# Теория сложности вычислений, IV семестр

**Весна 2016, лектор: Гирш Эдуард Алексеевич**

**Автор: Егор Суворов**

Собрано: 1 июня 2016 г. 23:47

---

## Содержание


**1 Билет 1** **4**
- 1.1 Типы задач . . . 4
- 1.2 Машины Тьюринга . . . 4
  - 1.2.1 Базовая модель . . . 4
  - 1.2.2 Модификации . . . 6
  - 1.2.3 Метрики . . . 7
- 1.3 Классы $P$ и $NP$ . . . 7
- 1.4 Сведение по Карпу . . . 8
- 1.5 Сведение по Левину . . . 9
- 1.6 Полные задачи . . . 9
- 1.7 Задача об ограниченной остановке . . . 10

**2 Билет 2** **10**
- 2.1 Универсальная МТ . . . 10

**3 Билет 3** **11**

**4 Билет 4** **12**
- 4.1 $Circuit - SAT$ . . . 12
- 4.2 $3 - SAT$ . . . 13

**5 Билет 5** **14**

**6 Билет 6** **15**

**7 Билет 7** **17**

**8 Билет 8** **18**

**9 Билет 9** **18**
- 9.1 Определения . . . 18
- 9.2 Полные задачи . . . 21
- 9.3 Коллапс . . . 21

**10 Билет 10** **23**
- 10.1 Идея . . . 23
- 10.2 Точное решение . . . 23

**11 Билет 11** **24**

**12 Билет 12** 25

**13 Билет 13** 26
13.1 NC и logspace-вычислимость . . . 26
13.2 Сводимости в P . . . 27
13.3 Следствия из P-полноты . . . 27

**14 Билет 14** 28
14.1 Основные понятия . . . 28
14.2 Арифметические операции . . . 29
14.3 Умножение матриц и достижимость . . . 31

**15 Билет 15** 31
15.1 NSpace[log n] <= NC2 . . . 31
15.2 DSpace[log n] <= NSpace[log n] . . . 32
15.3 NC1 <= DSpace[log n] . . . 32

**15 Билет 15** 32

**17 Билет 17** 33
17.1 Рандомизированные машины . . . 33
17.2 RP . . . 34
17.3 BPP и понижение ошибки . . . 34
17.4 PP и вложенность классов . . . 35
17.5 ZPP . . . 36

**18 Билет 18 (лемма Шварца-Циппеля)** 37

**19 Билет 19** 39

**20 Билет 20** 40
20.1 Основная идея . . . 40
20.2 Аккуратная оценка вероятности . . . 41
20.3 Следствие . . . 41

**21 Билет 21** 42
21.1 Определения . . . 42
21.2 Протокол для неизоморфизма . . . 43
21.3 Протокол для перманента . . . 44

**22 Билет 22** 45
22.1 Введение . . . 45
22.2 Арифметизация . . . 46
22.3 Интерактивный протокол . . . 47

**23 Билет 23** 48
23.1 Классы AM и MA . . . 48
23.2 Попарно независимые хэш-функции . . . 49
23.3 Протокол Гольдвассер-Сипсера . . . 50
23.4 Применение протокола Гольдвассер-Сипсера . . . 50

**24 Билет 24** 50

**25 Билет 25** 50
25.1 PCP . . . 50
25.2 Связь с неаппроксимируемостью . . . 51

# 1. Билет 1

## 1.1. Типы задач

**Def 1.1.** *Слово* — элемент $\{0,1\}^*$, длина слова $x$ обозначается $|x|$.

**Def 1.2.** *Язык* — некоторое множество слов (возможно, пустое или бесконечное).

*Замечание* 1.1. Мы считаем, что битовыми строками можно кодировать вообще любые объекты, которые можно описать. Кодировать будем, разумеется, наиболее разумно (если не сказано иное), чтобы размер строк был как можно меньше. Где важен тип кодирования — будем явно указывать.

*Пример* 1.1. Натуральные числа можно естесственно закодировать битовой строкой — представление числа в двоичной системе счисления.

*Пример* 1.2. Пары битовых строк: $(a, b)$ можно закодировать какой-то битовой строчкой. Тогда за $|(a, b)|$ естественным образом будет обозначаться длина этой строчки. В принципе, ожидаемо, что $|(a, b)| = \theta(|a| + |b|)$, по-хорошему надо выбрать один способ кодирования и для него доказать, но мы не стали.

**Def 1.3.** *Индивидуальная задача* — это пара $(a, b)$, где $a, b$ — слова. $a$ называется условием или входом, $b$ называется решением для данного входа.

*Замечание* 1.2. Это определение не нужно и дальше нигде встречаться не будет.

**Def 1.4.** *Массовая задача* $R$ — некоторое множество $R$ индивидуальных задач (возможно, пустое или бесконечное):

$$R \subseteq 2^{\{0,1\}^* \times \{0,1\}^*}$$

Также можно рассматривать как бинарное отношение на строках: $aRb \iff (a, b) \in R$ (читать как «$b$ является решением задачи $R$ с условием $a$») Мы будем писать и так, и так.

*Пример* 1.3. Задача $FRAC$ — множество таких пар натуральных чисел $(n, d)$, что $n \vdots d$. То есть у одного «условия» $(n)$ может быть несколько «решений» $(d)$, никаких проблем.

**Def 1.5.** Мы умеем решать *задачу поиска* для массовой задачи $R$, если есть алгоритм, который по условию $x$ находит произвольное решение $w$ такое, что $(x, w) \in R$ или же сообщает, что такого решения нет.

**Def 1.6.** Мы умеем решать *задачу распознавания* для массовой задачи $R$, если есть алгоритм, который по условию $x$ отвечает: есть ли хоть какое-нибудь решение $w$.

*Замечание* 1.3. Таким образом, если мы умеем решать задачу распознавания для $R$, то у нас есть алгоритм, разрешающий следующий язык:

$$L(R) = \{x \mid \exists w\colon (x, y) \in R\}$$

## 1.2. Машины Тьюринга

### 1.2.1. Базовая модель

Машина Тьюринга (МТ) — вычислительная модель. Есть много модификаций, но у каждой МТ есть следующее:

- Фиксированный конечный алфавит $\Sigma$, в котором имеется как минимум два специальных символа: пробел (␣) и специальный символ начала ленты ($\triangleright$).

- Фиксированное конечное число *лент*, бесконечных в одну сторону. Лента — это бесконечно много ячеек, занумерованных натуральными числами.
- Конечное множество *состояний* $q_i$.
- Конечное множество *переходов*. Каждый переход — это кортеж:

$$(q, c_1, c_2, \dots, c_l, c'_1, c'_2, \dots, c'_l, m_1, m_2, \dots, m_l r)$$

  при этом:

  - $l$ — число лент
  - $c_i, c'_i \in \Sigma$
  - $q, r$ — состояния
  - $m_i \in \{\leftarrow, \cdot, \rightarrow\}$ — *сдвиг головки* $i$

  Говорим, что это переход из $q$ в $r$ по символам на лентах $c_1, \dots, c_l$, который пишет на ленты $c'_1, \dots, c'_l$ и сдвигает головки в направлениях $m_i$.

- В зависимости от модификации могут вводиться какие-нибудь выделенные подмножества состояний (например, терминальные состояния, см. ниже) и лент, накладываться дополнительные ограничения.

Чтобы МТ что-нибудь вычисляла, вводится понятие *конфигурации* конкретной МТ $M$, оно состоит из:

- Для каждой ленты хранится одно натуральное число, соответствующее какой-то ячейке — *позиция головки на этой ленте*
- Для каждой ленты хранится отображение $f\colon \mathbb{N} \to \Sigma$, которое в конечном числе точек может возвращать что угодно, а в остальных возвращает $\llcorner\!\lrcorner$. Также можно дополнительно потребовать $f(1) = \rhd$, но это необязательно (соответствующие определения будут эквивалентны).
- Выделяется *текущее состояние* МТ $q$.

Дальше вводится понятие *корректного перехода из конфигурации* $A$ в какие-то другие конфигурации. Чтобы сгенерировать список этих конфигураций, надо сделать следующее:

1. Рассмотреть все переходы из текущего состояния конфигурации $A$.
2. Оставить из них только те переходы, у которых символ $c_i$ совпадает с символом на позиции головки на ленте $i$
3. Для каждого из оставшихся сказать, что мы можем перейти в конфигурацию, у которой текущее состояние изменено с $q$ на $r$, символы $c_i$ на лентах заменены на символы $c'_i$, соответственно, а позиции головок на каждой ленте независимо изменены в соответствии с $m_i$:
   - При $\leftarrow$ головка сдвигается на единицу влево
   - При $\cdot$ головка остаётся на месте
   - При $\rightarrow$ головка сдвигается на единицу вправо

*Замечание* 1.4. Можно формально определить бинарное отношение «быть корректным переходом» между состояниями, аккуратно расписав алгоритм выше.

*Замечание* 1.5. Никакая головка не должна уезжать за край ленты. Есть разные способы этого добиться.

### 1.2.2. Модификации

Все модификации МТ ниже в разном смысле эквивалентны друг другу, мы выбираем ту, в которой удобнее работать в данной задаче. Для доказательства эквивалентности двух моделей, надо выбрать машину в модели $A$ и построить эквивалентную ей машину в модели $B$. Под эквивалентностью можно понимать что-то из следующего:

- Обе машины решают задачу распознавания одного и того же языка
- Обе машины вычисляют одну и ту же функцию
- Обе машины используют одно и то же количество памяти (это может быть неправда при переделке ДМТ в НМТ)
- Обе машины работают за одно и то же время, или хотя бы время работы различается лишь не более, чем в фиксированную константу раз (это, опять же, не так для ДМТ и НМТ)

**𝔇𝔢𝔣 1.7.** *Детерминированная машина Тьюринга* (ДМТ) — если зафиксировать начальное состояние и множество символов под лентой, то переход либо есть и ровно один, либо его нет. Если перехода нет, то можно либо отвергать слово, либо переходить в какое-то спецсостояние (эквивалентно). Можно считать, что все переходы всегда должны быть (эквивалентно).

*Замечание* 1.6. Если у ДМТ зафиксировать конфигурацию, то следующая конфигурация (если есть) всегда определяется однозначно. Поэтому можно говорить о том, что ДМТ что-то однозначно вычисляет (или зацикливается).

**𝔇𝔢𝔣 1.8.** На ДМТ можно вычислять функции: на фиксированную (входную) ленту пишем вход, запускаем ДМТ, пока не завершиться, после этого ожидаем на фиксированной ленте (выходной) получить ответ.

**𝔇𝔢𝔣 1.9.** На ДМТ также можно вычислять ответы «да/нет»: говорим, что ДМТ обязана завершать работу в ровно одном из двух состояний — $q_{yes}$ или $q_{no}$, и именно этим определяется результат работы.

*Замечание* 1.7. Чтобы головка не уезжала налево за край ленты, можно считать по-разному:

- Переход, при котором головка уезжает налево некорректен
- Если головка уезжает налево, вычисление прекращается.

**𝔇𝔢𝔣 1.10.** *Недетерминированная машина Тьюринга* (НМТ) — без подобных ограничений, может быть несколько корректных переходов из одной конфигурации.

**𝔇𝔢𝔣 1.11.** НМТ принимает слово $x$, если есть такая последовательность корректных переходов, что машина останавливается в состоянии $q_{yes}$.

*Замечание* 1.8. Также иногда говорят, что НМТ-машина помимо входа принимает некоторую *подсказку*, которая указвает, какой переход делать в случае неоднозначности. Например: НМТ принимает слово, если она принимает его хотя бы с одной подсказкой и отвергает, если нет такой подсказки.

*Замечание* 1.9. Для НМТ может быть наглядно рисовать дерево вычислений: в корне стоит начальная конфигурация, детьми вершины-конфигурации являются конфигурации, в которые НМТ может перейти. В таком дереве лист является законченным вычислением. Тогда время работы — глубина дерева.

### 1.2.3. Метрики

**𝔇𝔢𝔣 1.12.** *Время работы $M$* — количество шагов.

**𝔇𝔢𝔣 1.13.** *Используемая $M$ память* — сумма следующей величины по всем лентам: максимальное правое положение головки на ленте.

*Пример* 1.4. Если машина на двух лентах сначала отвела первую головку вправо на 10, потом вернула, потом сделала то же самое со второй головкой, то используемая память — $10 + 10 = 20$.

**𝔇𝔢𝔣 1.14.** На НМТ время работы и память вычисляются так: независимо берётся максимум по всем веткам. То есть если в одной ветке время было 2 и память 5, а в другой — время 3 и память 4, то время работы машину будет 3, а память — 4.

**Утверждение 1.1.** Мы можем считать, что если у конкретной НМТ всегда каким-то разумным образом ограничено время работы и память в той последовательности переходов, которая ведёт к принятию слова (например, на входе $x$ будет сделано не более $f(|x|)$ переходов и не более $g(|x|)$ памяти), то она **всегда** использует не более $\mathcal{O}(g(|x|))$ памяти и $\mathcal{O}(f(|x|))$ времени.

▶ Давайте возьмём произвольную НМТ с функциями $f$ и $g$. Заставим эту НМТ параллельно с её основной работой считать используемую память и количество сделанных шагов на отдельных лентах (предполагаем, что значения $f$ и $g$ для конкретного входа мы можем вычислить быстро). Если в какой-то момент машина обнаруживает, что она сделала уже больше $f(|x|)$ шагов или использует более $g(|x|)$ памяти, то она точно уверена, что в этой ветке ей уже можно слово отвергать и завершаться. Если слово не принималось НМТ, то оно и не станет приниматься, а вот если принималось, то хотя бы одна ветка останется (но какие-то длинные могут и исчезнуть). Это называется «полиномиальный будильник» и может встречаться не только в НМТ, но и в ДМТ (например, если мы знаем, что ДМТ всегда находит ответ при его существовании за короткое время). ◀

*Замечание* 1.10. Иногда бывает так, что МТ должна использовать памяти меньше, чем вход. Тогда считают, что лента со входом доступна только на чтение (т.е. мы всегда пишем на неё тот же символ, что и прочитали) и в общую память она не входит.

## 1.3. Классы $P$ и $NP$

**𝔇𝔢𝔣 1.15.** Массовая задача $R$ *полиномиально ограничена* (п.о.), если существует полином $p$, ограничивающий длину кратчайшего решения для каждого входа, на котором вообще есть ответ:

$$\forall x\colon (\exists w\colon (x, w) \in R) \Rightarrow \exists w\colon (x, w) \in R \wedge |w| \leqslant p(|x|)$$

Длина решения зависит полиномиально от длины входа.

**𝔇𝔢𝔣 1.16.** Массовая задача $R$ *полиномиально проверяема* (п.п.), если существует ДМТ $M$ и полином $q$, ограничивающий время проверки решения:

$$(x, w) \in R \iff M((x, w)) = 1$$

Причём вычисление $M((x, w))$ всегда завершается не более, чем за $q(|(x, w)|)$ шагов.

*Замечание* 1.11. Можно считать, что $q$ на самом деле от двух переменных, можно считать, что он от $|x| + |w|$ — это всё будет эквивалентно.

**𝔇𝔢𝔣 1.17.** $\widetilde{NP}$ — класс п.о. п.п. задач поиска, т.е. это все п.о. п.п. массовые задачи, которым мы задаём вопрос «а какое есть решение у данного условия?»

*Замечание* 1.12. Дальше будет класс $NP$, в нём тоже будут лежать п.о. п.п. массовые задачи, однако им мы уже будем задавать вопрос «есть ли решение у данного условия?». Можно считать $\widetilde{NP}$ классом (некоторым множеством) массовых задач, а $NP$ — классом языков.

**Def 1.18.** $\widetilde{P}$ — подкласс $\widetilde{NP}$, причём для каждой задачи $R$ должно быть для каждого условия $x$ найти за полиномиальное от $x$ время какое-нибудь решение.

*Замечание* 1.13. Можно считать, что если решения нет, то мы это тоже за полиномиальное время узнаём. Или зацикливаемся. Это эквивалентно — добавляем будильник в МТ.

**Def 1.19.** $NP$ — все языки, порождаемые массовыми задачами из $\widetilde{NP}$. Т.е. задача $A \in \widetilde{NP}$ порождает язык из условий $x$ таких, что $\exists w\colon (x, w) \in A$.

**Def 1.20.** $P$ — все такие языки $A$, что: есть ДМТ $M$ и полином $q$, причём для любого слова $x$ $M(x) = A(x)$ и машина отработает не больше, чем за $q(|x|)$.

**Утверждение 1.2.** Это определение эквивалентно такому: $P$ — все языки, порождаемые $\widetilde{P}$.

▶ $L(\widetilde{P}) \subseteq P$ : взяли язык $A$ из левой части. Запустили алгоритм, ищущий решение. Если он нашёл, то условие принадлежит соответствующему языку, иначе не принадлежит.

$P \subseteq L(\widetilde{P})$ : взяли язык $A \in P$. Ему можно сопоставить массовую задачу $A'$:

$$(x, w) \in A' \iff (x \in A) \wedge (w = \varepsilon)$$

Очевидно, она лежит в $\widetilde{NP}$ и для каждого условия $x$ можно либо найти решение $\varepsilon$, либо сказать, что решений нет. Проверку существования решения за полиномиальное время сделать сумеем.

◀

**Утверждение 1.3.** Второе определение $NP$: это все такие языки, для которых существует полиномиальная ДМТ, которая проверяет слово из языка при наличии подсказки. При этом наличие хотя бы одной подсказки эквивалентно тому, что слово в языке.

**Утверждение 1.4.** Третье определение $NP$: это все такие языки, которые можно разрешить на полиномиальной НМТ.

## 1.4. Сведение по Карпу

Это то, что обычно подразумевается под «сведением задач» (в случае задач распознавания). Задача $L_1$ сводится по Карпу к $L_2$, если есть такая полиномиально вычислимая $f$, что:

$$x \in L_1 \iff f(x) \in L_2$$

То есть если хотим решить задачу $L_1$, то можно вычислить $f(x)$, решить задачу $L_2$ и сразу получить ответ на исходную. Тут важно, что ответ мы хотим получить сразу. Например, тогда подсказки для $L_2$ мы сразу можем назвать подсказками для $L_1$, а вот если бы, скажем, ответ инвертировали, то так сделать было бы нельзя.

**Утверждение 1.5.** Классы $P$, $NP$ замкнуты относительно сведения по Карпу, т.е. если $R_2 \in P$, то сводимая к ней $R_1$ тоже лежит в $P$

## 1.5. Сведение по Левину

Это то, что обычно подразумевается под «сведением задач» (в случае задач поиска). Опять же, использовать в будущем не будем.

Задача $R_1$ сводится по Левину к $R_2$, если есть три функции $f, g, h$ такие, что (тут по $x_i, y_i$ стоят кванторы всеобщности):

1. $f(x_1)$ по условию для $R_1$ возвращает за полиномиальное время условие для $R_2$.
2. Если есть решение $R_1$, то есть не сильно большее решение $R_2$:
$$R_1(x_1, y_1) \Rightarrow R_2(f(x_1), g(x_1, y_1))$$
Причём $g$ — п.о. (но необязательно полиномиально вычислима).
3. Если известно решение $R_2$, то его можно быстро переделать в решение $R_1$:
$$R_2(f(x_1), y_2) \Rightarrow R_1(x_1, h(f(x_1), y_2))$$
При этом функция $h$ принимает только $f(x_1)$, а не сам $x_1$, и работает за полиномиальное время.

*Замечание* 1.14. Смысл такой: если у нас есть условие для $R_1$, то мы можем посчитать условие для $R_2$ (функцией $f$), решить задачу $R_2$ и получить для неё ответ, получить ответ для $R_1$ (функцией $h$). Функция $g$ нужна только для того, чтобы гарантировать, что ответ на задачу $R_2$ будет не очень большой.

*Замечание* 1.15. Иногда «для красоты» требуют, чтобы функция $g$ тоже была полиномиально вычислима. Нам это вроде не было важно.

*Замечание* 1.16. Сведение по Левину имеет смысл только в том случае, если мы говорим о задачах с конкретными сертификатами. Сведение же по Карпу в некотором смысле более общее — оно сводит языки.

**Утверждение 1.6.** Если задача $R_1$ сводится по Левину к $R_2$, то язык $L(R_1)$ сводится по Карпу к $L(R_2)$

▶ Функция $f$ для сведения по Карпу будет взята из сведения по Левину. В самом деле, два случая:

**$x \in L(R_1)$:** Тогда мы точно знаем, что есть некоторый $y_1$, причём $(x, y_1) \in R_1$. Следовательно, по сведению по Левину есть такой $y_2 = g(x_1, y_1)$, что $(f(x_2), y_2) \in R_2$. Значит, $x \in L(R_1) \Rightarrow f(x) \in L(R_2)$.

**$x \notin L(R_1)$:** Хотим показать, что тогда $f(x) \notin L(R_2)$. Предположим противное: $f(x) \in L(R_2)$. Тогда есть некоторый $y_2$ такой, что $(f(x), y_2) \in R_2$. Тогда из сведения по Левину знаем, что $(f(x, h(f(x), y_2)) \in R_1$, т.е. $x \in L(R_1)$, противоречие.

◀

**Утверждение 1.7.** Классы $\widetilde{P}$, $\widetilde{NP}$ замкнуты относительно сведения по Левину, т.е. если $R_2 \in P$, то сводимая к ней $R_1$ тоже лежит в $P$

## 1.6. Полные задачи

Тут определяется и для классов задач поиска, и для классов задач распознавания.

**Def 1.21.** Задача $A$ — *трудная для класса* $C$ (или $C$*-трудная*), если произвольная задача $B \in C$ сводится к $A$ (по Карпу или Левину).

**Def 1.22.** Задача $A$ — *полная для класса* $C$ (или $C$*-полная*), если она $C$-трудная и лежит в классе $C$.

**Def 1.23.** Класс $NP$-полных задач обозначается $NPC$.

### 1.7. Задача об ограниченной остановке

**𝔇𝔢𝔣 1.24.** $1^t$ — так обозначается строка из $t$ единиц.

**𝔇𝔢𝔣 1.25.** Задача $\mathsf{BH}$ (*об ограниченной остановке*) — это множество троек $(M, x, 1^t)$ таких, что:

1. $M$ — недетерминированная машина Тьюринга
2. Существует подсказка $w$ для $M$ такая, что $M(x, w)$ останавливается и принимает $x$.

   *Замечание* 1.17. Конечно же, $|w| \leqslant t$, так как у нас машина после $t$ шагов будет остановлена.

**Теорема 1.1.** $\mathsf{BH}$ является $\mathsf{NP}$-трудной.

▶ Возьмём произвольный язык $L \in \mathsf{NP}$. По определению для него есть НМТ $M_L$ такая, что:

$$x \in L \iff M_L(x) = 1$$

причём $M_L$ принимает $x$ за время не большее $p(|x|)$, где $p$ — полином.

Давайте сводить $L$ к $\mathsf{BH}$ по Карпу, строя п.о. п.п. функцию $f$. Наша функция $f$ будет по входу $x$ выдавать машину $M_L$ (она константа и не зависит от $x$), потом — сам вход $x$, а потом — $1^{p(|x|)}$. Очевидно, что $x \in L \iff f(x) \in \mathsf{BH}$, просто по определению $\mathsf{NP}$.

Также понятно, что $f$ полиномиально вычислима и, следовательно, полиномиально ограничена (мы верим, что вычислить значение полинома и вывести столько единиц мы справимся).

**Теорема 1.2.** $\mathsf{BH}$ является $\mathsf{NP}$-полной.

▶ Эта задача уже точно является $\mathsf{NP}$-трудной, осталось показать, что она лежит в $\mathsf{NP}$.

Для этого просто строим универсальную недетерминированную машину Тьюринга, которая будет эмулировать данную на вход НМТ в течение не более $t$ шагов, а когда эмулируемой НМТ потребовалось принять недетерминированное решение, наша универсальная машина тоже будет принимать недетерминированное решение.

*Замечание* 1.18. Осторожно с построением универсальных машин: нам важно, чтобы замедление было максимум полиномиальным от $t$ и чтобы эмулируемая машина могла иметь произвольное число лент (а универсальная — лишь константное), см. билет 2.

# 2. Билет 2

## 2.1. Универсальная МТ

Тут мы говорим про ДМТ. Вход для универсальной ДМТ $U$ — пара из ДМТ и входа для неё — $(M, x)$. Интересно то, сколько шагов затратит $U$ на эмуляцию $M$, если $M$ работала за $t$ шагов.

*Замечание* 2.1. В общем случае алфавит $M$ может быть сильно больше алфавита $U$, но это лечится преобразованием машины $M$ в машину $M'$, которая пользуется алфавитом $\{0, 1, \sqcup, \triangleright\}$ и кодирует каждый старый символ фиксированным числом бит. Замедление $M'$ по сравнению с $M$ будет лишь константное.

Если нам требуется моделировать только $M$ с ограниченным числом лент $\leqslant k$, то это легко делается машиной $U$ с $k+1$ лентой с константным замедлением (т.е. $U_M$ будет работать для конкретной $M$ за $\mathcal{O}(t)$). На этой отдельной ленте мы храним описание $M$ и текущее состояние, а первые $k$ лент и позиции головок на них в точности соответствуют машине $M$. Прочитали за не зависящее от $t$ время символы под головками, нашли нужный переход, перешли.

Если у нас машина $U$ имеет только одну ленту, то эмулировать можно за время $\mathcal{O}(t^2)$. Мы храним описание $M$ в начале ленты, потом храним $k$ рабочих лент вперемешку (сначала первые символы, потом вторые символы и т.д.). На каждой ленте символ, на который указывает головка машины, мы специальным образом помечаем, чтобы не потерять. А дальше совершение шага такое:

1. Доехали до упора влево, до описания $M$.
2. Там мы где-то храним номер текущего состояния.
3. **TODO**

Если у нас все машины одноленточные (и $U$, и $M$), то эмулировать довольно просто, мы это делали на логике в прошлом семестре. У нас есть описание эмулируемой машины фиксированного размера, мы его таскаем по ленте вслед за головкой, получаем замедление в константное число раз.

А вот если лент у нас сколько угодно, то было бы разумно считать, что у $U$ число лент фиксированное, а вот эмулировать она может машину с произвольным числом лент. Можно либо делать за $\mathcal{O}(t^2)$ (см. 1.3.1 в Arora-Barak), либо за $\mathcal{O}(t \log t)$ (см. теорему 1.13 в Arora-Barak, это раздел 1.A). $\mathcal{O}(t^2)$ легко доделать до oblivious, а вот с доделкой $\mathcal{O}(t \log t)$ у меня возникли проблемы (это упражнение 9 в Arora-Barak к главе 1). **TODO**

# 3. Билет 3

**Def 3.1.** $f(n)$ *конструктивна по времени*, если есть такая ДМТ, что она по входу $1^n$ может вывести $1^{f(n)}$ за время $\mathcal{O}(f(n))$.

*Замечание* 3.1. В четырёх определениях ниже $f(n)$ должна быть конструктивна по времени и неубывать.

**Def 3.2.** $DTime[f(n)]$ — множество языков, разрешаемых на ДМТ за время $\mathcal{O}(f(n))$

**Def 3.3.** $DSpace[f(n)]$ — множество языков, разрешаемых на ДМТ с использованием $\mathcal{O}(f(n))$ памяти (если памяти мало, то входная лента становится read-only)

**Def 3.4.** $NTime[f(n)]$ — множество языков, разрешаемых на НМТ за время $\mathcal{O}(f(n))$

**Def 3.5.** $NSpace[f(n)]$ — множество языков, разрешаемых на НМТ с использованием $\mathcal{O}(f(n))$ памяти (если памяти мало, то входна лента становится read-only).

*Замечание* 3.2. Тут важно, что если мы используем определение НМТ с подсказкой, то ей запрещается читать старые биты подсказки: каждый бит читается максимум один раз.

*Пример* 3.1.

$$P = \bigcup_{poly(n)} = DTime[poly(n)] = \bigcup_{k=0}^{\infty} DTime[n^k]$$

**Def 3.6.**

$$PSPACE = \bigcup_{k=0}^{\infty} DSpace[n^k]$$

**Def 3.7.**

$$NPSPACE = \bigcup_{k=0}^{\infty} NSpace[n^k]$$

**Теорема 3.1.** Если $f(n)\log f(n) = o(g(n))$ (а $f$ и $g$ конструктивны по времени), то $\mathit{DTime}[f(n)] \subsetneq \mathit{DTime}[g(n)]$

▶ **TODO** ◀

**Теорема 3.2.** Если $f(n) = o(g(n))$ и для достаточно больших $n$ верно, что $g(n) \geqslant \log n$, то $\mathit{DSpace}[f(n)] \neq \mathit{DSpace}[g(n)]$.

▶ **TODO** ◀

Это про строгую вложенность $\mathit{DTime}[f(n)]$ (если функции отличаются на логарифм), $\mathit{NTime}[f(n)]$ (если функции отличаются чуть-чуть асимптотически в $n+1$ и $n$), $\mathit{NSpace}[f(n)]$ и $\mathit{DSpace}[f(n)]$ (если функции отличаются асимптотически)

**TODO** разделы 3.1-3.3 в Arora-Barak, теорема об иерархии по памяти там доказывается одновременно для $\mathit{NSpace}$ и $\mathit{DSpace}$

# 4. Билет 4

## 4.1. $\mathit{Circuit}-\mathit{SAT}$

**Def 4.1.** Массовая задача $\mathit{Circuit}-\mathit{SAT}$ — это множество пар $(C, \vec{x})$ таких, что:

1. $C$ — описание ацикличной булевой схемы с выделенными $k$ входами и одним выделенным выходом. Возможные гейты — какой-нибудь конечный базис (например, «и», «или», «не»), каждый элемент базиса имеет фиксированное число входов и один выход (который, впрочем, можно подключать куда угодно в схеме сколько угодно раз). Неважно, как базис брать за определение, всё будет эквивалетно.

2. $\vec{x}$ — битовый вектор длины $k$ такой, что если подать соответствующие биты на входы схемы и вычислить значения гейтов, то на выходном гейте будет единица.

*Замечание* 4.1. Например, элемент «и всех входов» для произвольного числа входов в схему брать мы не можем.

**Лемма 4.1.** $\mathit{Circuit}-\mathit{SAT} \in \mathit{NP}$

▶ Построим НМТ: она вначале сделает $k$ недетерминированных решений (выберет значения входных гейтов и запишет их на ленту), а дальше за линейное от размера схемы время вычислит значение выходного гейта (скажем, DFS'ом по гейтам, начиная с конечного). ◀

**Теорема 4.1.** $\mathit{Circuit}-\mathit{SAT}$ — $\mathit{NP}$-полная

▶ Нам осталось показать, что эта задача $\mathit{NP}$-трудная.

Возьмём произвольную задачу $L \in \mathit{NP}$, для неё, по определению, есть НМТ $M$, которая за время не большее $p(|x|)$ говорит «да», если слово действительно лежит в языке $L$.

Сведём $L$ к $\mathit{Circuit}-\mathit{SAT}$ (по Карпу). Пусть дали слово $x$ длины $n$, вычислим $t := p(|x|)$. Построим схему с $t$ входами размера $\mathcal{O}(t^2)$, эмулирующую работу $M$ в течение $t$ шагов. Входами схемы будет описание того, какой переход на каком шаге выбирает машина $M$ — всего $t \cdot |Q|$ бит (см. ниже).

1. Так как $M$ работает не более $t$ шагов, то она использует не более $t$ памяти (возможно, надо домножить на число лент, это ок).

2. Значит, конфигурацию $M$ (память лент, позиции головок и состояние) можно закодировать в виде битовой строчки константной длины. Будет удобно кодировать позицию лент не номером, используя $\log t$ битов, а последовательностью из $t$ бит, из которых ровно один равен единице — тот, где стоит головка. Аналогично будет удобно кодировать текущее состояние — не $\log|Q|$ битов, а $|Q|$ битов.
3. Дальше можно построить схему, которая по двумя конфигурациям и текущему переходу выдаёт единицу тогда и только тогда, когда из одной можно перейти в другую за один шаг таким переходом:
   (a) Сначала надо прочитать бит под каждой головкой и вывести его в отдельный гейт для головки — это мы сначала берём логическое «и» ленты с маской положения головки, остаётся один бит, а потом берём логическое «или» этих битов.
   (b) Потом надо проверить, что переход действительно ожидает эти биты под головками и это состояние.
   (c) Потом надо проверить, что конечное состояние действительно такое, как ожидает переход.
   (d) Потом надо проверить, что биты под головками поменялись правильным образом, головки правильно подвинулись.
   (e) Потом надо проверить, что биты не под головками остались неизменны.
4. Дальше делаем $t$ слоёв схемы, каждый слой — это гейты, кодирующие конфигурацию $M$. Первый слой фиксирован (исходная конфигурация, в неё зашит вход $x$), между соседними слоями стоит проверка корректности перехода, конечный слой должен принимать слово.

Теперь получаем, что наша схема выдаёт единицу тогда и только тогда, когда подсказка, поданная схеме на вход, заставляет машину $M$ пройти через $t$ шагов и завершиться в принимающем состоянии, что и требовалось. ◀

## 4.2. $3-SAT$

**Def 4.2.** Массовая задача $3-SAT$ — это множество таких пар $(\varphi, \vec{x})$, что:

1. $\varphi$ — это булева формула в 3-КНФ от $n$ переменных. Это как КНФ, но в каждом клозе формулы присутствует не более трёх термов.
2. $\vec{x}$ — вектор из $n$ бит, причём $\varphi(\vec{x}) = 1$.

**Лемма 4.2.** $3-SAT \in NP$

▶ Построим НМТ: она в начале делает $n$ недетерминированных переходов, выбирая значения переменных $x$ (и, например, записывая их на ленту), а потом посчитает значение формулы по значениям переменных, проверит, что получилась единица ◀

**Лемма 4.3.** Для произвольной функции $f(x,y)$ от двух бит условие $z = f(x,y)$ можно записать в 3-КНФ.

▶ Заметим, что выражение $(x \land y) \to z$ легко переписывается в 3-КНФ:

$$(x \land y) \to z$$
$$\Updownarrow$$
$$(\neg x \lor \neg y \lor z)$$

Доказать можно, расписав таблицу истинности.

Теперь давайте переберём четыре значения $(x,y)$, посчитаем $f(x,y)$ и запишем конъюнкцию четырёх 3-КНФ. Например, если $f(0,1) = 0$, то надо дописать 3-КНФ для выражения $(\neg x \land y) \to \neg z$. ◀

**Теорема 4.2.** $3 - SAT$ — $NP$-полная задача.

▶ Сведём $Circuit - SAT$ и $3 - SAT$.

Пусть нам дали схему с $n$ входами и $m$ гейтами (помимо входов). У нашей формулы будет $n+m$ переменных: $x_1, \dots, x_n, y_1, \dots, y_m$. Давайте запишем условия на $y_i$ в 3-КНФ по предыдущей лемме: мы знаем, как вычисляется каждый $y_i$ через остальные переменные. Также допишем в конец получившейся 3-КНФ значение $y_i$, соответствующего выходному гейту.

Т.о. наша формула будет истинна тогда и только тогда, когда все $y_i$ являются корректно посчитанными значениями гейтов и на выходном гейте единица. Значит, она выполнима тогда и только тогда, когда выполнима исходная схема. ◀

*Замечание* 4.2. Тут мы за полиномиальное время сводили только задачу выполнимости — у нас даже число переменных изменилось. Если бы хотели вычислить схему на произвольном входе формулой, то формула бы наверняка экспоненциально разраслась. Тут мы этого избежали за счёт увеличения числа переменных.

*Замечание* 4.3. Теорема Кука-Левина: задача $SAT$ является $NP$-полной. Выше мы доказали чуть более сильное утверждение.

# 5. Билет 5

**Теорема 5.1.** Пусть есть массовая задача $R \in \widetilde{NP}$, причём $L(R)$ — $NP$-полон. Пусть также есть алгоритм $A$ для разрешения языка $L(R)$. Тогда имеется алгоритм $B$ для решения задачи поиска $R$, работающий не более, чем в полином раз медленнее алгоритма $A$.

*Замечание* 5.1. Смысл такой: есть мы для $NP$-полной задачи умеем проверять существование ответа, то искать ответ мы умеем не сильно медленнее.

▶ Мы знаем, что так как $L(R) \in NPC$, то к $L(R)$ можно свести задачу $SAT$, т.е. есть такая п.о. полиномиально вычислимая $f$, что для любой формулы $\varphi$:

$$\varphi \in SAT \iff f(\varphi) \in L(R)$$

Значит, у нас есть алгоритм для проверки выполнимости формулы не медленнее, чем в полином раз медленнее $A$. Давайте теперь научимся, используя этот алгоритм, находить решение. Это просто: мы берём формулу, проверяем выполнимость. Если выполнима, то подставляем $x_1 = 0$, если формула всё ещё выполнима — мы угадали одну переменную, идём дальше. Иначе точно надо подставить $x_1 = 1$ и начать угадывать следующие переменные. Всего у нас будет не более $k$ запусков проверки на выполнимость, если имеется $k$ переменных.

Т.о. есть алгоритм $A'$, который ищет решение $SAT$ довольно быстро.

Давайте учиться по условию $x$ для задачи $R$ быстро искать ответ. У нас есть ДМТ $M$, которая по входу из условия и решения $R$ проверяет корректность. Аналогично доказательству теоремы Кука-Левина (4 билет) можно построить булеву формулу, эмулирующую $M$. В неё будет зашито условие $x$, а переменные будут в точности задавать решение для входа, формула будет верна тогда и только тогда, когда решение действительно является решением. А для этой формулы мы можем применить алгоритм $A'$, который быстро найдёт решение. ◀

*Замечание* 5.2. Краткая схема доказательства:

1. Сводим $SAT$ к нашему $NP$-полному языку, теперь умеем быстро проверять выполнимость.
2. Угадываем значения переменных по одной, теперь мы умеем быстро искать решения для булевых формул.

3. Строим по машине, проверяющей решение $R$, эквивалентную булеву формулу, решением которой будут являться в точности решения исходной задачи. Мы уже умеем быстро искать решения для булевых формул, успех.

*Замечание* 5.3. $\mathsf{NP}$-полнота существенна, например, про задачу поиска нетривиальных делителей $\mathsf{FACTOR}$ такого неизвестно. Мы знаем детерминированные полиномиальные (от длины числа) алгоритмы проверки на простоту (ABS-тест), но вот искать делители так же эффективно не умеем.

# 6. Билет 6

**𝔇𝔢𝔣 6.1.** Оракульная машина Тьюринга с оракулом для произвольной функции $f$ (не обязательно вычислимой) — это обычная машина Тьюринга (детерминированная или недетерминированная — неважно), у которой добавляется дополнительный переход «вызови оракула для функции $f$ на такой ленте и запиши ответ на такую ленту».

Обозначается как $M^f$ — машина $M$, у которой есть оракул $f$.

*Пример* 6.1. Например, можно в качестве оракула взять характеристическую функцию языка $3-\mathsf{SAT}$, тогда наша МТ сможет за один шаг узнавать выполнимость произвольной формулы в 3-КНФ.

*Замечание* 6.1. Можно брать несколько оракулов, не меняя определения: сказать, что у нас на вход оракулу помимо данных подаётся ещё и номер задачи, которую надо решать.

*Замечание* 6.2. Часто пишут $M^C$, где $C$ — целый класс языков. Подразумевается, что для каждой машины берётся какой-то один язык из $C$.

**𝔇𝔢𝔣 6.2.** Если $\mathcal{C}$ и $\mathcal{D}$ — классы языков, то можно ввести класс $\mathcal{C}^{\mathcal{D}}$, который состоит из языков вида $C^D$, где $D \in \mathcal{D}$ (язык), а $C$ — решатель для некоторого языка из $\mathcal{C}$.

*Пример* 6.2. $\mathsf{P}^A$ — класс языков, распознаваемых ДМТ с оракулом $A$ за полиномиальное время.

*Пример* 6.3. $\mathsf{NP}^A$ — класс языков, распознаваемых НМТ с оракулом $A$ за полиномиальное время.

**Утверждение 6.1.** Если $A \subseteq B$ — классы языков, то $C^A \subseteq C^B$.

▶ Должно быть очевидно — любой решатель с оракулом из $C^A$ должен лежать в $C^B$. ◀

*Замечание* 6.3. **Опасный момент**: то, что стоит внизу в обозначении оракула — не конкретный класс, а лишь модель вычислений, в которую добавляется оракул. В случае $\mathsf{P}$ это ДМТ, в случае $\mathsf{NP}$ это НМТ. Там можно поставить и другие классы со своими моделями вычислений. Однако не надо думать про это как про «основание» степени: например, даже если $\mathsf{P} = \mathsf{NP}$, то вовсе необязательно $\mathsf{P}^A = \mathsf{NP}^A$, модели вычислений-то совсем разные, а добавление оракула их ещё больше меняет.

**Теорема 6.1.** Есть такой язык $A$, что $\mathsf{P}^A = \mathsf{NP}^A$.

▶ Пусть $A$ — это язык из троек $(M, x, 1^n)$, причём ДМТ $M$ принимает вход $x$ быстрее, чем за $2^n$ шагов.

**𝔇𝔢𝔣 6.3.** $\mathsf{EXP}$ — это все языки, распознаваемые ДМТ за время $2^{poly(|x|)}$:

$$\mathsf{EXP} = \bigcup_{poly(n)} \mathsf{DTime}[2^{poly(n)}]$$

**Утверждение 6.2.** $A \in \mathsf{EXP}$.

▶ Просто запускаем ДМТ на входе, она делает не более $t := 2^n$ шагов, а моделировать МТ мы умеем за $\mathcal{O}(t^2) = \mathcal{O}(2^{2n}) = \mathcal{O}(2^{poly(n)})$. ◀

Покажем, что $EXP \subseteq P^A \subseteq NP^A \subseteq EXP$, откуда сразу $P^A = NP^A = EXP$:

**$EXP \subseteq P^A$:** Есть язык $L \in EXP$ и пусть нам дали слово $x$. По определению $EXP$ есть машина, которая принимает слова из языка за $2^{poly(|x|)}$, можно вычислить этот $poly(|x|)$ и спросить у оракула, примет ли машина слово за такое время — это и будет ответом.

**$P^A \subseteq NP^A$:** Очевидно, так как любая ДМТ с оракулом также является НМТ с оракулом и распознаёт тот же самый язык.

**$NP^A \subseteq EXP$:** Пусть есть язык $L \in NP^A$. Пусть нам дали слово $x$. Мы знаем, что если есть подсказка для НМТ, то она не длиннее, чем $p(|x|)$ ($p$ зависит только от $L$). Переберём все такие подсказки за $2^{p(|x|)}$, каждую проверим на корректность. Для проверки корректности требуется промоделировать не более $q(|x|)$ шагов НМТ, на каждом шаге может потенциально быть запущен оракул для входа размером не более $q(|x|)$, который отработает за $2^{q_1(q(|x|))}$, где $q_1$ — фиксированный многочлен. Значит, итоговое время работы будет не более:

$$2^{p(|x|)} \cdot q(|x|) \cdot 2^{q_1(q(|x|))} \leqslant 2^{p(|x|)+q(|x|)+q_1(q(|x|))}$$

Т.е. двойка в степени полином, что и требовалось.

◀

**Лемма 6.1.** Возьмём произвольный язык $B$ (возможно даже неразрешимый) и построим унарный язык $U_B$:

$$U_B = \{1^{|x|} \mid x \in B\}$$

Т.е. это все слова из $B$, в которых биты были заменены на единички. Тогда если $U_B \notin P^B$, то $P^B \neq NP^B$.

▶ Достаточно заметить, что $U_B \in NP^B$, тогда строгое включение будет доказано. Но это очевидно: можно считать, что НМТ требует в качестве подсказки для слова $1^n$ какое-нибудь слово $x$ длины $n$ из $B$ и при помощи оракула проверяет, что оно действительно лежит в $B$. Также НМт надо проверить, что ему на вход подали одни единицы. ◀

**Теорема 6.2.** Есть такой язык $B$, что $P^B \neq NP^B$.

▶ Воспользуемся предыдущей леммой и построим $B$ с нужным свойством ($U_B \notin P^B$). Строить его будет по шагам, на каждом шаге у нас для конечного множества строк будет решено, лежат они в $B$ или нет. Изначально это не известно ни для одной из строк. Для произвольной строки мы либо добавим/исключим её в $B$ на каком-то шаге, либо она никогда не возникнет. Если она никогда не возникнет, то мы считаем, что она не лежит в $B$.

Занумеруем машины с оракулом $B$ как $M_1, M_2, \dots$ (каждая машина имеет бесконечно много номеров). На $i$-м шаге мы будет достраивать $B$ так, чтобы машина $M_i$ не разрешала язык $B$ за время $2^n/10$ (при помощи диагонализации). Так как $M_i$ будет использовать оракул для $B$, то при каждом вызове оракула нам надо помещать или не помещать аргумент оракула в $B$.

Итак, $i$-й шаг. У нас для конечного множества строк известно, лежат ли они $B$ или нет. Выберем натуральное $n$, большее, чем длины все известных строк. Запустим $M_i$ на входе $1^n$ в течение $2^n/10$ шагов. Если она запустит оракул на строке, про которую мы уже что-то знаем, мы обязаны ответить корректно. Если же оракул запускается на строке, про которую мы ещё ничего не знаем, мы начинаем считать, что эта строка в $B$ не лежит. После того, как машина $M_i$ закончила работать, ни про одну из строк длины $n$ мы ещё не могли решить, что она лежит в $B$ (впрочем, про какие-то могли решить, что не лежат). Два варианта завершения работы $M_i$:

1. $M_i$ приняла строку $1^n$. Тогда говорим, что все строки длины $n$ не лежат в $B$. Никаких противоречий не возникнет, так как мы про строки длины $n$ не говорили, что они лежат в $B$. Получаем, что $M_i$ сказала неправду про язык $U_B$.

2. $M_i$ отвергла строку $1^n$. Тогда заметим, что точно существует строка длины $n$, про которую ещё ничего не известно (т.е. про неё не спрашивал оракул машины $M_i$), так как всего он спрашивал максимум про $2^n/10$ строк, а всего их $2^n$. Тогда скажем, что эта строка лежит в $B$. Получавем, что $M_i$ сказала неправду про язык $U_B$.

Мы так построили какой-то язык $B$. Докажем условие леммы от противного. Предположим, что существует полиномиальная машина $M$, разрешающая язык $U_B$ за полиномиальное время. Тогда при достаточно большом $k$ она начнёт отрабатывать за время, меньшее $2^k/10$. Тогда посмотрим на первый шаг после $k$-го, на котором встретилась эта машина $M$. На этом шаге машина точно завершилась, но по построению языка $B$ она выдала неправильный ответ. Противоречие.

Условие леммы доказано, стало быть, $\mathsf{P}^B \neq \mathsf{NP}^B$. ◀

# 7. Билет 7

**Def 7.1.** Задача $A$ *сводится по Тьюрингу* к задаче $B$, если есть машина $M^B$ решающая задачу $A$.

*Замечание* 7.1. Про время работы вообще ничего не говорим. Так можно сводить не только языки (т.е. задачи распознавания), но и задачи поиска.

*Замечание* 7.2. Это сведение для языков — более общий случай сведения по Карпу (там мы разрешаем вызвать оракула только один раз, только в конце, и ничего не можем делать с его ответом).

**Утверждение 7.1.** Классы $\mathsf{P}$ и $\widetilde{\mathsf{P}}$ замкнуты относительно сведения по Тьюрингу, т.е. $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$ и $\widetilde{\mathsf{P}}^{\widetilde{\mathsf{P}}} = \widetilde{\mathsf{P}}$

▶ Мы можем заменить все магические вызовы оракула на явный запуск оракула. Пусть машина работала $t$ шагов, где $t \leqslant p(n)$ ($n$ — длина входа машины). Тогда, очевидно, она не может скормить оракулу строчку длиннее $t$. Так как оракул работает за $q(k)$ (где $k$ — длина *входа оракула*, а $p$ — полином), то на каждом входе оракул будет работать не более $q(p(n))$ — тоже какой-то полином.

Если мы вместо обращения к оракулу будем явно вычислять функцию, то получим замедленее не более, чем в $q(p(n))$ раз, т.е. всё равно полином. ◀

*Замечание* 7.3. Обратите внимание, что в общем случае оракул может получать гораздо более длинный вход, чем сама машина получила на вход. Длина входа оракула ограничена только временем работы.

*Замечание* 7.4. А вот про классы $\mathsf{NP}$ и $\widetilde{\mathsf{NP}}$ мы такой замкнутости не знаем.

**Теорема 7.1.** • Для любой задачи поиска $R$ из $\mathsf{NP}$:

- Существует Ле́винский оптимальный алгоритм поиска $A$ (детерминированный) со следующим свойством:
- Для любого другого алгоритм $B$, решающего задачу $R$:
- Существует такой полином $p$, что:
- На любом входе $x$, где есть ответ:
- Время работы $A$ не более чем $p(t)$, где $t$ — время работы $B$.

▶ $A$ будет перебирать вообще все детерминированные машины: не только полиномиальные, не только останавливающиеся, а вообще все. Это можно сделать, перебирая все конечные строки (например, для описаний, не являющимися корректными машинами, можно считать, что они имеют право вести себя как угодно). Назовём эти машины $M_0, \dots, M_n, \dots$.

Давайте будем на шаге с номером $i = 2^l(1+2k)$ моделировать $k$-й шаг машины $M_l$, то есть на нечётных шагах моделируем шаги $M_1$, на делящихся только на 2 (но не на 4) — шаги $M_2$, на делящихся на 4 — $M_3$ и так далее. Если какая-нибудь машина завершается, то мы смотрим на её ответ и проверяем, не является ли он ответом на задачу. Если является, то выдаём, иначе продолжаем эмуляцию.

Заметим, что если есть какой-то алгоритм поиска (а какой-то точно есть — можно просто перебрать все решения), то мы на входе $x$ точно когда-то завершимся. Давайте предположим, что есть алгоритм $B$, которому соответствует машина $M_a$ и который на входе $x$ завершится за время $t$. Тогда мы обнаружим этот факт через $2^a(1+2t)$ шагов алгоритма (если не найдём ответ раньше), обозначим это число за $n$. Тогда заметим, что на каждом из предыдущих шагов мы тратили константное время не более $\mathcal{O}(n)$ времени на эмуляцию очередного шага очередной МТ и не более $\mathcal{O}(p(n))$ времени на проверку возможно выданного ответа (так как ответ не может быть больше времени работы МТ, а его проверка делается за полином). Значит, всего у нас время работы до завершения оптимального алгоритма на входе $x$ не более $n \cdot \mathcal{O}(n) \cdot \mathcal{O}(p(n))$, что есть полином от $n$, что есть полином от $t$, что и требовалось показать. ◀

*Замечание* 7.5. На входах без ответа наш алгоритм зацикливается.

*Замечание* 7.6. Это полезное знание, если у есть конечное число оптимальных алгоритмов $A_1, \dots, A_k$, причём каждый из алгоритмов быстро работает на каком-то своём множестве входов (но тормозит на других). Тогда оптимальный алгоритм будет работать на каждом выходе не более, чем в полином раз медленнее, чем *каждый* из этих алгоритмов. А так как их конечно, то мы даже можем сказать, что оптимальный алгоритм на каждом входе не сильно хуже, чем оптимальный для этого входа.

# 8. Билет 8

**Теорема 8.1.** Если $P \neq NP$, то есть задача из $NP \setminus P$, не являющаяся $NP$-трудной (и, следовательно, не являющаяся $NP$-полной).

▶ **TODO** хитрая диагонализация с очень медленно растущей функцией и явным построением такой задачи.

См. раздел 1.11 в конспекте Оли, раздел 2.2 в конспекте logic.pdmi.ras.ru/ hirsch/students/complexity1/
◀

# 9. Билет 9

## 9.1. Определения

**𝔇𝔢𝔣 9.1.** Если $L$ — язык, то $\bar{L}$ — его *дополнение*:

$$x \in L \iff x \notin \bar{L}$$

**𝔇𝔢𝔣 9.2.** Если $C$ — класс языков, то co-$C$ — класс дополнений:

$$L \in C \iff \bar{L} \in \text{co-}C$$

*Пример* 9.1. Язык $SAT$ лежит в $NP$. А его дополнение (это все невыполнимые булевы формулы плюс строчки, не кодирующие булевы формулы) лежит в co-$NP$.

**Утверждение 9.1.** Если $D$ — класс языков, то:

$$C^D = C^{\text{co-}D}$$

▶ Мы можем просто после каждого вызова оракула инвертировать ответ. ◀

**𝔇𝔢𝔣 9.3.** Полиномиальная иерархия состоит из трёх семейств классов, определяемых рекурсивно (при $i = 0$ по определению):

$$\begin{array}{lll}
\Sigma_0 = P & \Pi_0 = P & \Delta_0 = P \\
\Sigma_1 = NP^P & \Pi_1 = \text{co-}NP^P & \Delta_1 = P^P \\
\Sigma_2 = NP^{\text{co-}NP^P} & \Pi_2 = \text{co-}NP^{NP^P} & \Delta_2 = P^{NP^P} \\
\Sigma_3 = NP^{\text{co-}NP^{NP^P}} & \Pi_2 = \text{co-}NP^{NP^{\text{co-}NP^P}} & \Delta_2 = P^{NP^{\text{co-}NP^P}} \\
\vdots & \vdots & \vdots \\
\Sigma_{i+1} = NP_{\Pi^i} & \Pi_{i+1} = \text{co-}NP_{\Sigma^i} & \Delta_{i+1} = P^{\Sigma^i P}
\end{array}$$

*Замечание* 9.1. Класс $\Delta_i$ мы нигде не используем. Также классы могут обозначаться как $\Sigma^i P$ вместо $\Sigma_i$, но так длиннее писать.

**Утверждение 9.2.** При $i \geqslant 0$:

$$\begin{aligned}
\Sigma_i &= \text{co-}\Pi_i \\
\Sigma_{i+1} &= NP^{\Sigma_i} = NP^{\Pi_i} \\
\Pi_{i+1} &= \text{co-}NP^{\Sigma_i} = \text{co-}NP^{\Pi_i}
\end{aligned}$$

▶ Индукция по $i$. При $i = 0$ всё очевидно, в переходе мы просто пользуемся определением $\Sigma_i$ и $\Pi_i$ и тем фактом, что в оракуле можно менять класс языков на его дополнение. ◀

**𝔇𝔢𝔣 9.4.** Класс $PH$ (polynomial hierarchy):

$$PH = \bigcup_{i \geqslant 0} \Sigma_i$$

*Пример* 9.2.

$$\begin{aligned}
\Sigma_2 &= \Sigma^2 P = NP^{NP} = NP^{\text{co-}NP} \\
\Pi_2 &= \Pi^2 P = \text{co-}NP^{NP} = \text{co-}NP^{\text{co-}NP}
\end{aligned}$$

**Утверждение 9.3.**

$$PH = \text{co-}PH$$

▶ Включение co-$PH \subseteq PH$:

$$\text{co-}PH = \text{co-}\bigcup_{i \geqslant 0} \Sigma_i \subseteq \text{co-}\bigcup_{i \geqslant 0} \Pi_{i+1} = \bigcup_{i \geqslant 1} \text{co-}\Pi_i = \bigcup_{i \geqslant 1} \Sigma_i \subseteq \bigcup_{i \geqslant 0} \Sigma_i = PH$$

Включение co-$PH \supseteq PH$:

$$\text{co-}PH = \text{co-}\bigcup_{i \geqslant 0} \Sigma_i \supseteq \text{co-}\bigcup_{i \geqslant 1} \Sigma_i \supseteq \text{co-}\bigcup_{i \geqslant 1} \Pi_{i-1} = \bigcup_{i \geqslant 0} \text{co-}\Pi_i = \bigcup_{i \geqslant 0} \Sigma_i = PH$$

◀

**Теорема 9.1.** При $k \geqslant 1$:

- $L \in \Sigma_k$ тогда и только тогда, когда есть п.о. отношение $R \in \Pi_{k-1}$ такое, что:

$$x \in L \iff \exists y\colon R(x, y)$$

- $L \in \Pi_k$ тогда и только тогда, когда есть п.о. отношение $R \in \Sigma_{k-1}$ такое, что:

$$x \in L \iff \forall y\colon R(x, y)$$

*Замечание* 9.2. Это второе определение, через кванторы. Если его рекурсивно раскрыть, то получим, что $\Sigma_k$ — это чередующуюся цепочка из $k$ кванторов, начиная с $\exists$, а заканчивая п.о. п.п. отношением $R \in \mathsf{P}$. А $\Pi_k$ — то же самое, но начинается с $\forall$.

▶ Показываем эквивалентность индукцией по $k$, база при $k = 0$ и $k = 1$ очевидна.

Переход от $k - 1$ к $k$ (при $k \geqslant 2$). Сначала разбираемся с $\Sigma_k$. Надо показать в две стороны:

**⇐:** Пусть есть язык $L$ и отношение $R \in \Pi_{k-1}$ из условия. Построим машину из $\mathsf{NP}^R \subseteq \Sigma_k$, которая распознаёт язык $L$. Она будет недетерминированно угадывать $y$ и потом спрашивать у оракула корректность $R(x, y)$.

**⇒:** Пусть есть язык $L$ и НМТ $M$, использующая оракул для языка $L' \in \Sigma_{k-1}$ (тут мы пользуемся тем, что $X^{\Pi_{k-1}} = X^{\Sigma_{k-1}}$).

*Замечание* 9.3. Если бы $M$ не использовала оракул для $\Sigma_{k-1}$, а использовал бы его только для $\Sigma_{k-2}$, то переход был бы очевиден: в качестве отношения $R(x, y)$ можно было бы взять следующее: машина $M$ принимает вход $x$ по подсказке $y$. Тогда это отношение бы лежало в $\mathsf{P}^{\Sigma_{k-2}} \subseteq \text{co-}\mathsf{NP}^{\Sigma_{k-2}} = \Pi_{k-1}$. Однако нам так не повезло, надо чуть аккуратнее.

По индукционному предположению о $\Sigma_{k-1}$ это значит, что есть такое п.о. отношение $R' \in \Pi_{k-2}$:

$$x' \in L' \iff \exists y'\colon R'(x', y')$$

Формально говоря, $M$ принимает слово $x$ тогда и только тогда, когда есть последовательность недетерминированных выборов $y_0$, а также корректные ответы оракула для $L'$ на входах $x'_1, \ldots, x'_l$ (где его запускала $M$): $a_1, \ldots, a_l$.

Давайте введём отношение $R(x, y)$, а в $y$ закодируем следующее:

1. Саму последовательность детерминированных выборов $y_0$
2. Последовательность $a_i$
3. Для каждого $a_i = 1$ закодируем соответствующий $y'_i$ для отношения $R'$, позволяющий проверить ответ оракула для языка $L'$

Осталось лишь сделать так, чтобы $R$ могло убедиться в том, что для $a_i = 0$ ответ действительно ноль, т.е. что для всех подсказок $y'$ отношение $R'$ возвращает ноль. Скажем, что $R(x, y)$ имеет такой вид:

1. Для любых $y''_1, \ldots, y''_l$:
2. Для всех $i$ таких, что $a_i = 1$ верно $R'(x'_i, y'_i) = 1$
3. Для всех $i$ таких, что $a_i = 0$ верно $R'(x'_i, y''_i) = 0$
4. НМТ $M$ принимает вход $x$ по подсказке $y$, если ответы оракула на входах $x'_i$ равны $a_i$

Заметим, что это отношение лежит в $\text{co-}\mathsf{NP}^{R'}$ по построению — мы требуем принятие по всем подсказкам, иногда вызываем оракул для $R'$, и делаем дополнительную полиномиальную работу на последнем шаге. Т.о. $R \in \text{co-}\mathsf{NP}^{\Pi_{k-2}} = \Pi_{k-1}$, что и требовалось показать.

Аналогичным образом должно получаться и с $\Pi_k$. **TODO** ◀

## 9.2. Полные задачи

**𝔇𝔢𝔣 9.5.** Язык $QBF_k$ состоит из замкнутых булевых формул в предварённой форме с $k$ чередующимися кванторами, начиная с $\exists$, а под всеми кванторами — формула в ДНФ или КНФ.

*Замечание* 9.4. Один квантор может быть по нескольким переменным одновременно, важно именно количество смен кванторов.

**Утверждение 9.4.** Язык $QBF_k$ является $\Sigma_k$-полным.

▶ Возьмём произвольный язык $L \in \Sigma_k$. Мы хотим построить полиномиально вычислимую функцию $f$ такую, что:

$$x \in L \iff f(x) \in QBF_k$$

По определению $\Sigma_k$ мы знаем, что есть такое п.о. п.п. $R \in P$, что:

$$x \in L \iff \exists x_1 \forall x_2 \dots Q_k x_k \colon R(x, x_1, \dots, x_k)$$

Хочется каким-то образом преобразовать $R$ в формулу. Если бы нам разрешалось делать не формулы, а схемы, то мы бы могли, как в билете 4, построить схему квадратичного размера, которой на вход подаются $x_1, \dots, x_k$, в неё саму зашит $x$, и она вычисляет $R$. К сожалению, мы не умеем переделывать схемы в формулы напрямую — они экспоненциально разрастаются. Однако мы умеем (как в билете 4) делать это с добавлением промежуточных переменных, т.е. можно построить такую $\varphi$ полиномиального размера, что:

$$R(x, x_1, \dots, x_k) \iff (\exists y \colon \varphi(x, x_1, \dots, x_k, y))$$

Таким образом, если если $k$ нечётно, то последний квантор в формуле будет квантором существования и мы можем просто навесить ещё один, не сбивая чередование, получим честную формулу для $QBF_k$.

А вот если $k$ чётно и последний квантор — квантор всеобщности, то надо поступить хитрее. Надо записать $\neg R$ в виде булевой формулы $\varphi'$:

$$\begin{gathered}\neg R(x, x_1, \dots, x_k) \\ \Updownarrow \\ \exists y \colon \varphi'(x, x_1, \dots, x_k, y) \\ \Updownarrow \\ \neg \forall y \colon \neg\varphi'(x, x_1, \dots, x_k, y) \\ R(x, x_1, \dots, x_k) \iff \forall y \colon \neg\varphi'(x, x_1, \dots, x_k, y)\end{gathered}$$

Опять получим несбитое чередование кванторов, что и требуется. ◀

## 9.3. Коллапс

**Утверждение 9.5.** Если $PH = \Sigma_k$, то $PH = \Pi_k$

▶ Мы знаем, что co-$PH = PH$.

$$\begin{gathered}PH = \Sigma_k \\ \text{co-}PH = \text{co-}\Sigma_k \\ PH = \Pi_k\end{gathered}$$

◀

**Теорема 9.2.** Если $\Sigma_k = \Pi_k$ (при $k > 0$), то $PH = \Sigma_k = \Pi_k$ (иерархия коллапсирует до уровня $k$)

▶ Сначала покажем, что $\Sigma_{k+1} = \Pi_k$. Мы точно знаем нестрогое включение ($\Sigma_{k+1} \supseteq \Pi_k$, из определения с оракулами), покажем во вторую сторону. Пусть $L \in \Sigma_{k+1}$, т.е. есть такое $R \in \Pi_k = \Sigma_k$:

$$L = \{x \colon \exists y \colon R(x, y)\}$$

Так как $R \in \Sigma_k$, то есть такое $R' \in \Pi_{k-1}$, что:

$$R(x, y) \iff \exists z \colon R'(x, y, z)$$

Значит:

$$x \in L \iff \exists (y, z) \colon R'(x, y, z)$$

Отсюда немедленно $L \in \Sigma_k$.

Теперь делаем следствия:

$$\Sigma_{k+1} = \Pi_k = \Sigma_k$$
$$\text{co-}\Sigma_{k+1} = \text{co-}\Pi_k$$
$$\Pi_{k+1} = \Sigma_k = \Pi_k$$
$$\Sigma_{k+1} = NP^{\Sigma_k} = \Sigma_k$$
$$\Pi_{k+1} = \text{co-}NP^{\Sigma_k} = \Pi_k$$

Из последних двух строк автоматически получаем коллапс полиномиальной иерархии:

$$\Sigma_{k+i} = \Sigma_k = \Pi_k = \Pi_{k+i}$$

◀

**Теорема 9.3.** Если $\Sigma_k = \Sigma_{k+1}$, то $PH = \Sigma_k$ (иерархия коллапсирует до $\Sigma_k$)

▶ Как и в предыдущей теореме, достаточно показать, что $NP^{\Sigma_k} = \Sigma_k$ и $\text{co-}NP^{\Sigma_k} = \Pi_k$, тогда рекурсивное определение зациклится.

Первое нам дано в условии:

$$\Sigma_k = \Sigma_{k+1} = NP^{\Pi_k} = NP^{\Sigma_k}$$

Второе из него следует:

$$\text{co-}NP^{\Sigma_k} = \Pi_{k+1} = \text{co-}\Sigma_{k+1} = \text{co-}\Sigma_k = \Pi_k$$

◀

*Следствие* 9.3.1. Как следствие, иерархия коллапсирует до уровня $k$, так как если $PH = \Sigma_k$, то $PH = \Pi_k$.

*Следствие* 9.3.2. Если есть $\Sigma_{k+1}$-полная задача, лежащая в $\Sigma_k$, то иерархия коллапсирует до уровня $k$.

▶ Так как любая задача из $\Sigma_{k+1}$ сводится к задаче из $\Sigma_k$, то $\Sigma_{k+1} \subseteq \Sigma_k$ (так как уровни иерархии замкнуты относительно сведения). С другой стороны, $\Sigma_{k=1} \supseteq \Sigma_k$, стало быть, $\Sigma_{k+1} = \Sigma_k$ и иерархия коллапсирует до $\Sigma_k$. ◀

*Следствие* 9.3.3. Если существует $PH$-полная задача, то иерархия коллапсирует.

▶ Эта $PH$-полная задача лежит в конкретном уровне иерархии. Так как $\Sigma_{k+1} \supseteq \Pi_k$, то можно считать, что эта задача лежит в некотором $\Sigma_k$. Так как она $PH$-полная, то все задачи из $\Sigma_{k+1}$ к ней сводятся и иерархия коллапсирует до $\Sigma_k$. ◀

# 10. Билет 10

**Теорема 10.1.** Если $NP \subseteq P/poly$, то $PH = \Sigma_2$ (иерархия коллапсирует до $\Sigma_2$)

## 10.1. Идея

Покажем, что $\Sigma_3$-полный язык $QBF_3$ лежит в $\Sigma_2$, этого хватит для коллапса. Нам надо построить некоторое отношение $R \in P$ такое, что:

$$x \in QBF_3 \iff \exists \alpha\colon \forall \beta\colon R(x, \alpha, \beta)$$

Давайте рассуждать (точное решение будет ниже). Пусть нам дали формулу $F(x, y, z)$, хотим определить истинность следующего выражения (это задача $QBF_3$):

$$\exists x\colon \forall y\colon \exists z\colon F(x, y, z)$$

*Замечание* 10.1. Напоминаем, что $F$ — это булева формула от многих переменных, а $x$, $y$, $z$ — какие-то конечные множества булевых переменных. Очевидно, что $|x| + |y| + |z| \leqslant |F|$, иначе какая-то переменная в формуле не встречается и её можно убрать.

Заметим, что выражение $\exists z\colon F(x, y, z)$ — это в точности какая-то задача $A$ из $NP$ с условием $(F, x, y)$ и поиском решения $z$. Значит, по предпосылке теоремы, она лежит в $P/poly$, т.е. есть какое-то семейство схем $C_0, C_1, C_2, \ldots$ такое, что: схема $C_i$ решает $A$ для входов длины $i$:

$$\exists z\colon F(x, y, z) \iff (F, x, y) \in A \iff C_{|(F,x,y)|}((F, x, y)) = 1$$

К сожалению, мы не можем эти схемы напрямую использовать, несмотря на то, что $A$ — какая-то фиксированная задача, ни от чего не зависящая. Никто не сказал, что этим самые схемы можно конструктивно построить.

Схитрим: запихнём посторение этих схем в квантор существования. Их бесконечно много, но мы можем сказать, что нас интересуют только схемы до размера $\mathcal{O}(|F|)$, просто потому что так ограничен размер тройки $(F, x, y)$. Так как задача $A \in P/poly$, мы заранее знаем, каким полиномом ограничен размер каждой $C_i$, значит, можем завести нужное число переменных в формуле. Важно ещё как-то проверить, что имеющиеся схемы действительно решают задачу $A$. Это несложно сделать, так как схемы с небольшим входом можно просто встроить в формулу, а схемы с бо́льшими входами можно рекурсивно проверять через схемы с меньшими, например, должно быть верно следующее:

$$C(\forall x\colon \varphi(x))) = C(\varphi[x := 0]) \wedge C(\varphi[x := 1])$$

## 10.2. Точное решение

Нам надо показать, что есть такое п.о. п.п. отношение $R$, что:

$$x \in QBF_3 \iff \exists a\colon \forall b\colon R(x, a, b)$$

Перефразируя, надо построить такое $R$, что для любого $F$:

$$\exists a\colon \forall b\colon \exists c\colon F(a, b, c)$$
$$\Updownarrow$$
$$\exists \alpha\colon \forall \beta\colon R(F, \alpha, \beta)$$

*Замечание* 10.2. Тут латинскими буквами будут обозначаться последовательности из фиксированного числа булевых переменных (как в $QBF$ и вариациях), а греческими — битовые строки (подсказки для машин из полиномиальной иерархии).

Введём массовую задачу $A$ — это множество таких троек $(F, a, b)$ (тут $F$ — некоторая булева формула, а $a$ и $b$ — значения переменных для подстановки), что $\exists c\colon F(a, b, c)$. Очевидно, она лежит в $\mathsf{NP}$, так как по подсказке $(c)$ можно проверить решение. Значит, существует некоторое семейство формул полиномиального размера $C_0, C_1, C_2, \ldots$ такое, что:

$$\exists c\colon F(a,b,c)$$
$$\Updownarrow$$
$$C_{|(F,a,b)|}((F,a,b)) = 1$$

Раз семейство формул из $\mathsf{P/poly}$, то есть такой многочлен $p$, что $|C_i| \leqslant p(i)$. Значит, если мы рассмотрим схемы $C_0, \ldots, C_{|F|+|a|+|b|}$, то их можно закодировать битовой строкой $\gamma$ полиномиальной (от $|F|$) длины.

Введём специальное п.п. отношение $T(\gamma, \varphi)$, где $\varphi$ — произвольная формула длины не более $|F|$, использующая не более $|a| + |b|$ переменных (если использует больше, то полагаем $T = 1$). Это отношение проверяет, что описание схем $\gamma$ корректно решает $A(\varphi)$ в предположении, что для меньших формул ответ вычисляется корректно:

- Если в $\varphi$ нет переменных, то это какая-то константа, надо подставить в схему и проверить, что константа вычислена правильно
- Если в $\varphi$ есть переменные, то есть какой-то внешний квантор по переменной $x$. Надо вычислить $C(\varphi)$, $C(\varphi[x := 0])$ и $C(\varphi[x := 1])$ и проверить, что при комбинации последних двух значений получается первое. Комбинация — либо конъюнкция, либо дизъюнкция, в зависимости от квантора. Если это правда, то $C(\varphi)$ вычисляется правильно, если оба значения от меньших формул вычислены правильно.

*Замечание* 10.3. $T$ — это такой индукционный переход для доказательства корректности семейства схем $\gamma$.

Получаем следующую эквивалентность:

$$\exists a\colon \forall b\colon \exists c\colon F(a,b,c)$$
$$\Updownarrow$$
$$\exists \gamma, a\colon \forall \varphi, b\colon C_{|(F,a,b)|}(F,a,b) \land T(\gamma, \varphi)$$

Мы уже говорили, что $\gamma$ и $\varphi$ имеют полиномиальный размер относительно $|F|$, а выражение под кванторами тоже вычисляется за полином от $|F|$. Так что мы как раз решили задачу $\mathsf{QBF}_3$ в $\Sigma_2$, что и требовалось.

# 11. Билет 11

Раздел 1.11.2 в конспекте Оли.

**Теорема 11.1.** Для любого $n$ существует булева функция $f\colon \{0,1\}^n \to \{0,1\}$, которая не вычисляется схемами размера меньше $\frac{2^n}{10n}$

**Теорема 11.2.** Для любого $k$ неверно, что $\mathsf{PH} \subseteq Size[n^k]$.

**Теорема 11.3.** Для любого $k$ неверно, что $\Sigma_2 \cap \Pi_2 \subseteq Size[n^k]$.

*Замечание* 11.1. Это означает, что $\Sigma_2 \cap \Pi_2$ мы не умеем вычислять схемами какого-то фиксированного полиномиального размера. Про включение $\Sigma_2 \cap \Pi_2 \overset{?}{\subseteq} \mathsf{P/poly}$ мы ничего не знаем и не говорим.

# 12. Билет 12

**𝔇𝔢𝔣 12.1.** Язык $QBF$ состоит из замкнутых формул в предварённой форме с конечным количеством кванторов. а под всеми кванторами — формулама в ДНФ или КНФ.

*Замечание* 12.1. Отличие от $QBF_k$ в том, что в $QBF_k$ количество чередований квантор ограничено числом $k$. Схожесть в том, что и там, и там может быть произвольное (но конечное) число кванторов и переменных.

*Замечание* 12.2. Мы не знаем, лежит ли $QBF$ в каком-нибудь уровне полиномиальной иерархии.

**Лемма 12.1.** $QBF \in PSPACE$

▶ Давайте перебирать значения переменных, начиная с внешних. На это требуется экспоненциальное время, но всего полиномиальная память. После того, как знаем значения переменных, можно честно посчитать значение формулы под кванторами. Если очередной квантор — $\forall$, то надо, чтобы обе ветки перебора вернули `true`, если $\exists$, то надо, чтобы хотя бы одна ветка перебора вернула `true`. ◀

**Теорема 12.1.** $QBF$ — $PSPACE$-полная

▶ Нам осталось показать, что произвольная задача $L \in PSPACE$ сводится к $QBF$. Зафиксируем $L$ и какую-нибудь полиномиальную по памяти ДМТ $M$, разрешающую $L$. Не умаляя общности считаем, что в случае принятия слова машина $M$ очищает всю память и переходит в фиксированное состояние, т.е. конфигурация принятия ровно одна.

Пусть нам дали вход $x$ длины $n$. Идея такая: рассмотрим (чисто теоретически) граф конфигураций $M$, он имеет размер $2^{p(n)}$ (где $p(n)$ — время работы $M$). Одна его вершина кодируется $p(n)$ битами. Ребро из вершины в вершину есть тогда и только тогда, когда можно перейти из одной конфигурации в другую. Если мы сможем быстро возвести матрицу смежности этого графа (с добавленными петлями) в степень $2^{\lceil \log p(n) \rceil}$ и посмотреть на наличие ребра между начальной конфигурацией для входа $x$ и принимающей, то мы решим задачу.

Давайте научимся строить предикатные формулы $\varphi_i(a, b)$ полиномиального (от $n$) размера. Формула $\varphi_i(a, b)$ должна выдавать единицу тогда и только тогда, когда есть путь из конфигурации $a$ в $b$ длины не более $2^i$. У формулы $\varphi_i$ имеется $2p(n)$ булевых входов, сама она должна быть размера не более $q(p(n))$.

$\varphi_0$ строится довольно просто (как в теореме Кука-Левина): у нас есть константное число переходов в $M$, надо проверить каждый при помощи формулы полиномиальной длины (т.е. что символы под головками корректны до чтения и после записи и т.д.).

$\varphi_{i+1}(a, b)$ строится так: надо найти промежуточную вершину $c$ и проверить достижимость $\varphi_i(a, c) \wedge \varphi_i(c, b)$. К сожалению, это будет экспоненциальное разрастание, поэтому надо чуть схитрить, оставить лишь одно вхождение $\varphi_i$ и добавить кванторы:

$$\varphi_{i+1}(a, b) \overset{\text{Def}}{=} \exists c\colon \forall x, y\colon (((x, y) = (a, c) \vee ((x, y) = (c, b))) \to \varphi_i(x, y))$$

Получаем, что размер $\varphi_{i+1}$ лишь на $p(n)$ (размер одной переменной) больше размера $\varphi_i$, т.е. полиномиален от $n$ при фиксированном $i$.

Теперь для окончательной проверки надо записать формулу $\varphi_{\lceil p(n) \rceil}(A, B)$, где $A$ — единственная начальная конфигурация, а $B$ — единственная конечная. Это что-то полиномиального от $n$ размера. Потом надо привести эту формулу в предварённую нормальную формулу и получить готовый запрос к $QBF$. Для приведения в предварённую форму надо переименовать переменные и вынести все кванторы влево, не меняя их последовательности. ◀

*Следствие* 12.1.1. Если $PSPACE = PH$, то иерархия коллапсирует, так как тогда $QBF$ лежит в каком-то уровне иерархии $\Sigma_k$, а тогда $QBF_{k+1}$ лежит не выше, значит, иерархи коллапсирует до $\Sigma_k$.

# 13. Билет 13

## 13.1. $NC$ и logspace-вычислимость

**𝔇𝔢𝔣 13.1.** Семейство схем $\{C_n\}_{n\in\mathbb{N}}$ *равномерно*, если имеется полиномиальный алгоритм $A$ такой, что $A(1^n) = C_n$.

*Замечание* 13.1. Равномерность нужна, чтобы мы не могли зашивать в схему какие-то невычислимые функции, зависящие от размера входа.

**𝔇𝔢𝔣 13.2.** Функция $f$ называется *logspace-вычислимой*, если существуют вычисляющая её ДМТ со следующими свойствами:

1. Одна лента выделена под вход и не может изменяться на протяжении работы МТ (read-only)

2. Одна лента выделена под результат, машина не может с неё читать (только писать) и не может перемещать головку влево. Т.е. если МТ что-то записала и поехала дальше, эта запись уже не меняется

3. Суммарная используемая память на остальных (рабочих, read-write) лентах не превосходит $\mathcal{O}(\log n)$, где $n$ — длина входа функции.

**Утверждение 13.1.** Размер выхода logspace-вычислимой функции может быть лишь полиномиален от входа.

▶ Просто посчитаем число конфигураций, оно не меньше времени работы, а оно не меньше размеры выхода:

$$\mathcal{O}(2^{\mathcal{O}(\log n)}) = \mathcal{O}(2^{C\cdot\log n}) = \mathcal{O}\left(\left(2^{\log n}\right)^C\right) = \mathcal{O}(n^C)$$

◀

*Пример* 13.1. $f(x) = x+1$ является *logspace-вычислимой*: надо сначала запомнить позицию, начиная с которой идёт хвост вида `011...11` (это $\log n$ памяти), потом вывести начало $x$ до этой позиции, а потом — новый хвост `100...00`.

**𝔇𝔢𝔣 13.3.** Семейство схем *logspace-равномерно*, если оно равномерно и при этом алгоритм $A$ использует не более $\mathcal{O}(\log n)$ памяти (в том же смысле, что и logspace-вычислимые функции). То есть входная лента (с $n$ единицами) доступна только для чтения, выходная лента доступна только для записи и головка на ней двигается только вправо, в память считается только память рабочих лент.

**𝔇𝔢𝔣 13.4.** Класс $NC^i$ — это все задачи, которые решаются logspace-равномерными семействами схем глубины не более $\mathcal{O}(\log^i |x|)$. То есть задача $L$ лежит в этом классе тогда и только тогда, когда имеется полиномиальный (от длины входа) алгоритм $A$ такой, что он строит по длине входа некоторую схему $C_n$ с $n$ входами для которой потом верно:

$$|x| = n \Rightarrow C_n(x) = L(x)$$

*Замечание* 13.2. Logspace-равномерность тут нужна, чтобы алгоритм $A$ был не слишком умный и не мог бы что-то предподсчитать про входы размера $n$. Он должен просто выдавать схему (причём не зная входа, зная только его длину), а схема должна делать всю вычислительную работу.

**𝔇𝔢𝔣 13.5.** $NC = \bigcup_i NC^i$

**Лемма 13.1.** $NC \subseteq P$

▶ Возьмём задачу $L \in NC$. Для неё есть порождающий схему алгоритм $A$. Построим полиномиальный алгоритм проверки $x \in L$.

Пусть дали $x$. Давайте запустим $A(1^{|x|})$, он отработает за полиномиальное время и выдаст схему полиномиального от $|x|$ размера. Дальше мы её просто вычислим. ◀

## 13.2. Сводимости в $P$

*Замечание* 13.3. До этого моменты шли сведения, полиномиальные по времени. Теперь нам нужны более тонкие сведения.

**Def 13.6.** Когда мы говорим про сведение задач из класса $P$ друг к другу, мы будем предполагать, что сведение использует $\mathcal{O}(\log n)$ памяти и, следовательно, полиномиальное время.

**Def 13.7.** Язык $L$ — *$P$-полон*, если он принадлежит $P$ и любой другой язык из $P$ к нему сводится (за логарифмическую память).

**Лемма 13.2.** Композиция двух logspace-вычислимых функций (т.е. функций, которые при вычислениях не модифицируют входную ленту, на выходную ленту могут только дописывать в конец, при этом использующих $\mathcal{O}(\log n)$ памяти) тоже может быть вычислена с использованием $\mathcal{O}(\log n)$ памяти.

▶ Пусть надо посчитать $f(g(x))$, обозначим $n = |x|$. Для начала заметим, что размер выхода $g$ полиномиален от $n$ — это означает, что вычисление $f(g(n))$ (по известному $g(n)$) тоже требует $\mathcal{O}(\log n)$ памяти.

Давайте возьмём машину $A$, считающую $f$, и машину $B$, считающую $g$. Добавим к машине $A$ рабочие ленты машины $B$, а также ленту с позицией головки машины $A$ на входе (т.е. на выходе $B$) — это всё еще $\mathcal{O}(\log n)$ памяти. Попросим машину $A$ начать вычислять. Когда ей потребуется считать $i$-й бит $g(n)$, перезапустим с самого начала машину $B$, на ещё одной ленте будем поддерживать текущую позицию машины $B$ в выходе. Биты в выводе $B$ до $i$-го будем игнорировать, а вот после вывода $i$-го бита подадим его на вход машине $A$ и остановим машину $B$.

Таким образом у нас время работы может получиться квадратичным (мы на каждый бит входа, который использует $A$, перезапускаем $B$ с нуля), но память осталась логарифмической.

*Замечание* 13.4. В народном хозяйстве это означает, что сведение языков в $P$ транзитивно: если $A \to B$ и $B \to C$ (всё с логарифмической памятью), то также с логарифмической памятью можно свести $A \to C$ напрямую.

## 13.3. Следствия из $P$-полноты

**Теорема 13.1.** Если $L$ — $P$-полный, то:

$$L \in NC \iff P = NC$$

▶ **⇒:** Мы уже знаем, что $NC \subseteq P$, осталось показать второе включение. Возьмём произвольную задачу $L' \in P$. Мы знаем, что она сводится к $L$ с использованием логарифмической памяти при помощи некоторой функции $f(x)$.

Также из билета 15 мы знаем, что $DSpace[\log n] \subseteq NC^2$, то есть можно построить некоторую схему глубины $\mathcal{O}(\log^2 n)$, которая будет вычислять произвольный бит функции $f(x)$. Значит, можно вычислить все биты функции $f(x)$ (их полином), причём параллельно. После того, как схема по входу слова для $L'$ вычислит $f(x)$, можно добавить к ней схему, которая распознает $f(x) \overset{?}{\in} L$, получим всё ещё схему логарифмической глубины.

*Замечание* 13.5. Можно и явно повторить доказательство: построить матрицу смежности граф конфигураций машины для функции $f(x)$ схемой глубины $\mathcal{O}(1)$, потом возвести её за $\mathcal{O}(\log n)$ умножений в большую степень, каждое умножение — схема глубины $\mathcal{O}(\log n)$.

**⇐:** Раз $L$ является $P$-полной, то $L \in P = NC$, что и требовалось.

**Теорема 13.2.** Если $L$ — $P$-полный, то:

$$L \in DSpace[\log] \iff P = DSpace[\log]$$

▶ **⇒:** Очевидно, что $DSpace[\log] \subseteq P$ — если вычисление использует $C \cdot \log n$ памяти, то всего конфигураций не более полинома:

$$\mathcal{O}(2^{C \log n}) = \mathcal{O}\left(\left(2^{\log n}\right)^C\right) = \mathcal{O}(n^C)$$

Следовательно, время работы тоже ограничено полиномом.
Покажем второе включение. Возьмём произвольную задачу $L' \in P$. Так как $L$ — $P$-полна, то есть logspace-сведение $f$:

$$x' \in L' \iff f(x') \in L$$

Но так как $L \in DSpace[\log]$, то есть logspace-функция $g$:

$$x \in L \iff g(x) = 1$$

А мы знаем, что композиция logspace-функций снова logspace, т.о.:

$$x' \in L' \iff g(f(x')) = 1$$

Построили требуемую logspace-функцию, распознающую язык $L'$, значит, $L' \in DSpace[\log]$, что и требовалось.

**⇐:** Так как $L$ — $P$-полный, то $L \in P = DSpace[\log]$, что и требовалось.

◀

**Def 13.8.** Язык $CIRCUIT_EVAL$ — множество пар $(C, x)$, где $C$ — схема, $x$ — её вход, причём $C(x) = 1$.

**Утверждение 13.2.** $CIRCUIT_EVAL$ является $P$-полным языком.

▶ Очевидно, этот язык лежит в $P$ — берём и вычисляем схему.

Осталось показать, что любой язык $L \in P$ сводится к $CIRCUIT_EVAL$. Давайте придумаем функцию сведения $f$. Пусть нам дали на вход слово $x$. Давайте слово-в-слово повторим доказательство теоремы Кука-Левина (где мы сводили задачи из $NP$ к $SAT$) и построим многослойную схему, которая эмулирует работу машины Тьюринга, разрешающей язык $L$. Слои у нас все одинаковые, связи между ними — тоже, логарифма памяти точно хватит (логарифм памяти — это некоторое константное число int'ов, если угодно). Получим на выходе схему, у которой нет входов и которая вычисляется в единицу тогда и только тогда, когда машина принимает слово, что и требовалось. ◀

# 14. Билет 14

## 14.1. Основные понятия

Под «параллельными вычислениями» мы дальше будем подразумевать вычисления при помощи схем. Схема можем вычислять произвольную функцию от слов фиксированной длины.

**Def 14.1.** *Время* вычисления схемы $C$ — глубина схемы (максимальная длина от какого-нибудь входа до какого-нибудь выхода).

**Def 14.2.** *Работа* схемы — количество гейтов.

*Замечание* 14.1. Если у нас есть схема глубины $d$ и работы $w$, то мы можем вычислить её за время $d$, используя $w$ процессоров — каждому процессору поручим свой гейт. Поэтому на схемы можно смотреть как на параллельные вычисления. Это, впрочем, не самое оптимальное оптимальное решение.

**Def 14.3.** Говорим, что гейт лежит на уровне $i$ в схеме $C$, если имеется путь от какого-то входа до этого гейта длины $i$ и нет путей длиннее. Другими словами, если у нас $w$ процессоров, то этот гейт мы не можем посчитать раньше, чем в момент $i$, а ровно в него — можем.

*Замечание* 14.2. Это отличается от уровней бхода BFS'а, потому что BFS ищет кратчайшие пути, а нам надо длиннейшие.

**Теорема 14.1.** Принцип Брента: если есть схема $C$ глубины $d$ с $w$ гейтами, то при использовании $\lceil \frac{w}{d} \rceil$ процессоров её можно посчитать за время, не большее $2d$. То есть если упрощённо считать, что схема имеет на $i$-м уровне пример $\frac{w}{d}$ гейтов, которые можно считать параллельно, то мы не сильно ошибаемся.

▶ Всего в схеме $d$ уровней. Обозначим за $g_i$ количество гейтов на уровне $d$:

$$\sum_{i=1}^{d} g_i = w$$

Обозначим $k =: \lceil \frac{w}{d} \rceil$ — число используемых процессоров. Тогда давайте действовать так: сначала мы считаем все гейты на первом уровне (они независимы), потом все гейты на втором и так далее. Между уровнями никакого параллелизма не добавляем.

Если у нас на уровне $g_i$ гейтов, то мы можем их посчитать за время $\lceil \frac{g_i}{k} \rceil$ — сначала считаем параллельно первые $k$ гейтов, потом следующие $k$ и так далее. То есть суммарное время работы по всем уровням будет равно:

$$\sum_{i=1}^{d} \lceil \frac{g_i}{k} \rceil \leqslant \sum_{i=1}^{d} \left( \frac{g_i}{k} + 1 \right) \leqslant d + \sum_{i=1}^{d} \frac{g_i}{k} = d + \frac{\sum_{i=1}^{d} g_i}{k} = d + \frac{w}{\lceil \frac{w}{d} \rceil} \leqslant d + \frac{w}{(\frac{w}{d})} = 2d$$

◀

*Замечание* 14.3. Если у нас меньше процессоров, то время работы пропорционально возрастает — теперь на каждом уровне просто надо больше шагов.

*Замечание* 14.4. Если у нас больше процессоров, то время работы может не измениться, если на каждом уровне схемы ровно $\frac{w}{d}$ гейтов, зависящих от всех на предыдущем уровне.

## 14.2. Арифметические операции

**Лемма 14.1.** Посчитать выражение $a_1 \vee \ldots \vee a_n$ можно схемой глубины $\mathcal{O}(\log n)$

▶ Просто рисуем двоичное дерево: на первом уровне у нас есть отдельные $a_i$, на втором мы считаем логическое «или» пар соседних, на третьем — логическое «или» четвёрок соседних (пользуясь вторым уровнем) и так далее. ◀

*Замечание* 14.5. Аналогично можно действовать с любой ассоциативной операций, например, с $\oplus$ или с $\wedge$. Более того, точно так можно действовать даже когда операция в качестве аргументов принимает не биты, а последовательности из $k$ бит (где $k$ — константа). Тогда, правда, глубина домножится на глубину вычисления функции от двух аргументов (которая всё равно константа и на асимптотику не повлияет).

**Лемма 14.2.** Более того, если у нас имеются переменные $a_1, \dots, a_n$, то можно посчитать значения ассоциативной функции $f$ на префиксах:

$$f(a_1), f(a_1, a_2), f(a_1, a_2, a_3), \dots, f(a_1, \dots, a_n)$$

▶ Можно либо действовать так же и потом просто посмотреть на получившееся дерево как на дерево отрезков из алгоритмов (а запрос к дереву отрезков — это $\mathcal{O}(\log n)$ запросов к его вершинам), либо применить разделяй-и-властвуй.

Для этого рекурсивно считаем префиксные значения для первой половины переменных, для второй, а потом ко всем значениям из второй половины дописываем в начало $f(a_1, \dots, a_{n/2})$. ◀

**Теорема 14.2.** Сумму двух чисел длины $n$, записанных в двоичной системе счисления, можно посчитать схемой глубины $\mathcal{O}(\log n)$.

▶ Также есть на Викиконспектах.

Давайте сначала схемой глубины $\mathcal{O}(\log n)$ выясним, в каких разрядах будет при сложении возникать перенос. А потом отдельно выясним значение каждого разряда: это просто сумма двух разрядов исходных чисел плюс перенос (который мы уже знаем).

Давайте научимся считать функцию $f$ от двух чисел одинаковой длины $k$, она будет выдавать два бита:

1. Верно ли, что при сложении чисел возникнет перенос из разряда $k$?
2. Верно ли, что при сложении этих чисел и еще одной единицы возникнет перенос из разряда $k$?

Эта функция ассоциативна в том смысле, что если мы знаем $f(a, b)$ и $f(c, d)$, то легко можем посчитать $f(\overline{ac}, \overline{bd})$:

1. Перенос при сложении $\overline{ac} + \overline{bd}$ может случиться только в одном из двух случаев:
   - Перенос возникает уже при сложении $a + b$.
   - Возникает перенос при сложении $c + d$, а также при сложении $a + b + 1$
2. Аналогично для случая $\overline{ac} + \overline{bd} + 1$, но там уже надо будет смотреть на сложение не $c + d$, а сложение $c + d + 1$.

Теперь мы предыдущим леммам мы можем посчитать эту функцию на всех суффиксах двух слов и, тем самым, узнать, из каких разрядов будут переносы. ◀

**Теорема 14.3.** Произведение двух чисел длины $n$, записанных в двоичной системе счисления, можно посчитать схемой глубины $\mathcal{O}(\log n)$.

▶ Также есть на Викиконспектах.

Сначала построим элемент $3 \to 2$ глубины $\mathcal{O}(1)$, который по трём числам длины $2n$ будет строить два числа длины $2n$ с такой же суммой (перенос из разряда $2n$ отбрасывается). Он строится просто: $a = x \oplus y \oplus z$, а $b$ — это маска возникших переносов при суммировании $x + y + z$ (т.е. просто $((x \& y)|(x \& z)|(y \& z)) << 1)$.

Теперь умножение. Сначала строим $n$ чисел размера $2n$ каждое, как при умножении в столбик, это можно схемой глубины $\mathcal{O}(1)$. Дальше нужно сложить эти $n$ чисел. Давайте разобьём их на тройки, в каждой тройке запустим элемент $3 \to 2$, получим в полтора раза меньше чисел. Будем так повторять, пока не получим два числа (а это произойдёт через $\mathcal{O}(\log n)$ спусков). То есть за глубину $\mathcal{O}(\log n)$ мы свели умножение к сумме двух чисел, а её мы уже умеем делать за ещё $\mathcal{O}(\log n)$. ◀

## 14.3. Умножение матриц и достижимость

**Лемма 14.3.** Если есть две прямоугольные матрицы над полем $\mathrm{F}_2$ размеров $n \times m$ и $m \times k$, то их произведение (размера $n \times k$) можно посчитать схемой размера $\mathcal{O}(nmk)$ и глубины $\mathcal{O}(\log m)$.

▶ Давайте отдельно посчитаем каждый бит матрицы-ответа. Он является скалярным произведением строки и столбца исходных матриц. Давайте сначала схемой глубины $\mathcal{O}(1)$ посчитаем $m$ бит — произведения соответствующих элементов, а потом схемой глубины $\mathcal{O}(\log m)$ их сложим. ◀

*Замечание* 14.6. Можно сделать ровно то же самое, если мы не работаем в поле $\mathrm{F}_2$, а если у нас операция сложения является операцией «или».

**Лемма 14.4.** Если имеется ориентированный граф $G$ на $n$ вершинах и $A$ — его матрица смежности, то $(A+E)^n$ является его матрицей достижимости (т.е. единица в строке $i$ и столбце $j$ стоит тогда и только тогда, когда вершина $j$ достижима из вершины $i$).

*Замечание* 14.7. Под «+» в этой лемме и ниже подразумевается операция «$\vee$».

▶ Докажем следующее утверждение индукцией по $k$: $(A+E)^k$ — это матрица существования путей (маршрутов) длины не более $k$.

База при $k = 0$ очевидна (просто единичная матрица). Переход $k \to (k+1)$. Обозначим $B =: (A+E)^k$. Заметим, что $B(A+E) = BA + B$, т.е. в матрице $(A+E)^{k+1}$ будут, как минимум, все пути длины не более $k$. А что такое $BA$? Если посмотреть на определение умножения матриц, то получается, что в ячейке $BA$ единица стоит тогда и только тогда, когда был какой-то путь $i \to \alpha$ длины не более $k$ и путь $\alpha \to j$ длины не более 1, т.е. автоматически был путь $i \to j$ длины не более $k+1$. Что и требовалось.

А никакой путь в графе не может быть длинее $n$, значит, $(A+E)^n$ уже является матрицей достижимости). ◀

**Теорема 14.4.** Задача проверки достижимости вершины $b$ из вершины $a$ в орграфе $G$ на $n$ вершинах решается схемой глубины $\mathcal{O}(\log^2 n)$.

▶ Сначала надо схемой глубины $\mathcal{O}(1)$ построить матрицу смежности $A$ графа $G$, потом прибавить к ней $E$ за ещё одну операцию. Затем надо быстрым возведением в степень возвести $A+E$ в степень хотя бы $n$, для этого потребуется $\mathcal{O}(\log n)$ умножений матриц, каждое можно сделать за глубину $\mathcal{O}(\log n)$. Получаем итоговую глубину $\mathcal{O}(\log^2 n)$. ◀

# 15. Билет 15

## 15.1. $\mathit{NSpace}[\log n] \subseteq \mathit{NC}^2$

Пусть есть НМТ $M$, распознающая язык $L$ с $\mathcal{O}(\log n)$ памяти. Давайте построим logspace-равномерное семейство схем глубины $\mathcal{O}(\log^2 n)$, которое распознает этот же язык. Для этого будем строить алгоритм, выводящий схему для слов определённой длины как композицию двух logspace-функций:

1. Построение по входу $x$ схемы глубины $\mathcal{O}(1)$, выводящей описание графа псевдоконфигураций $M$ (т.е. конфигураций без учёта состояния входной ленты, так как она read-only).

2. Проверка достижимости принимающей псевдоконфигурации из начальной в данном графе.

Пункт 2 обсуждался в предыдущем билете (там получалась схема как раз глубины $\mathcal{O}(\log^2 n)$), так что сконцентрируемся на первом.

Пусть нам дали длину $n$. Мы знаем, что всего у $M$ будет $p(n)$ . Давайте построим граф на $p(n)$ вершинах. Из каждой конфигурации в каждую другу может либо вести ребро, либо не вести — это либо однозначно определено, либо зависит от конкретного бита во входе МТ (так как на вход указывает ровно одна головка). Соответственно, схема в каждом бите будет выводить либо константу, либо какой-то конкретный бит входа (либо его отрицание). Наш logspace-алгоритм просто перебирает все пары псевдоконфигураций (на это требуется $\mathcal{O}(\log n)$ памяти) и смотрит, возможен ли между ними переход в зависимости от бита на входной ленте.

В итоге наша схема состоит из двух частей: первая, глубины $\mathcal{O}(1)$, строит матрицу смежности графа псевдоконфигураций, а вторая, глубины $\mathcal{O}(\log^2 n)$, быстро возводит эту матрицу в степень.

## 15.2. $\mathsf{DSpace}[\log n] \subseteq \mathsf{NSpace}[\log n]$

Очевидно, так как любая ДМТ является заодно и НМТ.

## 15.3. $\mathsf{NC}^1 \subseteq \mathsf{DSpace}[\log n]$

**Лемма 15.1.** По описанию схемы глубины $d$ и её входам можно с использованием $\mathcal{O}(\log n + d)$ памяти вычислить её выходы.

▶ Будем по очереди вычислять значение на выходах.

Зафиксировали конкретный выход. Теперь давайте пойдём от него DFS'ом наверх, ко входам. В стеке вызовов будем хранить не номера вершин (это много памяти), а путь из этого выхода к текущей вершины: последоветельность нулей и единиц (в какого ребёнка пошли), т.е. $\mathcal{O}(d)$ памяти. Также надо будет хранить результаты текущих вычислений, но их не более $\mathcal{O}(d)$ — на каждом уровне у нас одна вершина сейчас вычисляется, у неё максимум один предок вычислен, он и хранится (если вычислено два — то мы уже и саму вершину посчитали).

В каждый момент нам надо будет знать лишь номер текущей вершины (его можно хранить), иногда надо будет находить гейт, поданный на вход вершине (надо пробежаться по описанию всей схемы), иногда надо будет подниматься вверх по стеку (пробежимся по всему стеку и по очереди найдём номера вершин от выхода до текущей).

Мы можем несколько раз посчитать значение одного и того же гейта, но в этом ничего страшного нет — мы всё равно посчитаем значение корректно и будем использовать не очень много памяти. ◀

Пусть у нас есть язык $L \in \mathsf{NC}^1$, мы хотим построить распознающий logspace-алгоритм. Давайте построим его как композицию logspace-алгоритма, генерирующего по длине входа схему и logspace-алгоритма, вычисляющего значение этой схемы на входе (у нас же схема глубины $\mathcal{O}(\log n)$).

# 15. Билет 15

Раздел 1.12 в конспекте Оли.

**Теорема 15.1.** Если $s(n)$ конструктивна по памяти, то $\mathsf{NSpace}[s(n)] \subseteq \mathsf{DTime}[2^{\mathcal{O}(s(n))}]$

*Замечание* 15.1. Этой теоремой не пользуемся, есть более крутая, см. ниже. Это просто пример.

▶

*Замечание* 15.2. У НМТ из одного состояния может вести сколько угодно переходов, хоть во все остальные состояния. Поэтому просто перебирать все возможные способы работы НМТ не поможет.

Так как НМТ работает за $s(n)$ шагов, то она использует не более $\mathcal{O}(s(n))$ памяти. Значит, у неё всего $2^{\mathcal{O}(s(n))}$ конфигураций. Можно явно построить граф таких конфигураций (квадрат экспоненты — всё ещё экспонента) и пройтись по нему DFS'ом, пытаясь найти путь из исходной конфигурации в какую-нибудь из терминальных. ◀

**Теорема 15.2.** Если имеется орграф на $n$ вершинах и наличие/отсутствие ребра в графе можно проверять за $\mathcal{O}(\log n)$ памяти, то задача проверка достижимости из вершины $a$ вершины $b$ лежит в $\mathit{DSpace}[\log^2 n]$.

▶ Введём предикат $s(a, b, i)$ — можно ли попасть из вершины $a$ в вершину $b$, сделав не более $2^i$ переходов. Покажем индукцией по $i$, что его можно вычислять с использованием $C \cdot (i+1) \cdot \log n$ памяти, где $C$ — некоторая фиксированная константа. Считаем, что на одной рабочей ленте МТ будет записано $a$, на другой — $b$, на третьей — $i$.

База при $i = 0$: это просто проверка на наличие ребра в графе из условия задачи, нам только важно, что $C$ не меньше той константы, что спрятана в ошке в условии. Переход: пусть мы хотим научиться вычислять $s(a, b, i)$. Сохраним $a$, $b$, $i$ на стек, потратим не более $3 \cdot \log n$ памяти. Потом переберём промежуточную вершину $z$, используя на это ещё $\log n$ памяти. После фиксации вершины $z$ посчитаем предикат $s(a, z, i - 1)$, используя $C \cdot i \cdot \log n$ памяти. Итого будет использовано не более $4 \cdot \log n + C \cdot i \cdot \log n \leqslant C \cdot (i + 1) \cdot \log n$ памяти (если $C \geqslant 4$). Посчитали, если результат отрицательный, то сразу возвращаем ноль, иначе считаем предикат $s(z, b, i - 1)$.

А чтобы теперь посчитать глобальную достижимость, надо посчитать при $i = \log n$, получим память $\mathcal{O}(\log^2 n)$. ◀

**Теорема 15.3.** Если $s(n)$ конструктивна по памяти, то $\mathit{NSpace}[s(n)] \subseteq \mathit{DSpace}[(s(n))^2]$

▶ Это следствие предыдущей теоремы: у НМТ есть не более $2^{C \cdot s(n)}$ конфигураций. Будем считать, что перед приёмом слова НМТ стирает все ленты, т.о. принимающая конфигурация ровно одна. Можно ли перейти из одной конфигурации в другую легко выяснить, прочитав обе конфигурации.

Т.о. мы находимся в условии предыдущей леммы: есть граф на $2^{C \cdot s(n)}$ вершинах, хотим проверить достижимость. Это можно сделать за $\mathcal{O}(C^2 \cdot s^2(n)) = \mathcal{O}(s^2(n))$. ◀

# 17. Билет 17

## 17.1. Рандомизированные машины

Случайные биты поступают независимо от входа и алгоритма. Потенциально битов бесконечно много (но мы используем только конечное число). Считаем, что каждый бит равновероятно 0 или 1. Можно и по-другому считать, но мы так делать не будем.

Можно воспринимать случайные биты либо как дополнительную ленту у МТ (тогда будет рандомизированная МТ), либо как дополнительный вход у МТ, на который накладываются какие-то ограничения.

*Замечание* 17.1. Если МТ не может использовать столько же памяти, сколько времени, то она может только читать ленту со случайными битами, а головку двигать только вправо. Это сделано чтобы нельзя было «возвращаться в прошлое» и использовать старые случайные биты вместо памяти.

Рандомизированные алгоритмы с ответом «да/нет» могут ошибаться в две стороны, могут в одну, могут не ошибаться (тогда точно рандомизируется время, иначе зачем нам рандомные биты вообще).

## 17.2. *RP*

**𝔇𝔢𝔣 17.1.** $L \in \mathsf{RP}$ (односторонняя ошибка), если имеется п.о. п.п. отношение $R$ такое, что для любого входа $x$:

1. $x \notin L \Rightarrow \forall w\colon (x,w) \notin R$
2. $x \in L \Rightarrow \Pr_w\{(x,w) \in R\} > \frac{1}{2}$

*Замечание* 17.2. Отличие от *NP* в том, что там нам требовалась хотя бы одна подсказка, а тут их должно быть хотя бы половина.

*Замечание* 17.3. Если алгоритм для языка из *RP* говорит «$x \in L$», то он не врёт — нашлась хотя бы одна подсказка. А вот если говорит «$x \notin L$», то это ещё неточно.

**Утверждение 17.1.** Мы можем считать, что машина всегда получает на вход фиксированное полиномиальное число случайных бит — $p(n)$ ($n$ — длина входа).

▶ Во-первых, понятно, что есть такой $p(n)$, что машина всегда использует не более $p(n)$ бит, достаточно взять $p$, ограничивающий время работы машины. Во-вторых заметим, что если в произвольный момент работы машины вставить чтение и игнорирование случайного бита, то это не повлияет на вероятности выдать какой-либо ответ — мы лишь расщепили один случай на два с таким же вердиктом и с вдвое меньшими вероятностями. Тогда давайте скажем, что перед тем, как машина выдаёт ответ, она считывает и игнорирует ещё $p(n) - k$ случайных бит, где $k$ — число использованных случайных бит. ◀

*Следствие* 17.0.1.

$$\Pr_w\{(x,w) \in R\} = \frac{|\{w \mid (x,w) \in R\}|}{2^{p(n)}}$$

**Теорема 17.1.** В *RP* можно понижать ошибку, т.е. для любого языка $L \in \mathsf{RP}$ и полинома $p(n)$ есть п.о. п.п. отношение $R$ такое, что:

1. $x \notin L \Rightarrow \forall w\colon (x,w) \notin R$
2. $x \in L \Rightarrow \Pr_w\{(x,w) \in R\} > 1 - \frac{1}{2^{p(|x|)}}$

▶ Давайте возьмём отношение $R$ из определения *RP* и повторим проверку $p(|x|)$ раз (это всё ещё полином). Вернем ответ «$x \in L$» только если хотя бы одна проверка вернула «да». Тогда если $x \notin L$, то мы «да» никогда не вернём. А если $x \in L$, то чтобы ошибиться, нам надо, чтобы каждая из проверок ошиблась, это произойдёт с вероятностью не больше $(1/2)^{p(|x|)}$. ◀

*Следствие* 17.1.1. В определении *RP* константа $\frac{1}{2}$ взята с потолка, можно брать любую, отделённую от единицы и нуля — всё получится эквивалентное.

## 17.3. *BPP* и понижение ошибки

**𝔇𝔢𝔣 17.2.** $L \in \mathsf{BPP}$ (двухсторонняя ограниченная ошибка), если имеется п.о. п.п отношение $R$ такое, что:

1. $x \notin L \Rightarrow \Pr_w\{(x,w) \in R\} < \frac{1}{3}$
2. $x \in L \Rightarrow \Pr_w\{(x,w) \in R\} > \frac{2}{3}$

**Утверждение 17.2.** Неравенство Чернова: если есть независимые случайные величины $x_i \in [0,1]$ и матожидание их суммы равно $\mu$, то для любого $0 < \varepsilon < 1$:

$$\Pr\{\sum x_i \geqslant (1+\varepsilon)\mu\} < e^{-\frac{\mu\varepsilon^2}{4}}$$

▶ Без доказательства — считается за известное. ◀

**Теорема 17.2.** В $\mathsf{BPP}$ тоже можно понижать ошибку, т.е. для любого языка $L \in \mathsf{BPP}$ и полинома $p(n)$ есть п.о. п.п. отношение $R$ такое, что:

1. $x \notin L \Rightarrow \Pr_w\{(x,w) \in R\} < \frac{1}{2^{-p(n)}}$
2. $x \in L \Rightarrow \Pr_w\{(x,w) \in R\} > 1 - \frac{1}{2^{-p(n)}}$

▶ Давайте возьмём отношение $R$ из определения $\mathsf{BPP}$ и повторим проверку $p(|x|)$ раз. Получим $p(|x|)$ ответов (обозначим это число за $k$). Возьмём тот, который чаще всех встречается (будем считать, что $k$ нечётно). Это всё ещё полином.

Теперь оценим вероятность ошибки. У нас есть $k$ случайных величин $x_i$ — возникла ли ошибка на шаге $i$. Каждая величина — либо ноль (с вероятностью $2/3$), либо единица (с вероятностью $1/3$). Матожидание суммы — $k/3$. Весь алгоритм ошибается тогда и только тогда, когда сумма $x_i$ получилась больше $k/2$ (т.е. ошиблись больше, чем в половине раундов). Пишем неравенство Чернова при $\varepsilon = 1/2$:

$$\Pr\{\sum x_i \geqslant \left(1 + \frac{1}{2}\right) \cdot \frac{k}{3}\} < e^{-\frac{\frac{k}{3} \cdot \frac{1}{4}}{4}}$$

$$\Pr\{\sum x_i \geqslant \frac{k}{2}\} < e^{-\frac{k}{48}} < 2^{-\frac{k}{48}} < 2^{-k}$$

Получили, что вероятность ошибки не больше $2^{-k}$, что и требовалось. ◀

*Следствие* 17.2.1. В определении $\mathsf{BPP}$ константы $\frac{1}{3}$ и $\frac{2}{3}$ взяты с потолка, можно брать любые, равные $x$ и $1-x$, где $x$ — константа, отделённая от $\frac{1}{2}$ и нуля.

*Замечание* 17.4. На самом деле можно даже брать вообще любые две константы, отделённые друг от друга, нуля и единицы, всё равно получится $\mathsf{BPP}$. Вроде доказывается тоже через Чернова, надо чуть аккуратнее. **TODO**

## 17.4. *PP* и вложенность классов

**𝔇𝔢𝔣 17.3.** $L \in \mathsf{PP}$ (двухсторонняя неограниченная ошибка), если имеется п.о. п.п отношение $R$ такое, что:

1. $x \notin L \Rightarrow \Pr_w\{(x,w) \in R\} \leqslant \frac{1}{2}$
2. $x \in L \Rightarrow \Pr_w\{(x,w) \in R\} > \frac{1}{2}$

*Замечание* 17.5. Этот класс почти никогда не нужен. Отличие от $\mathsf{BPP}$ в том, что ошибки могут подползать к $\frac{1}{2}$ сколь угодно близко с ростом $|x|$, и тогда их фиг отделишь.

**Теорема 17.3.**

$$\mathsf{NP} \subseteq \mathsf{PP}$$

▶ Давайте покажем, что $\mathsf{SAT} \in \mathsf{PP}$, после этого любую задачу из $\mathsf{NP}$ можно будет свести к $\mathsf{SAT}$ и она тоже окажется в $\mathsf{PP}$ (никаких проблем с вероятностью: она у нас берётся по случайным битам, никак не зависит от того, какие преобразования задачи мы делали).

Пусть есть формула с $n$ переменными. Давайте выберем их значения случайно. Если формула выполнилась, выводим «да». В противном случае выводим «да» с вероятностью $1/2$. Покажем, что вероятности какие надо.

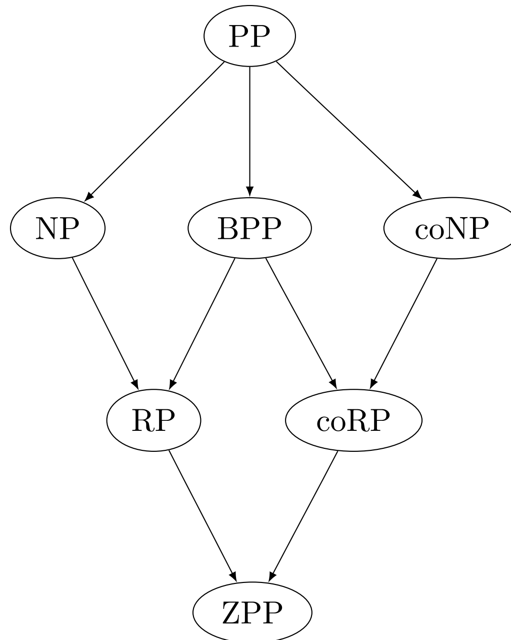
Пусть $x \notin \mathsf{SAT}$, т.е. формула невыполнима. Тогда мы скажем «$x \in \mathsf{SAT}$» с вероятностью ровно $1/2$, что и надо.

Пусть $x \in \mathsf{SAT}$, т.е. есть $k$ выполняющих наборов ($k > 0$). Тогда мы в них попадём с вероятностью $\frac{k}{2^n}$ и скажем «$x \in \mathsf{SAT}$», а второй способ сказать «да» — не попасть в выполняющий набор, но угадать со случайным битом. Итого вероятность сказать «да»:

$$\frac{k}{2^n} + \frac{2^n - k}{2^n} \cdot \frac{1}{2} = \frac{2k + 2^n - k}{2^{n+1}} = \frac{1}{2} + \frac{k}{2^{n+1}} > \frac{1}{2}$$

Что и требовалось. ◀

*Замечание* 17.6. Диаграмма Хассе для вложенности классов получается такая (стрелка от надкласса к подклассу):




## 17.5. $\mathsf{ZPP}$

**𝔇𝔢𝔣 17.4.**

$$\mathsf{ZPP} \stackrel{\text{Def}}{=} \mathsf{RP} \cap \text{co-}\mathsf{RP}$$

**Теорема 17.4.** Если $L \in \mathsf{ZPP}$, то есть вероятностная ДМТ, которая безошибочно распознаёт $L$ и *матожидание* времени её работы полиномиально.

▶ По определению $\mathsf{ZPP}$ имеется два полиномиальных вероятностных алгоритма: $A$ (для $\mathsf{RP}$) и $B$ (для co-$\mathsf{RP}$).

Давайте запустим их обоих, они отработают за полином. Если $A$ выдал «$x \in L$», то он точно прав (в эту сторону алгоритмы из $\mathsf{RP}$ не ошибаются). Аналогично, если $B$ выдал «$x \notin L$», то гарантировано $x \notin L$. А вот что делать в случае, когда $A$ выдал «$x \notin L$», а $B$ выдал «$x \in L$» — непонятно. Какой-то из двух алгоритмов точно ошибся, это произошло с вероятностью не более $\frac{1}{2}$.

Тогда давайте повторим раунд, и так до бесконечности. Матожидание времени работы (если считать, что время работы одного раунда равно $p(|x|)$):

$$p(|x|) \cdot \left(1 + \frac{1}{2}\left(1 + \frac{1}{2}(\dots)\right)\right) = p(|x|)\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right) = \mathcal{O}(p(|x|))$$

◀

**Утверждение 17.3.** Неравенство Маркова: если есть неотрицательная случайная величина $x$ с матожиданием $\mu$, то при $a > 0$:

$$\Pr\{x \geqslant a\} \leqslant \frac{\mu}{a}$$

▶ Без доказательства.

Неформально так: предположим обратное, сдвинем у $x$ все значения $< a$ в ноль, а все значения $\geqslant a$ — в $a$. Тогда матожидание может лишь уменьшиться (как и правая часть, т.е. неравенство станет строже), а левая часть не изменится. Но тогда $\mu = \Pr\{x \geqslant a\} \cdot a$, получим как раз равенство, успех. ◀

**Теорема 17.5.** Если для языка $L$ существует вероятностная ДМТ $M$, которая безошибочно его распознаёт и матожидание времени её работы полиномиально, то $L \in \mathit{ZPP}$.

▶ Пусть нам дали $x$ и мы знаем, что матожидание времени работы $M$ равно $p(|x|)$. Давайте запустим $M$ в течение $2p(|x|)$ шагов. Подставим в неравенство Маркова случайную величину $t$ (время работы $M$) с матожиданием $p(|x|)$ и $a = 2p(|x|)$:

$$\Pr\{t \geqslant 2p(|x|)\} \leqslant \frac{p(|x|)}{2p(|x|)} = \frac{1}{2}$$

Таким образом, получим, что с вероятностью $\frac{1}{2}$ мы узнаем точный ответ.

Теперь строим алгоритм для распознавания $L$ из $\mathit{RP}$. Он будет делать то же, что и выше. Узнали точный ответ — сказали, если не успели узнать — сказали «$x \notin L$», если это и ошибка, то она произойдёт с вероятностью не более $1/2$.

Аналогично для алгоритма из co-$\mathit{RP}$: если не успели узнать точный ответ — сказали «$x \in L$», в эту сторону ошибаться дозволяется. ◀

# 18. Билет 18 (лемма Шварца-Циппеля)

**Def 18.1.** Степень одночлена от нескольких переменных — сумма степеней по каждой переменной.

**Def 18.2.** Степень многочлена от нескольких переменных — максимум степеней его одночленов.

*Пример* 18.1.

$$\deg(x^2 + y^2) = 2$$

*Пример* 18.2.

$$\deg(x^2y + y^2) = 3$$

**Def 18.3.** Многочлен от нескольких переменных называется *тождественно равным нулю*, если все его коэффициенты равны нулю, т.е. нет ни одного одночлена с ненулевым коэффициентом. Обозначется $q(x_1, \dots, x_n) \equiv 0$.

**Лемма 18.1.** Пусть есть поле $F$, некоторое его конечное подмножество $S \subseteq F$ и многочлен $Q$ (не равный тождественно нулю) от $n$ переменных степени не больше $d$. Тогда если выбрать значения $x_i$ независимо и равновероятно из множества $S$, то вероятность попасть в корень многочлена не больше $\frac{d}{|S|}$.

▶ Индукция по числу переменных $n$.

**База:** $n = 1$. Мы знаем, что над полем у многочлена от одной переменной степени $d$ не более $d$ корней.

*Замечание* 18.1. Это следствие теоремы Безу, которая работает в произвольном поле: если $c$ — корень многочлена, то он делится на $x - c$ без остатка. Многочлен неотрицательной степени можно поделить не более $d$ раз на разные корни, разложить в произведение линейных и какого-то неприводимого, получить не более $d$ корней.

Значит, в множестве $S$ тоже лежит не более $d$ корней. Значит, вероятность попасть в случайный не более $\frac{d}{|S|}$.

**Переход:** $(n-1) \to n$. Пусть у нас всё доказано для многочленов от $n-1$ переменной и есть многочлен $Q(x_1, \dots, x_{n-1}, x_n)$. Давайте сгруппируем его одночлены по степеням $x_n$:

$$Q \in F[x_1, \dots, x_n]$$
$$Q(x_1, \dots, x_n) = x_n^d \cdot q_d(x_1, \dots, x_{n-1}) + \dots + x_1 \cdot q_1(x_1, \dots, x_{n-1}) + q_0(x_1, \dots, x_{n-1})$$
$$q_i \in F[x_1, \dots, x_{n-1}]$$

Так как $Q$ не является тождественным нулём, то хотя бы один многочлен из $q_i$ не является тождественным нулём. Давайте обозначим такое максимальное $i$ за $k$, получим:

$$Q(x_1, \dots, x_n) = x_n^k \cdot q_k(x_1, \dots, x_{n-1}) + \dots + q_0(x_1, \dots, x_{n-1})$$

Можно переобозначить этот многочлен за $Q_{x_1,\dots,x_{n-1}}(x_n)$.

Два случая:

*A*. $Q_{x_1,\dots,x_{n-1}}(x_n) \equiv 0$ (т.е. ноль независимо от $x_n$).

*B*. $Q_{x_1,\dots,x_{n-1}}(x_n) \not\equiv 0$

Пусть вероятность попасть в случай $A$ при случайных $x_1, \dots, x_{n-1}$ равна $p$, тогда вероятность попасть в $B$ равна $1-p$. Разберёмся в каждом случае, какая вероятность выбрать $x_n$ так, чтобы $Q(x_1, \dots, x_n) = 0$.

*A*. многочлен оказался тождественным нулём. Значит, независимо от выбора $x_n$ верно $Q_{\dots}(x_n) = 0$, вероятность — 1.

*B*. многочлен $Q_{x_1,\dots,x_{n-1}}(x_n) \not\equiv 0$. Этот многочлен имеет степень не больше $k$, значит, как в базе индукции, вероятность того, что он занулится при случайном $x_n$ не более $\frac{k}{|S|}$.

**Утверждение 18.1.** Многочлен $Q$ зануляется (при случайных $x_1, \dots, x_n$) с вероятностью не более $p + \frac{k}{|S|}$.

▶ В самом деле — с вероятность не более $p$ имеем случай $A$ (в котором многочлен зануляется) и с вероятностью не более 1 имеем случай $A$, в котором многочлен зануляется с вероятностью $\frac{k}{|S|}$. ◀

Оценим $p$ сверху. Чтобы $Q_{x_1,\dots,x_{n-1}}$ оказался тождественным нулём надо, в частности, чтобы его старший коэффициент ($q_k$) стал нулём. Так как $\deg q_k = d - k$, то, по предположению индукции, это событие произойдёт с вероятностью не большей $\frac{d-k}{|S|}$. Значит, $p < \frac{d-k}{|S|}$.

Таким образом, исходный многочлен $Q$ зануляется с вероятностью не более:

$$\frac{d-k}{|S|} + \frac{k}{|S|} = \frac{d}{|S|}$$

Что и требовалось показать.

◀

**𝔇𝔢𝔣 18.4.** *Задача проверки полинома на тривиальность*: имеется формальный полином $Q$ от $n$ переменных и некоторое фиксированное поле $F$. При этом мы можем вычислять значения этого полинома в произвольной точке поля за полиномиальное время.

Требуется проверить, является ли этот полином тождественным нулём (то есть равны ли все его коэффициенты нулю).

*Пример* 18.3. Например, может быть дан такой полином в поле $\mathbb{Z}_{2^{2n}}$ (он, очевидно, тождественно равен нулю):

$$(x_1 - x_2)(x_1 - x_2) \dots (x_1 - x_n) - (x_1 - x_2)(x_1 - x_2) \dots (x_1 - x_n)$$

Однако так сходу это может быть неясно. Если мы начнём честно раскрывать и приводить подобные слагаемые, мы получим экспоненциальный рост размера полинома, что нехорошо.

*Замечание* 18.2. Многочлен $2x$ в поле $\mathbb{R}$ не является тождественным нулём, а вот в поле $\mathbb{Z}_2$ — уже является.

**Утверждение 18.2.** Задача проверки полинома на тривиальность (в достаточно большом поле) лежит в co-$RP$ (ответ «да» — полином тождественный ноль, ответ «нет» — не тождественный ноль).

▶ Пусть дан полином от $n$ переменных степени $k$. Тогда давайте выберем в поле $F$ $n$ случайных чисел в качестве значений переменных и подставим их в полином. Если получился не ноль, то мы уверены в том, что полином не является тождественным нулём. Иначе по лемме вероятность того, что полином не является тождественным нулём не превосходит

$$\frac{k}{|F|}$$

Так как мы случайно попали в корень. Если поле имеет размер хотя бы $(1+\varepsilon)\cdot k$, то вероятность ошибки отделима от единицы. ◀

# 19. Билет 19

**Теорема 19.1.** $BPP \subseteq P/poly$

▶ Возьмём задачу распознавания $L \in BPP$. По определению $BPP$ имеется п.о. п.п. бинарное отношение $R$ такое, что вероятность $\Pr_w\{(x, w) \in R\}$ разная для случаев $x \in L$ и $x \notin L$. Давайте зафиксируем длину $x$ ($|x| = n$) и построим схему, которая будет корректно работать со словами длины $n$. Мы хотим найти одну подсказку $w$, на которой отношение $R$ будет вести себя безошибочно, а потом зашить эту подсказку в схему.

Для этого давайте сделаем из $R$ отношение $R'$ с пониженной ошибкой, которое будет запускать $R$ полиномиальное число раз и выбирать ответ, который более вероятен. Мы знаем, что можно понизить вероятность ошиби $R'$ до $\frac{1}{2^{poly(n)}}$ для произвольного полинома, давайте тогда выберем в качестве полинома $2n$. Тогда для $R'$ вероятность ошибиться на конкретном входе (в любую сторону) будет не более $2^{-2n}$:

$$\forall x\colon \Pr_w\{R'(x, w) \neq L(x)\} < 2^{-2n}$$

Обозначим за $W$ суммарное число подсказок (они ограничены по длине полиномом, так как

TODO ). Тогда:

$$\forall x\colon \frac{|\{w \mid R'(x,w) \neq L(x)\}|}{W} < 2^{-2n}$$
$$\forall x\colon |\{w \mid R'(x,w) \neq L(x)\}| < 2^{-2n} W$$
$$\sum_x |\{w \mid R'(x,w) \neq L(x)\}| < 2^n \cdot 2^{-2n} W$$
$$|\{x,w \mid R'(x,w) \neq L(x)\}| < 2^{-n} W$$
$$\sum_w |\{x \mid R'(x,w) \neq L(x)\}| < 2^{-n} W$$
$$\frac{\sum_w |\{x \mid R'(x,w) \neq L(x)\}|}{W} < 2^{-n}$$
среднее значение не может быть меньше минимального значения
$$\exists w\colon |\{x \mid R'(x,w) \neq L(x)\}| < 2^{-n}$$

Так как мощность множества — целое число, то существует подсказка $w$ такая, что $R'(x,w) = L(x)$ для всех входов $x$ фиксированной длины. Давайте тогда преобразуем $R'$ в схему полиномиального размера, а случайные биты положим константами из $w$. ◀

# 20. Билет 20

Ниже будем доказывать, что $BPP \subseteq \Sigma_2$.

## 20.1. Основная идея

Рассмотрим какой-нибудь язык $L \in BPP$. Будем считать, что есть п.о. п.п. бинарное отношение $R$, которое проверяет слова из языка с ошибкой не более $2^{-n}$, где $n$ — длина проверяемого слова, мы так понижать ошибку умеем. Обозначим за $m$ число использованных при проверке случайных бит. Выберем как-то число $k$ (как — поймём позже, но пока считаем, что оно является полиномом от $n$, где $n$ — длина проверяемого слова).

*Замечание* 20.1. Что происходит при небольших $n$, нас не интересует — мы можем намертво «зашить» ответы для всех коротких строк в обход основного случая.

*Замечание* 20.2. Спойлер: $k$ будет равно $m$ при $n > 1$.

Хотим показать, что $x \in L$ равносильно тому, что существует такой набор битовых строк $t_1, t_2, \ldots, t_k$ ($|t_i| = m$), что из любой битовой строчки $r$ можно получить правильную подсказку для $R$:

$$x \in L$$
$$\Updownarrow$$
$$\exists t_1, \ldots, t_k\colon \forall r\colon \left[\begin{array}{l} R(x, r \oplus t_1) \\ R(x, r \oplus t_2) \\ \vdots \\ R(x, r \oplus t_k) \end{array}\right.$$

*Замечание* 20.3. Смысл за этим стоит следующий: мы берём все подсказки $w$ такие, что $(x,w) \in R$, сдвигаем их по $k$ различным векторам в разных направлениях и тогда $x \in L \iff$ мы так покрыли все подсказки.

Заметим, что если $x \notin L$, то для каждого конкретного $t_i$ вероятность события $R(x, r \oplus t_i) = 1$ не превосходит $2^{-n}$. А вероятность того, что случится одно из $k$ событий (пусть и зависимых) вероятности не более $2^{-n}$ каждое, не превосходит $k \cdot 2^{-n}$. Если это число меньше единицы, то гарантированно найдётся такое $r$, что ни одно $t_i$ не сделает $R(x, r \oplus t_i)$. Значит, имеем ограничение на $k$:

$$k < 2^n$$

*Замечание* 20.4. Это экспоненциальное ограничение, а $k$ может быть максимум полиномиально, так как мы должны под квантором получить $k$ строчек. Так что это условие можно считать автоматически выполненным с некоторого $n$, чего нам достаточно — для мелких $n$ нам неважно.

А если $x \in L$, то хочется сказать, ситуация обратная — мы должны суметь покрыть тот маленький кусок подсказок, которые не удовлетворяют $R$, так как хороших подсказок у нас очень много.

## 20.2. Аккуратная оценка вероятности

Давайте покажем, что если $x \in L$, то существует подходящий набор $t_i$, заодно узнаем, какого $k$ нам хватит. Выберем все $t_i$ случайно равновероятно и оценим сверху вероятность события «нашлась хотя бы одна подсказка $r$, для которой всё сломалось». Если эта вероятность окажется меньше единицы, то нужный набор $t_i$ существует.

Обозначим за $\chi(P)$ обозначим характеристическую функцию предиката $P$ (ноль, если $P$ ложно и единица иначе):

$$\begin{gathered}
\Pr_{t_1,\dots,t_k}\{\exists r\colon R(x, r\oplus t_1) = \dots = R(x, r \oplus t_k) = 0\} = \\
= \mathop{\mathrm{E}}_{t_1,\dots,t_k} \chi\left(\exists r\colon R(x, r\oplus t_1) = \dots = R(x, r \oplus t_k) = 0\right) \leqslant \\
\leqslant \mathop{\mathrm{E}}_{t_1,\dots,t_k} \sum_r \chi\left(R(x, r\oplus t_1) = \dots = R(x, r \oplus t_k) = 0\right) = \\
= \sum_r \mathop{\mathrm{E}}_{t_1,\dots,t_k} \chi\left(R(x, r\oplus t_1) = \dots = R(x, r \oplus t_k) = 0\right) = \\
= \sum_r \Pr_{t_1,\dots,t_k}\{R(x, r\oplus t_1) = 0 \wedge \dots \wedge R(x, r \oplus t_k) = 0\} = \\
= \sum_r \prod_{i=1}^{k} \Pr_{t_i}\{R(x, r \oplus t_i) = 0\} = \sum_r \left(\Pr_t\{R(x, r\oplus t) = 0\}\right)^k = \sum_r \left(\Pr_t\{R(x, t) = 0\}\right)^k \leqslant \\
\leqslant \sum_r \left(\frac{1}{2^n}\right)^k = \sum_r \frac{1}{2^{nk}} = \frac{2^m}{2^{nk}}
\end{gathered}$$

Наше требование:

$$2^m < 2^{nk} \iff nk > m$$

В частности, можно положить $k = m$ (если $n > 1$), это полином от $n$, что и требовалось.

## 20.3. Следствие

*Следствие* 20.0.1. $\mathit{BPP} \subseteq \Pi_2$

▶ Пусть $L \in \mathit{BPP}$. Тогда $\bar{L} \in \mathit{BPP}$, так как $\mathit{BPP}$ замкнут относительно отрицания. Но тогда по теореме $\bar{L} \in \Sigma_2$. Значит, $\bar{\bar{L}} = L \in \Pi_2$, что и требовалось. ◀

# 21. Билет 21

## 21.1. Определения

**𝔇𝔢𝔣 21.1.** *Интерактивный протокол* — это описание процесса передачи информации между двумя независимыми устройствами: $P$ (Prover) и $V$ (Verifier). Передача информации происходит по раундам, в каждом раунде одно устройство передаёт какую-то информацию другому, обычно раунды чередуются (сначала Prover передаёт, потом Verifier, потом снова Prover...), кто начинает — входит в протокол. Полезно считать суммарный размер переданных данных. Prover может использовать случайные биты, может вычислять произвольные функции (в том числе невычислимые), Verifier обычно может вычислять лишь какие-то ограниченные по времени/памяти функции. Также Verifier может использовать случайные биты, которые Prover'у неизвестны.

**𝔇𝔢𝔣 21.2.** *Класс $IP$* — это класс языков $L$, распознаваемых при помощи интерактивного протокола:

1. Распознаваемая строчка $x$ известна изначально и Prover, и Verifier.
2. Суммарный размер переданных данных и число раундов должны быть п.о. размером $x$.
3. Время работы Verifier на каждом раунде должно быть п.о. размером $x$ (соответственно, общее время работы Verifier тоже п.о.).
4. После окончания протокола Verifier должен выдать ответ «да/нет»: верно ли, что $x \in L$.
5. Если $x \in L$, то у $P$ должна существовать стратегия, которая заставит Verifier сказать «да» с вероятностью, большей $\frac{2}{3}$ (по случайным битам Verifier и Prover)
6. Если $x \notin L$, то у $P$ нет стратегии, которая заставит Verifier сказать «да» с вероятностью, большей $\frac{1}{3}$

**Утверждение 21.1.** Можно считать, что Prover детерминирован.

▶ Возьмём недетерминированный Prover и переделаем его в детерминированный.

Заметим, что мы можем неуменьшить вероятность убеждения Verfier, детерминировав Prover так: давайте на последнем шаге Prover вместо отсылки случайной строки из некоторого множества с каким-то распределением посчитает для каждого возможного ответа вероятность того, что он убедит Verifier и отошлёт самую вероятную строку. Очевидно, это не ухудшит вероятность в убеждения Verfirier. То же самое потом сделаем с предпоследним шагом и так далее.

Если $x \in L$, то заметим, что стратегия, убеждающая Verifier всё еще существует — мы просто переделаем вероятностную стратегию, вероятность не ухудшится.

Если $x \notin L$, то вероятность убеждения Verifier'а не изменалсь — для каждой недетерминированной стратегии была детерминированная с хотя бы такой же вероятностью, а все детерминированные стратегии остались. ◀

*Замечание* 21.1. Доказывать принадлежность $IP$ обычно надо так: сначала придумать интерактивный протокол, а дальше доказать, что правильно действующий Verifier удовлетворяет всем условиям. При этом важно помнить, что Prover может отвечать как угодно (в частности, не соблюдать протокол). Обычно полезно считать, что если Verifier «поймал за руку Prover» (т.е. обнаружил, что Prover нарушает протокол), то сразу надо говорить «нет». На случай $x \in L$ это не влияет, так как Prover в этом случае может просто действовать в соответствии с протоколом. А в случае $x \notin L$ это обычно упрощает доказательство.

**Утверждение 21.2.** В классе $IP$ тоже можно понижать ошибку, как и в $BPP$, т.е. для произвольного полинома $p(|x|)$ можно добиться того, чтобы вероятность ошибки Verifier не превосходила $2^{-p(|x|)}$ для любого $x$.

▶ Пишем оценку Чернова **TODO**, делаем $\approx p(|x|)$ запусков протокола (в $p(|x|)$ больше раундов), а в конце Verifier выбирает тот ответ, который был чаще.

Если $x \in L$, то у Prover в каждом запуске есть стратегия, значит, у него в целом есть стратегия, при которой в каждом из запуском получится ответ «да».

Если $x \notin L$, то в каждом из $p(|x|)$ запусков вероятность получить ответ «да» от Verifier небольшая, оценка опять работает. ◀

*Замечание* 21.2. Аналогично можно считать, что константы у нас не $1/3$ и $2/3$, а какие-нибудь две различные.

*Замечание* 21.3. Ещё интерактивные протоколы можно использовать для вычисления функций (например, перманента матрицы). Происходить будет так: в первом же раунде Prover говорит Verifier'у ответ на задачу, а дальше идёт интерактивный протокол для массовой задачи поиска, в котором Prover доказывает Verifier'у, что это действительно корректный ответ.

## 21.2. Протокол для неизоморфизма

**Def 21.3.** *Язык пар неизоморфных графов*: это множество таких пар $(G, H)$, что $G$ и $H$ являются неизоморфными друг другу графами.

*Пример* 21.1. $(C_4, K_{2,2})$ не лежат в этом языке — цикл из четырёх вершин изоморфен полному двудольному графу на 4 вершинах.

*Пример* 21.2. $(C_6, K_{3,3})$ лежат в в этом языке — в этих графах просто разное число рёбер.

**Теорема 21.1.** Язык пар неизоморфных графов лежит в $IP$.

▶ Предъявим протокол: пусть известна пара $(G_0, G_1)$. Verifier для начала должен проверить, что ему на вход действительно закодировали пару графов. Дальше Verifier случайно выбирает один бит $a$, случайным образом перенумеровывает вершины графа $G_a$, получает граф $G'$ и посылает $G'$ Prover'у. После этого Prover обязан возвратить Verifier'у бит $a$. Если Prover возвращает не тот бит, ответ — «нет».

Такой Verifier точно полиномиален, а раунд вообще всего один.

Если $G_0$ и $G_1$ неизоморфны (т.е. $x \in L$), то Prover, увидя граф $G'$, сразу может сказать, кому из $G_0$ и $G_1$ он изоморфен, а кому — нет (это даже вычислимая функция). Значит, он точно может правильно ответить Verifier, во всех случаях.

Теперь пусть $G_0$ и $G_1$ изоморфны (т.е. $x \notin L$). Заметим, что тогда Prover получит на вход случайный граф, изоморфный одновременно и $G_0$, и $G_1$, причём все такие графы он получит равновероятно. На каждом входе Prover'а ровно в половине случаев этот вход был получен из $G_0$, а в половине — из $G_1$. Рукомахательное рассуждение: ясно, что Prover никакой информации из полученного графа не получает, значит, ему надо угадывать, причём равновероятно, а тогда вероятность угадать не больше $\frac{1}{2}$.

Более строго: пусть в случае получения помеченного графа $H$ Prover с вероятностью $p_H$ выдаёт единицу, а с вероятностью $1 - p_H$ — нолик. Тогда посчитаем вероятность того, что Prover угадал. Протокол может ошибиться только в том случае, если Prover угадал, в каком случае он находится. Посчитаем вероятность этого события, есть два случая (для фиксированного $H$):

1. Prover выдал 0 (вероятность этого — $p_H$), ответ был 0 (вероятность этого — $1/2$). Значит, вероятность этого случая — $\frac{p_H}{2}$.

2. Prover выдал 1 (вероятность этого — $1-p_H$), ответ был 1 (вероятность этого — $1/2$). Значит, вероятность этого случая — $\frac{1-p_H}{2}$.

Суммируем, получаем, что независимо от $H$ вероятность того, что Prover угадает, равна $\frac{1}{2}$, что и требуется для интерактивного протокола. ◀

## 21.3. Протокол для перманента

**Def 21.4.** *Перманент* матрицы $A$ ($\operatorname{perm} A$) порядка $n$ — это как определитель, только везде знак «плюс»:

$$\operatorname{perm} A \stackrel{\text{Def}}{=} \sum_{\sigma \in S_n} a_{1,\sigma(1)} \cdot a_{2,\sigma(2)} \cdot \ldots \cdot a_{n,\sigma(n)}$$

*Замечание* 21.4. Полиномиальный алгоритм вычисления определителя в некотором поле у нас есть (если мы считаем, что любые операции в поле идут за полином) — это алгоритм Гаусса приведения матрицы к верхнетреугольному виду. А вот для перманента такого счастья нет, его вычислять очень сложно (точнее, просто мы не умеем). Зато есть интерактивный протокол для проверки ответа, это прикольный факт.

*Замечание* 21.5. Тем не менее, раскладывать перманент по строке или столбцу всё-таки можно — так же, как определитель, но знак везде «плюс».

*Замечание* 21.6. Дальше мы везде будем считать, что в задачу проверки перманента как-то входит описание поля $F$, в котором можно эффективно вычислять результаты арифметических операций. Например, в виде машин Тьюринга, которые выполняют операции с этим полем. Также нам будет важно, что размер поля достаточно большой (больше $n^4$).

**Def 21.5.** Язык $\mathit{PERM}$ — это множество пар $(A, c)$, где $A$ — матрица порядка $n$, а $c$ — её перманент в поле $F$.

**Def 21.6.** Язык $\mathit{PERM}_2$ — это множество четвёрок $(A, B, c, d)$, где $A$ и $B$ — матрицы порядка $n$, а $c$ и $d$ — их перманенты, соответственно.

**Лемма 21.1.** Если $\mathit{PERM} \in \mathit{IP}$, то и $\mathit{PERM}_2 \in \mathit{IP}$.

▶ Пусть есть четвёрка $(A, B, c, d)$. Если $n = 1$, то Verifier может сам проверить. Интересный случай — $n > 1$.

Давайте рассмотрим такую матрицу над кольцом многочленов от одной переменной над $F$:

$$A'(x) = Ax + B(1-x) \in F[x]$$

Её перманент — это какой-то многочлен $p$, причём $p(0) = \operatorname{perm} B$, $p(1) = \operatorname{perm} A$. $p$ может иметь любые коэффициенты (в том числе все нули). Давайте спросим у Prover перманент такой матрицы в форме коэффициентов многочлена ($n$ элементов поля $F$).

*Замечание* 21.7. Мы не запускаем ещё раз протокол внутри — нам в протоколе надо поле, а не кольцо. Мы просто просим у Prover'а многочлен, проверим потом.

Проверим, во-первых, что $p(0) = d$ и $p(1) = c$, иначе сразу ответим «нет» — мы либо поймали за руку, либо исходная четвёрка не лежала в языке.

Дальше осталось проверить корректность многочлена. Для этого выбираем случайную точку $x$ из поля, подставляем её в многочлен (получаем значение) и в матрицу $A'$ (получаем матрицу над $F$). Дальше проверяем интерактивным протоколом для $\mathit{PERM}$, что $\operatorname{perm}(A'(x)) = p(x)$.

Очевидно, если Prover следует протоколу, то он нас всегда убедит. Оценим вероятность ошибки: исходная четвёрка в языке не лежит, но мы поверили Prover'у. Заметим, что для этого Prover должен был заведомо прислать некорректный многочлен $p$ (иначе бы не сошлось $p(0) = d \wedge p(1) = c$), то есть: Два случая:

1. $\operatorname{perm}(A'(x)) = p(x)$, то есть $(\operatorname{perm} A' - p)(x) = 0$, то есть случайно выбранный из поля $x$ попал в корень многочлена степени не выше $n$. Вероятность этого случая не больше $\frac{n}{|F|}$ (так как $p \not\equiv \operatorname{perm} A'$), в нём нас обманут с вероятностью 1 — ну, тут мы сами дураки, не угадали $x$.
2. $\operatorname{perm}(A'(x)) \neq p(x)$ (что происходит с вероятностью не больше единицы), то есть нас обманули уже во вложенном интерактивном протоколе. Но вероятность этого события не больше $\frac{1}{3}$.

Значит, вероятность ошибки не больше:

$$\frac{n}{|F|} \cdot 1 + 1 \cdot \frac{1}{3}$$

Если $|F| > 3n$, то мы получили вероятность ошибки, отделённую от единицы, её можно понижать. ◀

**Лемма 21.2.** Если $\mathsf{PERM}_2 \in \mathsf{IP}$, то задача $\mathsf{PERM}_k \in \mathsf{IP}$ (проверка перманентов сразу $k$ матриц) лежит в $\mathsf{IP}$.

▶ Аналогично предыдущей лемме сначала склеим первые две матрицы в одну, потом получившуюся с третьей, с четвёртой и так далее. А в конце сделаем один запуск интерактивного протокола.

*Замечание* 21.8. Сделать $n$ запусков интерактивного протокола мы не можем, так как тогда время работы протокола разрастётся экспоненциально.

Заметим, что Prover может нас обмануть и заставить сказать «да» только в том случае, если мы хотя бы на каком-то шаге выбрали неправильный $x$ (попавший в корень), либо уже в финальном запуске. Вероятность выбрать неправильный $x$ на каком-то шаге не превышает $n \cdot \frac{n}{|F|} = \frac{n^2}{|F|}$, то есть нам надо иметь поле размера хотя бы $3n^2$, чтобы получить отделённую от единицы вероятность ошибки. ◀

**Теорема 21.2.** $\mathsf{PERM} \in \mathsf{IP}$

▶ Взяли матрицу $A$ порядка $n$ и константу $c$. Если $n = 1$, то Verifier может сам проверить ответ.

Иначе разложим перманент по первой строке и попросим у Prover'а перманенты соответствующих миноров:

$$\operatorname{perm} A = a_1 \cdot A_1 + a_2 \cdot A_2 + \dots + a_n \cdot A_n$$

Во-первых, проверим, что сумма сошлась с константой $c$, иначе либо ответ сразу «нет», либо мы поймали за руку.

Дальше выполним интерактивный протокол для задачи $\mathsf{PERM}_{n-1}$ для проверки перманентов сразу всех матриц $A_i$. ◀

*Замечание* 21.9. Если оценивать сразу, а не через уменьшение ошибки (так вообще можно?), то можно взять поле размера приблизительно $n^3(1+\varepsilon)$ и получить ошибку не более $\frac{1}{1+\varepsilon}$. У нас всего в протоколе будет порядка $n^2$ выборов $x$, каждый неверен с вероятностью не более $\frac{n}{|F|}$, значит вероятность ошибки хоть где то не более $\frac{n^3}{|F|}$.

# 22. Билет 22

## 22.1. Введение

Будем доказывать двумя включениями.

**Лемма 22.1.**

$$IP \subseteq PSPACE$$

▶ Это простое включение: $IP \subseteq PSPACE$. Для начала добавим в протокол полиномиальный будильник, чтобы ограничить объём данных. Пусть дали вход $x$, хотим проверить $x \in L$. Надо нарисовать дерево возможных ответов Prover'а и Verifier'а. В каждой вершине считаем вероятность принятия строки Verifier'ом начиная из данного состояния (храним как несократимую дробь, длина числителя и знаменателя ограничены полиномом). Если в вершине ход делает Verifier — надо взять взвешенное среднее по всем детям (Verifier делает случайный переход). Если в вершине ход делает Prover — надо взять максимум по всем детям. Таким dfs'ом проходимся по дереву и в конце-концов посчитаем вероятность принятия $x$ при оптимальной стратегии Prover. Дальше надо лишь сравнить её с $1/3$ и $2/3$. ◀

Сложное включение: $PSPACE \subseteq IP$. Нам будет досточно построить протокол для $PSPACE$-полной задачи $QBF$. Тогда для построения интерактивного протокола для другой задачи $L \in PSPACE$ перед началом протокола Prover и Verifier независимо за полиномиальное время поймут, про какую $QBF$-задачу они говорят, а потом уже будут работать с этой $QBF$-задачей.

## 22.2. Арифметизация

Для начала перейдём от булевой формулы к арифметической формуле с арифметическими переменными, обычными арифметическими операциями и трёмя дополнительными унарными операциями. Отображение булевских операций такое:

- Формула от переменных $\to$ многочлен от тех же переменных
- `false` $\to 0$
- `true` $\to 1$
- $a \wedge b \to a \cdot b$
- $\neg a \to 1 - a$
- $a \vee b = \neg(\neg a \wedge \neg b) = 1 - (1-a)(1-b)$, эту операцию обозначим $a \odot b$

Также добавим ещё три операции над многочленами, они будут вести себя так:

- «Квантор всеобщности» по переменной:

  $$A_x P(x_1, \dots, x_k, x) := P(x_1, \dots, x_k, 0) \cdot P(x_1, \dots, x_k, 1)$$

  Степень многочлена от этой операции увеличивается не более, чем в два раза.

- «Квантор существования» по переменной:

  $$E_x P(x_1, \dots, x_k, x) := P(x_1, \dots, x_k, 0) \cdot P(x_1, \dots, x_k, 1)$$

  Степень многочлена от этой операции увеличивается не более, чем в два раза.

- Линеаризация:

  $$L_x P(x_1, \dots, x_k, x) := P \bmod (x^2 - x)$$

  Эта операция не меняет значение многочлена в точках, где все $x_i \in \{0, 1\}$, так как $x^k \bmod (x^2 - x) \equiv x^{1 \text{ или } 2}$, что равно $x$ при $x \in \{0, 1\}$. То есть в целом эту операцию можно пихать в любых местах, она нужна только для понижения степени многочлена по одной переменной до единицы. Если применить несколько таких операций по всем переменным, то степень многочлена упадёт до количества переменных.

Нам надо было проверить, что такая булева формула истинна (с точностью до первого квантора):

$$\forall x_1 \colon \exists x_2 \colon \dots Q_k x_k \colon \varphi(x_1, \dots, x_k)$$

Это в терминах наших новых операций равносильно тому, что следующее замкнутое арифметическое выражение равно единице:

$$A_{x_1} L_{x_1} E_{x_2} L_{x_1} L_{x_2} A_{x_3} L_{x_1} L_{x_2} L_{x_3} \dots p(x_1, x_2, \dots, x_n)$$

*Замечание* 22.1. Это выражение действительно замкнутое, имеет размер порядка $n^2 + |p|$, несмотря на то, что внутри есть многочлен от $n$ переменных. Все эти переменные убиваются операциями над многочленом.

Арифметизацию перед началом протокола Verifier выполнит самостоятельно.

## 22.3. Интерактивный протокол

Дальше Prover будет доказывать Verifier'у, что имеющаяся у него формула тождественно равна единице. Мы будем рекурсивно строить чуть более общий протокол, который позволяет проверять равенства такого вида для суффиксов формулы:

$$\left(q_{x_i} q_1 q_2 \dots q_\alpha p(x_1, \dots, x_{i-1}, x_i, \dots, x_n)\right) \Big|_{\substack{x_1 = r_1 \\ \vdots \\ x_{i-1} = r_{i-1}}} = c$$

Здесь $q_{x_i}$ — некоторый оператор по переменной $x_i$, $q_i$ — какие-то последующие операции, $r_i$ — подстановленные значения для переменных $x_1, \dots, x_{i-1}$, которые уже не будут убиваться встречающимися в формуле операторами, $c$ — значение, которому должно быть равно выражение.

*Замечание* 22.2. В случае $q = L$ также будет иметься фиксированное значение $r_i$ для переменной $x_i$ (т.е. она тоже подставляется в выражение).

Если операций не осталось, то мы уже знаем значения всех переменных и Verifier легко сам вычислит значение. А чтобы построить протокол для исходной задачи, надо просто начать с первой операции и $i = 1$ (т.е. никакие значения переменных ещё не известны).

По протоколу Prover должен первым шагом разложить следующий многочлен $x_i$ и прислать его коэффициенты:

$$s(x_i) := q_1 q_2 \dots q_\alpha p(r_1, \dots, r_{i-1}, x_i, x_{i+1}, \dots, x_n)$$

Это просто вся формула после $q_{x_i}$ с подставленными $r_1, \dots, r_{i-1}$. Дальше несколько случаев:

**$q = A$:** Тогда для начала надо проверить, что $s(0) \cdot s(1) = c$ (т.е. что оператор $A$ правильно посчитан в предположении корректности $s$). Если сошлось, то выбираем случайное $x_i$ *из некоторого поля* (т.е. не обязательно 0 или 1), считаем $s(r_i)$ самостоятельно и дальше рекурсивно просим Prover доказать, что многочлен $s$ действительно такой, как он нам сказал.

**$q = E$:** Аналогично, но проверять надо $s(0) \odot s(1) = c$

**$q = L$:** В этом случае у нас уже есть значение $r_i$ для подстановки в переменную $x_i$, надо проверить $s(r_i) = c$. А потом надо опять проверить, что $s$ нам дали корректный: выбираем новый случайный $r'_i$ *из поля* и просим Prover доказать, что многочлен $s$ корректен, доказав, что он в случайной точке равен тому, что мы сами посчитали.

На одном шаге мы облажаемся (т.е. попадём в корень $s(r_i) - s_{\text{реальный}}(r_i)$) с вероятностью не более $\frac{d}{|F|}$, а если мы нигде не облажаемся, то в конце-концов поймаем Prover'а за руку. Всего шагов у нас не более $d^2$, т.е. надо брать поле размера порядка $d^4$, чтобы получить ошибку $d$. А на любом шаге $d$ не более $2n$ из-за операторов линеаризации, что очень хорошо.

# 23. Билет 23

## 23.1. Классы $\mathsf{AM}$ и $\mathsf{MA}$

**𝔇𝔢𝔣 23.1.** Классы вроде $\mathsf{MAAMAMAMA}$ — это как $\mathsf{IP}$, но со следующими отличиями:

1. Prover зовётся Мерлин, а Verifier зовётся Артур
2. Случайные биты, которые использует Артур, становятся доступны Мерлину (но только после того, как он их сгенерировал)
3. Число раундов равно длине строки-названия класса
4. На $i$-м раунде передаёт информацию тот, чья буква стоит на $i$-м месте

*Замечание* 23.1. Бессмысленно пихать две одинаковые буквы подряд — зачем разбивать передачу данных на два раунда?

*Замечание* 23.2. Можно считать, что Мерлин знает, как работает Артур и поэтому на самом деле Артуру ничего говорить Мерлину (кроме случайных бит) не нужно. Ему нужно только в конце сказать, верит он или не верит. Требуется, чтобы Артур ошибался с вероятностью не более $\frac{1}{3}$.

*Замечание* 23.3. Есть ещё классы с односторонней ошибкой — $\mathsf{AM}_1$ и $\mathsf{MA}_1$, тут требуется, чтобы в случае $x \in L$ Артур никогда не ошибался.

*Замечание* 23.4. Иногда тут тоже зовут Prover/Verifier, но оговариваются, что у нас private coin (приватные случайные биты).

*Замечание* 23.5.

$$\mathsf{NP} \subset \mathsf{MA} \subset \mathsf{AM}$$

**Утверждение 23.1.** В $\mathsf{MA}$ можно понижать ошибку до $2^{-poly(n)}$ лишь полиномиальным увеличением времени работы ($n$ — размер входа).

▶ Мерлин присылает нам одну подсказку, мы её $\approx poly(n)$ раз тестируем, выдаём majority. Оценка — как обычно, через неравенство Чернова. В каждом раунде мы ошибёмся с вероятностью не более $1/3$. ◀

**Утверждение 23.2.** В $\mathsf{AM}$ можно понижать ошибку до $2^{-poly(n)}$ лишь полиномиальным увеличением времени работы ($n$ — размер входа).

▶ Пусть Артур выкинет не один набор случайных бит, а столько, сколько надо для $\approx poly(n)$ раундов. Мерлин пришлёт по подсказке на каждый раунд, мы их все тестируем, берём majority. Важное наблюдение для доказательства — все раунды независимы в том смысле, что Мерлин либо может на каких-то случайных битах нас убедить, либо не может. Если мог, но не воспользовался — он сам дурак.

Если $x \in L$, то в каждом раунде нам либо повезло с подсказкой (и Мерлин может ответить), либо не повезло. Majority этих событий с хорошей вероятностью «повезло».

Если $x \notin L$, то в каждом раунде нам либо не повезло с подсказкой (и Мерлин может ответить), либо повезло (и Мерлин не может). Majority этих событий с хорошей вероятностью «не повезло». ◀

## 23.2. Попарно независимые хэш-функции

**Def 23.2.** Семейство хэш-функций $H$, которые действуют из $\{0,1\}^n$ в $\{0,1\}^k$ является *попарно независимым семейством*, если:

1. Каждый элемент равновероятно (по функции) отправляется в любой хэш:
$$\forall x \in \{0,1\}^n\colon \forall \{0,1\}^k\colon \Pr_h\{h(x) = y\} = \frac{1}{2^k}$$
2. Каждые два разных элемента равновероятно (по функции) отправляется в любые два хэша (может, одинаковых):
$$\forall x, x' \in \{0,1\}^n\colon \forall y, y'\{0,1\}^k\colon (x \neq x') \Rightarrow \Pr_h\{h(x) = y \wedge h(x') = y'\} = \frac{1}{2^{2k}}$$

**Утверждение 23.3.** Без доказательства: для любого $n \geqslant 1$ существует поле с $2^n$ элементами.

**Теорема 23.1.** Если $k = n$, то есть попарно независимое семейство хэш-функций.

▶ Случай $n = 0$ неинтересен. Иначе скажем, что наши битовые строчки (и размера $n$, и размера $k$) соответствуют элементам поля. Тогда возьмём семейство из $2^{2n}$ хэш-функций, индексируемых двумя элементами поля:
$$h_{a,b}(x) = ax + b$$

Покажем, что каждый элемент равновероятно (по функции) отправляется в любой хэш. Зафиксируем $x$ и $y$, посчитаем число решений уравнения $ax + b = y$ (относительно $a$ и $b$). Очевидно, что для каждого возможного $a$ существует ровно одно подходящее $b$, т.е. всего решений $2^k$, а всего хэш-функций — $2^{2k}$, получили нужную вероятность.

Покажем, что для двух элементов тоже выполняется что надо. Зафиксируем $x \neq x'$, $y$, $y'$. Посчитаем число решений системы:
$$\begin{cases} ax + b = y \\ ax' + b = y' \end{cases} \iff \begin{cases} ax + b = y \\ a(x' - x) = y' - y \end{cases} \ominus\!\!\!\!\!\ominus \iff$$
$$\text{так как } x' - x \neq 0$$
$$\ominus\!\!\!\!\!\ominus \iff \begin{cases} b = y - ax \\ a = \frac{y' - y}{x' - x} \end{cases}$$

Получаем, что решение всегда единственно. Значит, вероятность действительно равна $\frac{1}{2^{2k}}$. ◀

*Следствие* 23.1.1. Если $k > n$, то есть семейство попарно независимых хэш-функций.

▶ Давайте построим семейство попарно независимых хэш функций из $\{0,1\}^k$ в $\{0,1\}^k$. После этого возьмём в точности его, а чтобы хэш-функции стали принимать строки короче (длины $n < k$), будем их просто добивать нулями. Так как у нас вероятности берутся по хэш-функциям, а по входам стоят только кванторы всеобщности, ничего не поменялось — мы лишь ослабили условия. ◀

*Следствие* 23.1.2. Если $k < n$, то есть семейство попарно независимых хэш-функций.

▶ Давайте построим семейство попарно независимых хэш функций из $\{0,1\}^n$ в $\{0,1\}^n$. После этого возьмём в точности его, а чтобы хэш-функции стали выдавать хэши короче (длины $k < n$), обрежем хэши. Теперь получим, что условие $h(x) = y$ на самом деле (в исходном семействе) равносильно дизъюнкции $2^{n-k}$ аналогичных условий для исходного семейства. Но все эти условия дизъюнктны. То есть вероятность можно просто сложить, получить, что надо: $2^{-n} \cdot 2^{n-k} = 2^{-k}$. Аналогично для второго условия. ◀

## 23.3. Протокол Гольдвассер-Сипсера

Раздел 1.16.2 в конспекте Оли, и викиконспекты.

**Теорема 23.2.** Пусть есть некоторое множество $S \subseteq \{0,1\}^n$ и для каждого $x \in S$ есть сертификат его принадлежности $S$ (полиномиальный от $n$, полином один на всё $S$), а если $x \notin S$, то такого сертификата нет. Также есть фиксированное число $K$.

Тогда можно построить протокол из $\mathsf{AM}$ такой, что если $|S| \geqslant k$, то с вероятностью $2/3$ протокол говорит «да», а если $|S| \leqslant K/2$, то с вероятностью $2/3$ говорит «нет»

## 23.4. Применение протокола Гольдвассер-Сипсера

Если у нас был протокол, который использовал приватные случайные биты, то можно его переделать в протокол, который использует публичные биты. Просто теперь Мерлин убеждает Артура, что множество случайных бит, в которых бы Verifier поверил Prover'у в закрытой игре, довольно большое. Сертификат элемента множества (случайных бит) — это как раз ответ Prover'а.

# 24. Билет 24

Раздел 1.16.1 в конспекте Оли и викиконспекты.

**𝔇𝔢𝔣 24.1.** Булева формула называется *одновыполнимой*, если у неё есть ровно один выполняющий набор.

*Замечание* 24.1. Язык таких формул называется $\mathsf{USAT}$ (unique SAT).

**Теорема 24.1.** По любой булевой формула $\varphi$ можно за полиномиальное время построить такой набор $\varphi_1, \dots, \varphi_m$, что:

- Если $\varphi$ невыполнима, то $\varphi_i$ тоже невыполнимы
- Если $\varphi$ выполнима, то с вероятностью хотя бы $1/2$ среди $\varphi_i$ есть одновыполнимая формула.

# 25. Билет 25

## 25.1. $\mathsf{PCP}$

**𝔇𝔢𝔣 25.1.** $\mathsf{PCP}[f(n), g(n)]$ — это класс таких языков, что существует полиномиальная машина Тьюринга $V$ (verifier), которая, используя $\mathcal{O}(f(n))$ битов рандома, неадаптивно обращаясь к $\mathcal{O}(g(n))$ битам подсказки, гарантированно принимает корректные подсказки и с вероятностью не более $\frac{1}{2}$ принимает некорректные подсказки.

**Теорема 25.1.** Без доказательства:

$$\mathsf{NP} = \mathsf{PCP}[\log n, 1]$$

**Теорема 25.2.** Будем доказывать в следующих билетах:

$$\exists k\colon \mathsf{NP} = \mathsf{PCP}[poly(k), 1]$$

## 25.2. Связь с неаппроксимируемостью

Из $PCP$-теоремы можно вывести, что некоторые задачи в предположении $P \neq NP$ не только плохо решаются, но даже плохо приближаются.

**Def 25.2.** $MAX - 3SAT$ — посчитать по данной формуле $\varphi$ в 3-КНФ, какое максимальное число клозов можно выполнить одновременно.

*Замечание* 25.1. Эта задача $NP$-трудная, так как к ней сводится $SAT$.

**Утверждение 25.1.** Есть алгоритм $A$ такой, что по любой формуле в 3-КНФ с $m$ клозами (из которых доля $\alpha$ могут быть выполнены) он выдаёт решение, которое выполняет не менее $\frac{m}{2}$ клозов.

▶ Жадный алгоритм: переберём все переменные по очереди, для каждой выбираем то значение, которое выполнит наибольшее число клозов из оставшихся. Без доказательства? **TODO** ◀

**Утверждение 25.2.** Без доказательства: есть алгоритм $A$ такой, что по любой формуле в 3-КНФ с $m$ клозами (из которых доля $\alpha$ могут быть выполнены) он выдаёт решение, которое выполняет не менее $^7/_8m$ клозов.

**Лемма 25.1.** Существует константа $0 \leqslant c \leqslant 1$ и полиномиальный алгоритм, который любую булеву формулу $\varphi$ в 3-КНФ может переделать в 3-КНФ формулу $\varphi'$, причём:

- Если $\varphi$ была выполнима, то $\varphi'$ тоже выполнима
- Если $\varphi$ была невыполнима, то $\varphi'$ сильно невыполнима, т.е. в ней нельзя выполнить долю клозов больше $c$

▶ Мы знаем, что $PCP[\log n, 1] = NP$. Мы знаем, что $3SAT \in NP$, тогда давайте возьмём PCP-алгоритм для него. Давайте переберём все возможные случайные биты, которые потребует PCP-алгоритм для проверки выполнимости $\varphi$, их всего полином. При конкретном наборе случайных бит можно записать формулу *константной длины* от битов доказательства, которая будет проверять соответствующие биты. Давайте эту формулу от констаного числа бит доказательства запишем в 3-КНФ, она всё ещё останется константной длины. Обозначим эту константу $C$.

Давайте теперь запишем конъюнкцию всех этих 3-КНФ блоков по всем случайным наборам, получим формулу $\varphi'$. Если $\varphi$ была выполнима, то и $\varphi'$ выполнима, так как PCP-алгоритм всегда принимает, независимо от случайных бит. В противном же случае, хотя бы половина «блоков» должна быть невыполнима (т.е. в каждом блоке хотя бы один клоз должен быть невыполним). Значит, всего хотя бы $\frac{1}{2C}$ клозов должно быть невыполнимо в формуле $\varphi'$ — эту долю и обозначим за $1 - c$. ◀

*Следствие* 25.2.1. Из этого сразу следует, что если есть полиномиальный алгоритм, решающий $MAX - 3SAT$ приближённо с долей больше $c$, то $P = NP$, так как мы умеем решать $3 - SAT$ точно. В самом деле: берём формулу $\varphi$, строим $\varphi'$, запускаем на ней наш чудо-алгоритм. Если $\varphi$ была выполнима, то он выдаст долю клозов больше $c$, иначе — не выдаст (так как это невозможно по построению $\varphi'$).