Содержание

1. Введение в теорию сложности

1.1 Основные классы

Алгоритмы позволяют для какой-то задачи сказать, за сколько она решается: дана задача $A \to$ решим ее за $\mathcal{O}(2^n)$

Теория сложности же позволяет сказать, что для какой-то задачи не существует алгоритма, решающего ее за какую-то асимптотику: дана задача $A \to$ не решается быстрее, чем за $\mathcal{O}(n \log n)$.

Алгоритмически не разрешимые задачи

Существуют ли неразрешимые задачи (то есть для которых нет решающих их алгоритмов)? На самом деле, таких задач больше, чем алгоритмов.

Рассмотрим следующие задачи: $A : \begin{cases} \text{input: } n \in \mathbb{N} \\ \text{output: } true/false \end{cases}$.

Любую такую задачу можно задать подмножеством натуральных чисел, на которых ответ true $A \subseteq \mathbb{N}$.

Задач по крайней мере столько, сколько множеств натуральных чисел — $|2^N|$. Алгоритмов счётное число (то есть $|N|$), ведь их всех можно пронумеровать: сначала выпишем все однобуквенные, затем все двухбуквенные, и так далее.

$|2^n| > |N|$ — тут можно либо сослаться на общую теорему Кантора ($2^A > |A|$), либо вспомнить школьные доказательства того, что $|2N| = |R|$ и $|R| > |N|$. Так что на самом деле «почти все» задачи неразрешимы.

Пример 1. Неразрешимая задача: задача остановки HALTING: *Дана программа, остановится ли она когда-нибудь на данном входе?*

$$HALTING : \begin{cases} \text{input: код программы и вход для этой программы} \\ \text{output: остановится ли запуск?} \end{cases}.$$

Теорема 1. *Halting problem алгоритмически не разрешима.*

Доказательство. От противного. Пусть есть алгоритм terminates(code, x), всегда останавливающийся, и возвращающий true только если code(x) останавливается. Рассмотрим программу:

```
    def invert(code):
        if terminates(code, code): while (true)
```

Запустим invert(invert), что может случится:

1. Он зависнет $\Rightarrow$ terminate(invert, invert)$=false \Rightarrow$ invert не зависает ?!

2. Он завершится $\Rightarrow$ terminate(invert, invert)$=true \Rightarrow$ invert зависает ?!

Противоречие. Значит, такого terminate не существует. □

Теорема 2. ***Теорема Успенского-Райса***
Любое нетривиальное свойство программ неразрешимо (то есть нет алгоритма, которые бы его проверял).

Поясним, что это значит. Будем говорить, что программы A и B эквивалентны ($A \sim B$), если для каждого входа A либо они обе зависают, либо обе останавливаются и печатают одно и тот же ответ (время работы и память при этом могут отличаться).

Определение 1. *Свойство программы* – это такой предикат $P(code)$ который для любых эквивалентных программ $\mathcal{A}$ и $\mathcal{B}$ выдаёт одно и то же: $\mathcal{A} \sim \lfloor \Rightarrow P(\mathcal{A}) = P(\mathcal{B})$. «Нетривиальное» означает, что хотя бы одна программа ему удовлетворяет, но не все программы.

Теорема 3. ***Теорема Гёделя о неполноте***
Если формальная система S непротиворечива, то в ней невыводимы обе формулы B и $!B$; иначе говоря, если система S непротиворечива, то она неполна, и B служит примером неразрешимой формулы.

Один из способов доказательства этой теоремы – через неразрешимость.

1.2 Decision/search problem

Определение 2. Если в задаче ответ – true/false, то это *decision problem* (задача распознавания). Иначе это *search problem* (задача поиска).

Пример 2.

1. Decision: проверить, есть ли x в массиве a.
2. Search: Найти позицию x в массиве a.
3. Decision: Проверить, есть ли путь из a в b в графе G.
4. Search: Найти сам путь.
5. Decision: Проверить, есть ли в графе клика размера хотя бы k.
6. Search: Найти максимальный размер клики (или саму клику).

Замечание. Decision problem f можно задавать, как язык (множество входов) $L = \{x : f(x) = true\}$.

1.3 DTime, P, EXP (классы для decision задач)

Определение 3. DTime$[f(n)]$ – множество задач распознавания, для которых $C > 0$ и детерминированный алгоритм, работающий на всех входах не более чем $C \cdot f(n)$, где n – длина входа.

Пример 3. IS_SORTED $\in$ DTime[n]
IS_SORTED $\in$ DTime[n^2]

Определение 4. P $= \bigcup\limits_{k>0}$ DTime$[n^k]$. Т.е. задачи, имеющие полиномиальное решение.

Определение 5. EXP$= \bigcup\limits_{k>0}$ DTime$[2^{n^k}]$. Т.е. задачи, имеющие экспоненциальное решение.

Пример 4. KNAPSACK: n предметов, рюкзак размера w, можно ли уложить $\geq k$ предметов?
Умеем решать за $\mathcal{O}(2^n \cdot n)$, за $\mathcal{O}(nw)$ (не полиномиальное решение!).
Длина входа: $\mathcal{O}(n \log n + W \log W + k \log n + n \log W) = |x|$.

Теорема 4. ***Об иерархии по времени***
$DTime[f(n)] \subsetneq DTime[f(n) \log^2 f(n)]$.

Доказательство. $\subset$ понятно, почему.
Но почему не $\subseteq$? Значит, существует задача, которая решает за $\mathcal{O}(f(n) \log^2 f(n))$ и не решается за $\mathcal{O}(f(n))$ шагов.
Задача: дана программа и вход. Завершится ли она на этом входе за $f(n) \log f(n)$ шагов?

□

Следствие 1. *P$\neq$ EXP.*

Доказательство. P $\subseteq$ DTime$[2^n] \subsetneq$ DTime$[2^{2n}] \subseteq$ EXP. □

1.4 NP (non-deterministic polynomial)

Задача $\to$ да + сертификат / нет.

Определение 6. $NP = \{L : \exists$ алгоритм M, работающий за полином от $|x|$, $\forall x\ (\exists y :\ M(x,y) = 1) \Leftrightarrow (x \in L)\}$.

Неформально: «NP – класс языков $L : \forall x \in L$, если нам дадут подсказку $y(x)$, то мы за полином сможем убедиться, что $x \in L$».
Ещё более неформально: «NP ˘ класс задач, к которым ответ можно проверить за полином».
Подсказку y так же называют *свидетелем/сертификатом* того, что x лежит в L.

Пример 5. Примеры NP-задач:

1. HAMPATH $= \{G \mid G$ – неорграф, в котором есть гамильтонов путь$\}$.

   Подсказка y – путь. M получает вход $x = G$, подсказку y проверяет, что y прост, $|y| = n$ и $\forall (e \in y)\ e \in G$.

2. k-CLIQUE – проверить наличие в графе клики размером k.

   Подсказка y – клика.

3. IS-SORTED – отсортирован ли массив? Она даже лежит в P.

Замечание. P $\subseteq$ NP (можно взять пустую подсказку).

Определение 7. coNP$= \{L \mid \overline{L} \in$NP$\}$

Определение 8. coNP$= \{L : \exists M$, работающий за полином от $|x|, \forall x\ (\exists y\ M(x,y) = 0) \Leftrightarrow (x \in L)\}$.

Определение 9. coNP$= \{L : \exists M$, работающий за полином от $|x|, \forall x\ (\exists y\ M(x,y) = 1) \Leftrightarrow (x \notin L)\}$.

Пример 6. Пример coNP задачи:
PRIME – является ли число простым. Подсказкой является делитель.
На самом деле PRIME $\in$ P, но этого мы пока не умеем понимать.

Замечание. Вопрос P $=$ NP или P $\neq$ NP остаётся открытым. Предполагают, что $\neq$.

1.5 NP-hard, NP-complete

Определение 10. $\exists$ *полиномиальное сведение (по Карпу)* задачи A к задаче B ($A \leq_P B$) $\Leftrightarrow \exists$ алгоритм f, работающий за полином, $(x \in A) \Leftrightarrow (f(x) \in B)$.

Замечание. f работает за полином $\Rightarrow |f(x)|$ полиномиально ограничена $|x|$.

Определение 11. $\exists$ *сведение по Куку* задачи A к задаче B ($A \leq_C B$) $\Leftrightarrow \exists M$, решающий A, работающий за полином, которому разрешено обращаться за $\mathcal{O}(1)$ к решению/оракулу B.

Ещё говорят «задача A сводится к задаче B».
В обоих сведениях мы решаем задачу A, используя уже готовое решение задачи B.
Другими словами доказываем, что «A не сложнее B». Различие в том, что в первом случае решением B можно воспользоваться только один раз (и инвертировать ответ нельзя), во втором случае – полином раз.

Определение 12. NP-hard $=$ NPh $= \{L : \forall A \in$ NP $\Rightarrow A \leq_P L\}$.

NP-трудные задачи – класс задач, которые не проще любой задачи из класса NP.

Определение 13. NP-complete = NPc = NPh $\cap$ NP.

NP-полные задачи – самые сложные задачи в классе NP.
Если мы решим хотя бы одну из NPc за полином, то решим все из NP за полином. Хорошая новость: все NP-полные по определению сводятся друг к другу за полином.

Замечание. Когда хотите выразить мысль, что задача трудная в смысле решения за полином (например, поиск гамильтонова пути), неверно говорить «это NP задача» (любая из P тоже в NP) и странно говорить «задача NP-полна» (в этом случае вы имеете в виду сразу, что и трудная, и в NP). Логично сказать «задача NP-трудна».

Лемма 1. *$A \le_P B, B \in P \Rightarrow A \in P$.*

Доказательство. Сведение f работает за n^s, B решается за $n^t \Rightarrow A$ решается за n^{st}. $\square$

Лемма 2. *$A \le_P B, A \in$NPh $\Rightarrow B \in$ NPh.*

Доказательство. $\forall L \in$NP $(\exists f : L$ сводится к A функцией $f(x)) \vee (A \le_P B$ функцией $g(x)) \Rightarrow L$ сводится к B функцией $g(f(x))$ за полином. $\square$

NP-полные задачи существуют!
Приведём простую и очень важную теорему. На экзамене доказательство можно сформулировать в одно предложение, здесь же оно для понимания расписано максимально подробно.

Определение 14. BH = BOUNDED-HALTING: вход $x = \langle \underbrace{11...1}_{k}, Mx\rangle$, проверить, $\exists$ ли такой $y : M(x,y)$ остановится за k шагов и вернёт $true$.

Теорема 5. *BH = BOUNDED-HALTING $\in$ NPc.*

Доказательство.

1. BH $\in$ NP

   Подсказка – такой y. Алгоритм – моделирование k шагов M за $\mathcal{O}(\mathrm{poly}(k))$.

   Важно, что если бы число k было записано, используя $\log_2 k$ бит, моделирование работало бы за экспоненту от длины входа, и нельзя было бы сказать «задача лежит в NP».

2. BH $\in$ NPh. То есть нужно доказать, что любой язык из NP к ней сводится.

   $L \in$ NP: $\exists y : \ A(x,y) = 1 \Leftrightarrow x \in L$

   A – полиномиальный алгоритм $\Rightarrow \exists P(x)$, ограничивающий время работы A. Программа A всегда отрабатывает за $P(|x|)$, если ее запустить с ограничением $P(|x|)$, то ничего не поменяется.

Рассмотрим $f(x) = (\underbrace{11...1}_{P(|x|)}, A, x)$. Получили полиномиальное сведение:

$x \in L \Leftrightarrow \exists y : A(x,y) = 1 \Leftrightarrow f(x) \in$ BH.

$\square$