lambda < \beta$.

Строка μ называется предпростой, если она непустая и для любого целого неотрицательного i будет верно, что $\mathrm{pref}_i(\mu) \leq \mathrm{suff}_i(\mu)$.

Расширением строки α длины i называется строка $\mathrm{ext}_i(\alpha) = \mathrm{pref}_i(\alpha^i)$.

*Если строка λ проста, то она предпроста.

*Если строка μ предпроста, то любой непустой префикс μ также предпрост.

*В случае, если размер алфавита бесконечен, строка является предпростой в том и только в том случае, если она является непустым префиксом простой строки.

*Строка μ проста в том и только в том случае, если она меньше всех своих нетривиальных циклических сдвигов.

Очевидно, что если строка простая, то она меньше всех своих нетривиальных циклических сдвигов. Рассмотрим теперь непустой собственный суффикс β строки λ, такой, что $\lambda = \alpha\beta$ для некоторой строки α. Если теперь $\beta \leq \lambda$, то поскольку $\lambda < \beta\alpha$, отсюда получаем, что $\beta \leq \lambda < \beta\alpha$, что влечёт за собой равенство $\lambda = \beta\gamma$ для некоторой строки γ, таким образом $\gamma < \alpha < \alpha\beta = \lambda < \gamma\beta$, что невозможно, так как $|\alpha| = |\gamma|$.

*Если строка λ проста, то строка λ^n предпроста для любого целого положительного n.

*Если строка λ проста, то для любого целого положительного i строка $\mathrm{ext}_i(\lambda)$ предпроста.

*Если строки λ_1 и λ_2 просты и $\lambda_1 < \lambda_2$, то строка $\lambda = \lambda_1\lambda_2$ проста.

Пусть строка α является непустым собственным суффиксом строки λ. Если α является суффиксом λ_2, то тогда $\lambda_1 < \lambda_2 \leq \alpha$; иначе $\alpha = \beta\lambda_2$ для некоторой непустой строки β; отсюда $\lambda_1 < \beta < \alpha$; в обоих случаях $\lambda_1 < \alpha$. Если теперь предположить, что $\alpha \leq \lambda$, то $\lambda_1 < \alpha \leq \lambda = \lambda_1\lambda_2$, то отсюда получаем, что $\alpha = \lambda_1\gamma$ для некоторой γ, такой, что $\gamma \leq \lambda_2$, однако γ является собственным суффиксом λ_2 и $\lambda_2 < \gamma$, что невозможно.

*Для любой строки α существует и единственно разложение вида $\alpha = \lambda_1 \dots \lambda_t$, такое, что все строки λ_i просты и $\lambda_1 \geq \dots \geq \lambda_t$. При этом если $t > 0$, то строка λ_t будет минимальным непустым суффиксом α.

Разобьём строку α на простые строки длины 1 и будем объединять их, пользуясь теоремой о конкатенации простых строк, пока это возможно. Полученное разбиение будет удовлетворять всем требованиям.

Если $t > 0$, то положим λ равным минимальному непустому суффиксу α. Строка λ будет простой по определению; она имеет вид $\lambda = \beta\gamma$, где β является непустым суффиксом одного из λ_i. Таким образом $\lambda_t \leq \lambda_i \leq \beta \leq \beta\gamma = \lambda \leq \lambda_t$, отсюда $\lambda = \lambda_t$. Из утверждения о минимальном суффиксе тривиальным образом следует утверждение о единственности разложения.

*Если $\mu = \alpha c$, $\lambda = \alpha d$ и $c < d$, то если строка μ предпроста, то строка λ проста.

Пусть $0 < i < |\mu|$. По определению предпростой строки $\mathrm{pref}_i(\mu) \leq \mathrm{suff}_i(\mu)$, кроме того $\mathrm{suff}_i(\mu) < \mathrm{suff}_i(\lambda)$, отсюда $\lambda < \mathrm{suff}_i(\lambda)$, что и требовалось.

*Если строка μ предпроста, и имеет разложение $\mu = \lambda_1 \dots \lambda_t$, то $\mu = \mathrm{ext}_{|\lambda'|}(\lambda_1)$.

Достаточно провести индукцию по длине μ. Если $|\mu| = |\lambda_1|$, то утверждение верно. Иначе, поскольку префикс μ длины $|\mu|-1$ предпрост, он является расширением λ_1. Таким образом, $\mu = \lambda_1^k\nu c$, где ν является префиксом λ_1. Поскольку строка μ предпростая, то $\mathrm{pref}_{|\nu|+1}(\mu) \leq \mathrm{suff}_{|\nu|+1}(\mu)$. Если выполнено равенство, то переход доказан. Если же выполнено строгое неравенство, то строка $\lambda_1^k\nu c$ будет простой, что противоречит определению разложения на простые строки.

*Непустая строка μ является предпростой в том и только в том случае, если $\mu = \mathrm{ext}_n(\lambda)$ для некоторой простой строки λ с длиной не больше n. Эта строка λ определяется единственным образом.

Докажем единственность λ. Если $\mu = \mathrm{ext}_n(\lambda_1) = \mathrm{ext}_n(\lambda_2)$, где $|\lambda_1| < |\lambda_2|$, то $\lambda_2 = \lambda_1^k\nu$, где $k > 0$ и ν является непустым префиксом λ_1. Но тогда $\nu < \lambda_2$, поскольку ν является собственным префиксом λ_2, и $\lambda_2 < \nu$, поскольку λ_2 простая, и ν является её собственным непустым суффиксом.

Достаточность условия доказана раньше.

*Если строка α имеет разложение вида $\alpha = \lambda_1 \dots \lambda_t$, её префикс β имеет вид $\beta = \lambda_1 \dots \lambda_{s-1}\gamma$, где γ является префиксом λ_s, и при этом разложение γ имеет вид $\gamma = \lambda'_1 \dots \lambda'_u$, то разложение β имеет вид $\beta = \lambda_1 \dots \lambda_{s-1}\lambda'_1 \dots \lambda'_u$.

*Если разложение строки α имеет вид $\alpha = \lambda_1 \dots \lambda_t$, где $t > 0$, то λ_1 является наиболее длинным простым префиксом α.

Если $\alpha = \lambda\beta$, где $\lambda > \lambda_1$, то мы могли бы получить другое разложение α на простые строки, используя конструкцию, аналогичную доказательству существования.

*Если λ_1 и λ_2 являются простыми строками с длинами не больше n, то $\operatorname{ext}_n(\lambda_1) = \operatorname{ext}_n(\lambda_2)$ тогда и только тогда, когда $\lambda_1 = \lambda_2$.

*Если строка λ проста и имеет длину не больше n, то первый член разложения строки $\operatorname{ext}_n(\lambda)$ совпадает с λ.

*Если строки λ_1 и λ_2 простые и существует такое неотрицательное целое i, что $\operatorname{ext}_i(\lambda_1) < \operatorname{ext}_i(\lambda_2)$, то $\lambda_1 < \lambda_2$.

*Если строки λ_1 и λ_2 простые и $\lambda_1 \geq \lambda_2$, то для любого целого неотрицательного n будет верно, что $\operatorname{ext}_n(\lambda_1) \geq \operatorname{ext}_n(\lambda_2)$.

*Существует биекция между простыми строками длины не больше n и предпростыми строками длины n, которая сопоставляет простой строке λ длины не больше n предпростую строку $\operatorname{ext}_n(\lambda)$ длины n, при этом данной предпростой строке μ длины n сопоставляется первый член её разложения на простые строки. Эта биекция сохраняет порядок.

*Любая строка α длины n представима в виде $\alpha = \theta^d$, где строка θ является примитивной.

*Существует биекция между множеством всех строк длины n и множеством всех пар, каждая из которых состоит из простой строки, длина d которой является делителем n и целого неотрицательного числа, меньшего d. При этой биекции паре (λ, i) сопоставляется строка $\operatorname{cyc}_i(\lambda)^{n/|\lambda|}$, где $\operatorname{cyc}_i(\lambda) = \operatorname{suff}_{|\lambda|-i}(\lambda)\operatorname{pref}_i(\lambda)$.

*Пусть $L_m(n)$ является числом m-арных простых строк длины n.

$$\sum_{d\backslash n} dL_m(d) = m^n, \quad m, n \geq 1.$$

$$L_m(n) = \frac{1}{n}\sum_{d\backslash n} \mu(d) m^{n/d}.$$

Теория графов

Замкнутые полукольца

Замкнутым полукольцом называется четвёрка $\langle S, +, \cdot, \Sigma \rangle$, где $+ : S \to S$ — операция сложения, $\cdot : S \to S$ — операция умножения, и $\Sigma : S^{\mathbf{N}} \to S$ — операция счётного суммирования. Эти операции должны удовлетворять следующим свойствам.

Пара $\langle S, + \rangle$ должна образовывать коммутативный моноид, то есть:

$$(a + b) + c = a + (b + c), \tag{1}$$

$$a + 0 = 0 + a = a, \tag{2}$$

$$a + b = b + a. \tag{3}$$

Здесь 0 — нейтральный элемент по сложению.

Пара $\langle S, \cdot \rangle$ должна образовывать моноид, то есть:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c), \tag{4}$$

$$a \cdot 1 = 1 \cdot a = a. \tag{5}$$

Здесь 1 — нейтральный элемент по умножению.

Операция умножения дистрибутивна относительно сложения:

$$a \cdot (b + c) = a \cdot b + a \cdot c, \tag{5}$$

$$(a + b) \cdot c = a \cdot c + b \cdot c. \tag{6}$$

Операция счётного суммирования является ассоциативной:

$$\Sigma_i a_i = a_0 + \Sigma_i a_{i+1}. \tag{7}$$

Операция счётного суммирования является коммутативной: для любой перестановки π множества натуральных чисел выполняется равенство:

$$\Sigma_i a_i = \Sigma_i a_{\pi(i)}. \tag{8}$$

Наконец, операция умножения является дистрибутивной относительно счётного суммирования:

$$a \cdot (\Sigma_i b_i) = \Sigma_i a \cdot b_i; \tag{9}$$

$$(\Sigma_i a_i) \cdot b = \Sigma_i a_i \cdot b. \tag{10}$$

Указанные десять свойств полностью определяют замкнутое полукольцо.

Определим при помощи этих операций новую операцию замыкания: замыканием элемента a называется элемент a^*, где $a^* = \Sigma_i a^i$, при этом $a^0 = 1$ и $a^{i+1} = a \cdot a^i$. Эта операция будем полезна в дальнейшем.

Суммирование путей в графе

Пусть задана квадратная матрица $w_{i,j}$ с элементами в заданном полукольце. Будем называть вершиной число, являющееся корректным номером строки и столбца этой матрицы. Путём в этой матрице мы называем последовательность из $k + 1$ вершин, где $k \geq 0$. Весом пути $(a_0, \ldots, a_k)$ называется произведение Π_k, где $\Pi_0 = 1$ и $\Pi_{i+1} = \Pi_i \cdot w(a_i, a_{i+1})$. Таким образом, $\Pi_k = w(a_0, a_1) \cdot w(a_1, a_2) \cdot \cdots \cdot w(a_{k-1}, a_k)$ (поскольку умножение ассоциативно, скобки ставить необязательно). Начальной вершиной пути называется вершина a_0, конечной вершиной пути называется вершина a_k. Наша задача состоит в том, чтобы посчитать для каждой пары вершин

i и j счётную сумму $s(i,j)$ всех путей из i в j. В силу коммутативности счётного суммирования определение корректно.

Нетрудно переформулировать эту задачу в графовой постановке, а именно, у нас задан полный простой ориентированный граф с петлями, при этом каждой паре вершин сопоставлен вес ребра между этими двумя вершинами, являющийся элементом полукольца. Для каждого пути определим его вес как произведение весов его рёбер в порядке от первого ребра к последнему. Требуется найти для каждой пары вершин сумму весов путей между ними.

Сделать это можно следующим образом. Пусть искомая матрица является матрицей w, тогда ответ будет в матрице s. Считаем, что размер всех матриц равен $n \times n$.

$a_0 \leftarrow w$
for $k \leftarrow 1 \mathinner{..} n$
  for $i \leftarrow 1 \mathinner{..} n$
    for $j \leftarrow 1 \mathinner{..} n$
      $a_k(i,j) \leftarrow a_{k-1}(i,j) + a_{k-1}(i,k) \cdot a_{k-1}(k,k)^* \cdot a_{k-1}(k,j)$
  $a_k(k,k) \leftarrow 1 + a_k(k,k)$
$s \leftarrow a_n$

Докажем правильность работы алгоритма. Будем доказывать по индукции следующее утверждение: $a_k(i,j)$ является суммой всех непустых путей из i в j, у которых все вершины, кроме первой и последней имеют номера не больше k (такие вершины будем называть промежуточными). Пустой путь из i в i включается в $a_k(i,i)$ в том и только в том случае, если $i \leq k$.

Для $k = 0$ очевидно, что утверждение верно. Докажем, что утверждение верно для заданного k в предположении, что оно верно для $k-1$. Каждый путь из i в j, который должен входить в счётную сумму $a_k(i,j)$ попадает в один и только в один из следующих классов:

- в первом классе находятся все пути, которые не используют в качестве промежуточных вершин вершину k;
- во втором классе находятся все остальные пути.

Очевидно, что в соответствии с определением массива a, суммой всех путей в первом классе будет $a_{k-1}(i,j)$, а также пустой путь, то есть 1, если $i = j = k$. Все пути во втором классе имеют следующий вид: сначала они идут от вершины i до вершины k, используя в качестве промежуточных вершин вершины с номерами, меньшими k; затем они ноль и более раз идут от вершины k до неё самой, опять используя промежуточные вершины с номерами меньше k; и, наконец, они идут от вершины k к вершине j, как и раньше, используя в качестве промежуточных вершин вершины с номерами меньше k. Таким образом, сумма всех таких путей будет равна $a_{k-1}(i,j) + a_{k-1}(i,k)a_{k-1}(k,k)^*a_{k-1}(k,j)$.

Из этого утверждения уже тривиальным образом следует корректность алгоритма, поскольку очевидно, что в конце работы матрица a_n является ответом.

Оценим теперь эффективность алгоритма. Очевидно, что он выполняет n^3+n операций сложения, $2n^3$ операций умножения и n^3 операций замыкания. Нетрудно заметить, что число операций замыкания тривиальным образом уменьшается до n.

Используемая дополнительная память есть n^3 ячеек для хранения элементов полукольца, однако её тривиальным образом можно уменьшить до $2n^2$ ячеек, поскольку на k-ом шаге из массива a используются только матрицы a_{k-1} и a_k.

Алгоритм Уоршелла

Рассмотрим полукольцо $\langle \mathbf{B}, \vee, \wedge, \bigvee \rangle$. Здесь $\mathbf{B} = \{0,1\}$, а три операции определены следующим образом:

$$0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1;$$

$$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1;$$

$$\bigvee_i a_i = \begin{cases} 1, & \text{если существует } i, \text{ такое, что } a_i = 1; \\ 0, & \text{иначе.} \end{cases}$$

Это полукольцо называется булевым полукольцом, поскольку в нём по существу реализованы операции классического булевого кольца.

Матрица, получающаяся в результате работы вышеприведённого алгоритма, называется булевым замыканием данной матрицы.

Для замыкания булевой матрицы как правило используют модификацию вышеприведённого алгоритма, которая называется алгоритмом Уоршелла.

for $k \leftarrow 1 \mathinner{..} n$
  $a_{k,k} \leftarrow 1$
for $k \leftarrow 1 \mathinner{..} n$
  for $i \leftarrow 1 \mathinner{..} n$
    for $j \leftarrow 1 \mathinner{..} n$
      $a_{i,j} \leftarrow a_{i,j} \vee (a_{i,k} \wedge a_{k,j})$

Корректность алгоритма следует из того, что $0^* = 1^* = 1$, а также из того, что величины $a_{i,k}$ и $a_{k,j}$ не изменяются на k-ом шаге, поэтому требуется всего одна матрица, которая вначале равна исходной матрице, а в конце равна её транзитивному замыканию.

В графовой формулировке этот алгоритм строит транзитивное замыкание графа, то есть если нам был дан ориентированный граф, заданный матрицей смежности, то этот алгоритм преобразует его в граф, в котором есть ребро (i, j) в том и только в том случае, если в исходном графе существовал путь от i до j.

В оригинальном алгоритме Уоршелла отсутствовали первые две строки, поэтому алгоритм строил положительное транзитивное замыкание, то есть в замыкании соответствующее ребро присутствовало в том и только в том случае, если в исходном графе из первой вершины во вторую можно было пройти по непустому пути.

Алгоритм Флойда

Рассмотрим полукольцо $\langle \mathbf{R} \cup \{-\infty, +\infty\}, \min, +, \inf \rangle$. Здесь $\mathbf{R}$ является множеством вещественных чисел, $\min$ и $+$ являются операциями минимума и сложения, расширенные соответствующим образом $(a + (-\infty) = (-\infty) + a = -\infty$ для $a < +\infty$, $a + (+\infty) = (+\infty) + a = +\infty)$, а $\inf$ является операцией взятия нижней грани.

Применение алгоритма суммирования путей к этому полукольцу имеет очевидную интерпретацию в терминах графов: если нам дан взвешенный граф, мы для каждой пары вершин найдём длину кратчайшего пути из первой вершины во вторую, при этом весом пути считается сумма весов его рёбер, если же кратчайшего пути нет (то есть для любого пути мы можем найти путь ещё короче), то соответствующее значение будет равно $-\infty$.

Нетрудно понять, как адаптировать алгоритм к случаю неполного графа, в котором отсутствуют некоторые рёбра. Для этого надо положить веса отсутствующих рёбер равными $+\infty$.

Очевидно, что операция замыкания удовлетворяет следующему тождеству:

$$a^* = \begin{cases} -\infty, & \text{если } a < 0; \\ 0, & \text{если } a \geq 0. \end{cases}$$

С учётом этого замечания реализация алгоритма не представляет особого труда.

Нетрудно заметить, что если в исходном графе все веса отличны от $-\infty$, то значение $-\infty$ может появиться только в том случае, когда в исходном графе были циклы отрицательного веса. Поэтому при нахождении кратчайших путей в большинстве случаев используют алгоритм, который либо находит длины кратчайших путей для всех пар вершин, либо сообщает, что в графе есть циклы отрицательного веса.

for $k \leftarrow 1 \mathinner{..} n$
  for $i \leftarrow 1 \mathinner{..} n$
    for $j \leftarrow 1 \mathinner{..} n$
      $a_{i,j} \leftarrow \min(a_{i,j}, a_{i,k} + a_{k,j})$
for $k \leftarrow 1 \mathinner{..} n$
  if $a_{k,k} < 0$
    return false
  else
    $a_{k,k} \leftarrow 0$
return true

Этот алгоритм называется алгоритмом Флойда. Если он возвращает **false** , то в графе есть циклы отрицательного веса. В остальных случаях он находит длины кратчайших путей между всеми парами вершин. Алгоритм корректен по тем же причинам, что и алгоритм Уоршелла. (Оригинальный алгоритм Флойда не обнулял значения на диагонали, поэтому исключал из рассмотрения пустые пути.)