Конспект по Дискретной математике.

Чепелин Вячеслав, Давыдов Иван, Амосов Алексей

Содержание

1 Лекция 1.

1.1 Введение определний

Множество - неопределенное понятие. A - какое-то множество. Мы умеем понимать:

$x \in A$ или $x \notin A$

В множестве несколько одинаковых элементов быть не может(вопрос бессмысленен: $\forall x, \forall A$: $x \in A$ или $x \notin A$).

Будем считать, что есть U ("универсум") и все множества в нем лежат.

Мы можем спокойно работать с множеством натуральных, целых, рациональных, вещественных, иногда комплексных. Обычно будет понятно из контекста какой у нас универсум.

Примеры задания множества

$\mathbb{B} = \{0,1\}$

$C := \{x \in A : P(x) = 1\}$, где P(x) - булевая функция (предикат).

С множеством можно делать много операций:

1) Объединение

$A \cup B = \{x : x \in A$ или $x \in B\}$

2) Пересечние

$A \cap B = \{x : x \in A$ и $x \in B\}$

3) Вычитание

$A \backslash B = \{x : x \in A$ и $x \notin B\}$

4) Исключающее объединение (xor)

$A \oplus B = (A \backslash B) \cup (B \backslash A)$

5) Отрицание

$\bar{A} = U \backslash A$

Произведение множеств:

Декартово(прямое) произведение X и Y обозначается $X \times Y$, на языке кванторов:

$X \times Y = \{(x, y) : x \in X, y \in Y\}$

$A \times A = A^2$

Нет доказательства того, что множество пар существует. В теории множеств это закреплено аксиоматически.

$(A \times B) \times C = A \times (B \times C)$ — пренебрегаем проблемой несоответсвия типов: Вместо $\{a, \{b, c\}\}$ и $\{\{a, b\}, c\}$ считаем их равными $\{a, b, c\}$ На языке C++ вместо $pair(a, pair(b, c))$ и $pair(pair(a, b), c)$ воспринимаем элементы объединения как $tuple(a, b, c)$

Произведение семейства множеств

Пусть есть множество значений $B = \{a, b, c\}$ и множество индексов $A = \{1, 2, 3\}$. Рассмотрим произведение семейства такого, что $\forall \alpha, \beta \in A : B_\alpha = B_\beta = B$

$\prod_{\alpha \in A} B_\alpha$ - множество всех способов выбрать одно из возможных значений B для каждого индекса A

Прим. лек. Элемент такого множества похож на map в C++: В качестве ключа выступает индекс из A, в качестве значения - элемент из B. Произведение выше - множество всех возможных map, которые можно составить при данных A и B.

Дизъюнктное объединение

Непересекающиеся множества называют дизъюнктными. Существует специальный знак объединения дизъюнктных множеств: $\sqcup$

$A \sqcup B$

Знак дизъюнктного объединения используется как синтаксический сахар, чтобы указать, что множества не пересекаются, или напомнить об этом.

Функция

$f : A \to B$

$X \subset Y$ — "мн-во X содержится в Y"; равносильно $x \in X \to x \in Y$

$f \subset A \times B$. Выполнено $\forall x \ \in A \exists! \, y \in B : (x, y) \in f$

$B^A = \{f : A \to B\}$ - множество всех функций перехода из A в B

$\{0, 1\}^A = 2^A = \mathbb{B}^A$ -каждому элементу сопоставили 1 или 0. Удобно брать подмножества.

Инъекция. Если $x \neq y$, то $f(x) \neq f(y)$

Сюръекция. $\forall y \in B : \exists x : f(x) = y$

Биекция = Инъекция + Сюръекция = Взаимнооднозначное соотвествие

Частичная функция.

$f : A_1 \subset A \to B$ - Переводит в B только часть множества A

Другие обозначения

$\varnothing$ - пустое множество

$A^0 = \{<>\}$ - пустой картеж = void = множество с ничего внутри

$\varnothing \times A = \varnothing$

$1 \times A = A$ - зачем-то приклеили каждому элементу ничего

Отношения

(бинарные) отношения между A и B

$R \subset A \times B$ - т.е. отношение R задает граф переходов из A в B

Прим. Функция - частный случай отношения. Не любое отношение является функцией, т.к. из одного аргумента отношение может вернуть множество значений.

Пример 1.

$A, B = \mathbb{N}$

$\leq \{(a, b) : a < b\}$ Обозначается aRb (инфиксная запись). Здесь мы задали критерий принадлежности пары к отношению

$<$ - такое отношение, что a должно быть меньше b.

Пример 2.

$\mathbb{N} \times 2^{\mathbb{N}}$

$R = \in$. $(17, \{1, 2, 17, 256\}) \in R$. $(2, \{x | x - \text{нечетный}\}) \notin R$

Пару $(\mathbb{N}, \mathbb{N})$ рассмотреть нельзя, так как она не принадлежит нашему универсуму.

$R \subset A \times A$ - отношение на A.

Отношения можно изображать в виде ориентированных графов. Виды отношений:

1) **Рефлексивность**.

$\forall x \in A : aRa$, то есть (a,a) входит в наше отношение.

2) **Антирефлексивность**.

$\forall x \in A : a \backslash Ra$, то есть (a,a) не входит в наше отношение.

3) **Симметричность**.

Если aRb, то bRa.

4) **Антисимметричность**

Если $a \neq b$ и aRb, то $a \backslash Rb$.

Эквивалентно $aRb, bRa \rightarrow a = b$

5) **Транзитивность**.

Если aRb и bRc, то aRc.

Классы отношений.

Отношение частичного порядка - 1,4,5 (строгий 2,4,5). Представление на графе будет выглядеть как направленный ациклический граф, но у каждой вершины есть петля

Отношение линейнего порядка - частичный и $\forall a, b : aRb$ или bRa.

Отношение полного порядка - линейный и $\forall$множество X имеет min элемент.

Отношение эвивалентности - 1,3,5.

X, R - эквивалентна на X, тогда $A \in X/R$

Классы эквивалентности

Отношение R на множестве X может разбиться на классы эквивалентности такие, что если a, b в одном классе, то aRb, если a, b в разных классах, то $a\backslash Rb$. Если представить на графе эквивалентное отношение R(наличие ребра), то все компоненты будут полными графами с петлями, а разные компоненты будут принадлежать разным классам эквивалентности.

2 Лекция 2.

2.1 Композиция

Пусть есть бинарные отношения $R \subset X \times Y$; $S \subset Y \times Z$.

Композиция $T = RS$.

Определение композиции:

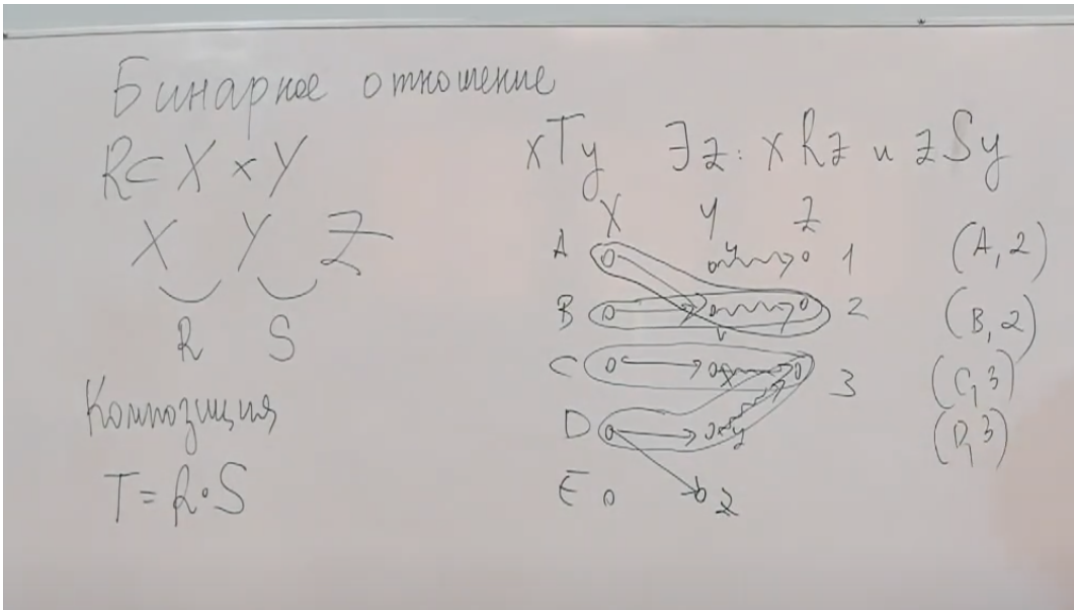
$$xTy \overset{def}{\Longleftrightarrow} \exists z : xRz, zSy$$

"Есть такой z, что через него можно добраться из x в y"

(Следовательно, композиция *не коммутативна*, $RS \neq SR$)

С точки зрения графов:

"Есть такая вершина, z, что можно по ней пройти сначала по ребру, соответствующему отношению R, потом по ребру, соответствующему ребру S"**Пример:** Пусть $R \subset X \times Y$ - отношение "человек x владеет собакой y а $S \subset Y \times Z$ - отношение "собака y носит ошейник x тогда композиция $T = RS$ будет отношением "собака человека x носит ошейник z"



Следовательно, присутствует *ассоциативность*: $T = (AB)C = A(BC)$

Композиция с собой: два элемента (вершины) находятся в отношении-композиции R^n, если между ними имеется путь длины ровно n.

2.2 Транзитивное замыкание

Пусть R - отношение "быть родителем тогда отношение "быть предком транзитивное замыкание R (обозначается R^+)

$$R^+ = \bigcup_{k=1}^{\inf} R^k$$

Если вершина должна включать себя в замыкание (например R - "быть сыном в дереве а R^* - "иметь на поддереве"), то следует объединить с R^0

$$R^0 = (x, x) : x \in X;$$

Это называется *рефлексивно-транзитивным замыканием* и обозначается R^*

$$R^* = \overset{\inf}{\underset{k=0}{U}} R^k$$

R^+ и R^* транзитивны:
$xR^*y \to xR^iy$
$yR^*z \to yR^jz$
Следовательно, $xR^{i+j}z$
А т.к. объединяем до бесконечности, $(i + j) \in [0; \inf)$, Значит из $xR^*y; yR^*z$ следует xR^*z, что является определением транзитивности. Более того, из всех транзитивных отношений S, для которых верно $R \subset S$, R^+ является минимальным (с наим. количеством ребер). Иначе говоря, $\forall T$, где T транзитивно и $R \subset T$, $R \subset T$, выполнено $R^+ \subset T$
Док-во по индукции:
База: $R \subset T$, т.к. это дано.
Докажем, что из $R^i \subset T$ следует $R^{i+1} \subset T$
$xR^{i+1}y \leftrightarrow x(R^iR)y \to \exists z : xRz, zR^iy$ по определению
$\to xTz, zTy$, а т.к. отношение T транзитивно, xTy
То есть $R^{i+1} \subset T$, ЧТД

2.3 Булевые функции:

$\mathbb{B}$ - множество из двух элементов (0, 1 или false, true, орёл и решка и т.д.)

Функция f, принимающая n элементов, принимает декартово произведение множеств, из которых берутся элементы:

$$f : A_1 \times A_2 \times ... \times A_n \longrightarrow B$$

Определение булевой функции:

$$f : \mathbb{B}^n \longrightarrow \mathbb{B}$$

Позже $f : \mathbb{B}^n \longrightarrow \mathbb{B}^k$ (В данном случае можно рассматривать как k отдельных булевых функций $f_1, f_2, ..., f_k$)

2.3.1 n = 0

$\mathbb{B}^0 = \{[]\}$, то есть мощность 1. Функций, принимающих void и возвращающих boolean две: **0** и **1** ("тождественный 0 и 1 соответственно")

2.3.2 n = 1

Функций, принимающих boolean и возвращающих boolean четыре:
$0 \to 0$
$0 \to 1$

x	y	**0**	AND	$!(x \impliedby y)$	x	y	$!(x \implies y)$	XOR	OR
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

x	y	NOR	$x = y$	$!y$	$x \impliedby y$	!x	$x \implies y$	NAND	**1**
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Таблица 1: Все бинарные функции

$1 \to 0$
$1 \to 1$

Свойство функции принимать k аргументов - "арность"(arity) (Унарность, бинарность, тернарность...)

$f(0) = 0$
$f(1) = 1$ - тождественная функция (identity function)

$f(0) = 1$
$f(1) = 0$ - функция отрицания (не x, not x, !x...)

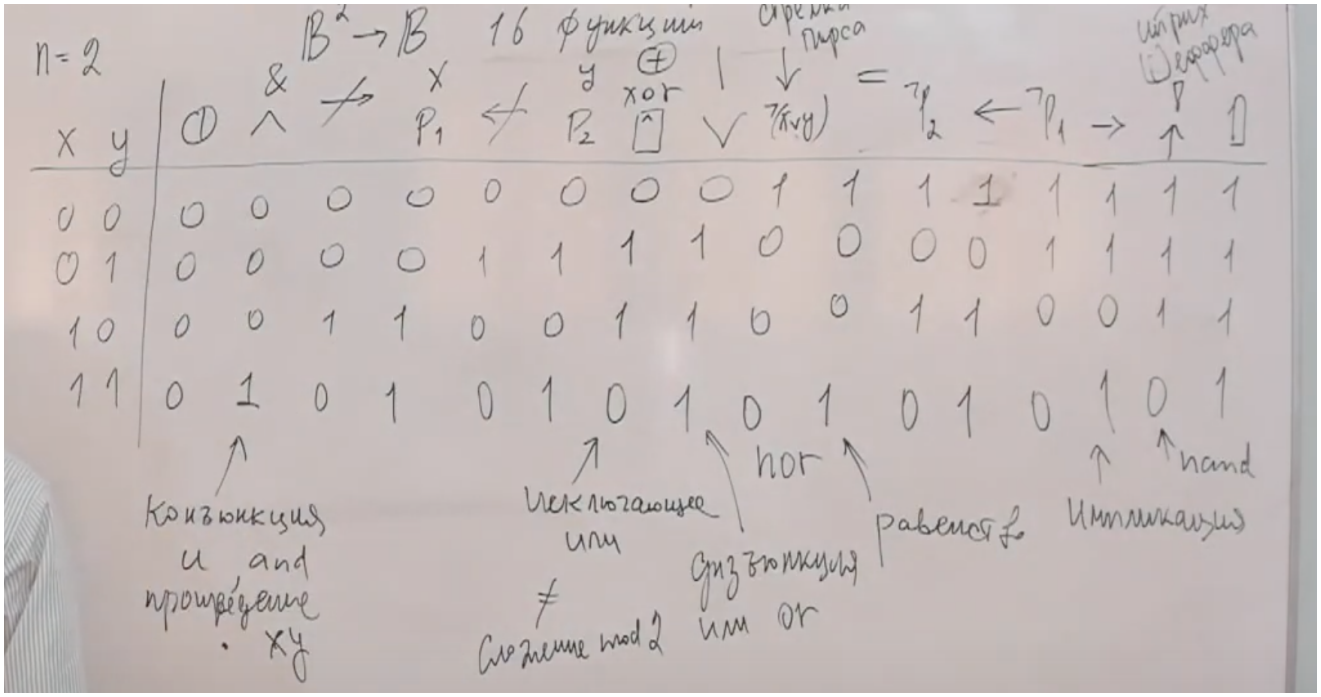
2.3.3 n = 2

Всего 16 возможных функций (бинарные ф-ии - одни из основных) Общая формула количества возможных функций: 2^{2^n}

Таблица в исполнении Станкевича:

x	y	z	med
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Таблица 2: Таблица истинности медианы



2.3.4 n = 3

tldw. Из интересных - медиана, она же "голосование"(возвращает тот аргумент, которого больше, см. таблица 2)

2.3.5 Задание булевых функций

1) Таблица истинности (плюс: есть у всех функций, прямолинейный способ; минус: асимптотика по памяти (2^{2^n})
2) Задание формулой (некой строкой): пусть мы выбрали некоторые базисные функции (система связок), например AND, OR, NOT, XOR. Для каждой базисной функции возьмем обозначение.

Каноническое обозначение: f(x, y)

Инфиксное обозначение: x f y (необходимо задавать приоритеты выполнения функций)

Пример: импликация - OR(NOT(x), y)

"Замыкание множества функций F" - множество всех функций, которые можно составить, используя функции из множества F (обозначается как $\overline{F}$)

Например, $\mathbf{0} \in \overline{XOR}$, но $\mathbf{1} \notin \overline{XOR}$

Теорема о стандартном базисе:

$$\overline{\{AND, OR, NOT\}} = \mathbb{BF}$$

($\mathbb{BF}$ — множество всех булевых функций). Более того, можно избавиться от AND или OR (см. закон де Моргана). Доказательство - существование СКНФ/СДНФ для любой таблицы истинности (тождественный 0 не имеет СДНФ, но можно выразить просто как NOT(OR(A,A)) или AND(A, NOT(A))).

(Пасхалко: есть по крайней мере одна базисная функция F из таблицы 1, для которой справедливо $\overline{F} = \mathbb{BF}$. Можете догадаться, какая? Почему? Есть ли другие?)

2.4 Конъюктивная нормальная форма

Конъюктивная нормальная форма

2.5 Дизъюнктивная нормальная форма

Дизъюнктивная нормальная форма

3 Лекция 3.

3.1 Базисы

Базис - набор булевых функций.

F называется **полным базисом**, если используя только булевые функций из F мы можем задать любую булеву функцию. Пример:

$\{\wedge, \vee, \neg\}$ — стандартный полный базис. Можем выкинуть $\wedge$ либо $\vee$ и он останется полным базисом.

Теорема. Пусть F - полный базис. G - множество функций. $\forall f \in F$: можно задать формулой над G. Тогда G - полный базис.

Доказательство.

Так как F - базис, через него можно выразить любую булеву функцию. Тогда давайте рассмотрим дерево операций над F для какой-то булевой функции и докажем, что для этой булевой функции мы можем построить дерево операций над G. Возьмем дерево операций над F и рассмотрим его конкретную вершину. Это какая-то функция f, про которую мы знаем, что $f \in F$. Раз так, то мы можем заменить ее на ее формулу над G. Сделаем так со всеми вершинами в дереве операций над F и получим дерево операций над G. Раз над G можно задать любую функцию, то G - базис. Q.E.D.

Так мы можем доказать, что $\{\vee, \neg\}$, $\{\wedge, \neg\}$ - полные базисы.

3.2 Полином Жегалкина

$\{\wedge, \oplus, 0\}$ — **арифметический базис.**

Выражение булевой функции в арифметическом базисе — **полином Жегалкина**.

Приведенный полином Жегалкина — вид полинома Жегалкина, в котором опущены все произведения с 0.

Теорема. $\forall$ функции $\exists!$ приведенный полином Жегалкина

Доказательство. Всего функций от n переменных - 2^{n^n}. Полиномов Жегалкина 2^{n^n}. Каждой функции соответсвует полином Жегалкина. У нас не может быть двух записей в виде полинома Жегалкина для одной и той же функции. Доказывается это от противного, если у нас есть два разных полинома Жегалкина Q_1 и Q_2, которые задают одну и ту же функцию, то $Q_1 \oplus Q_2 = 0$. Но раз Q_1 и Q_2 не совпадают, то $Q_1 \oplus Q_2 \neq 0$. Тогда это биекция между двумя множествами. Q.E.D.

3.3 Классы Поста

Классы Поста - это свойства булевых функций, такие, что если мы работаем над базисом, все функции которого принадлежат какому-то классу Поста, то и все функции, достижимые из этого базиса, также лежат в этом классе Поста.

Классы Поста:

1. **Сохраняющие 0** - булевы функции, которые от входного набора состоящего только из 0 выдают 0. $\{\vee, \wedge\}$ — сохраняют ноль.

Очевидно, любая функция над базисом из функций, сохраняющий 0 также будет сохранять 0.

2. **Сохраняющие 1** - булевы функции, которые от входного набора состоящего только из 1 выдают 1. $\{\vee, \wedge\}$ — сохраняют один.

   Также очевидно, что любая функция над базисом из функций, сохраняющий 1 также будет сохранять 1.

3. **Линейность** - функция линейна, если ее представление в виде полинома Жегалкина не имеет в записи $\wedge$.

   Доказательство линейности функции над базисом, состоящим из линейных функций также очевидно, если представить данную функцию в виде полинома Жегалкина, это упражнение я оставлю читателю.

4. **Монотонность** - функция монотонна, если при увеличении любого аргумента результат не уменьшается, то есть $\forall$ 2 наборов аргументов $x_1, \ldots, x_n$ и $y_1, \ldots, y_n$, таких, что $x_i \leq y_i$, верно $f(x_1, x_2, \ldots, x_n) \leq f(y_1, y_2, \ldots, y_n)$.

   Доказательство монотонности функции над базисом, состоящим из монотонных функций: Рассмотрим дерево операций для функии f, которая построена над базисом из монотонных функций. Тогда рассмотрим увеличение какого-то аргумента и докажем, что в этом случае значение f не уменьшится. Предположим это не так, значит значение в корне уменьшилось. Но так как в корне у нас монотонная функция, то значение какого-то из его сыновей уменьшилось. Продолжим этот спуск и получим, что значение какого-то аргумента уменьшилось. Противоречие. Q.E.D.

5. **Самодвойственность** - функция самодвойственна, если $\forall$ набора аргументов $x_1, \ldots x_n$, верно $F(x_1, \ldots x_n) \neq F(\overline{x_1} \ldots \overline{x_n})$. самодвойственности функции над базисом, состоящим из самодвойственных функций. Рассмотрим дерево операций для функии f, которая построена над базисом из самодвойственных функций. Тогда рассмотрим изменение всех аргументов и докажем, что в этом случае значение f также изменится. Начнем рассматривать вершины снизу вверх. Так как каждая вершина - это самодвойственная функция, и все аргументы поменялись на противополжные, то значение и в этой вершине сменится. Тогда по итогу в корне значение также сменится на противоположное. Q.E.D.

3.4 Критерий Поста

F – полный базис $\Leftrightarrow$ нем есть не самодвойственная, не сохраняющая ноль, не сохраняющая один, не линейная, не монотонная.

Доказательство.

1) Заметим, что необходимость этого утверждения была доказаны выше для каждого класса отдельно.

2) Докажем, что если набор F не содержится полностью ни в одном из данных классов, то он является полным.

   (a) Рассмотрим функцию, не сохраняющую ноль — f_0. Тогда $f_0(1)$ может принимать два значения:

      1. $f_0(1) = 1$, тогда $f_0(x, x, x, \ldots, x) = 1$
      2. $f_0(1) = 0$, тогда $f_0(x, x, x, \ldots, x) = \neg x$

      .

   (b) Рассмотрим функцию, не сохраняющую один — f_1. Тогда $f_1(0)$ может принимать два значения:

1. $f_1(0) = 0$, тогда $f_1(x, x, x, \ldots, x) = 0$
2. $f_1(0) = 1$, тогда $f_1(x, x, x, \ldots, x) = \neg x$

Таким образом, возможны четыре варианта:

a2b2. Мы получили функцию $\neg$. Используем несамодвойственную функцию f_s. По определению, найдется такой вектор x_0, что $f_s(x_0) = f_s(\overline{x_0})$. Пусть $x_0 = (x_{0_1}, x_{0_2}, \ldots, x_{0_k})$.

Рассмотрим $f_s(x^{x_{0_1}}, x^{x_{0_2}}, \ldots, x^{x_{0_k}})$, где либо $x^{x_{0_i}} = x$, при $x_{0_i} = 1$. Либо $x^{x_{0_i}} = \neg x$, при $x_{0_i} = 0$. Нетрудно заметить, что $f_s(0) = f_s(1) \Rightarrow f_s = const$. Таким образом мы получили одну из констант. Можем получить вторую, взяв отрицание от первой.

a2b1. Мы получили $\neg$ и $0\Rightarrow$ имеем константу, равную 1, поскольку $\neg 0 = 1$

a1b2. Мы получили $\neg$ и $1\Rightarrow$ имеем константу, равную 0, поскольку $\neg 1 = 0$

a1b1. Мы получили 1 и 0. Рассмотрим немонотонную функцию f_m. Существуют такие $x_1, x_2, \ldots, x_n$, что $f_m(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) = 1$, $f_m(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) = 0$, зафиксируем все $x_1, x_2, \ldots, x_n$, тогда $fm(x_1, x_2, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) = \neg x$. Такие $x_1, x_2, \ldots, x_n$ являются x-ами при которых ломается монотонность

В итоге имеем три функции: $\neg, 0, 1$ в каждом из случаев. Покажем, что из них можно получить или или и.

Используем нелинейную функцию f_l. Среди нелинейных членов f_l (ее представления в виде полинома Жегалкина), выберем тот, в котором минимальное количество элементов.

Все аргументы кроме двух в этом члене приравняем единице, оставшиеся два назовем x_1 и x_2.

Все элементы, не входящие в данный член, примем равными нулю. Тогда эта функция будет представима в виде $g_l = x1x2[\oplus x1][\oplus x2][\oplus 1]$, где в квадратных скобках указаны члены, которые могут и не присутствовать (остальные слагаемые будут равны нулю, поскольку в них есть как минимум один аргумент, не входящий в выбранный член, так как в выбранном члене минимальное число элементов).

Рассмотрим несколько вариантов:

1) Присутствует член $\oplus 1$.

   Возьмем отрицание от g_l и член $\oplus 1$ исчезнет.

2) Присутствуют три члена, без $\oplus 1$:

   $g_l = x1x2 \oplus x1 \oplus x2$. Составив таблицу истинности для этой функции нетрудно заметить, что она эквивалентна функции или.

3) Присутствуют два члена, без $\oplus 1$.

   Построив две таблицы истинности для двух различных вариантов, заметим, что в обоих случаях функция истинна только в одной точке, следовательно, СДНФ функции g_l будет состоять только из одного члена. Если это так, то не составляет труда выразить или через не и g_l.

4) Присутствует один член. Выразим $\wedge$ через $\neg$ и g_l аналогично пункту 3.

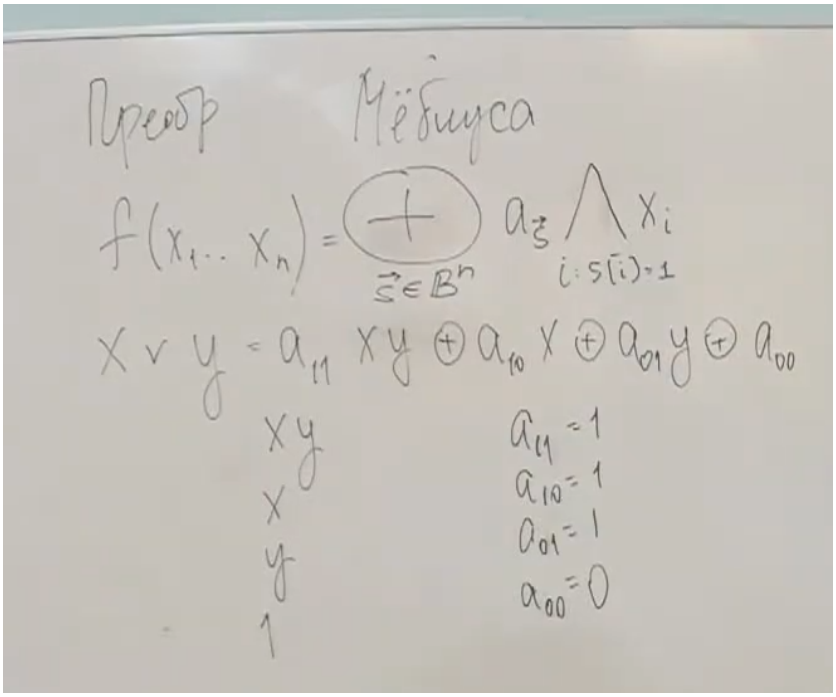
В итоге получим функцию $\neg$,а также функцию или либо и. Любую булеву функцию, не равную тождественному нулю, можно представить в форме СДНФ, то есть выразить в данном базисе.

Значит, полученные функции образуют полную систему, поскольку с их помощью можно выразить любую булеву функцию. Из этого следует, что F — полная система функций, что и требовалось доказать.

4 Лекция 4.

4.1 Преобразование Мёбиуса

Преобразование Мёбиуса позволяет переводить коэффициенты из полинома Жегалкина в таблицу истинности и наоборот.



Давайте заведем бинарные коэффициенты для полинома Жегалкина, которые будут отражать, берем мы или не берем то или иное слагаемое. Всего таких коэффицентов будет 2^n, столько же, сколько и всего возможных членов полинома Жегалкина. Пронумеруем их так, если в члене нашего полинома есть iый член, то в индексе коэффицента на iом месте будет 1, иначе 0 (например для произведения xz в полинома Жегалкина для функции от трех переменных индекс коэффицента будет 101). Получаем, что индекс коэффицента - булевый вектор какой-то длины, а его уже можно однозначто перевести в числовой коэффицент через двоичную систему счисления.

Посмотрим на формулу с картиночки и начнем осознавать, откуда она взялась: $f(x_1 \dots x_n) = \bigoplus_{\overrightarrow{s} \in \mathbb{B}^n} a_{\overrightarrow{s}} \bigwedge_{i:s[i]=1} x_i$ - это просто формула полинома Жегалкина. Давайте разберемся, когда у нас $a_{\overrightarrow{s}} \bigwedge_{i:s[i]=1} x_i = 1$, очевидно это происходит, только в том, случае, если выполняется это: $\forall i : s[i] = 1, x_i = 1$. Назовем такое отношение **доминированием** и будем обозначать как $\leq$. Тогда нашу формулу можно переписать как:

$$\bigoplus_{\overrightarrow{s} \in \mathbb{B}^n} a_{\overrightarrow{s}} \bigwedge_{i:s[i]=1} x_i = \bigoplus_{\overrightarrow{s} \leq \overrightarrow{x}} a_{\overrightarrow{s}}$$

Отлично у нас получилась формула перехода от одного базиса к другому, и, как и в линейной алгебре, ее можно записать с помощью матрицы перехода, независимо от того, какие функции у нас были. Например для четырех переменных она выглядит так:

1	0	0	0
1	1	0	0
1	0	1	0
1	1	1	1

При умножении данной таблицы на коэффиценты в полиноме Жегалкина, мы получим таблицу истинности, сейчас мы докажем, что при повторном умножении, мы обратно получим коэффиценты полинома Жегалкина, то есть докажем, что такая таблица обратна сама себе.

Давайте доказывать, хотим такую формулу:

$$a_t = \bigoplus_{x \leq t} f_x$$

Но мы уже знаем, что верно:

$$\bigoplus_{x \leq t} f_x = \bigoplus_{x \leq t} \bigoplus_{s \leq x} a_s = \bigoplus_{x,s:s \leq x \leq t} a_s$$

Хорошо тогда давайте разберемся сколько раз мы взяли конкретный s. Если t не доминирует над s, то очевидно 0. Иначе мы взяли a_s 2 в степени количества битов i, таких что в $t[i] = 1$ и $s[i] = 0$. Очевидно что это число нечетное, только если их 0, то есть на самом деле a_s влияет на сумму, только если $s = t$, а значит справа у нас изначально было написано a_t. *QED.*

4.2 Функциональные элементы

Рассмотрим новый способ представления булевых схем - в виде ориентированного графа. Вершины в нем - это булевые значения, а ребра - операции с ними.

- длина максимального пути между вершиной, названной входной и вершиной, названной выходной.

4.2.1 Топологическая сортировка

Топологическая сортировка - такая сортировка вершин ориентированного графа, что если в ней вершина u стоит раньше вершины v, то не сушествует ребра из v в u.

Теорема. Топологическая сортировка существует, если в графе нет циклов и наоборот.

В левую сторону доказать совсем легко, если в графе есть цикл и топологическая сортировка в нем тоже существует, то мы можем найти самую правую вершину этого цикла в топологической сортировке и тогда ребро из нее будет вести влево, что противоречит определению топологической сортировки.

В правую сторону доказать немного сложнее, но все также нетрудно. Найдем вершину, из которой не ведет ни одного ребра в другие вершины, если такой нет, то цикл обязательно существует ведь мы можем запуститься из любой вершины и ходить по ребрам, пока не вернемся в уже посещенную вершину и получим цикл. Тогда удалим эту вершину, а в топологическую сортировку допишем ее слева, что не может сломать ее, ведь из этой вершины нет ни одного ребра в те, которые мы допишем позже. Продолжая этот алгоритм, мы и получим топологическую сортировку.

5 Лекция 5.

5.1 Сумматор

Я пропущу сборку обычного сумматора, для него вам нужно просто придумать 2 схемы - битовый сумматор без переполнения и с переполнением и использовать второй n раз.

5.2 Двоичный каскадный сумматор

Примечателен этот вариант сумматора тем, что имеет глубину $O(\log_2 n)$.

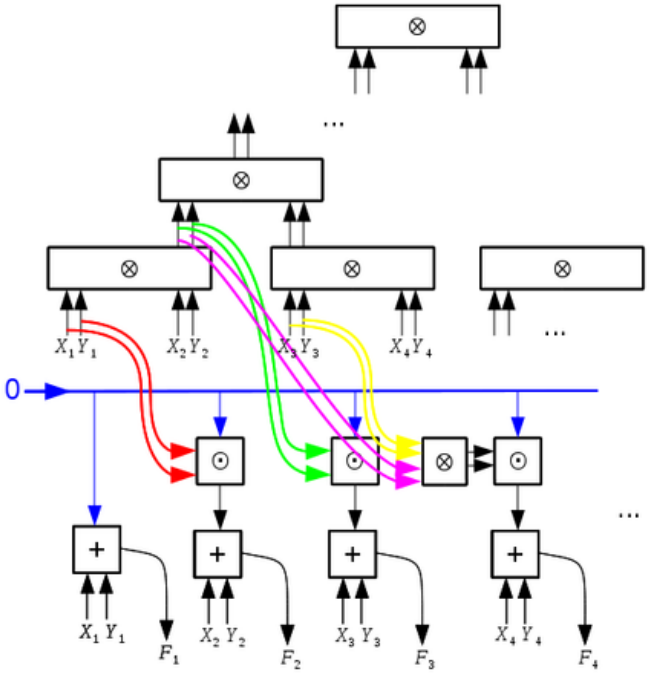
Чтобы его собрать для начала разберемся с тем, как устроен битовый сумматор с переполнением.

1. Если в нем складываются биты 0 и 0, то неважно, что ему пришло в качестве переполнения, он передаст далее 0, этот вариант мы назовем k.
2. Если в нем складываются биты 1 и 0, то он передаст дальше то, что пришло ему в качестве переполнения, этот вариант мы назовем p.
3. Если в нем складываются биты 1 и 1, то неважно, что пришло ему в качестве переполнения дальше он передаст 1, этот вариант мы называем g.

Тогда мы можем завести операцию композиции на этих вариантах, таблица которой будет выглядеть так:

1\2	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

Тогда, если мы сможем быстро считать переполнение, которое приходит к конкретному биту, то и чему равен бит в итоге, узнать будет несложно. Чтобы насчитывать такое и получить логарифмическую глубину мы возьмем за основу структуру дерева отрезков, только теперь она будет работать немного по-другому.



Функции у конретной вершины в дереве будут такие:

1. Получить от предка копозицию прошлых переполнений, которые не являются частью этого поддерева(если корень то получить k).
2. Если вершина является листом, то передать полученную от предка композицию дальше в алгоритм, а предку вернуть значения переполнения этого бита.
3. Передать левому ребенку композицию, полученную у предка, обратно получить композицию переполнения всего правого поддерева.
4. Сделать композицию результата предка и левого сына и передать все это правому.
5. Получить от правого композицию переполнений правого поддерева, сделать композицию предка, левого поддерева и правого поддерева и вернуть все это предку.

Этот алгоритм насчитает все переполнения, а глубина у него $O(\log_2 n)$, так как переполнения любого бита сначала будет только подниматься от листа до какой-то вершины, а потом опускаться обратно до какого-то листа, и пройдет путь не более $2\log_2 n$.

Полученные переполнения мы по отдельности сложим с битами обычным битовым сумматором (считая k нулем, а g единицей), за $O(n)$ и получим сумму.

5.3 Вычитатор

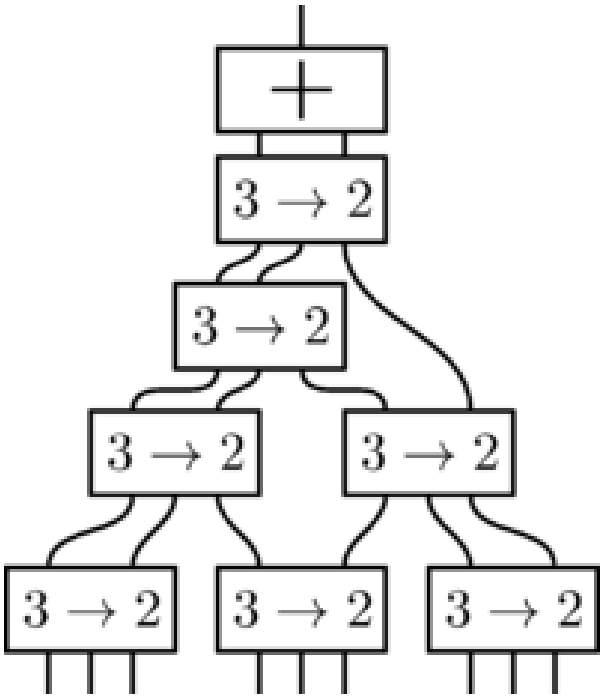
Если мы вспомним формулу обратного числа в дополнении до 2, то она будет равна: $-y = \sim y + 1$. Инверсию всех битов можно сделать заранее, за $O(n)$, добавить единицу можно не делая отдельное сложение с 1, если мы передадим на корень не k, а g, что и будет косвенно означать сложение с 1.

5.4 Умножатор

При умножении двух n битных чисел получается $2n$ битное число, тогда для начала вспомним, как мы умножаем числа в столбик. При умножении чисел длины n в столбик мы складываем n чисел длины не более $2n$. Воспользуемся деревом Уоллеса для того, чтобы свести сумму n чисел к сумме двух, которые мы уже честно сложим двоичным каскадным сумматором.

5.4.1 Дерево Уоллеса

Предположим, что у нас есть три числа, которые мы хотим сложить, длины не более n, сумма которых по длине также не превысит n, тогда мы можем разбить их на биты и каждый бит сложить битовым сумматором с переполнением. Полученные биты без переполнения будут объединены и станут первым числом, объединенные биты переполенения сдвинутые на 1 влево станут вторым числом. Эти 2 числа по сумме будут равным трем до, значит с помощью такой схемы мы за глубину 1 можем уменьшить количество элементов, которые нам надо сложить.



6 Лекция 6.

6.1 Нижняя граница на размер функциональной схемы

Линейные программы - еще один способ задать булеву функцию, в котором удобнее ее рассматривать для всяких оценок. Выберем какой-то базис и переменный, n из которых будут начальными, которые мы будем получать на вход, для остальных заведем правила, по которым они будут вычисляться из предыдущих (расположены эти элементы будут в топологическом порядке).

Пусть у нас есть всего 1 операция в базисе - стрелка Пирса, тогда давайте оценим количество возможных линейных программ для n-арной функции с t операциями. Первая операция может быть одной из n^2, вторая одной из $(n+1)^2$ и так далее. Тогда сумма:

$$n^2 * (n+1)^2 \cdots * (n+t)^2 \leq (n+t)^{2t}$$

Подставим в эту формулу константу, например $\frac{2^n}{3n}$:

$$\left(n + \frac{2^n}{3n}\right)^{\frac{2*2^n}{3n}}$$

Теперь если мы подсчитаем отношения этого количества ко всем возможным функциям, а их я напомню - 2^{2^n}, то поймете, что оно довольно быстро стремится к 0.

6.2 Верхняя граница на размер функциональной схемы

Пусть у нас есть n-арная булева функция для которой мы хотим построить функциональную схему. Заведем две переменные k и s от которых в итоге и будет зависить размер нашей схемы.

Начнем с того, что представим нашу таблицу истинности, как таблицу с 2^k строчками и 2^{n-k} столбцами, просто выделив $n-k$ элементов как столбцы. Теперь объединим строчки в блоки по s строк. Хорошо, давайте научимся строить схему для таблицы из одного столбца и s строк. Вариантов, в каком блоке такая подтаблица может оказаться - $\frac{2^k}{s}$, вариантов для каждого блока - 2^s. Построим демультиплексор, на k элементах, чтобы определять в какой из строк мы оказались, на это уйдет 2^k элементов схемы, далее для каждого из типов блоков соберем элементы или, которые в сумме позволят его получить, на это уйдет максимум s умноженное на количество вариаций блоков, то есть:

$$\frac{2^k}{s} * 2^s * s = 2^{k+s}$$

Хорошо, теперь нам нужно собрать столбец целиком, для этого мы возьмем полученные блоки и сделаем с нужными или, всего элеметов или у нас на это уйдет:

$$2^{n-k} * \frac{2^k}{s}$$

Осталось собрать столбики в таблицу, чтобы это сделать снова построим демультиплексор, но теперь на $n-k$ вершинах, на что уйдет 2^{n-k} элементов, а потом сделаем или для каждого

выхода демультиплексора, с нужным для него столбцом, на что уйдет всего 2^{n-k} элементов. После этого все результаты надо объединить за еще 2^{n-k} элементов.

Получается всего мы потратили:

$$2^k + 2^{n-k} + 2^{k+s} + 2^{n-k} * \frac{2^k}{s} + 2^{n-k} = O(2^{k+s} + \frac{2^n}{s})$$

Если выбрать $k = \log_2 n$, а $k = n - 2\log_2 n$, то мы и получим сложность порядка $O(\frac{2^n}{n})$.

7 Лекция 7.

7.1 Префиксный код

Префиксный код - принцип однозначного кодирования, при котором для любых двух кодов a и b, ни a не является префиксом b, ни b не является префиксом a.

Доказательство того, что декодирование в таком случае можно сделать однозначно настолько простое, что я оставлю его читателю.

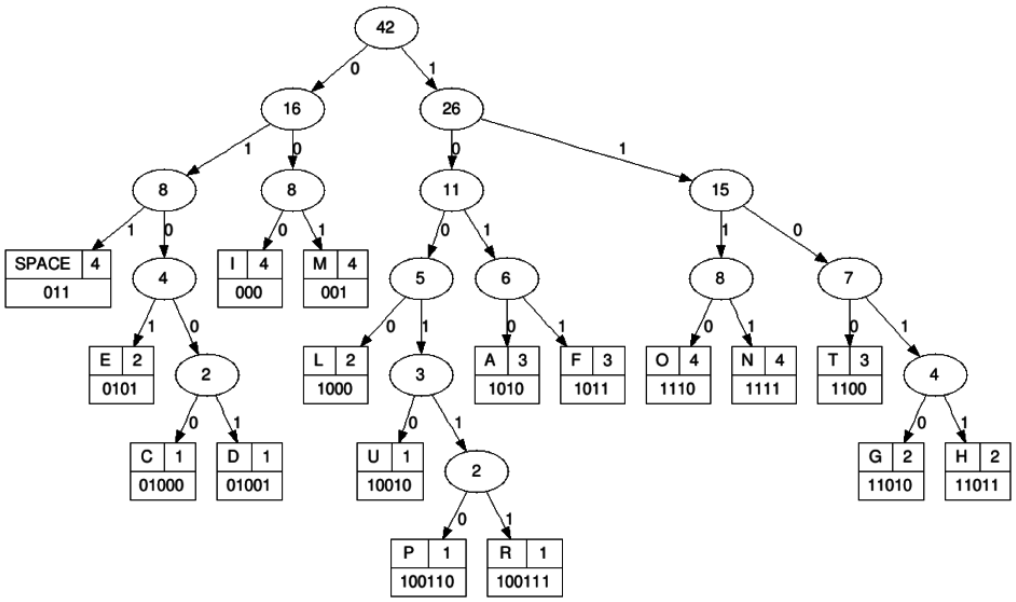
7.2 Код Хаффмана

Код Хаффмана - самый эффективный способ кодировать информацию в том случае, если каждый символ должен кодироваться целым числом бит и, если мы напрямую не пользуемся закономерностями данных, которые кодируем.

Чтобы закодировать алфавит кодом Хаффмана мы должны иметь частоту появления каждого символа в тексте, который мы кодируем. Построим дерево Хаффмана на основе этих частот, каждый символ станет листом со своим весом, равным частоте появления этого символа в тексте, далее мы будем выполнять следующий алгоритм:

1. Находим два элмента без родителя с наименьшим весом
2. Создаем им родителя с весом, равным суммарному весу этих вершин

Чтобы теперь задать бинарный код для каждого символа напишем на разных ребрах из одной вершины 0 и 1 и теперь, чтобы получить код для какого-то символа, нужно пройтись от корня до этого листа, и конкатенация цифр на ребрах и будет его кодом.



Докажем, что это кодирование оптимальнее любого другого.

Лемма В дереве какого-то из оптимальных кодирований два символа с минимальным весом находятся на максимальной глубине и имеют общего родителя. **Доказательство** Пусть это

не так, посмотрим на 2 символа на максимальной глубине, имеющие общего родителя (такие обязательно найдутся, так как у нас нет вершин у которых всего 1 потомок, иначе их можно удалить). Эти вершины должны иметь максимальный вес, ведь если это не так, то мы можем поменять их с другими, которые имеют больший вес, а значит предыдущий код был не оптимальным. *QED*

Заметим, что раз 2 вершины с максимальным весом находятся на максимальной глубине, то мы можем заменить их на их родителя, длина кода которого на 1 меньше, а вес тогда будет равен сумме их весов, получается мы уменьшаем общее количество символов на 1. Как раз так и строится дерево Хаффмана, а значит код Хаффмана является оптимальным.

7.3 Неравенство Крафта-Макмиллана

В общем случае неравенство Крафта-Макмиллана утверждает: Для того, чтобы для набора длин кодовых слов алфавита мощностью r: $l_1, l_2 \dots l_n$ существовал однозначно декодируемый код, необходимо и достаточно

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$

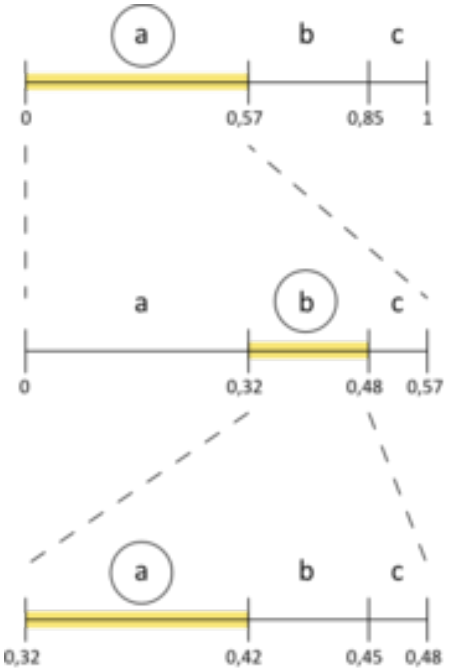
За доказательством сюда.

8 Лекция 8.

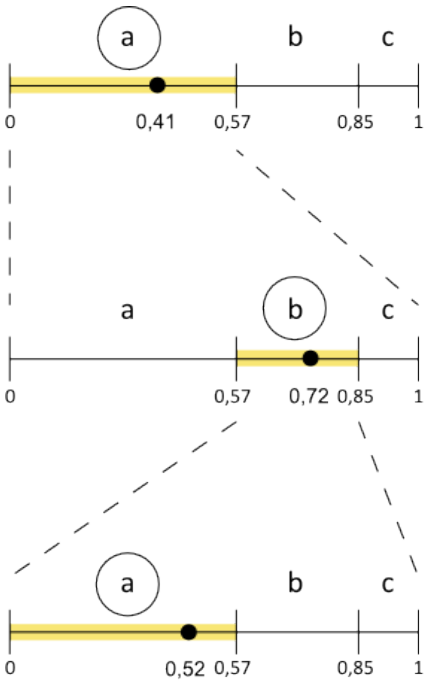
8.1 Арифметическое кодирование

Арифметическое кодирование является улучшенной версией кода Хаффмана, потому что позволяет кодировать символы нецелым числом бит. Алгоритм кодирования устроен так:

1. Начните работу с отрезка от 0 до 1.
2. Поделите отрезок на части в соотношении, равном отношению частот символов
3. В зависимости от текущего символа в кодируемой строке перейдите к нужному отрезку
4. Если строка не кончилась перейдите к пункту 2, иначе выберите на полученном отрезке точку вида $\frac{p}{2^q}$ с наименьшим q, вашим кодом будет двоичный вектор длины q, в котором записано число p



Декодирование работает точно так же, мы снова делим отрезок на подотрезки, но теперь мы уже знаем в какой из них нам нужно, так как мы знаем точку, которая ответу принадлежит, перейдем в нужный отрезок и продолжим алгоритм.



8.2 LZW

Алгоритм LZW - алгоритм для сжимания данных, основанный на поиске повторяющихся кусков данных и сжимания их. Для сохранения данных о закономерностях предлагается использования буфера. В общем виде алгоритм действия такой:

1. Добавление всех символов алфавита в буфер с наименьшими номерами
2. Инициализируем скользящее окно X как пустую строку
3. Обработка очередной символа S из текста.
4. Если S — это символ конца сообщения, то выдать код для X, иначе:

   4.1. Фраза XS есть в словаре: Приписать к X символ S,
   Перейти к шагу 3.

   4.2 Фразы XS нет в словаре: Выдать код для входной фразы X, добавить XY в словарь и присвоить X значение S. Перейти к Шагу 3.

Алгоритм декодирования:

1. Все возможные символы заносятся в словарь. В скользящее окно X заносится первый код декодируемого сообщения.
2. Считать очередной код S из сообщения.
3. Если S — это конец сообщения, то выдать символ, соответствующий коду X, иначе:

   3.1 Если фразы под кодом XS нет в словаре, вывести фразу, соответствующую коду X, а фразу с кодом XS занести в словарь.

   3.2 Иначе присвоить входной фразе код XS и перейти к Шагу 2 .

8.3 Алгоритм Барроуза-Уиллера

Алгоритм Барроуза-Уиллера также как и LZW производит поиск паттернов в данных, но вместо того, чтобы сжимать их, он просто превращает их в последовательности из одинаковых символов для увеличения эффективности дальнейшего сжатия. Работает он так:

1. Выписываем все циклические сдвиги данных в отсортированном порядке.
2. В качестве результата берем последний столбец, для декодирования запоминаем, какой из символов этого столбца является символом исходной строки.

Вход	Перестановки	Отсортированные	Выход
НАНАНА&	НАНАНА&	АНАНА&**Н**	**ННН&ААА**
	&НАНАНА	АНА&НА**Н**	
	А&НАНАН	А&НАНА**Н**	
	НА&НАНА	НАНАНА**&**	
	АНА&НАН	НАНА&Н**А**	
	НАНА&НА	НА&НАН**А**	
	АНАНА&Н	&НАНАН**А**	

Почему так происходит: Предположим у нас есть паттерн s, который встречается много раз в исходной строке. Тогда если мы возьмем и отрежем у этого паттерна первый символ, пусть он будет a, тогда в тех случаях, когда этот символ будет в конце циклического сдвига, остальная часть паттерна будет в его начале, а значит, с большой вероятностью, эти символы будут идти подряд.

Декодирование будет устроено так:

1. Изначально у нас будет записан только результат кодирования, который мы отсортируем
2. Приписываем к полученной таблице слева результат кодирования и сортируем
3. Если таблица не готова, то перейти к шагу 2, иначе мы берем строку.

Это декодирование основано на том, что у нас выписаны все циклические сдвиги, а значит на самом деле переход от iого столбца к $i+1$ому это какая-то перестановка, причем перестановка постоянная. Тогда эту перестановку можно получить если перебросить первый столбец в конец, у нас получится пара - отсортированный столбец и наш результат левее, значит если так повторять, то мы просто будем применять эту перестановку и получать предыдущий столбец. Сложность декодирования выходит $O(n^2 \log n)$.

Но если мы 1 раз построим перестановку и будем применять ее только к элементу последнего столбца, который принадлежит искомой строке, то мы можем восстановить закодированные данные без восстановления всей таблицы, что будет работать уже за $n \log n$.

9 Лекция 9.

9.1 Избыточное кодирование

Избыточное кодирование - кодирования с использованием большего количества информации, чем находится в самом сообщении, для того чтобы уметь находить или исправлять ошибки в данных.

Функция Хэмминга - функция $H(x, y)$, равная количеству i, при которых $x[i] \neq y[i]$. Функция Хэмминга является метрикой для множества строк, так как удовлетворяет всем аксиомам метрики(это просто доказать, так что докажите сами).

Код c обнаруживает k ошибок, если $k < \min\limits_{\forall x,y \in \mathbb{S}, x \neq y} H\Big(c(x), c(y)\Big)$, здесь и далее $\mathbb{S}$ - это множество строк, которые мы кодируем. Докажем, что этот код обнаруживает k ошибок. Возьмем код какой-либо строки - x, в которой сделаем k ошибок. Тогда полученная строка y не может быть никаким кодом для любой другой строки, ведь если это так, то у нас есть два кода x и y, между которыми расстояние k, а значит $k \not< \min\limits_{\forall x,y \in \mathbb{S}, x \neq y} H\Big(c(x), c(y)\Big)$.

Код c исправляет k ошибок, если $2k < \min\limits_{\forall x,y \in \mathbb{S}, x \neq y} H\Big(c(x), c(y)\Big)$. Работать исправление будет так, мы считаем функцию Хэмминга для всех возможных строк, которые мы могли закодировать, и выбираем ту, до которой расстояние не превышает k. Очевидно, что раз мы допускаем, что ошибок в сообщении не больше k, то такая строка есть. Докажем что она единственна. Пусть это не так, тогда существует $\exists z \exists x, y \in \mathbb{S} : H(c(x), z) \leq k$ и $H(c(y), z) \leq k$. Тогда из того, что функция Хэмминга у нас метрика, следует: $2k \not< \min\limits_{\forall x,y \in \mathbb{S}, x \neq y} H\Big(c(x), c(y)\Big)$.

Для дальнейшего удобства позаимствуем из математического анализа определение шара $B(O, R)$, который будет определен над множеством строк и для которого метрикой будет функция Хэмминга. Объем шара будем обозначать $V(n, R)$, он не зависит от центра, так как $\forall B(O_1, R), B(O_2, R) : \forall x \in B(O_1, R)$ можно привести в соответствие $x \oplus O_1 \oplus O_2$, который принадлежит $B(O_2, R)$, а значит существует биекция между любыми двумя шарами одинакового радиуса, а значит и их объемы равны.

Лемма. Если у нас есть кодирование c, обнаруживающее k ошибок, то $\forall x, y \in \mathbb{S} : B(c(x), k)B(c(y), k) = \varnothing$. Это переосмысление нашего доказательства того, что кодирование, обнаруживающее k ошибок действительно это делает, и определения шара.

9.2 Теорема: Граница Хэмминга

. Если c - код для множества мощностью M, исправляющий k ошибок и использующий l бит для кодирования, то

$$M * V(l, k) \leq 2^l$$

.

Доказательство. Просто возьмем шары для каждого закодированного элемента, они все попарно непересекаются. Тогда в них лежит $M * V(l, k)$, что должно быть меньше, чем количество всевозможных кодов, что равно 2^l

9.3 Теорема: Граница Гилберта

Если $M * V(l, 2k) \leq 2^l$, то существует код, кодирующий множество мощностью M, исправляющий k ошибок и использующий l бит для кодирования.

Доказательство. Построим такое кодирование. Возьмем любой элемент из множества, которые мы хотим кодировать и выберем для него любой код. Тогда у нас недоступными для выбора следующего кода становится не более $V(lm2k)$ кодов, выкинем их. Продолжим, пока не зададим каждому элементу свой код. Кодов хватит, так как на выборе i кода мы выбросим не более $(i-1)*V(l, 2k)$ вершин, и $(i-1)*V(l, 2k) < M*V(l, 2k) \leq 2^n$. Значит такой код есть.

Общая формула $V(n, R)$:

$$V(n, R) = \sum_{i=0}^{R} C_n^i$$

Код Хэмминга.

Принцип работы: Найдем наименьшее k, что $k + n < 2^k$. Тогда код Хэмминга кодирует последовательность так: по строке длины n он выдает код длина которого в двоичной системе счисления равна $n + k$, в котором все биты с индексом равным степени двух - контрольные, а остальные n - исходные данные, которые мы бехз изменений перенесем из той строки, которую кодируем. Контрольные биты будем насчитывать так, на 2^i позиции будет находиться xor всех чисел, у которых i бит равен 1. Работает все это дело за $O(n \log(n))$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
r_0	r_1	x_1	r_2	x_2	x_3	x_4	r_3	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	r_4	x_{12}	x_{13}	x_{14}	x_{15}
1	1	1	1	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0	1

Алгоритм исправления ошибки: Давайте по полученной строке снова насчитаем контрольные биты по тем же правилам. Теперь давайте найдем индекс изменненого бита. Изначально предположим его равным нулю, начнем проходиться по контрольным битам, если iый насчитанный нами бит не совпадает с тем, что передали нам в строке, то ошибка у нас в числе, у которого в битовой записи на iом месте стоит 1, значит к ответу надо добавить 2^i. Полученный бит мы просто меняем на противоположный и радуемся. Декодировка также работает за $O(n \log(n))$.

Удивительно, но код Хемминга совпадает с границей Хэмминга, что делает его оптимальным кодированием с исправлением 1 ошибки.

10 Лекция 10.

10.1 Комбинаторные объекты

Комбинаторные объекты - объекты, обладающие между собой некоторыми зависимостями.

Мы будем рассматривать **линейное представление** комбинаторных объектов - их запись в виде строки над некоторым алфавитом(не обязательно конечным).

Мы хотим научиться:

1. Считать количество комбинаторных объектов
2. Уметь перечислять комбинаторные объекты
3. Нумеровать комбинаторные объекты
4. Выдавать по номеру объект, а по объекту номер

Лексикографический порядок - для двух различных строк A и B, строка A лексиграфически меньше строки B, если:

1. Строка A - префикс строки B
2. $\exists i < \min(|A|, |B|) : \forall j < i : A_j = B_j$ и $A_i \neq B_i$

Очевидно, что этих двух условий достаточно, чтобы задать порядок на строчках. Тогда мы можем задать порядок на всех линейных комбинаторных объектах.

10.2 Вектора фиксированной длины

Вектор фиксированной длины - это упорядоченный набор из n (возможно повторяющихся) элементов из алфавита A.

Подсчет количества векторов длины n ни для кого не сотавит затруднения - всего их $|A|^n$.

С нумерацией также проблем не возникает. Пусть $k = |A|$. Тогда давайте зададим каждому элементу из множества A свое число от 0 до k. Теперь дадим каждому элементу вектора свой коэффицент, который будет равен k в стенени индекса этого элемента в векторе, то есть для первого элемента это будет k^0(для удобства все в 0-индексации), для второго k^1 и так далее. Теперь перемножим коэффиценты элементов в векторе на числа, обозначающие данный элемент и получим искомое значение. По факту это запись числа в k-ичной системе счисления, и доказательство единственности зашифровки и расшифровки аналогичное.

Из способа нумеровать объекты становится очевиден способы выдавать по номеру объект, мы просто смотрим на остаток от номера по модулю k - это и будет первый элемент вектора, теперь поделим номер на k (с округлением вниз) и аналогично получим второй элемент вектора. Так и продолжим наш алгоритм, пока не получим все элементы вектора.

Если у вас уже упорядочены объекты, то вы можете сортировать их лексикографически.

10.3 Двоичные вектора без двух единиц подряд

Не стану вдаваться в подробности, потому что это не очень интересный пример, мы его посмотрим и забудем.

Воспользуемся идеей разделяй и влавствуй и отделим последний элемент нашего вектора. Если это 0, то ограничений на оставшийся вектор нет, а если это 1, то мы точно знаем, что предыдущий элемент нашего вектора это 0, а оставшийся на оставшийся вектор не накладывается никаких ограничений. Тогда мы можем вывести реккурентную формулу:

$$F(x) = F(x-1) + F(x-2)$$

Пристальным взглядом можно заметить, что это числа Фиббоначи, а значит мы можем посчитать их по формуле, которую спокойно можно загуглить(не беспокойтесь, на экзамене по дм она вам явно не пригодится).

10.4 Перестановки

Перестановка - последовательность длины n, такая что все числа в ней от 1 до n, при этом в нем нет одинаковых, то есть все числа присутствуют. Можно также это делать не с числами, но полученные перестановки будут изоморфны обычным, так что для удобства дальше мы будем работать только с числами.

Количество перестановок длины n равно $n * (n-1) * (n-2) * \cdots * 2 * 1 = n!$.

Очевидно, что перестановки можно лексиграфически нумеровать, так и будем делать.

Давайте научимся по номеру перестановки находить ее. Что происходит, если мы удаляем первый элемент перестановки, у нас остается $(n-1)$ элемент в любом порядке, то есть перестановка на $(n-1)$ элемент. Заведем обозначение на кол-во перестановок длины k - P_k. Тогда на первой позиции стоит 1, если итоговый номер не превышает P_{n-1}. Если же это не так, давайте выкинем все перестановки, начинающиеся с 1, тогда номер искомой уменьшается на P_{n-1}. Для перестановок, начинающихся с 2 мы можем сделать аналогичные умозаключения и понять, стоит в начале перестановки 2 или нет. Продолжая такой алгоритм мы поймем, что стоит на первом месте, тогда можно отбросить это число и продолжить алгоритм для второго числа. Получается такой алгоритм:

1. Имея перестановку длины n и номер искомой перестановки k и набор еще не поставленных чисел a в отсортированном порядке, ставим на первой место перестановки $a[k/P_{n-1}]$.
2. Уменьшаем n на 1, k меняем на $kmodP_{n-1}$, а из массива a выбрасываем поставленное число и, если он у нас еще не пуст - повторяем алгоритм с начала.

Ну, раз мы умеем по номеру находить перестановку нам не составит труда запустить обраный алгоритм и узнавать номер перестановки по самой перестановке.

10.5 Размещения

Размещения - количество способов упорядоченно выбрать k различных чисел из множества мощностью n.

Обозначается A_n^k и имеет формулу $A_n^k = \frac{n!}{(n-k)!}$.

10.6 Сочетания

Сочетания(биномиальный коэффицент) - количество способов неупорядоченно выбрать k различных чисел из множества мощностью n.

Обозначается C_n^k или $\binom{n}{k}$ и имеет формулу $C_n^k = \frac{n!}{k!(n-k)!}$(доказательство будет чуть ниже).

Возникают проблемы с тем, что сочетания - не линейный объект и для того, чтобы их решить, введем **канонизацию**.

Канонизация - довольная абстрактный метод, который говорит: если у нас один и тот же объект может представляться несколькими способами, один из них назовем каноническим, и чтобы упорядочить два объекта, упорядочим их канонические виды. В данном случае, как и во большинстве других, каноническим видом сочетания будет перестановка множества, лексикографически минимальная, среди всех остальных(Станкевич назвал этот порядок возрастающим, но я буду переиспользовать лексикографический порядок).

Формула как раз доказывается таким образом, мы берем размещения и говорим, при канонизации у нас все перестановки длины k сводятся к одной строке, значит кол-во сочетаний из n по k = (кол-во размещений из n по k)/(кол-во перестанок длины k).

Быстро расскажу про школьный способ считать количество сочетаний, мы снова применяем разделяй и влавствуй для какого-то элемента и если он лежит в множестве, то у нас C_{n-1}^{k-1} способов выбрать оставшиеся элементы, а если он не лежит, то C_{n-1}^k способов. Снова получаем реккуренту:

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

для которой можно по индукции доказать, что формула верна, но это выглядит непоследовательно и случайно, а делается довольно просто, поэтому это задания я оставлю читателю, как простейшее упражнение на закрепление формулы сочетаний.

10.7 Код Грея

Код Грея(для двочиных векторов длины n) - перечисления двочиных векторов в таком порядке, что расстояние Хэмминга между двумя соседними векторами равно 1.

Код Грея называют **циклическим**, если длина Хэмминга между первой и последней строкой также равна 1.

Алгоритм построения циклического кода Грея(зеркального) для двоичных векторов - Пусть мы хотим построить код Грея для двоичных векторов длины n. Построим его для двоичных векторов длины $n-1$ и допишем к ним ко всем в начало 0. Теперь снова этот же код Грея для двоичных векторов длины $n-1$, перевернем его(очевидно, что при этом он останется кодом Грея) и допишем ко всем векторам в нем 1 в начало. Теперь я утверждаю, что я если мы объединим два этих набора векторов, то получим код Грея для векторов длины n. Понятно, что коды Грея длины $n-1$, к которым мы приписали один и тот же символ, сломаться не могли, значит, если поломка и произошла, то она произошла на одном из стыков, но это тоже невозможно, ведь там различие только в приписанном в начало символе. Для базы $n = 1$ любой порядок векторов будет кодом Грея, значит этот алгоритм может построить код Грея для любого n.

10.8 Формула включений-исключений

Формально, формула включений-исключений говорит: Если мы умеем считать мощность множеств и мощность их пересечений, то мы можем также посчитать мощность их объединений.

Формула: Для семейства множеств $A_1, A_2, \ldots, A_n$:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{I \subset \{1,2,\ldots,n\} I \neq \emptyset} (-1)^{|I|+1} \left(\bigcap_{i \in I} A_i\right)$$

Доказательство:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \left|\bigcup_{i=1}^{n-1} A_i \cup A_n\right| = \left|\bigcup_{i=1}^{n-1} A_i\right| + |A_n| - \left|\bigcup_{i=1}^{n-1} (A_i \cap A_n)\right| =$$

$$= \sum_{I \subset \{1,2,\ldots,n-1\}} (-1)^{|I|+1} \left(\bigcap_{i \in I} A_i\right) + |A_n| + \sum_{J \subset \{1,2,\ldots,n\}} (-1)^{|J|+2} \left(\bigcap_{i \in J} A_i \cup A_n\right)$$

Посмотрим на эту формулу. Первая сумма это сумма по всем множествам, в которых нет A_n, вторая - A_n, а третья, это в точности все множества, которые содержат A_n и ещё хоть что-то, причем все с выражения с правильным знаком. Тогда это в точности формула, которую мы хотим доказать.

11 Лекция 11.

11.1 Генерация комбинаторных объектов

Алгоритм един:

1. Проверим является ли полученный префикс комбинаторным объектом.
2. Попытаемся дополнить полученный префикс единичным элементом.
3. Проверим, является ли полученный префикс корректным префиксом и, если это так, то рекурсивно запустим от него этот алгоритм с первого пункта.

Код всех генераций ко на языке с++ будет тут после того, как автор конспектов их напишет в лабу и она закончится.

11.2 Числа Каталана

Очень классная последовательность, которая очень часто встречается в комбинаторных задачах при подсчете объектов. Формул *иого* числа Каталана много - $C_n = \frac{C_{2i}^i}{i+1}$.

Через подсчет псп - $C_n = \sum_{k=0}^{n-1} C_k * C_{n-1-k}$.

11.3 Поиск *кого* комбинаторного объекта

Алгоритм снова един:

1. Проверим является ли полученный префикс комбинаторным объектом, если так, то если номер объекта, который мы ищем - 0, то выведем его и выйдем, иначе просто уменьшим номер объекта, который мы ищем на 1.
2. Попытаемся дополнить полученный префикс единичным элементом, которые мы будем обходить в лексикографическом порядке.
3. Проверим, является ли полученный префикс корректным префиксом и, если это так, то рекурсивно запустим от него этот алгоритм с первого пункта.

Этот алгоритм можно немного ускорить, так как нам не обязательно проходить по всем эелементам, если мы умеем быстро считать кол-во ко с каким-то префиксом, то второй пункт можно ускорить. Код всех поисков ко по их номеру на языке с++ будет тут после того, как автор конспектов их напишет в лабу и она закончится.

11.4 Получение следующего и предыдущего комбинаторного объекта

Алгоритм опять един:

1. Найти самый последний элемент, который можно увеличить.
2. Увеличить его минимально возможным образом.
3. Приписать лексиграфически наименьший хвост.

12 Лекция 12.

12.1 Разбиение

12.1.1 Разбиение на множества

Разбиение любого множества на набор множеств, так, что каждый элемент исходного множества оказывается ровно в одном множестве набора.

12.1.2 Разбиение на слагаемые

Разбиение числа на множество чисел, так, что сумма этих чисел равна самому числу.

ное **числа Белла** B_n - количество способов разбить на множества множество из n эелементов.

Число Стирлинга 2ого рода из n по k - $\{{n \atop k}\} = S_2(n,k)$

Формула числа Стирлинга 2ого рода:

$$\{{n \atop k}\} = \{{n-1 \atop k-1}\} + k * \{{n-1 \atop k}\}$$

Чтобы это доказать посмотрим на какой-то элемент из n, если его удалить и он был один в множестве, то кол-во множеств и элементов уменьшилось и это $\{{n-1 \atop k-1}\}$, если же он лежит с кем-то в множестве, то при его удалении мы получим одну и ту же конфигурацию k раз, значит их можно посчитать как $k * \{{n-1 \atop k}\}$.

Число Белла тогда можно посчитать как:

$$B_n = \sum_{k=0}^{n} \{{n \atop k}\}$$

Число Стирлинга 1ого рода $S_1(n,k)$ - число способов разбить множество мощностью n на k циклов, где цикл - размещения в котором два элемента считаются одинаковыми, если являются циклическими сдвигами друг друга.

Формула числа Стирлинга 1ого рода:

$$[{n \atop k}] = [{n-1 \atop k-1}] + (n-1) * [{n-1 \atop k}]$$

Доказательство этого аналогично доказательству формулы для числа Стирлинга 2ого рода.

Пока что не ясно зачем это, но было обращено внимание на последовательности 1 7 6 1 для чисел Стирлинга 2ого рода, и 6 11 6 1 для чисел Стирлинга 1ого рода, если вы их видите, то с большой вероятностью вы видите числа Стирлинга.

Свойства чисел Стирлинга 1ого рода - $\sum_{k=0}^{n} [{n \atop k}] = n!$

12.2 Пути Дика

Рассмотрим плоскость и путь в нем берущий начало в точке (0,0), выполняющий следующие требования:

1. Каждое звено которого направлено вправо на 1 и либо на 1 вверх, либо на 1 вниз.
2. Путь Дика обязательно заканчивается в точке с Y-координатой равной нулю.
3. Y-координата любой вершины в пути Дика никогда не становится отрицательной.

Проще всего провести аналогию с графиком акций криптовалюты, они обе начинаются с нуля, растут или падают, но в конце концов всегда обесцениваются и оказываются в 0, но при этом не могут быть отрицательны ни в какое время.

12.3 Пентагональная формула Эйлера.

Не будем вдаваться тут особо в подробности о том, что это такое, я скажу так, вас об этом вряд ли спросят, а уж запоминать вы это точно не захотите(Станок мельком что-то об этом упомянул и это скорее та самая легендарная дополнительная литература по дм).

12.4 Бинарные деревья.

Бинарные деревья - корневые деревья, ограниченные тем, что у каждой вершины не больше двух сыновей, один из которых считается правым, а второй левым. Жесткого ограничения на степень вершины нет, причем четкого предписания, что если сын один, то он обязательно левый или правый тоже нет, поэтому, например, одинаковые по строению деревья, после предписания какие сыновья левые, а какие правые, могут получится разными.

Утверждение: Количество бинарных деревьев на n вершинах - C_n(n-ое число Каталана)

Доказательство:

1. Проведем аналогию с псп, для которого уже все доказано. Пусть вершина - пара открывающейся и закрывающейся скобки, ее левый сын, это та псп, что находится внутри этой пары скобок, а правый сын, та псп, что находится правее нее. Получаем, что из любого дерева мы можем однозначно получить какое-то псп, а также из любой псп можем однозначным образом получить бинарное дерево(если неясно почему, докажите сами), значит это биекция, а значит утверждаемое верно.

13 Лекция 13.

13.1 Перестановки(глубже)

13.1.1 Композиция перестановок

Перестановки можно переменожать, как это было названо на лекции, но здесь далее я буду использовать композицию, потому что это подходит по смыслу больше. При композиции перестановок a и b мы просто переставляем массив по перестановке a, а потом по перестановке b. Записывается это в обратном порядке - ba, а также всегда равно какой-то перестановке c, для которой верно: $c[i] = b[a[i]]$, так проще всего и запомнить правильный порядок записи ba.

Перестановки **ассоциативны**, доказывается это так, посмотрим куда попадет iый элемент перестановки: $a(bc)[i] = a[b[c[i]]]$ и $(ab)c[i] = a[b[c[i]]]$.

У перестановок есть **нулевой элемент**, называемая стабильной перестановкой - $1, 2, \ldots, n$

Также существует и **обратный элемент**, для перестановок это та перестановка, при композиции с которой наша исходная перестановка P станет стабильной. Доказательство существования такой перестановки может показаться неочевидным, но на самом деле все станет кристально понятно, если мы представим перестановку в виде ориентированного графа, это представление, кстати, самое удобное, так вот в нем становится понятно, что перестановка это просто набор циклов, и если все ребра в нем повернуть в обратную сторону, то мы также получим какую-то перестановку, которая при композиции с исходной и будет давать стабильную, эту перестановку мы и назовем обратной.

13.2 Циклические классы

Мы уже забежали вперед в прошлой секции и представили перестановку в виде графа, так вот **циклические классы** - это множество циклов данной перестановки, если представить ее в виде графа. Для каждого класса оттуда можно задать его **мощность** - это просто длина цикла.

Инволюция - перестановка, в которой мощности циклических классов не превышает 2.

Порядок перестановки P - степень k, при возведении P в которую у нас получится стабильная перестановка.

Давайте научимся находить порядок перестановки P. Представим ее в виде графа и вспомним, что это все циклы какой-то длины. При композиции текущего массива с P мы, по факту, из каждой вершины ходим по ребру, который ведет из этой вершины в перестановке. А так как мы уже помним, что P - это набор циклов каких-то длин, то нам нужно, чтобы итоговое число перемещений по ребрам было кратно всем мощностям циклов, то есть НОК всех мощностей циклов.

Важный факт: Перестановку можно хранить как матрицу, но никто так не делает, так что идем дальше.

13.3 Теорема Кэли о конечных группах

Любая конечная группа H вложена в S_n.

Уточнение: S_n - множество всех перестановок из n элементов.

Занумеруем элементы группы H, а дальше построим таблицу умножения для этой группы. Теперь сопоставим каждому элементу свою перестановку из n элементов - строку из данной таблицы умножения, соответствующую данному элемент(так как в группе при умножении на разные элементы мы будем получать разные результаты, то это действительно перестановка). Сопоставим элементу a перестановку L_a, тогда знаем мы такое: $L_{ab}h = (ab)h = a(bh) = L_aL_bh$, значит $L_{ab} = L_aL_b$.

14 Лекция 14.

14.1 Подсчет с точностью до действия группы

На группе заводятся действия, которые удовлетворяют следующим аксиомам:

1) $h(g(x)) = (h * g)(x)$, набор действий это группа и для них определено умножение
2) $ex = x$, существует нейтральный элемент

Для действия может быть **неподвижная точка**, для которой верно:

$$I_g = \{x | x : gx = x\}$$

Для элемента x существует **стабилизатор** - действия, которые его не изменяют:

$$St_x = \{g | gx = x\}$$

Мы получили отношения эквивалентности, это можно проверить, проверив рефлексивность транзитивность и симметричность. X/G - множество классов эквивалентности, называемая **орбитами**.

14.2 Лемма Бёрнсайда

Формула -

$$|X/G| = \frac{\sum\limits_{g \in G} |I_g|}{|G|}$$

Выглядит это буквально как среднее арифметическое размеров групп, им оно на самом деле и является.

Доказательство:

Нарисуем таблицу всех действий на всевозможные объекты. Рассмотрим какой-нибудь столбец элемента x_i, это будет его орбита, из-за отношения эквивалентности. Некоторые орбиты могут оказаться одинаковыми - для эквивалентных элементов, если будем находить несколько таких, то выкинем повторяющиеся. Столбцов тогда останется столько, сколько у нас всего орбит - $|X/G|$. Строк - $|G|$. А каждый элемент встречается теперь всего один раз, в том столбце, на которой орбите он лежит, но лежит он там столько раз, сколько действий превращают его в самого себя, то есть $|St_x|$ раз. Тогда мы получили такую формулу: $|X/G| * |G| = \sum\limits_x St_x$. Ну а эта формула уже очевидно приводится к нужной, так как сумма мощностей неподвижных точек по всем g равна сумме стабилизаторов по всем x.

14.3 Теорема Пойя

Число орбит - $|X/G| = \frac{1}{|G|} \sum\limits_g S^{C(G)}$ или $|X/G| = \frac{1}{|G|} \sum\limits_{i=1}^{n} k_i * S^i$, где k_i - количество действий, перестановка которых имеет k циклов. Тут также доказательство я лучше вынесу в удобную ссылочку.

15 Информация о курсе

Поток — y2024.
Группы M3138-M3142.
Преподаватель — Станкевич Андрей Сергеевич.

Это первый семестр курса по дискретной математике, всем успехов!