Лекции по теории алгоритмов
(Зюзьков В. М., мехмат)

Использование компьютеров связано с возможностью алгоритмического решения задач и эффективного вычисления функций. Между тем в математике широко используются функции, заданные неэффективными определениями. Это приводит к тому, что некоторые функции поддаются вычислению с помощью алгоритма, скажем на компьютере, как только для этого будет составлена надлежащая программа, тогда как другие функции, заданные неэффективным определением, могут требовать творческого подхода для вычисления своих значений. Столь же часты доказательства разрешимости задач, не сопровождаемые алгоритмами их решения.

В действительности класс задач, доступных классическим средствам, в некотором трудно уточняемом смысле строго шире класса задач, решаемых алгоритмически. В главе проясняется смысл этого утверждения, и излагаются некоторые математические модели вычислимости. За последние десятилетия стало ясно, что различие между быстро и долго решаемыми задачами не менее философски и практически важно, чем различие между алгоритмически разрешимыми и неразрешимыми, и теория сложности вычислений стала одной из центральных в логике (и вообще в математике). Поэтому также рассматриваются основные понятия теории сложности алгоритмов.

1. Алгоритмы и вычислимые функции

1.1. Понятие алгоритма и неформальная вычислимость

Под *алгоритмом* понимается способ преобразования представления информации. Слово *algorithm* – произошло от имени аль-Хорезми – автора известного арабского учебника по математике (от его имени произошли также слова «алгебра» и «логарифм»).

Интуитивно говоря, алгоритм – некоторое формальное предписание, действуя согласно которому можно получить решение задачи.

Алгоритмы типичным образом решают не только частные задачи, но и классы задач. Подлежащие решению частные задачи, выделяемые по мере надобности из рассматриваемого класса, определяются с помощью параметров. Параметры играют роль исходных данных для алгоритма.

Основные особенности алгоритма

- *Определенность***.** Алгоритм разбивается на отдельные шаги (этапы), каждый из которых должен быть простым и локальным.
- *Ввод.* Алгоритм имеет некоторое (быть может, равное нулю) число входных данных, т.е. величин, заданных ему до начала работы.
- *Вывод***.** Алгоритм имеет одну или несколько выходных величин, т. е. величин, имеющих вполне определенное отношение к входным данным.
- *Детерминированность.* После выполнение очередного шага алгоритма однозначно определено, что делать на следующем шаге.

Обратите внимание, что мы не требуем, чтобы алгоритм заканчивал свою работу для любых входных данных.

Примеры алгоритмов широко известны: изучаемые в школе правила сложения и умножения десятичных чисел или, скажем, алгоритмы сортировки массивов. Для алгоритмически разрешимой задачи всегда имеется много различных способов ее решения, т. е. различных алгоритмов.

Примеры «почти» алгоритмов: медицинский и кулинарный рецепты. Кстати, почему такие рецепты во многих случаях нельзя рассматривать как алгоритмы?

Данное здесь определение алгоритма не является, конечно, строгим, но оно интуитивно кажется вполне определенным. К сожалению, для решения некоторых задач не су-

ществует алгоритма. Установление таких фактов требует введение строгого понятия алгоритма.

Мы будем рассматривать алгоритмы, имеющие дело только с натуральными числами. Можно доказать, что это не является потерей общности, так как объекты другой природы можно закодировать натуральными числами. Для пользователей компьютеров такое утверждение должно быть очевидным.

Пусть $\boldsymbol{N}$ обозначает множество натуральных чисел {0, 1, 2, ...}. Объекты, которые мы будем рассматривать, будут функциями с областью определения $D_f \subseteq \boldsymbol{N}^k$ (k – целое положительное число) и с областью значений $R_f \subseteq \boldsymbol{N}$. Такие функции будем называть *k-местными частичными*. Слово «частичная» должно напомнить о том, что функция определена на подмножестве $\boldsymbol{N}^k$ (конечно, в частном случае может быть $D_f = \boldsymbol{N}^k$, тогда функция называется *всюду определенной*).

Вычислимые функции

Назовем k-местную функцию f: $\boldsymbol{N}^k \to \boldsymbol{N}$ *эффективно вычислимой* (или просто *вычислимой*), если существует алгоритм A, её вычисляющий, то есть такой алгоритм A, что:

1. Если на вход алгоритма A поступил вектор $\boldsymbol{x} = \langle x_1, x_2, ..., x_k \rangle$ из D_f, то вычисление должно закончиться после конечного числа шагов и выдать $f(\boldsymbol{x})$.
2. Если на вход алгоритма A поступил вектор $\boldsymbol{x}$, не принадлежащий области определения D_f, то алгоритм A никогда не заканчивается.

Несколько замечаний по поводу этого определения:

1. Понятие вычислимости определяется здесь для частичных функций (областью определения которых является некоторое подмножество натурального ряда). Например, нигде не определенная функция вычислима, в качестве A надо взять программу, которая всегда зацикливается.
2. Можно было бы изменить определение, сказав так: «если $f(\boldsymbol{x})$ не определено, то либо алгоритм А не останавливается, либо останавливается, но ничего не печатает на выходе». На самом деле от этого ничего бы не изменилось (вместо того, чтобы останавливаться, ничего не напечатав, алгоритм может зацикливаться).
3. Входами и выходами алгоритмов могут быть не только натуральные числа, но и двоичные строки (слова в алфавите {0, 1}), конечные последовательности слов и вообще любые, как говорят «конструктивные объекты».
4. Множество эффективно вычисляемых функций мы не отождествляем с множеством «практически вычисляемых» функций, так как не накладываем на первое множество никаких ограничений, связанных с современными вычислительными машинами. Хотя каждое входное натуральное число, должно быть конечным, тем не менее не предполагается верхняя граница размера этого числа, так, например, количество цифр числа может быть больше числа электронов во Вселенной. Точно также нет никакой верхней границы на число шагов, которые может сделать алгоритм для конкретных $\boldsymbol{x}$ из области определения.

Рассматривая теорию алгоритмов, мы можем ссылаться на программистский опыт, говоря об алгоритмах, программах, интерпретаторах и т. д. Это позволяет нам игнорировать детали построения тех или иных алгоритмов под тем предлогом, что читатель их легко восстановит (или хотя бы поверит). Но в некоторых случаях этого недостаточно, поэтому мы собираемся дать строгое определение нового множества функций, которое в некотором смысле будет совпадать с множеством вычислимых функций. Мы дадим две различные формализации понятия вычислимой функции.

1.2. Частично-рекурсивные функции

Определения

Этот подход к формализации понятия алгоритма принадлежит Гёделю и Клини (1936).

Стивен Коул Клини

Основная идея Гёделя состояла в том, чтобы получить все вычислимые функции из существенно ограниченного множества базисных функций с помощью простейших алгоритмических средств.

Множество исходных функций таково:

- постоянная функция $0(\boldsymbol{x}) = 0$;
- одноместная функция следования $s(x) = x+1$;
- функция проекции pr_i, $1 \le i \le k$, $pr_i(\boldsymbol{x}) = x_i$.

Нетривиальные вычислительные функции можно получать с помощью композиции (суперпозиции) уже имеющихся функций. Этот способ явно алгоритмический.

- Оператор суперпозиции. Говорят, что k-местная функция $f(\boldsymbol{x})$ получена с помощью суперпозиции из m-местной функции $\varphi(y_1, y_2, ..., y_m)$ и k-местных функций $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_m(\boldsymbol{x})$, если $f(\boldsymbol{x}) = \varphi(g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), ..., g_m(\boldsymbol{x}))$.

Второй (несколько более сложный) способ действует так.

- Примитивная рекурсия. При $n \ge 0$ из n-местной функции f и $(n+2)$-местной функции g строится $(n+1)$-местная функция h по следующей схеме:
  $h(\boldsymbol{x}, 0) = f(\boldsymbol{x})$,
  $h(\boldsymbol{x}, y+1) = g(\boldsymbol{x}, y, h(\boldsymbol{x}, y))$.

При $n=0$ получаем (a – константа):
$h(0) = a$;
$h(y+1) = g(y, h(y))$.

Два упомянутых способа позволяют задать только всюду определенные функции. Частично-определенные функции порождаются с помощью третьего гёделева механизма.

- Оператор *минимизации*. Эта операция ставит в соответствие частичной функции f: $\boldsymbol{N}^{k+1} \to \boldsymbol{N}$ частичную функцию h: $\boldsymbol{N}^k \to \boldsymbol{N}$, которая определяется так ($\boldsymbol{x} = \langle x_1,..., x_k\rangle$):
  1) область определения $D_h = \{\boldsymbol{x} \mid$ существует $x_{k+1} \ge 0$, $f(\boldsymbol{x}, x_{k+1}) = 0$ и $\langle \boldsymbol{x}, y\rangle \in D_f$ для всех $y \le x_{k+1}\}$;
  2) $h(\boldsymbol{x})$ = наименьшее значение y, при котором $f(\boldsymbol{x}, y) = 0$.

Оператор минимизации обозначается так $h(\boldsymbol{x}) = \mu y[f(\boldsymbol{x},y) = 0]$. Очевидно, что даже если f всюду определено, но нигде не обращается в 0, то $\mu y[f(\boldsymbol{x}, y) = 0]$ нигде не определено. Естественный путь вычисления $h(\boldsymbol{x})$ состоит в подсчете значения $f(\boldsymbol{x}, y)$ последовательно для $y = 0,1, 2,...$ до тех пор, пока не найдется y, обращающее $f(\boldsymbol{x}, y)$ в 0. Этот алгоритм не остановится, если $f(\boldsymbol{x}, y)$ нигде не обращается в 0.

Все функции, которые можно получить из базисных функций за конечное число шагов только с помощью трех указанных механизмов, называются *частично-рекурсивными*. Если функция получается всюду определенная, то тогда она называется *общерекурсивной*. Если функция получена без механизма минимизации, то в этом случае она называется *примитивно-рекурсивной*.

Любую примитивно-рекурсивную функцию можно вычислить с помощью цикла в форме *for*, так как верхнюю границу для числа повторений можно указать заранее. Оператор минимизации позволяет описать функции, которые нельзя вычислить за заранее ограниченное число итераций, для вычисления их значений требуется цикл в форме *while*.

Можно легко показать [16, с.136], что введение фиктивных переменных, а также перестановка и отождествление переменных не выводят за пределы класса примитивно-

рекурсивных функций и класса частично-рекурсивных функций. Это проще всего объяснить на примерах.

Введение фиктивных переменных. Если $g(x_1, x_3)$ – примитивно-рекурсивная функция и $f(x_1, x_2, x_3) = g(x_1, x_3)$, то $f(x_1, x_2, x_3)$ – примитивно-рекурсивная функция.

Перестановка переменных. Если $g(x_1, x_2)$ - примитивно-рекурсивная функция и $f(x_2, x_1) = g(x_1, x_2)$, то f есть также примитивно-рекурсивная функция.

Отождествление переменных. Если $g(x_1, x_2, x_3)$ – примитивно-рекурсивная функция и $f(x_1, x_2) = g(x_1, x_2, x_1)$, то $f(x_1, x_2)$ есть также примитивно-рекурсивная функция.

Примеры рекурсивности

Рассмотрим примеры частично-рекурсивных функций. Все эти примеры и много других можно найти в [14, 16].

- Сложение двух чисел
  $sum: <x,y> \to x+y$.
  Эта функция является общерекурсивной в силу примитивной рекурсии
  $sum(x,0) = pr_1(x) = x$,
  $sum(x,y+1) = s(sum(x,y)) = sum(x,y)+1$.

- Умножение двух чисел
  $prod: <x,y> \to x\ y$.
  Используем примитивную рекурсию
  $prod(x,0) = 0(x) = 0$,
  $prod(x,y+1) = sum(prod(x,y),x)$.

- Усеченное вычитание 1
  $\delta(x) = x-1$, если $x>0$,
  $\delta(0) = 0$.
  Эта функция примитивно-рекурсивна, действительно,
  $\delta(0) = 0 = 0(x)$,
  $\delta(y+1) = y = pr_2(<x,y>)$.

- Усеченная разность
  $x \div y = x - y$, если $x \geq y$,
  $x \div y = 0$, если $x < y$.
  Эта функция примитивно-рекурсивна, действительно,
  $x \div 0 = x$,
  $x \div (y+1) = \delta(x \div y)$.

- Модуль разности
  $|x-y| = x-y$, если $x \geq y$,
  $|x-y| = y-x$, если $x<y$.
  Эта функция примитивно-рекурсивна в силу суперпозиции
  $|x-y| = (x \div y)+(y \div x)$.

- Факториал
  Действительно,
  $0! = 1$,
  $(y+1)! = prod(y!,y+1)$.

- $\min(x,y)$ – наименьшее из чисел x и y
  В силу суперпозиции: $\min(x,y) = x \div (x \div y)$.

- Знак числа
  $sg(x) = 0$, если $x = 0$,
  $sg(x) = 1$, если $x>1$.
  В силу рекурсии
  $sg(0) = 0$,
  $sg(y+1) = 1$.

- $rm(x, y)$ – остаток от деления y на x, если $x\neq0$, и y, если $x=0$.
  В силу рекурсии и суперпозиции
  $rm(x,0) = 0$,
  $rm(x,y+1) = prod(s(rm(x,y)),sg(|x–s(rm(x,y))|))$.

Используя функции, для которых уже установлено, что они являются частично-рекурсивными, мы получаем все новые и новые частично-рекурсивные функции. Существуют критерии, которые позволяют установить частичную рекурсивность сразу для обширных классов функций (см., например, [16, с. 135–151]).

Используя минимизацию (μ-оператор) можно получать частично-определенные функции из всюду определенных функций. Например, полагая $f(x,y)$ есть частично-рекурсивная функция $|x–y^2|$, мы обнаруживаем, что $g(x) = \mu y[f(x,y)=0]$ – не всюду определенная функция

$g(x) = \sqrt{x}$, если x есть точный квадрат и неопределенна в противном случае.

Таким образом, тривиально используя μ-оператор вместе с суперпозицией и рекурсией, можно построить больше функций, исходя из основных, чем только с помощью суперпозиции и рекурсии (так как эти операции порождают из всюду определенных функций – всюду определенные). Существуют, однако, и общерекурсивные (всюду определенные) функции, для построения которых нельзя обойтись без минимизации.

Приведем пример функции, не являющейся примитивно рекурсивной, хотя и вычислимой в интуитивном смысле.

Определим последовательность одноместных функций F_n: $\boldsymbol{N}\to\boldsymbol{N}$, $n\in\boldsymbol{N}$, следующим образом:

$$F_0(x) = x+1,$$
$$F_{n+1}(x) = \underbrace{F_n(F_n(\ldots F_n}_{x+1 \text{ раз}}(1)\ldots))$$

Поэтому $F_1(x) = x+2$, $F_2(x) = 2x+3 \approx 2x$, $F_3(x) \approx 2^x$,

$F_4(x) \approx 2^{2^{\cdot^{\cdot^{2}}}}$ (башня из $x+1$ двойки) и т. д.

Имеем следующие свойства:

1) для каждого n функция $\lambda x.\ F_n(x)$ является примитивно рекурсивной;
2) $F_n(x) > 0$,
3) $F_n(x+1) > F_n(x)$,
4) $F_n(x) > x$,
5) $F_{n+1}(x) \geq F_n(x+1)$,
6) для каждой k–местной примитивно рекурсивной функции $\lambda x_1x_2\ldots x_k.f(x_1,x_2,\ldots,x_k)$ существует такое n, что F_n мажорирует f, т.е. $f(x_1,x_2,\ldots,x_k) \leq F_n(\max(x_1,x_2,\ldots,x_k))$ для всех $x_1,x_2,\ldots,x_k$,
7) функция $\lambda nx.\ F_n(x)$ не является примитивно рекурсивной [7, с. 53] (эта функция известно как *функция Аккермана*).

Функцию Аккермана можно определить и в традиционной записи:
$f(0, y) = y+1$,
$f(x+1,0) = f(x,1)$,
$f(x+1,y+1) = f(x, f(x+1,y))$

Позднее мы приведем доводы в пользу правдоподобности того, что понятие частично-рекурсивной функции есть точный математический эквивалент интуитивной идеи (эффективно) вычислимой функции.

1.3. Машины Тьюринга

Алан Тьюринг

Рассмотрим еще один способ определения вычислимых функций, следуя в изложении [22, с. 12–14]. Формулировка, выраженная в терминах воображаемой вычислительной машины, была дана английским математиком Аланом Тьюрингом в 1936 г. Главная трудность при нахождении этого определения была в том, что Тьюринг искал его до создания реальных цифровых вычислительных машин. Познание шло от абстрактного к конкретному: фон Нейман был знаком с работой Тьюринга, и сам Тьюринг позднее сыграл вдохновляющую роль в развитии вычислительных машин.

На неформальном уровне мы можем описывать машину Тьюринга как некий черный ящик с лентой. Лента разбита на ячейки и каждая ячейка может содержать пустой символ 0, либо непустой символ 1. Лента потенциально бесконечна в обе стороны в том смысле, что мы никогда не придем к ее концу, но в любое время лишь конечное число ячеек может быть непустым. В начале лента содержит числа входа, в конце – число–выход. В промежуточное время лента используется как пространство памяти для вычисления.

Если мы откроем черный ящик, то обнаружим, что он устроен очень просто. В любой момент времени он может обозревать лишь одну ячейку памяти. Устройство содержит конечный список инструкций (или *состояний*) $q_0, q_1,..., q_n$. Каждая инструкция может указать два возможных направлений действий; одного нужно придерживаться, если на обозреваемой ячейке ленты находится 0, а другого, – если там находится 1. В любом случае следующее действие может состоять из таких трех типов элементарных шагов:

- символ (возможно, такой же, как старый) пишется на обозреваемой ячейке ленты, при этом предыдущий символ стирается;
- лента сдвигается на одну ячейку влево или вправо;
- указывается следующая инструкция.

Таким образом, список инструкций определяет некоторую функцию перехода, которая по данной инструкции и обозреваемому символу указывает три компоненты того, что нужно делать. Мы можем формализовать эти идеи, взяв в качестве машины Тьюринга эту функцию перехода.

Машина Тьюринга

Машина Тьюринга – это функция M такая, что для некоторого натурального числа n, область определения этой функции есть подмножество множества $\{0, 1,..., n\}\times\{0, 1\}$, а область значений есть подмножество множества $\{0, 1\}\times\{Л, П\}\times\{0, 1, ..., n\}$.

Например, пусть $M(3,1) = <0, Л, 2>$. Подразумеваемый смысл этого состоит в том, что как только машина дойдет до инструкции q_3, а на обозреваемой ячейке написан сим-

вол 1, она должна стереть 1 (оставляя на ячейке 0), передвинуть ленту так, чтобы обозреваемой ячейкой стала левая соседняя ячейка от той, которая обозревалась, и перейти к следующей инструкции q_2. Если $M(3,1)$ не определено, тогда как только машина дойдет до инструкции q_3, а на обозреваемой ячейки написан символ 1, то машина останавливается. (Это единственный путь остановки вычисления.)

Такая подразумеваемая интерпретация не включена в формальное определение машины Тьюринга, но она мотивирует и подсказывает формулировки всех следующих определений. В частности, можно определить, что означает для машины M передвижение (за один шаг) от одной конфигурации до другой. Нам не нужно здесь давать формальных определений, так как они являются простыми переводами наших неформальных идей.

Входные и выходные данные – это строчки из 1, разделенные 0. Пусть $<n>$ будет строчкой из 1 длины n+1. Тогда

$$<n_1> 0 <n_2> 0 \ldots 0 <n_k>$$

получено комбинацией k строчек из 1, каждая отделена от другой 0.

Наконец, мы можем определить вычислимость.

Вычислимость по Тьюрингу

Пусть $D_f \subseteq \boldsymbol{N}^k$ – область определения k-местной функции f: $D_f \to \boldsymbol{N}$. Функция f называется *вычислимой по Тьюрингу*, если существует машина Тьюринга M такая, что как только M начинает с инструкции q_0, обозревая самой левый символ строки

$$<n_1> 0 <n_2> 0 \ldots 0 <n_k>,$$

(вся остальная часть ленты пуста), тогда:

- если $f(n_1, n_2,..., n_k)$ определено, то M, в конце концов, остановится, обозревая самый левый символ строки $<f(n_1, n_2,..., n_k)>$, при этом часть, находящаяся справа от этой строчки, пустая;
- если $f(n_1, n_2,..., n_k)$ не определено, то M никогда не останавливается.

Заметим, что имеется бесконечное множество машин Тьюринга, для каждой вычислимой функции своя. Более того, для любой вычислимой функции имеется бесконечное множество машин Тьюринга, вычисляющих эту функцию.

Пример. Построим машину Тьюринга, вычисляющую сумму n_1+n_2.
Зададим функцию M следующим образом:

$M(0, 1) = <1, П, 0>;$
$M(0, 0) = <1, П, 1>;$
$M(1, 1) = <1, П, 1>;$
$M(1, 0) = <0, Л, 2>;$
$M(2, 1) = <0, Л, 3>;$
$M(3, 1) = <0, Л, 4>;$
$M(4, 1) = <1, Л. 4>;$
$M(4, 0) = <0, П, 5>.$

Посмотрим, как происходит сложение 1+1. В текущей строке символов обозреваемый символ выделен.

номер инструкции	текущая строка символов	комментарий
0	0***1***10110	прохождение через первое слагаемое
0	01***1***0110	
0	011***0***110	заполнение промежутка
1	0111***1***10	прохождение через второе слагаемое
1	01111***1***0	
1	011111***0***	конец второго слагаемого
2	01111***1***0	стирание 1

3	0111***1***00	стирание второй 1
4	011***1***000	движение назад
4	01***1***1000	
4	0***1***11000	
4	***0***111000	остановка
5	0***1***11000	

Мы должны заметить, что многие детали нашего определения машины Тьюринга до некоторой степени произвольны. Если бы было более одной ленты, то класс вычислимых функций остался бы неизменным, хотя некоторые функции могли бы быть вычислены более быстро. Аналогично, мы могли бы допускать больше символов, чем 0 и 1, или же у нас могла бы быть лента бесконечная только в одну сторону от начальной точки вместо имеющейся бесконечной в обоих направлениях. Ни одно из этих изменений не затрагивает класса вычислимых функций. Что действительно существенно в этом определении – это разрешение произвольно большого количества материала для запоминающего устройства и произвольно длинных вычислений.

1.4. Тезис Чёрча

За последние 60 лет было предложено много различных математических уточнений интуитивного понятия алгоритма. Два из этих подхода мы разобрали. Перечислим некоторые другие альтернативные способы, которые предлагались следующими авторами:

- Гёдель–Эбран–Клини. Общерекурсивные функции, определенные с помощью исчисления рекурсивных уравнений [16, с. 261–278].
- Чёрч. λ-определимые функции, определенные в рамках λ-исчисления [3].
- Пост. Функции, определяемые каноническими дедуктивными системами [7, с. 66–72].
- Марков. Функции, задаваемые некоторыми алгоритмами (известные под названием нормальными алгоритмами) над конечным алфавитом [16, с.228–250].
- Шепердсон–Стерджис. МНР–вычислимые функции [7].

Между этими подходами (в том числе, и два выше рассмотренных) имеются большие различия; каждый из них имеет свои преимущества для соответствующего описания вычислимости. Следующий замечательный результат получен усилиями многих исследователей.

Алонсо Чёрч

Теорема 1 (Основной результат). [7, с. 57]. Каждое из вышеупомянутых уточнений эффективной вычислимости приводит к одному и тому же классу вычислимых функций.

Вопрос: насколько хорошо неформальное и интуитивное понятие эффективно вычислимой функции отражено в различных формальных описаниях?

Чёрч, Тьюринг и Марков каждый в соответствии со своим подходом выдвинули утверждение (тезис) о том, что класс определенных ими функций совпадает с неформально определенным классом вычислимых функций. В силу основного результата все эти утверждения логически эквивалентны. Название *тезис Чёрча* теперь применяется к этим и аналогичным им утверждениям.

Тезис Чёрча

Интуитивно и неформально определенный класс эффективно вычислимых функций совпадает с классом частично-рекурсивных функций.

Сразу же заметим, что этот тезис не является теоремой, а скорее утверждение, которое принимается на веру, причем вера подкрепляется следующими аргументами [7, с. 75–76].

- Фундаментальный результат: многие независимые инварианты уточнения интуитивного понятия вычислимости привели к одному и тому же классу функций.
- Обширное семейство эффективно вычислимых функций принадлежит этому классу. Конкретные функции, рассмотренные в 6.1.2, образуют исходную часть этого семейства, которую можно расширять до бесконечности методами из 6.1.2 или более мощными и сложными методами.
- Никто еще не нашел функцию, которую можно было признать вычислимой в неформальном смысле, но которую нельзя было бы построить, используя один из формальных методов.

Приведем парадокс, показывающий насколько мы подходим к опасной границе между непротиворечивым и противоречивым в обсуждении вычислимых функций [8, с. 184].

Легко доказать, что множество всюду определенных вычислимых функций f: $\boldsymbol{N}\to\boldsymbol{N}$ является перечислимым, т. е. их можно перенумеровать в виде последовательности $f_1, f_2, f_3, \ldots$.

Определим теперь новую функцию g формулой

$$g(n) = f_n(n)+1.$$

Она не входит в нашу последовательность, поскольку при n=1 она отличается от f_1, при n=2 – от f_2 и т. д. Следовательно, она не вычислима.

С другой стороны, ясно, что она вычислима, так как $f_n(n)$ вычислима, а прибавив 1 к $f_n(n)$, мы получим $g(n)$.

Этот парадокс можно объяснить. На самом деле мы здесь используем две формальные системы. В рамках одной системы (скажем, элементарной арифметики *EA*), мы описываем вычислимые функции f, а то, что g является вычислимой функцией мы получаем в рамках другой формальной системы, в которой уже используется возможность упорядочить f. Вторая формальная система является надсистемой или метасистемой относительно первой.

1.5. Теорема о рекурсии

После изучения эффективно вычислимых операций на множестве *чисел*, естественно рассмотреть вопрос о том, существует ли понятие подобного рода и для операций над *функциями*. Существенная разница между функциями и числами (если их рассматривать как множества основных объектов) состоит в том, что первые являются, как правило, бесконечными объектами, а вторые – конечными.

Для простоты изложения мы будем рассматривать частичные функции только одного аргумента, но все определения и полученные результаты обобщаются и на случай функций от нескольких аргументов.

Обозначим через $\boldsymbol{F}$ класс всех частичных функций из $\boldsymbol{N}$ в $\boldsymbol{N}$. Словом *оператор* мы будем обозначать функцию Ф: $\boldsymbol{F}\to\boldsymbol{F}$, причем будем рассматривать лишь такие операторы Ф, область определения которых совпадает со всем классом $\boldsymbol{F}$.

Основная проблема при попытке дать определение понятию вычислимого (или эффективного) оператора Ф: $\boldsymbol{F}\to\boldsymbol{F}$ состоит в том, что как вводимая функция f, так и выводимая функция Ф(f) являются, скорее всего, бесконечными объектами и, следовательно, не могут быть заданы за конечное время. В то же время, согласно нашему интуитивному представлению об алгоритмических процессах, мы должны в некотором смысле уметь проводить их за конечное время.

Чтобы разобраться, как можно преодолеть эту трудность, рассмотрим следующие операторы из $\boldsymbol{F}$ в $\boldsymbol{F}$:

1) $\Phi_1(f) = 2f$,
2) $\Phi_2(f) = g$, где $g(x) = \sum_{y \le x} f(y)$.

Эти операторы, несомненно, очень естественны и записаны в явном виде. На интуитивном уровне их вполне можно было бы считать вычислимыми. Попытаемся сейчас разобраться, почему. Пусть $f \in \boldsymbol{F}$. Рассмотрим случай $g_1 = \Phi_1(f)$. Заметим, что любое конкретное значение $g_1(x)$ (если оно определено) может быть вычислено за конечное время по одному единственному значению $f(x)$ функции f. В случае $g_2 = \Phi_2(f)$ для вычисления значения $g_2(x)$ (если оно определено) необходимо знать *конечное* число значений $f(0), f(1), \ldots, f(x)$. Тем самым в обоих случаях любое определенное значение выводимой функции ($\Phi_1(f)$ или $\Phi_2(f)$) может быть эффективно вычислено за *конечное* время при использовании лишь *конечной* информации о вводимой функции f. В этом и состоит суть приведенного ниже определения рекурсивного оператора.

Для того чтобы дать определению точный смысл, условимся о некоторых технических деталях. Если f, g – функции, то будем говорить, что g *продолжает* f, если $D_f \subseteq D_g$ и $f(x) = g(x)$ для всех $x \in D_f$. Это отношение функций f, g записывается как $f \subseteq g$.

Под «конечной частью» функции f понимается некоторая конечная функция θ, которая продолжается до f. (Функция θ называется *конечной*, если её область определения есть конечное множество.)

Из приведенных выше соображений ясно, что в определении вычислимого оператора должны участвовать эффективные вычисления с конечными функциями. Придадим этому точный смысл, закодировав каждую конечную функцию θ натуральным числом $g(\theta)$. Удобный для наших целей кодирующий алгоритм определяется следующим образом.

Пусть p_k ($k = 0, 1, 2, \ldots$) обозначает последовательность простых чисел 2, 3, 5, 7, 11, … Для данного $\theta \in \boldsymbol{F}$ положим

$$g(\theta) = \prod_k p_k^{\theta(k)+1}, \text{ где } k \in D_\theta \neq \varnothing,$$

$$g(\theta) = 0, \text{ если } D_\theta = \varnothing.$$

Имеется простой алгоритм, с помощью которого можно для любого числа z определить, верно ли $z = g(\theta)$ для некоторой конечной функции θ, и, если ответ на вопрос положителен, выяснить, принадлежит ли некоторое заданное x множеству D_θ, и если да, то вычислить значение $\theta(x)$.

Сформулируем теперь наше определение.

Рекурсивный оператор

Оператор Φ: $\boldsymbol{F} \to \boldsymbol{F}$ называется рекурсивным (или вычислимым), если существует вычислимая функция $\phi(z, x)$, такая, что для всех $f \in \boldsymbol{F}$ и $x, y \in \boldsymbol{N}$

$\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\phi(g(\theta), x) = y$.

Число $g(\theta)$ – кодовый номер функции θ.

Пример. Оператор $\Phi(f) = 2f$ является рекурсивным оператором. Чтобы убедиться в этом, положим

$$\phi(z, x) = \begin{cases} 2\theta(x), \text{ если } z = g(\theta) \text{ и } x \in D_\theta, \\ \\ \text{не определена в остальных случаях.} \end{cases}$$

В силу тезиса Чёрча функция ϕ вычислима. Далее, для любых f, x, y мы имеем

$\Phi(f)(x) = y \Leftrightarrow x \in D_f$ и $y = 2f(x) \Leftrightarrow$ существует $\theta \subseteq f$, для которой $x \in D_\theta$ и $y = 2\theta(x)$
$\Leftrightarrow$ существует $\theta \subseteq f$, для которой $\phi(g(\theta), x) = y$.

Отсюда Φ – рекурсивный оператор.

Важное свойство рекурсивных операторов состоит в том, что они *непрерывны* и *монотонны* в следующем смысле.

Непрерывный оператор

Оператор Φ: $\boldsymbol{F} \to \boldsymbol{F}$ называется непрерывным, если для любой функции $f \in \boldsymbol{F}$ и любых x, y: $\Phi(f)(x) = y$ тогда и только тогда, когда существует конечная функция $\theta \subseteq f$, такая, что $\Phi(\theta)(x) = y$.

Монотонный оператор

Оператор Φ: $\boldsymbol{F} \to \boldsymbol{F}$ называется монотонным, если для любых функций f, $h \in \boldsymbol{F}$, таких, что $f \subseteq h$, выполняется условие $\Phi(f) \subseteq \Phi(h)$.

Теорема 2. Любой рекурсивный оператор является непрерывным и монотонным. Доказательство см., например, [7, с. 194–195].

Неформально использование слова «непрерывный» для описания свойства оператора можно объяснить следующим образом. Можно показать, что если функция h находится «близко» от функции f (в том смысле, что они совпадают на конечном множестве D_θ, где $\theta \subseteq f$), то для непрерывного оператора Φ функция $\Phi(h)$ расположена «близко» от функции $\Phi(f)$.

Первая теорема о рекурсии Клини представляет собой теорему о неподвижной точке для рекурсивных операторов, и ее зачастую называют также теоремой о неподвижной точке (из теории рекурсии).

Теорема 3 (теорема о неподвижной точке). Пусть Φ: $\boldsymbol{F} \to \boldsymbol{F}$ – рекурсивный оператор. Тогда существует вычислимая функция f, которая является наименьшей неподвижной точкой для Φ, т.е

1) $\Phi(f) = f$,
2) если $\Phi(h) = h$, то $f \subseteq h$.

Отсюда если функция f всюду определена, то она является единственной неподвижной точкой Φ.

Доказательство см., например, [7, с. 203–204].

Теорема вызывает недоумение: пусть оператор Φ задан так: функция $f(x)$ отображается в функцию $f(x)+1$. Это отображение конечно можно задать с помощью алгоритма (т.е. оператор рекурсивный), но разве у этого оператора есть неподвижная точка? Да, это функция, которая нигде не определена.

Наш программистский опыт говорит о том, что определения, даваемые в терминах примитивной рекурсии, имеют смысл. В случае более сложных рекурсивных определений (см., например, функцию Аккермана) это не так очевидно – вполне можно представить себе, что функций, удовлетворяющих тому или иному определению, вообще не существует. Именно здесь оказывается полезной теорема 3. Рекурсивные определения весьма общего типа могут быть представлены уравнением вида

$$f = \Phi(f),$$

где Ф – некоторый рекурсивный оператор. Теорема о неподвижной точке показывает, что такое определение имеет смысл: всегда существует даже вычислимая функция, которая ему удовлетворяет. Поскольку в математике требуется, чтобы определения описывали различные понятия однозначно, можно сказать, что данное рекурсивное определение определяет наименьшую неподвижную точку оператора Ф. Таким образом, согласно теореме 3, класс вычислимых функций замкнут относительно рекурсивного определения очень общего типа. Таким образом, понятно, почему эту теорему называют теоремой о рекурсии (точнее первой теоремой о рекурсии, есть также вторая теорема о рекурсии).

1.6. Куины

«Куином» называется программа, которая на своем выходе генерирует собственный исходный текст. Этот термин предложил Д. Хофштадтер в честь американского философа и математика *Willard van Orman Quine* (1908 – 2000).

Докажем, что куины существуют для любого универсального языка программирования. Действительно, теорему Клини о рекурсии (или о неподвижной точке) можно сформулировать в следующем виде: «Нельзя найти алгоритма, преобразующего программы, который бы по каждой программе давал бы другую (не эквивалентную ей)».

Пусть теперь дано преобразование программ:

$$F: p \rightarrow \{\text{программа, которая на любом входе печатает } p\}.$$

В силу теоремы Клини существует программа p такая, что $F(p) = p$, т. е. существует программа, печатающая (на любом входе) свой собственный текст.

Написать куин – хорошая и популярная задача для любителей программирования. Неформальную инструкцию на русском языке можно представить следующим образом:

напечатать два раза, второй раз в кавычках, такой текст: «напечатать два раза, второй раз в кавычках, такой текст:»

Сравнительно легко можно написать куин на функциональном языке, таком как лисп. Написание куинов на процедурных языках требует некоторых хитростей. Вот пример на Паскале:

```
var a:array[1..7] of string;
    i:integer;
begin
a[1]:='var a:array[1..7] of string;';
a[2]:='    i:integer;';
a[3]:='begin';
a[4]:='for i:=1 to 3 do writeln(a[i]);';
a[5]:='for i:=1 to 7 do writeln(#97#91,i,#93#58#61#39,a[i],#39#59);';
a[6]:='for i:=4 to 7 do writeln(a[i]);';
a[7]:='end.';
for i:=1 to 3 do writeln(a[i]);
for i:=1 to 7 do writeln(#97#91,i,#93#58#61#39,a[i],#39#59);
for i:=4 to 7 do writeln(a[i]);
end.
```

Чтобы понять эту программу полезно иметь в виду соответствие между символами и кодами:

a	[	]	:	=	;	'
97	91	93	58	61	59	39

Предложенный текст куина на Паскале имеет существенный недостаток: он использует значения кодов ASCII – но коды не являются частью самого языка Паскаль. Следующий куин исправляет этот недостаток (автор Dan Hoey: hoey@aic.nrl.navy.mil):

```
program s;const bbb='program s;const bbb';a='a';b='b';bb=');writeln(';
aa='''';ab='=''';ba=''';';
aaa='begin writeln(bbb,ab,bbb,ba,a,ab,a,ba,b,ab,b,ba,b,b,ab,bb,ba';
aba='a,a,ab,aa,aa,ba,a,b,ab,ab,aa,ba,b,a,ab,aa,ba,ba';
abb='a,a,a,ab,aaa,ba);writeln(a,b,a,ab,aba,ba);writeln(a,b,b,ab,abb,ba';
baa='b,a,a,ab,baa,ba);writeln(b,a,b,ab,bab,ba);writeln(aaa,bb';
bab='aba,bb);writeln(abb);writeln(bb,baa);writeln(bb,bab)end.';
begin writeln(bbb,ab,bbb,ba,a,ab,a,ba,b,ab,b,ba,b,b,ab,bb,ba);writeln(
a,a,ab,aa,aa,ba,a,b,ab,ab,aa,ba,b,a,ab,aa,ba,ba);writeln(
a,a,a,ab,aaa,ba);writeln(a,b,a,ab,aba,ba);writeln(a,b,b,ab,abb,ba
);writeln(b,a,a,ab,baa,ba);writeln(b,a,b,ab,bab,ba);writeln(aaa,bb
);writeln(aba,bb);writeln(abb);writeln(bb,baa);writeln(bb,bab)end.
```

Как видим, вопреки общепринятому мнению, куин на процедурном языке не обязан содержать итерацию или рекурсию.

1.7. Некоторые алгоритмически неразрешимые проблемы

> ... Найти задачу – не меньшая радость, чем отыскать решение.
>
> Томас де Куинси

> – Это же проблема Бен Бецалеля. Калиостро же доказал, что она не имеет решения... Как же искать решения, когда его нет? Бессмыслица какая-то...
> – Бессмыслица – искать решение, если оно и так есть. Речь идет о том, как поступать с задачей, которая решения не имеет.
>
> А. и Б. Стругацкие.
> Понедельник начинается в субботу

Решение вопроса о том, обладают ли натуральные числа данным свойством, являются часто встречающейся задачей математики. Поскольку свойства чисел можно выразить с помощью подходящего предиката, то решение задачи сводится к выяснению того, является ли данный предикат разрешимым или нет (т.е., является ли характеристическая функция предиката вычислимой или нет, см. 2.2). Задачи с произвольными универсумами во многих случаях можно переформулировать в виде задач с натуральными числами, если использовать подходящее кодирование.

В контексте разрешимости предикаты часто называются *проблемами.*

Имея точное определение вычислимости, удалось доказать, что некоторые проблемы неразрешимы.

- Теорема Черча о неразрешимости логики предикатов. Не существует алгоритма, который для любой формулы логики предикатов устанавливает, общезначима она или нет.
- Проблема остановки неразрешима [7, с. 108]. Не существует никакого общего алгоритма, позволяющего установить, остановится ли некоторая конкретная программа (на любом языке программирования), запущенная после введения в неё некоторого конкретного набора данных. Смысл этого утверждения для теоретического программиро-

вания очевиден: не существует совершенно общего метода проверки программ на наличие в них бесконечных циклов.

- Не существует никакого общего алгоритма, позволяющего установить, вычисляет ли некоторая конкретная программа (на любом языке программирования) постоянную нулевую функцию [7, с. 110]. То же самое справедливо и для любой другой конкретной вычислимой функции. И как следствие, можно утверждать, что вопрос о том, вычисляют ли две данные программы одну и ту же одноместную функцию, также неразрешим. Тем самым, мы получаем, что в области тестирования компьютерных программ, мы имеем принципиальные ограничения.

Диофантовы уравнения [7, с. 114]. Современная математика вообще изобилует разрешимыми и неразрешимыми проблемами. Одна из проблем связана с диофантовыми уравнениями.

Пусть $p(x_1, x_2, ..., x_n)$ – многочлен от переменных $x_1, x_2, ..., x_n$ с целыми коэффициентами. Тогда уравнение

$p(x_1, x_2, ..., x_n) = 0,$

для которого мы ищем только целые решения, называется диофантовым уравнением. Диофантовы уравнения не обязательно имеют решения. Например, не имеет решения уравнение $x^2–2 = 0$.

Десятая проблема Гильберта, сформулированная в 1900 году, состоит в том, чтобы установить, существует ли алгоритм, с помощью которого можно было бы проверить, имеет ли данное диофантово уравнение решение. В 1970 году советский математик Ю. Матиясевич доказал, что такого алгоритма не существует. Доступное доказательство этого можно найти в [14].

Отметим также, что доказательство теоремы Гёделя о неполноте использует теорию алгоритмов. Элементарное доказательство приведено в [23]. Занимательному изложению вопросов вычислимости, вплоть до получения доказательства теоремы Гёделя, посвящена книга Р. Смаллиана [20].

2. Элементарная арифметика и неполнота

2.1. Элементарная арифметика

Попытаемся получить независимую формализацию средств математического рассуждения, использующих только понятие натурального числа (не упоминая ни действительных чисел, ни – тем более – произвольных множеств Кантора). Это самая надежная часть арсенала математики, не скомпрометировавшая себя парадоксами. Естественно назвать ее **элементарной арифметикой** (по аналогии с термином «элементарная теория чисел» – в отличие от аналитической теории чисел). Аксиоматическое построение этой дисциплины известно как *система аксиом Пеано*.

Элементарная арифметика – это теория первого порядка *EA*, в языке которой имеются:
1. Предметная константа **0**.
2. Двухместные функторы + и ×, одноместный функтор *S*.
3. Двухместный предикат =.

В качестве универсума при стандартной интерпретации элементарной арифметики рассматривается множество натуральных чисел, причем предметной константе **0** соответствует число ноль. Термы интерпретируются как натуральные числа. Значение функции $S(t)$ для любого терма t, обозначающего натуральное число, интерпретируется как (непосредственно) следующее число за t. Термы, имеющие вид $S(\mathbf{0})$, $S(S(\mathbf{0}))$, $S(S(S(\mathbf{0})))$ и т. д. интерпретируются как натуральные числа 1, 2, 3 и т.д. Знаки + и × интерпретируются как обозначения операций сложения и умножения.

Теория *EA* использует только предикатный символ равенства «=» (понимаемый как равенство натуральных чисел). Если t_1, t_2 – термы *EA*, то $t_1=t_2$ – элементарная формула *EA*. Из элементарных формул с помощью логических связок и кванторов строятся более сложные формулы *EA*, причем $\exists x$ мы понимаем как «существует натуральное число», а $\forall x$ – как «для всех натуральных чисел».

Средствами языка *EA* легко записываются простейшие утверждения о свойствах натуральных чисел, например:

«$x < y$» $\Leftrightarrow \exists z(\neg(z=\mathbf{0})\& \ y=x+z)$;

«x – четное число» $\Leftrightarrow \exists y(x=y+y)$;

«x – простое число» $\Leftrightarrow$ «$S(\mathbf{0})<x$» & $\neg\exists y\exists z$(«$y<x$»& «$z<x$» & $x=y\times z$);

«существует бесконечно много простых чисел» $\Leftrightarrow \forall x\exists y$(«$x<y$» & «$y$ – простое число»).

Все, что мы до сих пор говорили о нашем «понимании» языка элементарной арифметики, является нашим личным делом и к определению теории *E*A не относится. В этой теории о свойствах введенных нами символов известно только то, что сформулировано в аксиомах. Часть этих свойств уже содержится в логических аксиомах и правилах вывода, которые безоговорочно принимаются в *EA*. Для определения наиболее существенных свойств мы вводим собственные аксиомы *EA*:

Собственные аксиомы (схемы аксиом) элементарной арифметики:

P_1: ($P(\mathbf{0})$ & $\forall x(P(x)\supset P(S(x)))\supset\forall z P(z)$ (принцип математической индукции, P – произвольная формула),

P_2: $S(t_1) = S(t_2) \supset t_1 = t_2$,

P_3: $\neg(\ S(t)=\mathbf{0})$,

P_4: $t_1 = t_2 \supset (t_1 = t_3 \supset t_2 = t_3)$,

P_5: $t_1 = t_2 \supset S(t_1) = S(t_2)$,

P_6: $t+\mathbf{0} = t$,

P_7: $t_1 + S(t_2) = S(t_1+ t_2)$,

P_8: $\mathbf{0}\times t = \mathbf{0}$,

P_9: $S(t_1) \times t_2 = t_1\times t_2 + t_2$.

Аксиомы P_4 и P_5 обеспечивают некоторые необходимые свойства равенства. Аксиомы P_3 и P_2 обеспечивают существование нуля и операции «непосредственно следующий». Аксиомы P_6–P_9 представляют собой рекурсивные равенства, служащие определениями операций сложения и умножения.

С помощью *MP* из схемы аксиом P_1 мы можем получить следующее правило индукции: из $P(\mathbf{0})$ и $\forall x(P(x) \supset P(\mathrm{S}(x)))$ выводится $\forall x P(x)$.

Терм $S(\ldots S(\mathbf{0})\ldots)$, где символ S повторяется k раз, кратко будем обозначать $\boldsymbol{k}$. Таким образом, натуральное число k в *EA* интерпретируется как терм $\boldsymbol{k}$. Термы такого рода **0**, **1**, **2**, … принято называть нумералами – стандартными обозначениями конкретных натуральных чисел. Очевидно, термы *EA* – это обозначения полиномов (от нескольких, вообще говоря, переменных) с натуральными коэффициентами. Например, терм $((x\times x)+((\mathbf{2}\times x)\times y))+(y\times y)$ представляет полином $x^2+2xy+y^2$.

Сейчас удобно на примере системы *EA* пояснить, почему необходима аксиома $\boldsymbol{A_4}$ для определения теории первого порядка – она предотвращает коллизию переменных.

Для теории первого порядка должна выполняться аксиома

$\boldsymbol{A_4}$. $\forall x\ A(x) \supset A(t)$, где $A(t)$ есть формула теории $\boldsymbol{T}$ и t есть терм теории $\boldsymbol{T}$, свободный для x в $A(x)$.

Рассмотрим теорию *EA*. Пусть $A(x) \equiv \exists b(b=x+1)$. Тогда $\forall x A(x) \equiv \forall x\exists b(b=x+1)$ – истинная формула при стандартной интерпретации *EA*. Возьмем терм $t \equiv b$, тогда для терма t имеем следующую формулу $\forall x\exists b(b=x+1) \supset \exists b(b=b+1)$. Формула $\exists b(b=b+1)$ ложна при стандартной интерпретации *EA*, следовательно, формула $\forall x\ A(x) \supset A(t)$ – не общезначима.

Вывод в теории *EA* организован таким образом, что все теоремы элементарной арифметики являются истинными утверждениями математики[16].

Основным средством вывода теорем в теории *EA* является, как и следовало ожидать, схема индукции. Рассмотрим в качестве примера вывод формулы $\mathbf{0}+x=x$ (она отличается от аксиомы $x+\mathbf{0}=x$!). Обозначим $\mathbf{0}+x=x$ через $A(x)$. Сначала мы должны доказать $A(\mathbf{0})$, т.е. $\mathbf{0}+\mathbf{0}=\mathbf{0}$, но это частный случай упомянутой только что аксиомы. Теперь можем доказать $A(x)\supset A(S(x))$. Предполагая воспользоваться теоремой дедукции, возьмем $A(x)$ в качестве гипотезы:

$A(x)$ или $\mathbf{0}+x=x$ (гипотеза),

$\mathbf{0}+S(x)=S(\mathbf{0}+x)$ (частный случай аксиомы),

$\mathbf{0}+x=x \supset S(\mathbf{0}+x)=S(x)$ (свойство равенства),

$S(\mathbf{0}+x)=S(x)$ (modus ponens),

$\mathbf{0}+S(x)=S(x)$ или $A(S(x))$ (транзитивность равенства).

По теореме дедукции отсюда следует $\vdash A(x)\supset A(S(x))$, а затем $\vdash \forall x(A(x)\supset A(S(x)))$. Так как $A(\mathbf{0})$ уже доказано, то по схеме индукции получаем $\vdash \forall x A(x)$ или $\vdash \mathbf{0}+x=x$.

Аналогично доказываются другие простые теоремы *EA*. Следует помнить, однако, что перед тем как доказывать какую-либо теорему (например, коммутативность умножения: $x\times y=y\times x$), полезно уже знать некоторые теоремы. Таким образом, даже доказательство простых теорем *EA* содержит в себе творческий момент – он состоит в наиболее рациональном выборе порядка, в котором эти теоремы следует доказывать.

Следующее утверждение является эмпирически установленным фактом: все рассуждения обычной (интуитивной) теории чисел, которые не апеллируют к произвольным действительным числам и функциям, могут быть формально воспроизведены в *EA*.

2.2. Арифметические функции и отношения

Как можно трактовать в теории *EA* операцию возведения в степень, если соответствующего символа в языке теории нет? Другой вопрос: мы пишем формулы, заменяющие в *EA* интуитивно понимаемые предикаты вроде «x – простое число»:

«$\mathbf{1}<x$» & $\neg\exists y\exists z$(«$y<x$»& «$z<x$» & $x=y\times z$);

Какие требования мы должны предъявлять к такого рода формулам? Если кто-то предлагает формулу и утверждает, что она «выражает» то-то и то-то, как это проверить? Для таких ситуаций вводятся понятия выразимости предикатов и представимости функций.

Арифметическими функциями называются функции, у которых область определения и множество значений состоят из натуральных чисел, а арифметическим отношением является всякое отношение, заданное на множестве натуральных чисел. Так, например, умножение есть арифметическая функция с двумя аргументами, а выражение $x+y<z$ определяет некоторое арифметическое отношение с тремя аргументами. Арифметические функции и отношения являются понятиями интуитивными и не связанными ни с какой формальной системой.

Выразимые отношения

Арифметическое отношение $R(x_1,\ldots, x_n)$ называется *выразимым* в *EA*, если существует формула $A(x_1,\ldots, x_n)$ теории *EA* с n свободными переменными такая, что для любых натуральных чисел $k_1,\ldots, k_n$

1) если $R(k_1,\ldots, k_n)$ истинно, то $\vdash A(\boldsymbol{k_1},\ldots, \boldsymbol{k_n})$,
2) если $R(k_1,\ldots, k_n)$ ложно, то $\vdash \neg A(\boldsymbol{k_1},\ldots, \boldsymbol{k_n})$.

Легко проверить, что формула $x=y$ выражает в *EA* обычное интуитивно понимаемое равенство натуральных чисел. В самом деле, для любых m, n: а) если $m=n$, то $\vdash \boldsymbol{m}=\boldsymbol{n}$ (термы $\boldsymbol{m}$, $\boldsymbol{n}$ в этом случае просто совпадают), б) если $m\neq n$, то $\vdash \neg(\boldsymbol{m}=\boldsymbol{n})$ (проверьте).

Интересно было бы получить какую-то характеристику класса тех предикатов, которые выразимы в *EA*. Если теория *EA* противоречива, то в ней доказуема любая формула и поэтому – в смысле принятого нами определения – в *EA* можно выразить любой предикат (например, формула $x=y$ выражает любой двухместный предикат). Многочисленные примеры выразимых отношений с доказательствами см. в [16, с.132–135]. Почти любое арифметическое отношение является выразимым в элементарной арифметике. Любое одноместное отношение (свойство) натуральных чисел определяет некоторое множество натуральных чисел, а именно, состоящее из чисел, обладающих данным свойством. Нетрудно проверить, что множество различных формул в теории первого порядка является счетным множеством. С другой стороны множество всех подмножеств, составленных из натуральных чисел, имеет мощность больше счетного (теорема Кантора), отсюда следует, что существуют свойства натуральных чисел, невыразимые в *EA*.

Представимые функции

Арифметическая функция $f(x_1,\ldots, x_n)$ называется *представимой* в *EA*, если существует формула $A(x_1,\ldots, x_n, y)$ формальной арифметики со свободными переменными $x_1,\ldots, x_n, y$ такая, что для любых натуральных чисел $k_1,\ldots, k_n, k_{n+1}$ таких, что $f(k_1,\ldots, k_n) = k_{n+1}$

1) $\vdash A(\boldsymbol{k_1},\ldots, \boldsymbol{k_n}, \boldsymbol{k_{n+1}})$,
2) $\vdash \forall y\,(\neg(\boldsymbol{k_{n+1}} = y) \supset \neg A(\boldsymbol{k_1},\ldots, \boldsymbol{k_n}, y))$.

Все функции, которые мы в интуитивном смысле считаем вычислимыми (т. е. значения которых вычисляются с помощью алгоритмов), оказываются представимыми в *EA* (см. теорему 5.). Если теория *EA* противоречива, то в ней представима любая функция.

Характеристической функцией данного отношения $R(\boldsymbol{x})$ (в данном случае $\boldsymbol{x}$ сокращенная запись для $<x_1, \ldots, x_n>$) называется функция $C_R(\boldsymbol{x})$, задаваемая условиями:

$$C_R(\boldsymbol{x}) = \begin{cases} 1, \text{ если } R(\boldsymbol{x}) \text{ истинно,} \\ 0, \text{ если } R(\boldsymbol{x}) \text{ ложно.} \end{cases}$$

Теорема 4. Отношение $R(\boldsymbol{x})$ выразимо в теории *EA* тогда и только тогда, когда характеристическая функция $C_R(\boldsymbol{x})$ представима в *EA*.

Теорема 5. [16, с. 158] Всякая функция, представимая в *EA*, является частично-рекурсивной и наоборот.

Оказывается, что класс выразимых отношений в *EA* также оказывается очень широк. Но сначала нам потребуется определение.

Отношение R называется **разрешимым**, если его характеристическая функция C_R является частично-рекурсивной.

Теорема 6. Всякое отношение, выразимое в *EA*, является разрешимым и наоборот.

Справедливость этой теоремы следует из теорем 5 и 4.

2.3. Гёделева нумерация

Пусть дана некоторая формальная система. Предложение, утверждающее, что некоторая последовательность формул образует (или не образует) вывод некоей формулы, уже не является доказательством в самой этой формальной системе. Это утверждение о системе; такие утверждения обычно называют *метаматематическими*. Необходимо тща-

тельно различать математические и метаматематические рассуждения. Нарушение этого условия приводит к парадоксам. Пример см. в разделе 1.4.

У Гёделя возникла великая идея *арифметизации метаматематики*, т.е. замены утверждений о формальной системе эквивалентными высказываниями о натуральных числах с последующим выражением этих высказываний в формальной системе. Идея арифметизации стала ключом к решению многих важных проблем математической логики. Арифметизацией данной теории первого порядка мы называем всякую функцию g, отображающую инъективно множество всех символов, выражений и конечных последовательностей выражений во множество целых положительных чисел. При этом требуется: (*i*) чтобы значение функции g можно было вычислить с помощью некоторого алгоритма, (*ii*) чтобы существовал алгоритм, позволяющий для каждого m определить, является ли m значением функции g, и в случае, если является, то построить тот объект x, для которого $m = g(x)$. Значение функции g для символа (выражения, последовательности выражений) называется *гёделевым номером* соответствующего объекта.

Существуют различные способы построения гёделевых номеров для системы *EA*[16, с. 151–152]. Изложим кратко одну из возможных гёделевых нумераций. Рассмотрим алфавит теории *EA*. Выберем 10 основных символов этого алфавита: **0**, S, (,), +, ×, $\supset$, $\neg$, $\forall$ и =. Другие логические связки можно выразить через $\supset$, $\neg$ и $\forall$. Например, $A \vee B \equiv \neg A \supset B$, а $\exists x A(x) \equiv \neg \forall x \neg A(x)$.

Таким образом, основной символ алфавита получает в качестве гёделева номера число, не превосходящее 10. Например, соответствие может быть задано таблицей:

0	S	(	)	+	×	$\supset$	$\neg$	$\forall$	=
1	2	3	4	5	6	7	8	9	10

Предметным переменным сопоставляются простые числа, большие 10 (скажем, x получает номер 11, y – номер 13 и т. д.).

Формула получает номер по правилу, которое лучше всего пояснить на примере. Рассмотрим формулу $\forall x\ (x \times S(\mathbf{0}) = x)$, которая утверждает, что число не меняется при умножении на 1. Эта формула содержит 12 символов, причем некоторые из них (скажем, x) входят несколько раз. Возьмем первые 12 простых чисел, возведем каждое из них в степень, равную номеру соответствующего символа и перемножим полученные числа. Найденное так число

$$2^9 \times 3^{11} \times 5^3 \times 7^{11} \times 11^6 \times 13^2 \times 17^3 \times 19^1 \times 23^4 \times 29^{10} \times 31^{11} \times 37^4 =$$
$$351246696379113496496255178346205341140836151634379449650656150131286227200 0$$

и будет гёделевым номером этой формулы.

Последовательность формул (которая может составлять доказательство) $F_1, F_2, \ldots, F_m$ получает гёделевый номер

$$2^{G_1} \times 3^{G_2} \times \ldots \times p_m^{\,G_m},$$

где p_m есть m-е простое число, а $G_1, G_2, \ldots$ – гёделевы номера соответственно формул $F_1, F_2, \ldots$.

Таким образом каждой формуле или последовательности формул ставится в соответствие единственное натуральное число. Не всякое натуральное является геделевым номером, но всякий гёделевый номер однозначно определяет соответствующее выражение. Это следует из теоремы о единственности разложения на простые множители: для целого числа, большего 1, существует единственный (с точностью до порядка) способ записать его в виде произведения степеней простых чисел.

Оказывается функцию, осуществляющую гёделеву нумерацию формул элементарной арифметики, можно сделать даже примитивно-рекурсивной.

Гёделева нумерация устанавливает также счетность формул теории первого порядка.

Заметим, что метаматематическое утверждение «$\forall x$ есть начальная часть формулы $\forall x\ (x \times S(\mathbf{0}) = x)$» отражается внутрь теории, переходя в чисто арифметическое предложение «гёделевый номер выражения $\forall x$, равный $2^9 \times 3^{11}$, является делителем гёделевого номера полной формулы».

2.4. Лемма о рефлексии

– Он целовал вас, кажется?
– Боюсь, что это так.
– Но как же вы позволили?
– Ах, он такой чудак.
Он думал, что уснула я
И всё во сне стерплю,
Иль думал, что я думала,
Что думал он: я сплю!
Ковентри Патмор «О поцелуе»

Попытаемся воспроизвести рассмотренные ранее (раздел 2.1) парадоксы средствами теории *EA*. Чтобы воспроизвести классический парадокс лжеца, мы должны построить формулу *Q*, «утверждение» которой состояло бы в том, что «Q можно опровергнуть в EA» (в теории *EA* ложным считается то, что можно опровергнуть исходя из аксиом). Но как добиться, чтобы формула относилась к себе самой, «говорила о себе»?

Формулы *EA* «умеют говорить» только о натуральных числах. Чтобы формула *Q* могла говорить «о формулах и о себе», все формулы должны быть закодированы натуральными числами с помощью гёделевой нумерации. Пусть функция g осуществляет гёделеву нумерацию. Если теперь для некоторой формулы $A(x)$ и другой формулы B удается установить, что $\vdash A(g(B))$, то можно сказать: в теории *EA* доказано, что формула B «обладает свойством A».

Теорема 7 (лемма о рефлексии). Пусть $A(x)$ произвольная формула формальной арифметики, имеющая единственную свободную переменную x. Тогда можно построить замкнутую формулу B, такую, что $\vdash B \sim A(g(B))$.

Неформально говоря, для любого выразимого свойства формул, можно подобрать замкнутую формулу, «утверждающую», что она этим свойством обладает.

В своей знаменитой теореме о неполноте (см. ниже) К. Гёдель использовал конструкцию, которая составляет доказательство леммы о рефлексии, однако в общем виде он эту лемму не сформулировал. Доказательство этой леммы в различных формах см. в [15, с. 79; 3, с. 153–154; 17, с. 128; 21, с.15–16].

Мы приведем эскиз доказательства. Предполагаем, что в ***EA*** существует счетное множество функций (операций), причем каждый функциональный символ имеет свой код Гёделя. Если s – синтаксический объект в ***EA***, то пусть $\#s$ обозначает его геделевский номер («код»), а $\langle s \rangle \equiv \langle \#s \rangle$ – соответствующий терм в ***EA***. В этих обозначениях формулировка теоремы следующая:

Пусть $A(x)$ – формула системы ***EA*** с единственной свободной переменной x. Тогда существует формула B системы ***EA***, такая, что

$\vdash_{EA} B \sim A(\langle B \rangle)$.

(B говорит: «*Я* обладаю свойством A»)

Доказательство.

Введем на множестве натуральных чисел $\boldsymbol{N}$ бинарное отношение $\sim$:

$n_1 \sim n_2 \Leftrightarrow$ существуют формулы A_1 и A_2 в ***EA***, такие, что $n_1 = \#A_1$, $n_2 = \#A_2$ и $\vdash A_1 \sim A_2$. Для теоремы достаточно показать, что для данной формулы $A(x)$ существует такое $n \in \boldsymbol{N}$, что

$$n \sim \#A(\langle n \rangle). \qquad (1)$$

Действительно, после этого взяв формулу B такую, что $n = \#B$ будем иметь $\vdash B \sim A(\langle n \rangle)$. Так как $\langle n \rangle = \langle \#B \rangle \equiv \langle B \rangle$, то получим, что $\vdash B \sim A(\langle B \rangle)$.

Для доказательства (1) введем следующее определение.

Для n, $m \in \boldsymbol{N}$ положим $h(n, m)$ = код результата подстановки терма $\langle m \rangle$ вместо единственной свободной переменной в формулу с кодом n.

Функция h является вычислимой. Действительно, зная n, можно проверить, является ли n формулой. Если это так, то алгоритмическим способом можно отыскать свободную переменную в формуле и убедиться, что она единственная. По номеру m определяем соответствующий терм. Подстановка терма $\langle m \rangle$ вместо свободной переменной в формулу описывается алгоритмом, как и последующее вычисление геделевского номера. По тезису Черча, функция h – частично рекурсивна. Она не определена в тех случаях, когда n не является номером формулы, или формула не обладает единственной свободной переменной, или m не является номером терма.

Когда значение функции определено, то имеем равенство

$$h(\#F(x), m) = \#F(\langle m \rangle). \qquad (2)$$

Поскольку h – частично рекурсивная функция, то она представима в ***EA***, т. е. существует формула $H(x_1, x_2, x_3)$ в ***EA*** со свободными переменными x_1, x_2, x_3, такая, что

$\vdash H(\langle n \rangle, \langle m \rangle, \langle h(n, m) \rangle)$,

$\vdash \forall y(\neg(y = \langle h(n, m) \rangle) \supset \neg H(\langle n \rangle, \langle m \rangle, y))$.

Для любых термов x_1, x_2 если и существует, то единственный терм y, обладающим свойством $H(x_1, x_2, y)$. Поэтому мы можем ввести в ***EA*** (не усиливая первоначальную теорию [1]) новый двухместный функциональный символ sub, такой, что

$$\vdash sub(\langle n \rangle, \langle m \rangle) = \langle h(n, m) \rangle. \qquad (3)$$

Положим теперь натуральные $v = \#A(sub(x, x)))$ и $n = h(v, v)$. Тогда

$n = h(v, v) = h(\#A(sub(x, x)), v) = \#A(sub(\langle v \rangle, \langle v \rangle)$ (в силу (2)) $\sim \#A(\langle h(v,v) \rangle) = \#A(\langle n \rangle)$.

Отношение $\sim$ между числами имеет место потому, что в силу (3) из свойства равенства следует, что

$$\vdash \mathrm{A}(sub(\langle n \rangle, \langle m \rangle)) \sim \mathrm{A}(\langle h(n, m) \rangle).$$

2.5. Теорема Гёделя о неполноте

Когда вопрос простой – ответ таким же будет.
Когда ж вопрос скрипит в зубах
Песком, попавшим в кашу,
Ответ – как прут, торчащий из земли.
Дверь без двери (Мумонкан)

Итак, лемма об рефлексии позволяет, по-видимому, воспроизвести парадокс лжеца средствами EA. Какие это будет иметь последствия?

Формула–аналог утверждения «Я лгу» должна утверждать: «Меня можно опровергнуть в EA» (вместо истинности и ложности в EA фигурируют доказуемость и опровержимость). Если

F: «$\neg F$ доказуема в EA»,

то

$\neg F$: «$\neg F$ недоказуема в EA»,

[1] Более строго см. [15, с.75–76].

поэтому с тем же успехом мы можем рассматривать формулу

F: «F недоказуема в EA»,

она также «равносильна» парадоксу лжеца. Именно такой формулой занимался в свое время К. Гёдель, и мы не будем нарушать традицию.

Формулу F: «F недоказуема в EA» мы могли бы получить из леммы о рефлексии, если бы сумели изобразить в виде формулы EA свойство «формулу с номером x можно доказать в EA».

Определим отношение на множестве замкнутых термов в элементарной арифметике $Prove(d, t)$:

« формула с гёделевым номером t имеет доказательство с номером d».

Разумеется, по номеру можно определить, является ли он номером доказательства, принадлежащего теории EA. В самом деле, по номеру последовательности можно восстановить все входящие в нее формулы и порядок их расположения. Затем мы можем проверить, является ли каждая из этих формул аксиомой EA (логической или собственной) или же она получена из предыдущих формул последовательности с помощью правил вывода. Если для всех формул это так – анализируемый номер представляет доказательство. Можно построить даже программу, реализующую эту процедуру проверки без всякого вмешательства человека.

Геделева нумерация формул и доказательств элементарной арифметики проводится таким образом, чтобы было выразимым отношение $Prove$, т.е. чтобы выполнялось утверждение

«теорема y имеет доказательство x тогда и только тогда, когда $\vdash Prove(g(x),g(y))$».

Введем формулу $Pr(t)$ как сокращение для формулы $\exists d\ Prove(d,t)$. Предикат $Pr(t)$ утверждает доказуемость теоремы с гёделевым номером t.

Описанную выше гёделеву нумерацию можно провести таким образом, чтобы для любой формулы A элементарной арифметики из $\vdash A$ следует $\vdash Pr(g(A))$.

Нам понадобится следующее определение.

ω-противоречие

Формула в теории первого порядка $C(y)$ с одной свободной переменной y такая, что
а) $\vdash \exists y C(y)$,
б) для каждого n: $\vdash \neg C(\boldsymbol{n})$
называется ω-противоречием.

Формула $C(y)$ не дает «настоящего» противоречия (доказательства формулы вместе с ее отрицанием). Однако, если подобная формула $C(y)$ имеется, это означает все же, что в теории «не все в порядке».

Можно доказать, что «настоящее» противоречие (т.е. формула D такая, что $\vdash D$ и одновременно $\vdash \neg D$) означает также ω-противоречие.

Теорема 8 (теорема Гёделя о неполноте, 1931). Можно построить замкнутую формулу G из языка EA, такую, что

1) если G доказуема в EA, то теория EA противоречива,
2) если $\neg G$ доказуема в EA, то теория EA ω-противоречива.

Эскиз доказательства. Взяв в лемме о рефлексии формулу $\neg Pr(t)$ («формула с номером t не имеет доказательства в EA»), получим замкнутую формулу G, такую, что

$$\vdash G \sim \neg Pr(g(G)).$$

Формула G утверждает, что она недоказуема в EA. Теперь можно попытаться выяснить какое из двух утверждений $\vdash G$ или $\vdash \neg G$ выполнено. Разбор случаев приводит соответственно к утверждениям, что теория EA противоречива или ω-противоречива. Но мы не можем узнать, как «на самом деле» обстоит с доказуемостью G.

Если теория *EA* ω-непротиворечива, то теорема Гёделя утверждает, что обе формулы G и $\neg G$ нельзя доказать средствами *EA*. Но тогда недоказуемая формула G при стандартной интерпретации на универсуме натуральных чисел является истинной, поскольку она утверждает свою недоказуемость.

Таким образом, у нас выбор: 1) теория *EA* ω-противоречива или 2) *EA* ω-непротиворечива (и, следовательно, просто непротиворечива), но существует истинная формула, которую доказать нельзя. Второй вариант говорит, что теория *EA* неполна.

Почему этой теореме придается такое большое значение? Сначала введем общепринятый термин. Замкнутую формулу F из языка теории $\boldsymbol{T}$ называют *неразрешимой в* $\boldsymbol{T}$, если ни F, ни $\neg F$ нельзя доказать средствами $\boldsymbol{T}$ (F предсказывает вполне определенное свойство «объектов» теории $\boldsymbol{T}$, однако это предсказание нельзя средствами $\boldsymbol{T}$ ни доказать, ни опровергнуть).

Не следует, однако, думать, что нами *доказана* неполнота теории *EA*. Неразрешимость средствами *EA* формулы G будет доказана только..., если удастся доказать, что теория *EA* ω-непротиворечива (т.е. что в ней не могут возникать ω-противоречия). До тех пор мы вправе утверждать только, что доказали *несовершенство* аксиом *EA* – эти аксиомы либо ω-противоречивы, либо с их помощью нельзя решить некоторые проблемы, касающиеся натуральных чисел (одна такая проблема выражена в формуле G – несмотря на все наши разговоры о том, что G «занимается» собственной доказуемостью, G – замкнутая формула в языке *EA* и как таковая выражает вполне определенное свойство натуральных чисел).

Несовершенную систему аксиом следует совершенствовать. Может быть, мы «забыли» какие-то важные аксиомы? Следует найти их, присоединить к аксиомам *EA*, и в результате мы получим... совершенную систему?

К сожалению, рассуждения К. Гёделя проходят и для любого расширения *EA*. Никакие новые аксиомы не могут привести к «совершенной» системе аксиом арифметики. Метод Гёделя позволяет доказать *принципиальное* несовершенство всякой системы аксиом арифметики: каждая такая система неизбежно является либо ω-противоречивой, либо недостаточной для решения некоторых проблем, касающихся свойств натуральных чисел.

Из теоремы Гёделя вытекает следующий интересный факт. Согласно закону исключенного третьего, принятого в классической логике,

$$\vdash A \vee \neg A$$

для любой формулы A. Однако следует ли отсюда, что если A – замкнутая формула, то либо $\vdash A$, либо $\vdash \neg A$? Если взять формулу G, то из теоремы Геделя о неполноте следует, что ни *EA*, ни какая-либо другая серьезная математическая теория этим идеальным свойством обладать не могут – несмотря на постулирование закона исключенного третьего в их аксиомах!

Мы сформулировали теорему Гёделя для теории *EA*. Формальная арифметика *EA* представляет простейший уровень математических рассуждений – в которых участвуют только целые числа (и не участвуют произвольные действительные числа, не говоря уже о произвольных множествах Кантора). Более сложные рассуждения формализуются и более сложными (по сравнению с *EA*) формальными теориями. «Силу» этих более сложных теорий составляет прежде всего их способность обсуждать более сложные объекты (действительные числа, функции действительных и комплексных переменных и т.д.), которые недоступны в *EA*.

Каким образом выделить в некоторой формальной теории $\boldsymbol{T}$ ту ее часть, которая относится к компетенции *EA*? Этот вопрос решается очень естественно с помощью так называемых *относительных интерпретаций*. Чтобы воспроизвести в теории $\boldsymbol{T}$ арифметику, прежде всего какие-то объекты из области значений переменных $\boldsymbol{T}$ должны быть объявлены натуральными числами. Это связано с выделением в языке $\boldsymbol{T}$ некоторой формулы $N(x)$ (с единственной свободной переменной x), которая «утверждает», что x является натуральным числом. Далее, необходимо отобразить в теории $\boldsymbol{T}$ элементарные формулы *EA*,

т.е. формулы вида $t_1 = t_2$, трактующие о значениях полиномов t_1, t_2 с натуральными коэффициентами.

Имея относительную интерпретацию *EA* в ***T***, в теории ***T*** можно доказать любое свойство натуральных чисел, которое доказуемо в *EA*. Учитывая роль системы натуральных чисел в математике, формальную теорию, в которой относительно интерпретируема теория *EA* (и которая содержит в этом смысле полноценное понятие натурального числа), будем называть **фундаментальной теорией**. Простейшей из фундаментальных теорий является, конечно, сама теория *EA*.

Теорема Гёделя, оказывается, справедлива для любой фундаментальной теории.

Понятие об ω-противоречивости несколько «портит» теорему К. Гёделя о неполноте. Однако сам Гёдель не пытался освободиться от него, впервые это удалось только Б. Россеру в 1936 г. – в теореме Россера вместо ω-противоречивости фигурирует «обычная» противоречивость.

Теорема 9 (теорема Гёделя в форме Россера). В языке всякой фундаментальной теории ***T*** найдется замкнутая формула $R_{\boldsymbol{T}}$ («выражающая» некоторое свойство натуральных чисел), такая, что если $\vdash_{\boldsymbol{T}} R_{\boldsymbol{T}}$ или $\vdash_{\boldsymbol{T}} \neg R_{\boldsymbol{T}}$, то теория ***T*** противоречива.

В этой формулировке сделано еще одно усиление теоремы о неполноте, не имеющее отношения к методу Россера. Сейчас речь идет о теории ***T*** с *произвольным* языком первого порядка, которая содержит в себе элементарную арифметику. Это освобождает нас от подозрений, что принципиальное несовершенство всякой системы аксиом арифметики кроется в неудачном выборе языка *EA*.

«Принцип несовершенства» К. Гёделя

Всякая фундаментальная теория несовершенна – она либо противоречива, либо недостаточна для решения всех возникающих в ней проблем.

В частности, если удалось аксиоматизировать всю математику, то она была бы несовершенной.

В качестве примера нефундаментальной теории можно назвать **арифметику Пресбургера**, которая получается из *EA* удалением символа умножения. В 1929 г. М. Пресбургер доказал полноту и непротиворечивость этой теории. В силу теоремы Геделя–Россера тем самым была доказана и ее нефундаментальность.

2.6. Нестандартное расширение *E*A

Поскольку *EA* является неполной системой (формула *G* – неразрешима), то ее можно пополнить (как можно дополнить абсолютную геометрию).

Стандартное пополнение: добавить в качестве аксиомы формулу *G*. (Это соответствует расширению абсолютной геометрии в эвклидовом смысле.) Такое добавление кажется довольно безвредным и даже желательным, поскольку *G* всего навсего утверждает некоторую истину о системе натуральных чисел.

Нестандартное пополнение: (если следовать аналогии с ситуацией аксиомы параллельности) добавить в качестве аксиомы формулу $\neg G$. Но как мы можем даже подумать о такой ужасной, отвратительной вещи? В конце концов, если перефразировать Саккери, не является ли то, что утверждает $\neg G$, «противным самой природе натуральных чисел».

Супернатуральные числа

Проблема с подходом Саккери к геометрии (при доказательстве постулата о параллельных) заключалась в том, что, что он основывался на жестком понятии о том, что истинно и что ложно; он хотел доказать только то, что он считал истинным с самого начала. Рассмотрим беспристрастно, что означает добавление к системе *EA* новой аксиомы $\neg G$.

Подумаем только, на что была бы похожа современная математика, если бы люди не решили в свое время добавить к ней аксиом типа:

$\exists a(a+a=\mathbf{1})$ (рациональные числа);

$\exists a(\boldsymbol{S}a=\mathbf{0})$ (отрицательные числа);

$\exists a(a\times a=\mathbf{2})$ (иррациональные числа);

$\exists a(\boldsymbol{S}(a\times a)=\mathbf{0})$ (мнимые числа).

Хотя каждое из этих утверждений «противно природе ранее известных числовых систем», каждое из них в то же время означает значительное и замечательное расширение понятия целых чисел. $\neg G$ пытается открыть нам глаза на такую возможность. В прошлом каждое новое расширение системы натуральных чисел встречалось в штыки. Это заметно по названиям: «иррациональные», «мнимые». Оставаясь верными традиции, давайте назовем числа, которые порождает $\neg G$, супернатуральными, поскольку они противоречат всем понятиям разума и здравого смысла.

Если мы собираемся добавить $\neg G$ в качестве новой аксиомы *EA*, мы должны постараться понять, каким образом эта строчка может существовать с ω-противоречием. Ведь $\neg G$ утверждает, «что выполнено $\vdash \exists y Prove(y, g(G))$». При этом члены пирамидальной семьи с успехом утверждают, что

$\vdash \neg Prove(\mathbf{0}, g(G))$

$\vdash \neg Prove(\mathbf{1}, g(G))$

$\vdash \neg Prove(\mathbf{2}, g(G))$

$\vdash \neg Prove(\mathbf{3}, g(G))$

…

Это сбивает с толку, поскольку кажется совершеннейшим противоречием. Наша проблема заключается в том, что, так же как и в случае с расширенной геометрией, мы упрямо отказываемся модифицировать интерпретацию символов, несмотря на то, что прекрасно понимаем, что имеем дело с модифицированной системой. Мы хотим обойтись без добавления *хотя бы одного* символа – что, разумеется, оказывается невозможным.

Проблема разрешается, если мы интерпретируем ∃ как «существует некое *обобщенное* натуральное число» вместо «существует некое натуральное число». Одновременно с этим нам потребуется соответствующим образом изменить интерпретацию ∀. Это означает, что, кроме натуральных, мы открываем дверь для неких новых чисел. Это супернатуральные числа. Натуральные и супернатуральные числа вместе составляют *обобщенные натуральные числа*.

Кажущее противоречие теперь испаряется, поскольку пирамидальная семья все еще утверждает, «что никакое натуральное число не является номером доказательства *G*». Строчки этой семьи ничего не упоминают о супернатуральных числах, поскольку для них не существует *символов*. С другой стороны, $\neg G$ утверждает, что существует такое обобщенное натуральное число, которое является номером доказательства *G*. Противоречия больше нет. $EA+\neg G$ превращается в непротиворечивую систему, если её интерпретация включает супернатуральные числа.

Варианты теории чисел и банкиры

Так ли это важно и необходимо, чтобы существовали разные варианты теории чисел? Если бы вы спросили банковского работника, думаю, что бы сначала не поверил, что вы говорите серьезно – а потом пришел в ужас. Как 2 + 2 может быть отлично от 4? Если бы 2 +2 не было бы равно 4, разве не зашаталась бы мировая экономика от невыносимой неуверенности, не развалилась бы она от подобного удара? На самом деле, этого не произошло бы. Прежде всего, нестандартное расширение *EA* не угрожает справедливости равенства 2+2=4. Оно отличается от привычной нам арифметики только тем, как она обращается с понятием бесконечности. В конце концов, любая теорема *EA* остается теоремой в любой расширенной версии *EA*!

Боязнь, что новые варианты математических теорий изменят старые, хорошо известные факты, отражает непонимание отражения математики с действительностью. *Математика дает нам ответы на вопросы о реальном мире только после того, как мы выбрали, какой тип математики мы используем в данный момент.* Даже если бы существовал соперничающий вариант теории чисел, в котором 2+2=3 было бы теоремой, у банкиров не было бы причин выбирать именно этот вариант! Эта теория не отражает того, как ведут себя деньги. Мы приспосабливаем математику к действительности, а не наоборот.

Варианты теории чисел и метаматематики

Как может затронуть математических логиков существование супернатуральных чисел? Теория чисел играет в логике две роли: (1) когда она аксиоматизирована, она становится объектом изучения, и (2) используемая неформально, она является необходимым орудием, при помощи которых могут изучаться формальные системы. Это напоминает уже знакомое нам различие между использованием и упоминанием: в роли (1) теория чисел упоминается, в роли (2) она используется.

Математики решили, что теория чисел, хотя она и не подходит для подсчета облаков, вполне годится для изучения формальных систем, так же как банковские работники решили, что арифметика действительных чисел годится для их операций. Подобные решения принимаются вне математики; они показывают, что мыслительные процессы, задействованные в изучении математики, так же, как и в других областях человеческой деятельности, включают «запутанные иерархии», где мысли на одном уровне могут влиять на мысли на другом уровне. При этом четкого разделения на уровни не существует, как могут думать последователи формалистского взгляда на математику.

Формалистская философия утверждает, что математики имеют дело с абстрактными символами и что им совершенно все равно, соответствуют ли эти символы окружающей действительности. Однако, это весьма искаженная картина, что становится особенно ясным в метаматематике. Используя саму теорию чисел для получения новых фактов о формальных системах, метаматематики показывают, что они считают эфирные создания, называемые «натуральными числами», частью реального мира, а не просто плодом воображения. Именно поэтому я упомянул ранее о том, что в некотором роде можно ответить на вопрос, какой из вариантов теории чисел является «истинным». Дело в том, что математическим логикам приходится выбирать в какой из вариантов теории чисел «поверить». В частности, они не могут оставаться в стороне от принятия или не принятия супернатуральных чисел, поскольку эти варианты теории чисел дают разные ответы на вопросы метаматематики.

2.7. Теорема Гудстейна

Хотя неразрешимое самоссылочное утверждение Гёделя, несомненно, также говорит о каком-то свойстве натуральных числах, математикам хотелось бы также обнаружить более «естественное» верное, но недоказуемое в арифметике Пеано утверждение. Одно из таких утверждений есть теорема Гудстейна.

Наследственное представление

Наследственным представлением натурального числа называется его представление в виде суммы степеней с основанием b, причем показатели степени также представляются в виде суммы степеней числа b и т. д., пока процесс не остановится.

Например, наследственное представление 266 по основанию 2 есть

266	=	$2^8 + 2^3 + 2$
	=	$2^{(2^{2+1})} + 2^{2+1} + 2$

Последовательность Гудстейна

Для данного наследственного представления числа n по основанию b пусть $F_b(n)$ – неотрицательное целое число равное результату синтаксической замены в представлении n каждого b на b+1 (т. е. F_b есть оператор замены b на b+1).

Так как $266 = 2^{(2^{2+1})} + 2^{2+1} + 2$, то замена основания 2 на 3 дает

$$F_2(266) = 3^{(3^{3+1})} + 3^{3+1} + 3.$$

В построении следующей последовательности повторяется применение оператора F_b с последующим вычитанием 1. Первые шесть членов суть

$G_0(266)$	=	$266 = 2^{(2^{2+1})} + 2^{2+1} + 2$
$G_1(266)$	=	$F_2(266)–1 = 3^{(3^{3+1})} + 3^{3+1} + 2$
	=	443426488243037769948249630619149892886 (39 цифр)
$G_2(266)$	=	$F_3(G_1)–1 = 4^{(4^{4+1})} + 4^{4+1} + 1$
	=	3231700607…853611059596231681 (617 цифр)
$G_3(266)$	=	$F_4(G_2)–1 = 5^{(5^{5+1})} + 5^{5+1}$ (10922 цифры)
$G_4(266)$	=	$F_5(G_3)–1 = 6^{(6^{6+1})} + 6^{6+1} – 1$
	=	$6^{(6^{6+1})} + 5\cdot 6^6 + 5\cdot 6^5 + \ldots + 5\cdot 6 + 5$
$G_5(266)$	=	$F_6(G_4)–1 = 7^{(7^{7+1})} + 5\cdot 7^7 + 5\cdot 7^5 + \ldots + 5\cdot 7 + 4$

Последовательность $\{G_k(n)\}$ называется последовательностью Гудстейна.

Теорема Гудстейна

Для любого n существует такое k, что $G_k(n) = 0$.

Кажется невероятным, но это так. А чтобы в это поверить, я рекомендовал бы читателю самостоятельно проделать вышеописанную процедуру, для начала – с числом «3».

$G_0(3) = 2 + 1$
$G_1(3) = F_2(3) –1 = 3$
$G_2(3) = F_3(G_1) –1 = 4 – 1 = 3$
$G_3(3) = F_4(G_2) –1 = 2$
$G_4(3) = F_5(G_3) –1 = 1$
$G_5(3) = F_6(G_6) –1 = 0$

Попробуем проверим утверждение теоремы для n=4:

2^2
$3^3 – 1 = 2\times3^2+2\times3+2$
$2\times4^2+2\times4+1$
$2\times5^2+2\times5$
$2\times6^2+ 6 +5$
$2\times7^2+ 7 +4$
$2\times8^2+ 8 +3$
$2\times9^2+ 9 +2$
$2\times10^2+ 10 +1$

$2\times11^2+11$
$2\times12^2+12-1 = 2\times12^2+11$
$2\times13^2+10$
…
2×23^2
$2\times24^2-1 = 24^2+23\times24+23$
$25^2+23\times25+22$
…
$47^2+23\times47$
$48^2+23\times48-1 = 48^2+22\times48+47$
$49^2+22\times49+46$
…
$95^2+22\times95$
$96^2+22\times96-1 = 96^2+21\times96+95$
$97^2+21\times97+94$
…
$191^2+21\times191$
$192^2+21\times192-1 = 192^2+20\times192+191$
$193^2+20\times193+190$
…
$383^2+20\times383$
$384^2+20\times384-1 = 384^2+19\times384+383$
$385^2+19\times385+382$
…
$767^2+19\times767$
$768^2+19\times768-1 = 768^2+18\times768+767$
$769^2+18\times769+766$
…
$1535^2+18\times1535$
$1536^2+18\times1536-1 = 1536^2+17\times1536+1535$
$1537^2+17\times1537+1534$
…
…
…

Эта последовательность доходит до числа из 121 210 695 цифр (для сравнения заметим, что 1000000! содержит около 5 с половиной миллионов цифр), но потом числа только уменьшаются (так как уже не содержат в наследственном представлении основания) вплоть до 0.

Секрет поведения последовательности Гудстейна заключается в следующем. Наследственное представление числа *n* по основанию *b* имитирует ординальное представление для всех ординалов меньших чем некоторое число. Для таких ординалов оператор *Fb* делает этот ординал фиксированным, в то время как вычитание 1 его уменьшает. Но так как множество ординалов вполне упорядоченно, то из этого следует, что последовательность Гудстейна сходится к нулю.[2]

---

[2] Более строго. Введем обозначения: ω – ординал, равный порядковому типу множества натуральных чисел; $\varepsilon_0 = \sup(\omega, \omega^\omega, \omega^{(\omega^\omega)}, ...)$. Возьмем наследственное представление числа *n* и заменим все основания сразу на ординал ω. Тогда последовательность Гудстейна для *n* становится убывающей последовательностью ординалов, меньших ε_0.

Гудстейн доказал теорему в 1944 году, используя трансфинитную индукцию [25]. В 1982 году Л. Кирби и Дж. Парис получили результат о том, что теорема Гудстейна формально недоказуема в рамках аксиоматической системы элементарной арифметики [26].

2.8. Об аксиоматизации

Теорема К. Гёделя о неполноте породила множество рассуждений о том, что аксиоматический метод недостаточен для реконструкции «живого, содержательного» математического мышления. Аксиоматику сравнивали с прокрустовым ложем, которое не в состоянии вместить все богатство содержательной математики. Но разве могут в математике какие-либо доказательные рассуждения происходить иначе, как по схеме «посылки – заключение»? Если так и всякое математическое рассуждение сводится к цепи заключений, то можно спросить: эти заключения происходят по *определенным* правилам (т.е. таким, которые не меняются от одного случая к другому и от одного математика к другому)? И если правила являются определенными, то, будучи функцией человеческого мозга, могут ли они быть такими, что их нельзя никак явно сформулировать? Если какие-либо «правила» нельзя явно сформулировать, то, следовательно, нельзя *доказать* их определенность! Ну, а полагать, что в математике кроме рассуждений (по определенным правилам) имеются «объекты», существующие независимо от этих рассуждений, означает впасть в обыкновенный платонизм работающего математика.

Таким образом, преждевременно говорить об ограниченности аксиоматизации – границы ее применимости, по-видимому, совпадают с границами применимости самой математики.

В процессе развития математических теорий аксиоматизация и интуиция взаимодействуют. Аксиоматизация «проясняет» интуицию, когда та «запуталась в себе». Но аксиоматизация влечет за собой и неприятные последствия: многие рассуждения, которые в интуитивной теории опытный специалист проводит очень быстро и представляет компактно, в аксиоматической теории оказываются очень громоздкими. Поэтому после замены интуитивной теории аксиоматической (особенно если эта замена неэквивалентна по причине недостатков интуитивной теории) специалисты развивают новую интуицию, которая восстанавливает способность теории к творческому развитию. Пример тому – история аксиоматизации теории множеств. Когда в интуитивной теории множеств Кантора в 1890-х гг. были обнаружены противоречия, от них удалось избавиться путем аксиоматизации. Естественно, что созданная аксиоматическая теория множеств Цермело–Френкеля отличалась от интуитивной теории Кантора не только формой, но и отдельными аспектами содержания. Для работы в новой теории специалисты развили модифицированную интуицию (в том числе особую интуицию множеств и классов). Ныне вполне нормальной считается работа в теории Цермело–Френкеля на интуитивном уровне. Именно на таком уровне доказываются серьезные новые теоремы этой теории.

Какую пользу дает аксиоматизация?

Во-первых, аксиоматизация позволяет «подправить» интуицию: устранить неточности, двусмысленности и парадоксы, которые иногда возникают из-за неполной контролируемости бессознательных процессов. Самый впечатляющий пример – историю аксиоматизации теории множеств, мы только что отметили.

Во-вторых, аксиоматизация позволяет подвергнуть подробному исследованию отношения между принципами теории (прежде всего, установить их зависимость или независимость), а также между этими принципами и теоремами теории. Для доказательства конкретной теоремы иногда требуются не все аксиомы теории, а только их часть. Исследования такого рода могут привести к созданию более общих теорий, которые применимы

в различных конкретных теориях. Характерными примерами являются теория групп и многочисленные ее алгебраические ответвления.

В-третьих, нередко после аксиоматизации удается установить недостаточность данной теории для решения отдельных проблем, естественно возникающих в ней. Именно так произошло с континуум–гипотезой в теории множеств. В таких случаях можно ставить вопрос о необходимости совершенствования системы аксиом теории, о развитии альтернативных вариантов теории и т.д.

Построение формальной системы определяется, как правило, нуждами практики, но аксиоматические системы можно изучать независимо от каких-либо прикладных задачах. Поэтому существует, например, несколько вариантов формальных теорий множеств, причем не все они приводят к одним результатам (например, есть теории, в которых существуют множества с промежуточной мощностью между счетными и континуумом, и есть теории, в которых таких множеств нет). Можно спросить, «а что на самом деле?». Каков «подлинный мир множеств»? Никакого «подлинного мира множеств», не зависящего от аксиом, с помощью которых он исследуется, разумеется, не существует. Математика даёт ответы на вопросы о реальном мире только после того, как мы выбрали, какой тип математики мы используем в данный момент.

3. Сложность вычислений

Применение математики во многих приложениях требует, как правило, использования различных алгоритмов. Для решения многих задач не трудно придумать комбинаторные алгоритмы, сводящиеся к полному перебору вариантов. Но здесь вступает в силу различие между математикой и информатикой: в информатике недостаточно высказать утверждение о существовании некоторого объекта в теории и даже недостаточно найти конструктивное доказательство этого факта, т.е. алгоритм. Мы должны учитывать ограничения, навязываемые нам миром, в котором мы живем: необходимо, чтобы решение можно было вычислить, используя объем памяти и время, приемлемые для человека и компьютера.

Если дана задача, как найти для её решения эффективный алгоритм? А если алгоритм найден, как сравнить его с другими алгоритмами, решающими ту же задачу? Как оценить его качество? Вопросы такого рода интересуют и программистов, и тех, кто занимается теоретическим исследованием вычислений.

Введем в первую очередь некоторые понятия, связанные с асимптотической оценкой функций.

3.1. Асимптотические обозначения

Хотя во многих случаях эти обозначения используются неформально, полезно начать с точных определений. На протяжении этого раздела встречающиеся в тексте функции отображают целые числа в целые, в том случае, если из контекста это непонятно. Содержание пункта взято из [9].

Θ-обозначение

Если $f(n)$ и $g(n)$ – некоторые функции, то запись $\boldsymbol{f(n) = \Theta(g(n))}$ означает, что найдутся такие c_1, $c_2 > 0$ и такое n_0, что $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всех $n \ge n_0$.
В этом случае говорят, что $g(n)$ является **асимптотически точной оценкой** для $f(n)$.

Разумеется, это обозначение следует употреблять с осторожностью: Установив, что $f_1(n) = \Theta(g(n))$ и $f_2(n) = \Theta(g(n))$, не следует заключать, что $f_1(n) = f_2(n)$!

Определение $\Theta(g(n))$ предполагает, что функции $f(n)$ и $g(n)$ асимптотически неотрицательны, т.е. неотрицательны для достаточно больших значений n. Заметим, что если f и g строго положительны, то можно исключить n_0 из определения (изменив константы c_1 и c_2 так, чтобы для малых n неравенство также выполнялось).

Это отношение симметрично: если $f(n) = \Theta(g(n))$, то $g(n) = \Theta(f(n))$.

Пример. Проверим, что $(1/2)n^2–3n = \Theta(n^2)$. Согласно определению надо указать положительные константы c_1, c_2 и число n_0 так, чтобы неравенства

$$c_1n^2 \le n^2/2 - 3n \le c_2n^2$$

выполнялось для всех $n \ge n_0$. Разделим на n^2:

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2.$$

Видно, что для выполнения второго неравенства достаточно положить $c_2 = ½$. Первое будет выполнено, если, например, $n_0 = 7$ и $c_1 = 1/14$.

Другой пример использования формального определения: покажем, что $6n^3 \ne \Theta(n^2)$. В самом деле, пусть найдутся такие c_2 и n_0, что $6n^3 \le c_2n^2$ для всех $n \ge n_0$. Но тогда $n \le c_2/6$ для всех $n \ge n_0$ – что явно не так.

Отыскивая асимптотически точную оценку для суммы, мы можем отбрасывать члены меньшего порядка, которые при больших n становятся малыми по сравнению с основным слагаемым. Заметим также, что коэффициент при старшем члене роли не играет (он может повлиять только на выбор констант c_1 и c_2). Например, рассмотрим квадратичную функцию $f(n) = an^2 + bn + c$, где a, b и c – некоторые константы и $a>0$. Отбрасывая члены младших порядков и коэффициент при старшем члене, находим, что $f(n) = \Theta(n^2)$. Чтобы убедиться в этом формально, можно положить $c_1 = a/4$, $c_2 = 7a/4$ и $n_0 = 2\times\max(|b|/a, \sqrt{|c|/a})$. Вообще, для любого полинома $p(n)$ степени d с положительным старшим коэффициентом имеем $p(n) = \Theta(n^d)$.

Упомянем важный частный случай использования Θ-обозначений: $\Theta(1)$ обозначает ограниченную функцию, отделенную от нуля некоторой положительной константой при достаточно больших значениях аргумента.

O- и Ω-обозначения

Запись $f(n) = \Theta(g(n))$ включает в себя две оценки: верхнюю и нижнюю. Их можно разделить.

Если $f(n)$ и $g(n)$ – некоторые функции, то

- запись $\boldsymbol{f(n) = O(g(n))}$ означает, что найдётся такая константа $c > 0$ и такое n_0, что $f(n) \le cg(n)$ для всех $n \ge n_0$;
- запись $\boldsymbol{f(n) = \Omega(g(n))}$ означает, что найдётся такая константа $c > 0$ и такое n_0, что $0 \le cg(n) \le f(n)$для всех $n \ge n_0$.

Теорема 10.

1. Для любых двух функций $f(n)$ и $g(n)$ свойство $f(n) = \Theta(g(n))$ выполнено тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.
2. Для любых двух функций $f(n)$ и $g(n)$ свойство $f(n) = O(g(n))$ выполнено тогда и только тогда, когда $g(n) = \Omega(f(n))$.

Как мы видели, $an^2 + bn + c = \Theta(n^2)$ (при $a>0$). Поэтому $an^2 + bn + c = O(n^2)$. Другой пример: при $a>0$ можно записать $an + b = O(n^2)$ (положим $c = a + |b|$ и $n_0 = 1$). Заметим, что в этом случае $an + b \ne \Omega(n^2)$ и $an + b \ne \Theta(n^2)$.

Работая с символами O, Θ и Ω мы имеем дело с *односторонними* равенствами – эти символы могут стоять только справа от знака =.

Сравнение функций

Введенные нами определения обладают некоторыми свойствами транзитивности, рефлексивности и симметричности.

Транзитивность

$f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ влечет $f(n) = \Theta(h(n))$.
$f(n) = O(g(n))$ и $g(n) = O(h(n))$ влечет $f(n) = O(h(n))$.
$f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ влечет $f(n) = \Omega(h(n))$.

Рефлексивность

$f(n) = \Theta(f(n)), f(n) = O(f(n)), f(n) = \Omega(f(n))$

Симметричность

$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

3.2. Алгоритмы и их сложность

Класс однородных вычислительных задач мы будем называть *проблемой* (также используется понятие *массовая задача* или *абстрактная задача*). Индивидуальные случаи проблемы Q мы будем называть *частными случаями* проблемы Q. Мы можем, например, говорить о проблеме умножения матриц. Частные случаи этой проблемы суть пары матриц, которые нужно перемножить.

Более формально, мы принимаем следующую абстрактную модель вычислительной задачи. Абстрактная задача есть произвольное бинарное отношение Q между элементами двух множеств – множества *условий* (или входных данных) I и множества *решений* S. Например, в задаче умножения матриц входными данными являются две конкретные матрицы – сомножители, а матрица – произведение является решением задачи. В задаче SHORTEST-PATH (поиска кратчайшего пути между двумя заданными вершинами некоторого неориентированного графа $G = (V,E)$) условием (элементом I) является тройка, состоящая из графа и двух вершин, а решением (элементом S) – последовательность вершин, составляющих требуемый путь в графе. При этом один элемент множества I может находиться в отношении Q с несколькими элементами множества S (если кратчайших путей между данными вершинами несколько).

Нам бы хотелось связать с каждым частным случаем проблемы некоторое число, называемое его *размером*, которое выражало бы меру количества входных данных. Например, размером задачи умножения матриц может быть наибольший размер матриц-сомножителей. Размером задачи о графах может быть число ребер данного графа.

Решение задачи на компьютере можно осуществлять с помощью различных алгоритмов. Прежде чем подавать на вход алгоритма исходные данные (то есть элемент множества I), надо договориться о том, как они представляются «в понятном для компьютера виде»; мы будем считать, что исходные данные закодированы последовательностью битов. Формально говоря, *представлением* элементов некоторого множества M называется отображение e из M во множество битовых строк. Например, натуральные числа 0, 1, 2, 3, … обычно представляют битовыми строками 0, 1, 10, 11, 100, … (при этом, например, $e(17) = 10001$).

Фиксировав представление данных, мы превращаем абстрактную задачу в *строковую*, для которой входным данным является битовая строка, представляющая исходное данное абстрактной задачи. Естественно считать размером строковой задачи длину строки.

Асимптотическая временная сложность

Будем говорить, что алгоритм **решает** строковую задачу за **время O($T(n)$)**, если на входном данном битовой строки длины n алгоритм работает время O($T(n)$).

В качестве временной оценки работы алгоритма вместо общего числа шагов мы можем подсчитывать число шагов некоторого вида, таких как арифметические операции при алгебраических вычислениях, число сравнений при сортировке или число обращений к памяти.

Можно было подумать, что колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения компьютеров, уменьшит значение эффективных алгоритмом. Однако происходит в точности противоположное. Так как компьютеры работают все быстрее и мы можем решать все большие задачи, именно сложность алгоритма определяет то увеличение размера задачи, которое можно достичь с увеличением скорости машины.

Следуя [2], рассмотрим это более подробно. Допустим, у нас есть пять алгоритмов A_1–A_5 со следующими временными сложностями:

Алгоритм	Асимптотическая временная сложность
A_1	O(n)
A_2	O($n \log n$)
A_3	O(n^2)
A_4	O(n^3)
A_5	O(2^n)

Пусть единицей времени будет одна миллисекунда и мультипликативные константы в точных оценках временной сложности во всех алгоритмах равны 1. Тогда алгоритм A_1 может обработать за одну секунду вход размера 1000, в то время как A_5 – вход размера не более 9. В таблице 1 приведены размеры задач, которые можно решить за одну секунду, одну минуту и один час каждым из этих пяти алгоритмов.

Таблица 1. Границы размеров задач, определяемые скоростью роста сложности.

Алгоритм	Асимптотическая временная сложность	Максимальный размер задачи		
		1 с	1 мин	1 ч
A_1	O(n)	1000	6×10^4	$3{,}6\times10^6$
A_2	O($n \log n$)	140	4893	$2{,}0\times10^5$
A_3	O(n^2)	31	244	1897
A_4	O(n^3)	10	39	153
A_5	O(2^n)	9	15	21

Предположим, что следующее поколение компьютеров будет в 10 раз быстрее нынешнего. В таблице 2 показано, как возрастут размеры задач, которые мы сможем решить благодаря этому увеличению скорости.

Таблица 2. Эффект десятикратного ускорения

Алгоритм	Асимптотическая временная сложность	Максимальный размер задачи	
		До ускорения	После ускорения
A_1	O(n)	S_1	10 S_1

A_2	O($n \log n$)	S_2	Примерно 10 S_2 для больших S_2
A_3	O(n^2)	S_3	3,16 S_3
A_4	O(n^3)	S_4	2,15 S_4
A_5	O(2^n)	S_5	S_5 + 3,3

Заметим, что для алгоритма A_5 десятикратное увеличение скорости увеличивает размер задачи, которую можно решить, только на три, тогда как для алгоритма A_3 размер задачи более чем утраивается.

Вместо эффекта увеличения скорости рассмотрим теперь эффект применения более действенного алгоритма. Вернемся к таблице 1. Если в качестве основы для сравнения взять 1 мин, то, заменяя алгоритм A_4 алгоритмом A_3, можно решить задачу в 6 раз большую, а заменяя A_4 на A_2, можно решить задачу, большую в 125 раз. Эти результаты производят гораздо большее впечатление, чем двукратное улучшение, достигаемое за счет десятикратного увеличения скорости. Если в качестве основы для сравнения взять 1 ч, то различие оказывается еще значительнее. Отсюда мы заключаем, что асимптотическая скорость алгоритма служит важной мерой качественности алгоритма, причем такой мерой, которая обещает стать еще важнее при последующем увеличении скорости вычислений.

Несмотря на то, что основное внимание здесь уделяется порядку роста величин, надо понимать, что больший порядок сложности алгоритма может иметь меньшую мультипликативную постоянную, чем малый порядок роста сложности другого алгоритма. В таком случае алгоритм с быстро растущей сложностью может оказать предпочтительнее для задач с малым размером – возможно, даже для всех задач, которые нас интересуют.

3.3. Сложность задач

Сложность задачи – это асимптотическая временная сложность наилучшего алгоритма, известного для ее решения.

Основной вопрос теории сложности: насколько успешно или с какой стоимостью может быть решена заданная проблема Q? Мы не имеем в виду никакого конкретного алгоритма решения Q. Наша цель – рассмотреть все возможные алгоритмы решения Q и попытаться сформулировать утверждение о вычислительной сложности, внутренне присущей Q. В то время как всякий алгоритм A для Q дает верхнюю оценку величины сложности Q, нас интересует нижняя оценка. Знание нижней оценки представляет интерес математически и, кроме того, руководит нами в поиске хороших алгоритмов, указывая, какие попытки заведомо будут безуспешны.

Быстрыми являются линейные алгоритмы, которые обладают сложностью порядка O(n), где n – размерность входных данных. К линейным алгоритмам относится школьный алгоритм нахождения суммы десятичных чисел, состоящих из n_1 и n_2 цифр. Сложность этого алгоритма – O($n_1 + n_2$). Есть алгоритмы, которые быстрее линейных, например, алгоритм двоичного поиска в линейном упорядоченном массиве имеет сложность O(log n), n – длина массива.

Другие хорошо известные алгоритмы – деление, извлечение квадратного корня, решение систем линейных уравнений и др. – попадают в более общий класс полиномиальных алгоритмов.

Полиномиальным алгоритмом (или алгоритмом полиномиальной временной сложности, или алгоритмом принадлежащим классу **P**) называется алгоритм, у которого временная сложность равна O(n^k), где k – положительное целое число. Алгоритмы, для временной сложности которых не существует такой оценки, называются *экспоненциальными* и такие задачи считаются *труднорешаемыми*. Понятие полиномиально разрешимой зада-

чи принято считать уточнением идеи «практически разрешимой» задачи. Чем объясняется такое соглашение?

Во-первых, используемые на практике полиномиальные алгоритмы обычно действительно работают довольно быстро. Конечно, трудно назвать практически разрешимой задачу, которая требует времени $\Theta(n^{100})$. Однако полиномы такой степени в реальных задачах почти не встречаются.

Второй аргумент в пользу рассмотрения класса полиномиальных алгоритмов – тот факт, что объем этого класса не зависит от выбора конкретной модели вычислений (для достаточно широкого класса моделей) [9]. Например, класс задач, которые могут быть решены за полиномиальное время на последовательной машине с произвольным доступом (RAM), совпадает с классом задач, полиномиально разрешимых на машинах Тьюринга. Класс будет тем же и для моделей параллельных вычислений, если, конечно, число процессоров ограничено полиномом от длины входа.

В-третьих, класс полиномиально разрешимых задач обладает естественными свойствами замкнутости. Например, композиция двух полиномиальных алгоритмов (выход первого алгоритма подается на вход второго) также работает полиномиальное время. Объясняется это тем, что сумма, произведение и композиция многочленов снова есть многочлен.

Приведем примеры классификации задач по их сложности.

Класс P

- Рассортировать множество из n чисел. Сложность поведения в среднем порядка $O(n \log n)$ для быстрого алгоритма Хоара [18, стр.316–321].
- Найти эйлеровый цикл на графе из m ребер. В силу теоремы Эйлера мы имеем необходимое и достаточное условие для существования эйлерова цикла и проверка этого условия есть алгоритм порядка $O(m)$.
- Задача Прима-Краскала. *Дана плоская страна и в ней n городов. Нужно соединить все города телефонной связью так, чтобы общая длина телефонных линий была минимальной.* В терминах теории графов задача Прима-Краскала выглядит следующим образом: *Дан граф с n вершинами; длины ребер заданы матрицей* $(d[i,j])$, $i,j = 1,...,n$. *Найти остовное дерево минимальной длины.* Эта задача решается с помощью жадного алгоритма сложности $O(n \log n)$[18, стр.357–358].
- Кратчайший путь на графе, состоящем из n вершин и m ребер. Сложность алгоритма $O(m\,n)$ [18, стр.377–382].
- Связные компоненты графа. Определяются подмножества вершин в графе (связные компоненты), такие, что две вершины, принадлежащие одной и той же компоненте, всегда связаны цепочкой дуг. Если n – количество вершин, а m – количество ребер, то сложность алгоритма $O(n+m)$ [18, стр.364–365].
- Быстрое преобразование Фурье [2, стр. 284–302], требующее $O(n \log n)$ арифметических операций, – один из наиболее часто используемых алгоритмов в научных вычислениях.
- Умножение целых чисел. Алгоритм Шёнхаге-Штрассена [2, стр. 304–308]. Сложность алгоритма порядка $O(n \log n \log \log n)$. Отметим, что школьный метод для умножения двух n-разрядных чисел имеет сложность порядка $O(n^2)$.
- Умножение матриц. Алгоритм Штрассена [2, стр. 259–261] имеет сложность порядка $O(n^{\log 7})$, для умножения двух матриц размера $n \times n$. Очевидный алгоритм имеет порядок сложности $O(n^3)$.

Класс E: задачи, экспоненциальные по природе

К экспоненциальным задачам относятся задачи, в которых требуется построить множество всех подмножеств данного множества, все полные подграфы некоторого графа или же все поддеревья некоторого графа.

Существует масса примеров задач с экспоненциальной сложностью. Например, чтобы вычислить $2^{(2^k)}$ для заданного натурального k, нам только для записи конечного ответа потребуется около 2^n шагов (где n – число цифр в двоичной записи k), не говоря даже о самом вычислении.

Задачи не попадающие ни в класс P, ни в класс E

На практике существуют задачи, которые заранее не могут быть отнесены ни к одному из рассмотренных выше классов. Хотя в их условиях не содержатся экспоненциальные вычисления, однако для многих из них до сих пор не разработан эффективный (т.е. полиномиальный) алгоритм.

К этому классу относятся следующие задачи [13, с. 207]:

- задача о выполнимости: существует ли для данной булевской формулы, находящейся в КНФ, такое распределение истинностных значений, что она имеет значение истина?
- задача коммивояжера (Коммивояжер хочет объехать все города, побывав в каждом ровно по одному разу, и вернуться в город, из которого начато путешествие. Известно, что переезд из города i в город j стоит $c(i, j)$ рублей. Требуется найти путь минимальной стоимости.);
- решение систем уравнений с целыми переменными;
- составление расписаний, учитывающих определенные условия;
- размещение обслуживающих центров (телефон, телевидение, срочные службы) для максимального числа клиентов при минимальном числе центров;
- оптимальная загрузка емкости (рюкзак, поезд, корабль, самолёт) при наименьшей стоимости;
- оптимальный раскрой (бумага, картон, стальной прокат, отливки), оптимизация маршрутов в воздушном пространстве, инвестиций, станочного парка;
- задача распознавания простого числа; самый лучший в настоящее время тест на простоту имеет сложность порядка $O(L(n)^{L(L(L(n)))})$, где $L(n)$ – количество цифр в числе n (выражение $L(L(L(n)))$ стремиться к бесконечности очень медленно; первое число, для которого $L(L(L(n))) = 2$, равно $10^{999999999}$) [1, стр. 102].

4. NP-полнота

В этом разделе мы рассмотрим класс задач, называемых «***NP***-полными». Для этих задач не найдены полиномиальные алгоритмы, однако не доказано, что таких алгоритмов не существует. Примеры таких задач приведены в конце предыдущего раздела. Изучение ***NP***-полных задач связано с так называемым вопросом ***P≠NP***. Этот вопрос был поставлен в 1971 году и является сейчас одной из наиболее интересных и сложных проблем теории вычислений.

Большинство специалистов полагают, что ***NP***-полные задачи нельзя решить за полиномиальное время. Дело в том, что если хотя бы для одной ***NP***-полной задачи существует решающий ее полиномиальный алгоритм, то и для всех ***NP***-полных задач такие алгоритмы существуют. В настоящее время известно очень много ***NP***-полных задач – многие из них практически важные. Все попытки найти для них полиномиальные алгоритмы оказались безуспешными. По-видимому, таких алгоритмов нет вовсе.

Зачем программисту знать о ***NP***-полных задачах? Если для некоторой задачи удается доказать ее ***NP***-полноту, есть основания считать ее практически неразрешимой. В этом случае лучше потратить время на построение приближенного алгоритма, чем продолжать искать быстрый алгоритм, решающий ее точно.

Содержание раздела 4 взято из [9]. Для понимания материала могут потребоваться элементарные понятия теории графов. См., например, [2].

4.1. Задачи разрешения и задачи оптимизации

В теории ***NP***-полноты рассматриваются только **задачи разрешения** – задачи, в которых требуется дать ответ «да» или «нет» на некоторый вопрос. Формально задачу разрешения можно рассматривать как функцию, отображающую множество условий I во множество $\{0,1\}$ (1 = «да», 0 = «нет»). Многие задачи можно тем или иным способом преобразовать к такому виду. Например, с задачей SHORTEST-PATH поиска кратчайшего пути в графе связана задача разрешения PATH: «по заданному графу $G = (V, E)$, паре вершин u, $v \in V$ и натуральному числу k определить, существует ли в G путь между вершинами u и v, длина которого не превосходит k». Пусть четверка $i = \langle G, u, v, k\rangle$ является условием задачи PATH. Тогда PATH(i) = 1, если длина кратчайшего пути между вершинами u и v не превосходит k, и PATH(i) = 0 в противном случае.

Часто встречаются **задачи оптимизации**, в которых требуется минимизировать или максимизировать значение некоторой величины. Прежде чем ставить вопрос о ***NP***-полноте таких задач, их следует преобразовать в задачу разрешения. Обычно в качестве такой задачи разрешения рассматривают задачу проверки, является ли некоторое число верхней (или нижней) границей для оптимизируемой величины. Так, например, мы перешли от задачи оптимизации SHORTEST-PATH к задаче разрешения PATH, добавив в условие задачи границу длины пути k.

Если после этого получается задача, не имеющая полиномиального алгоритма разрешения, то тем самым установлена трудность исходной задачи. В самом деле, если для оптимизационной задачи имеется быстрый алгоритм, то и полученную из неё задачу разрешения можно решить быстро (надо просто сравнить ответ этого алгоритма с заданной границей). Таким образом, можно исследовать временную сложность задач оптимизации.

Мы будем использовать строковое представление задач. Для каждого представления e множества I входов абстрактной задачи Q мы получаем свою строковую задачу, которую мы в дальнейшем обозначаем $e(Q)$. Мы не будем подробно описывать используемое представление в конкретных задачах, считая, что оно выбрано достаточно разумно и экономно (целые числа задаются двоичной записью, конечные множества – списком элементов и т.п.). Нам понадобятся также следующие определения.

Будем говорить, что функция f: $\{0,1\}^* \to \{0,1\}^*$ ($\{0,1\}^*$ обозначает множество битовых строк) **вычислима за полиномиальное время**, если существует полиномиальный алгоритм A, который для любого $x \in \{0, 1\}^*$ выдает результат $f(x)$.

Рассмотрим теперь множество I условий произвольной абстрактной задачи разрешения. Два представления e_1 и e_2 этого множества называются **полиномиально связанными**, если существуют две вычислимые за полиномиальное время функции f_{12} и f_{21}, для которых $f_{12}(e_1(i)) = e_2(i)$ и $f_{21}(e_2(i)) = e_1(i)$ для всякого $i \in I$. Это значит, что e_1–представление входа может быть за полиномиальное время получено из e_2–представления и наоборот. В этом случае не имеет значения, какое из двух полиномиально связанных представлений выбрать, как показывает следующая теорема.

Теорема 11. Пусть Q – абстрактная задача разрешения с множеством условий I, а e_1 и e_2 – полиномиально связанные представления для элементов множества I. Предположим, что множество всех строк, которые являются e_1–представлениями элементов Q, разрешимо за полиномиальное время, и что аналогичное свойство выполнено для представления e_2. Тогда свойства $e_1(Q) \in P$ и $e_2(Q) \in P$ равносильны.

Доказательство. Утверждение симметрично, так что достаточно доказать его в одну сторону. Предположим, что задача $e_1(Q)$ разрешима за время $O(n^k)$ для некоторого фиксированного числа k. По предположению для всякого условия $i \in I$ представление $e_1(i)$ может быть получено из представления $e_2(i)$ за время $O(n^c)$ (где c – некоторая константа, $n = |e_2(i)|$, $|s|$ – обозначает длину строки s). Для решения задачи $e_2(Q)$, получив на входе $e_2(i)$, мы сперва вычислим $e_1(i)$, а затем применим алгоритм, разрешающий $e_1(Q)$ к строке $e_1(i)$. Сколько времени займет наше вычисление? Преобразование $e_2(i)$ в $e_1(i)$ требует полиномиального времени. Следовательно, $|e_1(i)| = O(n^c)$, поскольку длина выхода алгоритма не превосходит времени его работы. Решение задачи с условием $e_1(i)$ занимает $O(|e_1(i)|^k) = O(n^{ck})$ времени. Итак, время вычисления оказалось полиномиальным. (Мы пропустили важный момент: получив на входе некоторую строку, мы должны сначала проверить, что она является e_2–представлением некоторого входа; по предположению это можно сделать за полиномиальное время.)

Представление объекта будем обозначать угловыми скобками: $<G>$ – это стандартное представление объекта G. При этом множество всех строк, являющихся представлениями, является полиномиально разрешимым (существует полиномиальный алгоритм, проверяющий по строке, представляет ли она какой-либо объект), а различные «разумные» способы представления данных оказываются полиномиально связанными, так что можно воспользоваться теоремой 11 и не описывать представление детально, если нас интересует лишь вопрос о полиномиальности задачи. Таким образом, в дальнейшем мы не будем делать различия между абстрактной задачей и её строковым представлением, как это обычно и делают.

4.2. Формальные языки

Для задач разрешения удобно использовать терминологию теории формальных языков. **Алфавитом** Σ называется любой конечный набор символов. **Языком** L **над алфавитом** Σ называется произвольное множество строк символов из алфавита Σ (такие строки называются **словами** в алфавите Σ). Например, можно рассмотреть $\Sigma = \{0,1\}$ и язык $L = \{10, 11, 101, 111, 1011, 1101, 10001,\ldots\}$, состоящий из двоичных записей простых чисел.

Задача разрешения (точнее, соответствующая ей строковая задача разрешения) является языком над алфавитом $\Sigma = \{0,1\}$.

Например, задаче PATH соответствует язык PATH = $\{<G, u, v, k>: G = (V,E)$ – неориентированный граф, $u, v \in V, k \geq 0$ – целое число, и в графе G существует путь из u в v, длина которого не превосходит $k\}$.

Мы будем использовать одно и то же название – в данном случае PATH – для обозначения задачи и соответствующего языка.

Алгоритм допускает слово

Говорят, что алгоритм A *допускает слово* $x \in \{0, 1\}^*$, если на входе x алгоритм выдает результат 1 ($A(x) = 1$).

Алгоритм отвергает слово

Говорят, что алгоритм A *отвергает слово* $x \in \{0, 1\}^*$, если на входе x алгоритм выдает результат 0 ($A(x) = 0$).

Заметим, что алгоритм может не остановиться на входе x или дать ответ, отличный от 0 и 1. В этом случае он и не допускает и не отвергает слово x.

Алгоритм допускает язык

Говорят, что алгоритм *A допускает язык L*, если алгоритм допускает те и только те слова, которые принадлежат языку *L*.

Алгоритм *A*, допускающий некоторый язык *L*, не обязан отвергать всякое слово $x \notin L$.

Алгоритм распознает язык

Говорят, что алгоритм *A распознает язык L*, если *A* допускает все слова из *L*, а все остальные слова отвергает.

Язык допускается за полиномиальное время

Язык *L допускается за полиномиальное время*, если имеется алгоритм *A*, который допускает данный язык, причем всякое слово $x \in L$ допускается алгоритмом за время $O(n^k)$, где *n* – длина слова *x*, а *k* – некоторое не зависящее от *x* число.

Язык распознается за полиномиальное время

Язык *L распознается за полиномиальное время*, если имеется алгоритм *A*, который распознает данный язык, причем время работы алгоритма на каждом слове длины *n* не больше $O(n^k)$.

Теперь можно переформулировать определение сложностного класса ***P***.

$P = \{L \subset \{0, 1\}^*$: существует алгоритм *A*, распознающий язык *L* за полиномиальное время$\}$.

На самом деле в данной ситуации нет разницы между языками, допускаемыми и распознаваемыми за полиномиальное время.

Теорема 12. $P = \{L$: *L* допускается за полиномиальное время$\}$.

Доказательство. Если язык распознается некоторым алгоритмом, то он и допускается тем же алгоритмом. Остается доказать, что если язык *L* допускается полиномиальным алгоритмом *A*, то он распознается некоторым (возможно, другим) полиномиальным алгоритмом *B*. Пусть алгоритм *A* допускает язык *L* за время $O(n^k)$. Это значит, что существует константа *c*, для которой *A* допускает любое слово длины *n* из *L*, сделав не более $T = cn^k$ шагов. (Формально говоря, это верно для достаточно длинных слов *x*; мы опускаем очевидные детали.) С другой стороны, слова не из *L* алгоритм не допускает (ни за какое время).

Новый алгоритм *B* моделирует работу алгоритма *A* и считает число шагов этого алгоритма, сравнивая его с известной границей *T*. Если за время *T* алгоритм *A* допускает слово *x*, алгоритм *B* также допускает это слово и выдает 1. Если же *A* не допускает *x* за указанное время, алгоритм *B* прекращает моделирование и отвергает слово (выдает 0). Замедление работы за счёт моделирования и подсчета шагов не так уж велико и оставляет время работы полиномиальным.

4.3. Проверка принадлежности языку и класс *NP*

Для определения сложностного класса ***NP*** нам потребуется ряд новых понятий, в том числе рассмотрение задачи о гамильтоновом цикле.

Гамильтоновым циклом в неориентированном графе $G = (V, E)$ называется простой цикл, который проходит через все вершины графа. Графы, в которых есть гамильтонов цикл, называются **гамильтоновыми**.

Задача о гамильтоновом цикле состоит в выяснении, имеет ли данный граф G гамильтонов цикл. Формально говоря,

$$\text{HAM-CYCLE} = \{<G>: G \text{ – гамильтонов граф}\}.$$

Как решать такую задачу? Можно перебрать все перестановки вершин данного графа и проверить, является ли хотя бы одна из них гамильтоновым циклом. Оценим время работы такого алгоритма. Если мы используем представление графа с помощью матрицы инцидентности, то число вершин m в графе будет $\Omega(\sqrt{n})$, где $n = |<G>|$ – длина представления графа G. Имеется $m!$ различных перестановок вершин графа, и время работы алгоритма равно $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, т.е. не является полиномиальным. Таким образом, наивный алгоритм не дает эффективного решения задачи. (На самом деле, есть основания предполагать, что полиномиального алгоритма для неё вообще не существует.)

Пусть вы заключили пари с приятелем, который утверждает, что (нарисованный перед вами на доске) граф является гамильтоновым. При этом вы не можете быстро проверить так это или нет. Тем не менее приятель может выиграть пари, если каким-то образом отгадает гамильтонов цикл и предъявит его вам: проверка того, что данный цикл является гамильтоновым, проста. Нужно лишь проверить, что предъявленный цикл проходит через все вершины графа и что он действительно идет по рёбрам.

Проверяющий алгоритм

- Назовем проверяющим алгоритмом алгоритм A с двумя аргументами; первый аргумент мы будем называть (как и раньше) входной строкой, а второй – **сертификатом**. Мы говорим, что алгоритм A с двумя аргументами **допускает** вход x, если существует сертификат y, для которого $A(x, y) = 1$.
- **Языком, проверяемым алгоритмом** A, мы назовем язык

$L = \{x \in \{0, 1\}^*: \exists y \in \{0, 1\}^*$, для которого $A(x, y) = 1\}$.

Другими словами, алгоритм A проверяет язык L, если для любой строки $x \in L$ найдется сертификат y, с помощью которого A может проверить принадлежность x к языку L, а для $x \notin L$ такого сертификата нет. Например, в задаче HAM-CYCLE сертификатом была последовательность вершин, образующая гамильтонов цикл.

Класс *NP*

Сложностный класс ***NP*** – это класс языков, для которых существуют проверяющие алгоритмы, работающие полиномиальное время, причем длина сертификата также ограничена некоторым полиномом.

Более точно, язык L принадлежит классу ***NP***, если существует такой полиномиальный алгоритм A с двумя аргументами и такой многочлен $p(x)$ с целыми коэффициентами, что

$L = \{x \in \{0, 1\}^*:$ существует сертификат y, для которого $|y| \le p(|x|)$ и $A(x, y) = 1\}$.

В этом случае мы говорим, что алгоритм A **проверяет язык L за полиномиальное время**.

Несколько слов о названии: сокращение ***NP*** происходит от английских слов Nondeterministic Polynomial (time), что переводится как недетерминированное полиномиальное время. Первоначально класс ***NP*** определялся в терминах так называемых недетерминированных вычислений.

Мы уже знаем одну задачу из класса ***NP*** – это задача HAM-CYCLE. Кроме того, всякая задача из ***P*** принадлежит также и ***NP***. Действительно, если есть полиномиальный алгоритм, распознающий язык, то легко построить проверяющий алгоритм для того же

языка – проверяющий алгоритм может просто игнорировать свой второй аргумент (сертификат). Таким образом, $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

В данное время неизвестно, совпадают ли классы ***P*** и ***NP***, но большинство специалистов полагают, что нет.

Интуитивно класс ***P*** можно представлять себе как класс задач, которые можно быстро решить, а класс ***NP*** – как класс задач, решение которых может быть быстро проверено.

На практике решить самому задачу часто намного труднее, чем проверить уже готовое решение, особенно если время работы ограничено. По аналогии можно думать, что в классе ***NP*** имеются задачи, которые нельзя решить за полиномиальное время.

4.4. *NP*-полнота и сводимость

По-видимому, наиболее убедительным аргументом в пользу того, что классы ***P*** и ***NP*** различны, является существование так называемых ***NP***-полных задач. Этот класс обладает удивительным свойством: если какая-нибудь ***NP***-полная задача разрешима за полиномиальное время, то и все задачи из класса ***NP*** разрешимы за полиномиальное время, т.е. $\boldsymbol{P} = \boldsymbol{NP}$. Несмотря на многолетние исследования, ни для одной ***NP***-полной задачи не найден полиномиальный разрешающий алгоритм.

В частности, задача HAM-CYCLE (о гамильтоновом цикле) является ***NP***-полной. Таким образом, научившись решать её за полиномиальное время, мы получим полиномиальные алгоритмы для всех задач класса ***NP***. Переформулировка: если множество $\boldsymbol{NP} \backslash \boldsymbol{P}$ не пусто, то HAM-CYCLE $\in \boldsymbol{NP} \backslash \boldsymbol{P}$.

Неформально говоря, ***NP***-полные языки являются самыми «трудными» в классе ***NP***. При этом трудность языков можно сравнивать, сводя один язык к другому. В этом разделе мы даём определение сводимости, затем определяем класс ***NP***-полных задач и устанавливаем полноту многих задач.

Говоря неформально, задача Q сводится к задаче R, если задачу Q можно решить для любого входа, считая известным решение задачи R для какого-то другого входа. Например, задача решения линейного уравнения сводится к задаче решения квадратного уравнения (линейное уравнение можно превратить в квадратное, добавив фиктивный старший член). Если задача Q сводится к задаче R, то любой алгоритм, решающий R, можно использовать для решения задачи Q, т.е. Q «не труднее» R.

Сводимость за полиномиальное время

Говорят, что язык L_1 сводится за полиномиальное время к языку L_2 (запись: $L_1 \leq_P L_2$), если существует такая функция f: $\{0, 1\}^* \to \{0, 1\}^*$, вычислимая за полиномиальное время, что для любого $x \in \{0, 1\}^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

Функцию f называют **сводящей функцие**й, а полиномиальный алгоритм F, вычисляющий функцию f, – **сводящим алгоритмом**.

Теорема 12. Если язык $L_1 \subseteq \{0, 1\}^*$ сводится за полиномиальное время к языку $L_2 \subseteq \{0, 1\}^*$ и $L_2 \in \boldsymbol{P}$, то $L_1 \in \boldsymbol{P}$.

Доказательство. Пусть A_2 – полиномиальный алгоритм, распознающий язык L_2, а F – полиномиальный алгоритм, сводящий язык L_1 к языку L_2. Построим алгоритм A_1, который будет за полиномиальное время разрешать язык L_1.

Рисунок 4 иллюстрирует построение. Получив вход $x \in \{0, 1\}^*$, алгоритм A_1 (с помощью алгоритма F) получает $f(x)$ и с помощью алгоритма A_2 проверяет, принадлежит ли

слово $f(x)$ языку L_2. Результат работы алгоритма A_2 на слове $f(x)$ и выдается алгоритмом A_1 в качестве ответа.

Определение полиномиальной сводимости гарантирует, что алгоритм A_1 дает правильный ответ; он полиномиален, поскольку полиномиальны алгоритмы F и A_1.

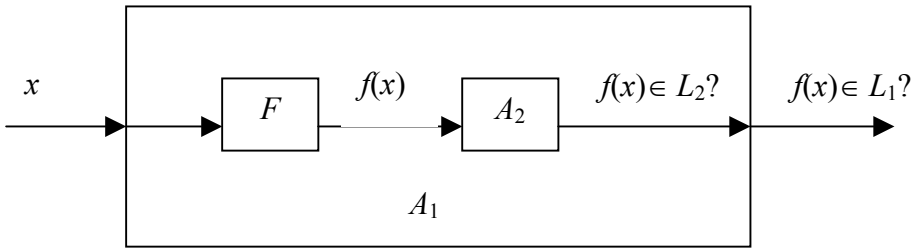

Рис. 1. Принадлежность слова x к языку L_1 можно проверить, использовав сводящий алгоритм F и алгоритм A_2, разрешающий язык L_2.

Понятие сводимости позволяет придать точный смысл утверждению о том, что один язык не менее труден, чем другой (с точностью до полинома). Запись $L_1 \leq_P L_2$ можно интерпретировать так: сложность языка L_1 не более чем полиномиально превосходит сложность языка L_2. Наиболее трудны в этом смысле ***NP***-полные задачи.

***NP*-полнота**
Язык $L \subseteq \{0, 1\}^*$ называется ***NP***-полным, если 1) $L \in$ ***NP***; 2) $L' \leq_P L$ для любого $L' \in$ ***NP***.

Класс ***NP***-полных языков будем обозначать ***NPC***.

Основное свойство ***NP***-полных языков состоит в следующем.

Теорема 13. Если некоторая ***NP***-полная задача разрешима за полиномиальное время, то ***P*** = ***NP***. Другими словами, если в классе ***NP*** существует задача, не разрешимая за полиномиальное время, то все ***NP***-полные задачи таковы.

Доказательство. Пусть L – ***NP***-полный язык, который одновременно оказался разрешимым за полиномиальное время ($L \in$ ***P*** и $L \in$ ***NPC***). Тогда для любого языка L' по свойству 2 определения ***NP***-полного языка имеем $L' \leq_P L$. Следовательно, $L' \in$ ***P*** (теорема 12), и первое утверждение теоремы доказано.

Второе утверждение теоремы является переформулировкой первого.

Таким образом, гипотеза ***P*** ≠ ***NP*** означает, что ***NP***-полные задачи не могут быть решены за полиномиальное время. Большинство экспертов полагают, что это действительно так; предполагаемое соотношение между классами ***P***, ***NP*** и ***NPC*** показано на рис. 2.

Конечно, мы не можем быть уверены, что однажды кто-нибудь не предъявит полиномиальный алгоритм для решения ***NP***-полной задачи и не докажет тем самым, что ***P***=***NP***. Но пока этого никому не удалось, и поэтому доказательство ***NP***-полноты некоторой задачи является убедительным аргументом в пользу того, что она является практически неразрешимой.

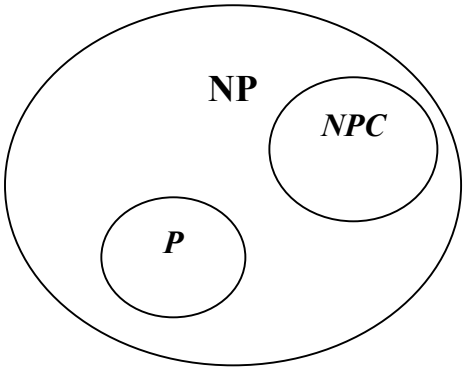

Рис. 2. Предполагаемое соотношение между классами ***P***, ***NP*** и ***NPC***.
Классы ***P*** и ***NPC*** содержатся в ***NP*** (что очевидно), и, можно полагать, не пересекаются и не покрывают всего ***NP***.

Первой задачей, для которой была доказана её ***NP***-полнота была задача о выполнимости пропозициональных формул:

SAT = {<φ>: φ – пропозициональная и выполнимая формула}.

В неформальной постановке:

Условие. Дана формула исчисления высказываний φ.

Вопрос. Существует ли такое распределение истинностных значений высказывательных переменных, при которых формула φ выполнима?

Теорема 14 (теорема Кука). Задача SAT является ***NP***-полной [6, с.56–63].

Можно сказать так: после доказательства этой теоремы была пробита брешь в стене и установлена ***NP***-полнота одной задачи, после этого было доказано с помощью сводимости ***NP***-полнота большого числа задач [6, 9]. В разделе 3 мы перечислили некоторые задачи, которые не попадают ни в класс ***P***, ни в класс ***E***. Все они являются ***NP***- полными.

Литература

1. Акритас А. Основы компьютерной алгебры с приложениями: пер. с англ. – М.: Мир, 1994. – 544 с.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. – 536с.
3. Барендрегт Х. Ламбда–исчисление. Его синтаксис и семантика. – М.: Мир, 1985.– 606с.
4. Верещагин Н. К., Шень А. Лекции по математической логике и теории алгоритмов. Часть 2. Языки и исчисления. М.: МЦНМО, 2000.–288 с.
5. Грэй П. Логика, алгебра и базы данных: Пер. с англ.– М.: Машиностроение, 1989. – 359с.
6. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982.
7. Катленд Н. Вычислимость. Введение в теорию рекурсивных функций: Пер. с англ.–М.: Мир, 1983.–256с.
8. Кац М., Улам С. Математика и логика. Ретроспектива и перспективы: Пер. с англ. – М.: Мир, 1971. –254с.
9. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 2001.– 960с.

10. Кроновер Р. М. Фракталы и хаос в динамических системах. Основы теории. – М.: Постмаркет, 2000. – 352 с.
11. Логический подход к искусственному интеллекту: От классической логики к логическому программированию: Пер. с франц. / Тейз А., Грибоман П., Луи Ж и др.– М.: Мир, 1990. – 432с.
12. Логический подход к искусственному интеллекту: От модальных логики к логике баз данных: Пер. с франц. / Тейз А., Грибоман П., Юлен Г. и др.– М.: Мир, 1998. – 496с.
13. Лорьер Ж.-Л. Системы искусственного интеллекта – М.: Мир, 1991. – 568с.
14. Манин Ю. И. Вычислимое и невычислимое. – М.: «Советское радио», 1980. – 128с.
15. Манин Ю. И. Доказуемое и недоказуемое. – М.: «Советское радио», 1979. – 168с.
16. Мендельсон Э. Введение в математическую логику - М.: Наука, 1976.- 320с.
17. Подниекс К. М. Вокруг теоремы Гёделя. Рига: «Зинатне», 1992. – 191с.
18. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. Пер. с англ. – М.: Мир, 1980. – 478 с.
19. Смаллиан Р. Как же называется эта книга? – М.: Мир, 1981.– 238с.
20. Смаллиан Р. Принцесса или тигр? – М.: Мир, 1985. – 221с.
21. Справочная книга по математической логике: В 4-х частях /Под ред. Дж. Барвайса. – Ч. IV. Теория доказательств и конструктивная математика: Пер. с англ.– М.: Наука, 1983. – 392с.
22. Справочная книга по математической логике: В 4-х частях /Под ред. Дж. Барвайса. – Ч. III. Теория рекурсии: Пер. с англ.– М.: Наука, 1982.– 360с.
23. Успенский В. А. Теорема Гёделя о неполноте – М.: Наука, 1982. – 112с.
24. Хофштадтер Д. Гёдель, Эшер, Бах: эта бесконечная гирлянда. – Самара: Издательский Дом «Бахрах-М», 2001. – 752 с.
25. Goodstein, R. L. «On the Restricted Ordinal Theorem.» *J. Symb. Logic*, **9** (1944), 33–41.
26. Kirbi L., Paris J. «Accessible independence result for Peano arithmetic.» *Bulletin of the London Mathematical Society*, **14** (1982), 285–93.