Алгоритмическая проблема – это проблема, в которой требуется найти единственный метод (алгоритм) для решения бесконечной серии однотипных единичных задач. Такие проблемы называют также **массовыми проблемами**. Массовая проблема представляет собой бесконечную серию индивидуальных задач.

Массовая проблема *П* является **алгоритмически разрешимой**, если соответствующая характеристическая функция *f*, которая определяется соотношением

$$f(\pi)=\begin{cases}1 \Leftrightarrow \text{инд.задача } \pi \in \Pi \text{ имеет ответ "Да"}\\ 0 \Leftrightarrow \text{инд.задача } \pi \notin \Pi \text{ имеет ответ "Нет"},\end{cases}$$

является вычислимой.

Решая конкретную массовую проблему, следует иметь в виду, что она может оказаться алгоритмически неразрешимой, поэтому необходимо иметь представление о технике доказательства неразрешимости.

7.1. Нумерация алгоритмов

Понятие нумерации алгоритмов – важное средство для их исследования, в частности для доказательств несуществования единого алгоритма для решения той или иной массовой проблемы.

Поскольку любой алгоритм можно задать конечным описанием (словом), а множество всех конечных слов в фиксированном конечном алфавите счетно, то множество всех алгоритмов счетно.

Это означает наличие взаимно однозначного соответствия между множеством $\boldsymbol{N}$ натуральных чисел и множеством всех алгоритмов, рассматриваемых как подмножество множества A^* всех слов в алфавите A, выбранном для описания алгоритмов (φ: $N \rightarrow A^*$).

Такая функция называется **нумерацией алгоритмов**.

Если $\varphi(n) = A$, то число n называется **номером алгоритма *A***.

Из взаимной однозначности отображения φ следует существование обратной функции φ^{-1}, восстанавливающей по описанию алгоритма A_n его номер в этой нумерации $\varphi^{-1}(A_n) = n$. Очевидно, что различных нумераций много.

Нумерация всех алгоритмов является одновременно и нумерацией всех вычислимых функций в следующем смысле: номер функции f – это номер некоторого алгоритма, вычисляющего f. Ясно, что в любой нумерации всякая функция будет иметь бесконечное множество различных номеров.

Существование нумераций позволяет работать с алгоритмами как с числами. Это особенно удобно при исследовании алгоритмов над алгоритмами. Отсутствие именно таких алгоритмов часто приводит к алгоритмически неразрешимым проблемам.

7.2. Нумерация машин Тьюринга

Опишем теперь более конкретный процесс нумерации всех машин Тьюринга.

Зафиксируем счетные множества символов

$$\{a_0, a_1, \ldots, a_i, \ldots\} \text{ и } \{q_0, q_1, \ldots, q_j, \ldots\}$$

и будем считать, что внешние алфавиты и алфавиты внутренних состояний всех машин Тьюринга берутся из этих множеств.

При этом будем считать, что a_0 принадлежит всем внешним алфавитам машин и интерпретируется как пустой символ, а буквы q_1, q_0 принадлежат всем внутренним алфавитам машин и всегда означают заключительное и начальное состояния, соответственно.

Опишем теперь единый способ представления информации о машинах с помощью кодирования.

Каждому символу из множества $\{L, R, E, a_0, a_1, \ldots, a_i, \ldots, q_0, q_1, \ldots, q_j, \ldots\}$ поставим в соответствие двоичный набор согласно таблице:

	Символ	Код	Количество нулей в коде
Символы сдвига	R	10	1
	L	100	2
	E	1000	3
Символы алфавита ленты	a_0	10000	4
	a_1	1000000	6
	…	…	…
	a_i	100…00	$2i+4$
	…	…	…
Символы алфавита состояний	q_0	100000	5
	q_1	10000000	7
	…	…	…
	q_i	100…00	$2j+5$
	…	…	…

Далее, команде I машины Тьюринга T, имеющей вид $qa \Rightarrow q'a'\alpha_k$, ставится в соответствие двоичный набор вида

$$\text{Код}(I)=\text{Код}(q)\text{Код}(a)\text{Код}(q')\text{Код}(a')\text{Код}(\alpha_k), \qquad \textbf{(7.1)}$$

в котором коды букв приписаны друг к другу.

Пусть машина T имеет алфавиты

$$A=\{a_0, a_1, \ldots, a_i, \ldots\} \text{ и } Q=\{q_0, q_1, \ldots, q_j, \ldots\}.$$

Упорядочим команды машины T в соответствии с лексикографическим порядком левых частей команд:

$$q_1, a_0, q_1, a_1, \ldots, q_1, a_m, q_2, a_0, q_2, a_1, \ldots, q_2, a_m, \ldots, q_n, a_0, q_n, a_1, \ldots, q_n, a_m.$$

Пусть $I_1, \ldots, I_{n(m+1)}$ – соответствующая последовательность команд. Тогда машине T поставим в соответствие двоичный набор вида

$$\text{Код}(T)=\text{Код}(I_1)\text{Код}(I_2)\ldots\text{Код}(I_{n(m+1)}), \qquad \textbf{(7.2)}$$

полученных приписыванием друг к другу кодов команд.

Пример 7.1. Пусть задана машина Тьюринга T. $A = \{a_0, a_1\}$ и $Q = \{q_0, q_1\}$.

$$T: \begin{cases} q_1 a_0 \Rightarrow q_0 a_0 E, \\ q_1 a_1 \Rightarrow q_0 a_1 E. \end{cases}$$

Тогда имеем Код(T) = $10^7 10^4 10^5 10^4 10^3 10^7 10^6 10^5 10^6 10^3$.

Легко видеть, что машины вычисляет функцию $f(x) = x$.

Ясно, что указанное кодирование является алгоритмической процедурой.

Если имеется код машины Тьюринга, то можно однозначно восстановить систему ее команд.

Для этого надо выделить подслова, начинающиеся с единицы, справа от которой стоят нули (причем последний нуль этого подслова в коде машины Тьюринга является «левым соседом» следующей единицы).

Пятерка таких подслов образует команду. Далее, легко видеть, что имеется алгоритмическая процедура, позволяющая по произвольному слову, состоящему из нулей и единиц, выяснять будет ли это слово служить кодом некоторой машины Тьюринга.

Будем теперь рассматривать код машины Тьюринга как двоичную запись натурального числа. Данное число назовем **номером машины Тьюринга.**

Поскольку все коды начинаются с единицы, то разным кодам соответствуют разные числа. Упорядочим машины Тьюринга по возрастанию чисел, представляемых их кодами, и пронумеруем их

$$T_0, T_1, \ldots, T_n, \ldots \qquad \textbf{(7.3)}$$

Номер машины T в этом упорядочении будем обозначать n_T.

Указанное упорядочение является эффективным в том смысле, что существует алгоритм, который по n выдает $Код(T_n)$, и существует алгоритм, который, наоборот, по $Код(T_n)$ выдает n_T.

Если обозначить через $f_n(x)$ одноместную функцию, которую вычисляет машина Тьюринга T_n, то в результате получим нумерацию всех одноместных частично рекурсивных функций:

$$f_0(x), f_1(x), \ldots, f_n(x), \ldots \qquad \textbf{(7.4)}$$

Согласно доказанному, каждая одноместная частично рекурсивная функция представлена в этой последовательности.

Можно доказать, что каждая такая функция представлена в последовательности (7.4) бесконечное число раз.

Аналогично можно определить нумерацию n-местных функций.

7.3. Примеры алгоритмически неразрешимых проблем

Основной метод, применяемый в доказательствах алгоритмической неразрешимости, который называется **методом сводимости**, базируется на следующем рассуждении:

Пусть имеем две массовые проблемы Π_1 *и* Π_2. *Пусть имеется алгоритм А, который по всякой индивидуальной задаче* $\pi_1 \in \Pi_1$ *строит индивидуальную задачу* $\pi_2 \in \Pi_2$, *такую, что выполнено:* π_1 *имеет ответ «ДА» тогда и только тогда, когда* π_2 *имеет ответ «ДА». В этом случае говорят, что проблема* Π_1 *сводится к проблеме* Π_2. *Если проблема* Π_1 *неразрешима, то проблема* Π_2 *также неразрешима*.

Возможность применения данного метода зависит от имеющегося запаса проблем, алгоритмическая неразрешимость которых уже установлена.

Рассмотрим примеры неразрешимых проблем:

1. **Проблема самоприменимости машин Тьюринга**. Рассмотрим машины Тьюринга, внешние алфавиты которых содержат символы 0 и 1 (наряду с другими). Для каждой машины Тьюринга T построим Код(T), который является (0,1)-словом, и запустим машину T в начальной конфигурации

$$q_1\text{Код}(T).$$

Если машина T останавливается (т.е. попадает в заключительное состояние) через конечное число шагов, то она называется **самоприменимой**, в противном случае – **несамоприменимой**.

Заметим, что имеются как самоприменимые, так и несамоприменимые машины Тьюринга.

Например,

1) ***несамоприменима*** машина T_1, у которой все команды имеют вид

$$q_i a_j \Rightarrow q_i a_j E$$

(в правых частях команд нет заключительного состояния);

2) ***самоприменима*** машина T_2, у которой все команды имеют вид

$$q_i a_j \Rightarrow q_0 a_j E$$

(в правых частях всех команд имеется заключительное состояние).

Проблема самоприменимости состоит в том, чтобы *по любой машине Тьюринга T определить, является ли она самоприменимой или нет*.

Условимся, что *машина Тьюринга M решает проблему самоприменимости, если для любой машины T начальную конфигурацию q_1Код(T) она переводит в $q_0$1, если машина T самоприменима, и в $q_0$0, если машина T несамоприменима*.

Теорема 7.1. *Не существует машины Тьюринга M, решающей проблему самоприменимости, т.е. проблема самоприменимости алгоритмически неразрешима*.

Доказательство. Предположим противное, т.е. пусть существует машина Тьюринга *M*, решающая проблему самоприменимости в указанном выше смысле.

Построим новую машину M_0, добавив новое состояние q_0' и объявив его заключительным, а также добавив новые команды для состояния q_0, которое было заключительным в M:

$$q_0 1 \Rightarrow q_0 1E, \tag{7.5}$$

$$q_0 0 \Rightarrow q_0' 0E. \tag{7.6}$$

Рассмотрим теперь работу машины M_0.

Пусть T – произвольная машина.

Если T – самоприменима, то начальная конфигурация $q_1 Код(T)$ перейдет с помощью команд машина M через конечное число шагов в конфигурацию $q_0 1$, далее применяется команда (7.5), и машина M_0 никогда не остановится.

Если T – несамоприменима, то начальная конфигурация $q_1 Код(T)$ перейдет с помощью команд машина M через конечное число шагов в конфигурацию $q_0 0$, далее применяется команда (7.6), и машина M_0 остановится.

Значит, машина M_0 применима к кодам несамоприменимых машин T и неприменима к кодам самоприменимых машин T.

Теперь рассмотрим

$$Код(M_0)$$

и применим машину M_0 к начальной конфигурации

$$q_1 Код(M_0).$$

Сама машина M_0 является либо самоприменимой, либо несамоприменимой.

Если M_0 самоприменима, то по доказанному она никогда не остановится, начав с $q_1 Код(M_0)$, и потому она несамоприменима.

Если M_0 несамоприменима, то по доказанному, она останавливается через конечное число шагов, начав с $q_1 Код(M_0)$, и потому она самоприменима.

Получили противоречие, которое является следствием допущения существования машины M_0, решающей проблему самоприменимости, что и требовалось доказать.

2. Проблема останова. *Проблемой останова* называют проблему, заключающуюся в том, чтобы по любой машине Тьюринга T и слову P в ее внешнем алфавите узнать, применима ли T к начальной конфигурации q_1P.

Проблема останова алгоритмически неразрешима, так как если бы она была разрешимой, то, взяв в качестве P слово $Код(T)$, мы получили бы разрешимость проблемы самоприменимости.

3. Проблема распознавания истинности формул элементарной арифметики. Формулы строятся с помощью арифметических действий (сложения и умножения), логических операций (логических связок и кванторов) и знака равенства. Проблема состоит в том, чтобы найти алгоритм, который по любой такой формуле определяет, истинна она или нет на натуральном ряду (Неразрешимость этой проблемы установил К. Гедель в 1931 году).

4. Проблема разрешения для логики предикатов первого порядка. Нужно найти алгоритм, который бы проверил общезначимость формулы алгебры предикатов (неразрешимость этой проблемы установил А. Чёрч в 1936 г.).

5. Проблема сочетаемости Поста. Конечное множество V пар слов в некотором алфавите называется *сочетаемым*, если для некоторых пар $(A_1, B_1),\ldots, (A_S, B_S)$ из V выполнено равенство $A_1\ldots A_S=B_1\ldots B_S$. Нужно найти алгоритм, который по всякому множеству V пар слов узнает, сочетаемо оно или нет (неразрешимость данной проблемы установил Э. Пост в 1946 г.).

6. Десятая проблема Гильберта. Эта проблема из 23, поставленных Гильбертом, формулируется так: «Пусть задано диофантово уравнение (т.е. уравнение вида $p(x_1, \ldots, x_n) = 0$, p – многочлен) с произвольными неизвестными и целыми рациональными коэффициентами. Указать способ, при помощи которого возможно после конечного числа операций установить, разрешимо ли уравнение в целых рациональных числах» (неразрешимость 10-й проблемы Гильберта установлена Ю.В. Матиясевичем в 1970 г.).

7.4. Характеристики сложности вычислений

В общей теории алгоритмов изучается лишь принципиальная возможность алгоритмического решения задачи.

При рассмотрении конкретных задач не обращается внимания на ресурсы времени и памяти для соответствующих им разрешающих алгоритмов.

Однако нетрудно понять, что принципиальная возможность алгоритмического решения задачи еще не означает, что оно может быть практически получено.

Поэтому возникает необходимость в уточнении понятия «**алгоритмическая разрешимость**».

Для этого были введены **характеристики сложности алгоритмов**, которые позволяют судить об их практической реализуемости. Выше было установлено, что различные алгоритмические модели приводят к одному и тому же классу алгоритмически вычислимых функций.

Поскольку основные сложностные характеристики переносятся от одной модели вычисления к другой, в качестве модели вычислительного устройства достаточно рассмотреть машину Тьюринга, а в качестве характеристик сложности – необходимое время (число шагов вычисления) и память (размер используемой ленты).

Пусть T – машина Тьюринга, вычисляющая функцию $f(x)$. Обозначим через $t_T(x)$ число шагов работы машины T, вычисляющей значение $f(x)$, если $f(x)$ определено, и будем считать, что $t_T(x)$ не определено, если не определено значение $f(x)$. Функция $t_T(x)$ называется **временной сложностью машины *T***.

Пусть значение $f(x)$ определено. **Активной зоной** при работе машины T на числе x называется минимальное множество $S_T(x)$ ячеек ленты, удовлетворяющее следующим условиям:

1) множество $S_T(x)$ содержит все ячейки, которые являются активными, т.е. используются при вычислении $f(x)$;

2) множество $S_T(x)$ вместе с любыми двумя ячейками содержит все промежуточные ячейки ленты.

Ленточной сложностью машины *T* называется функция $s_T(x)$, которая равна мощности активной зоны $S_T(x)$, если значение $f(x)$ определено, и $s_T(x)$ не определено, если не определено значение $f(x)$.

Если в машине Тьюринга T алфавит A состоит из m элементов, множество состояний Q – из n элементов и значение $f(x)$ определено, то справедливы следующие неравенства, позволяющие оценивать величину ленточной сложности через временную и наоборот:

$$s_T(x) \le x+1+t_T(x), \quad t_T(x) \le (n+1)^2 s_T^2(x) m^{s_T(x)}.$$

Приведенные оценки показывают, что для исследования сложности алгоритма в качестве основной характеристики достаточно рассмотреть временную сложность.

7.5. Переборные и распознавательные задачи. Классы сложности P и NP и их взаимосвязь

Трудоемкостью, или **сложностью**, алгоритма решения массовой задачи Z называется функция f, ставящая в соответствие каждому натуральному числу n максимальное время $f(n)$ работы алгоритма по всем входным словам данной задачи, имеющим длину n.

Анализ эффективности алгоритма заключается в выяснении вопроса о том, насколько быстро растет функция $f(n)$ с ростом n. При ответе на этот вопрос обычно используются O-символика.

Говорят, что неотрицательная функция $f(n)$ не превосходит по порядку функцию $g(n)$, и используют запись

$$f(n) = O(g(n)),$$

если существует такая константа C, что выполняется неравенство

$$f(n) \le Cg(n) \text{ для любого } n \in N .$$

Выражениям «трудоемкость (сложность) алгоритма составляет $O(g(n))$», «решение задачи требует порядка $O(g(n))$» или «алгоритм решает задачу за время $O(g(n))$» обычно придается именно этот смысл.

Например, трудоемкость $O(1)$ означает, что время работы соответствующего алгоритма не зависит от длины входного слова.

Алгоритм с трудоемкостью $O(n)$ называется **линейным**. Линейный алгоритм просматривает входную информацию определенное количество раз. Для подавляющего большинства практических задач линейный алгоритм является неулучшаемым по порядку. Поэтому отысканием линейного алгоритма (при его существовании) обычно заканчивается алгоритмическое решение данной массовой задачи.

Алгоритм, сложность которого равна $O(n^c)$, где c – константа, называется **полиномиальным**. Говорят, что массовая **задача *Z* решается эффективно**, если существует алгоритм, имеющий полиномиальную сложность. Таким образом, эффективное вычисление представляет собой вычисление с полиномиальной сложностью.

Обычно появление какого-либо полиномиального алгоритма приводит в конце концов к алгоритму, трудоемкость которого оценивается по порядку полиномом небольшой степени.

В большинстве случаев эта степень не превосходит трех и оценка, как правило, имеет один из следующих видов: $O(n^3)$, $O(n^2)$, $O(n\sqrt{n})$, $O(n\log n)$, $O(n)$. Например, полиномиальными являются алгоритмы нахождения кратчайших маршрутов во взвешенных графах.

Алгоритм, сложность которого равна $O(c^n)$, где c – константа, $c \geq 2$, называется **экспоненциальным**.

Пример 7.2. Рассмотрим задачу проверки эквивалентности булевых функций $f(x_1, x_2, \ldots, x_n)$ и $g(x_1, x_2, \ldots, x_n)$. Как известно, решение задачи состоит в составлении таблиц истинности функций f и g, т.е. в переборе всевозможных наборов $(\delta_1, \delta_2, \ldots, \delta_n) \in \{0, 1\}^n$. Для проверки 2^n соотношений $f(\delta_1, \delta_2, \ldots, \delta_n) = g(\delta_1, \delta_2, \ldots, \delta_n)$ эта задача требует экспоненциального времени и поэтому решается неэффективно.

Массовая задача, представленная в примере 7.2, относится к типу, так называемых, **переборных задач.**

Распознавательными массовыми задачами называются такие задачи, решение которых заключается в получение ответа «да» или «нет».

Всякий алгоритм решения такой задачи, будучи примененным к соответствующему входу, работает какое-то время, а затем, сообщив ответ «да» либо «нет», останавливается.

Для некоторых задач их изначальные постановки являются распознавательными. Например, проверка тождественной истинности формулы.

Однако на практике чаще исходная задача является **оптимизационной**. В **оптимизационной** задаче требуется выбрать из множества допустимых решений X такое решение x, вес (стоимость или какая-либо другая характеристика) $w(x)$ которого минимален.

Каждой оптимизационной задаче поставим в соответствие ее **распознавательный вариант**, который выглядит следующим образом. По заданным множеству X, весовой функции w и числу k требуется выяснить, существует ли элемент $x \in X$, для которого $w(x) \leq k$.

Очевидно, что из наличия полиномиального алгоритма решения оптимизационной задачи следует существование эффективного решения для соответствующей распознавательной задачи.

Класс, состоящий из всех распознавательных массовых задач, каждая из которых может быть решена некоторым полиномиальным алгоритмом, обозначается через ***P***.

Определим класс переборных задач, который содержит класс *P* и включает задачи, имеющие полиномиальные алгоритмы для проверки позитивной информации. Для этого введем понятие недетерминированного алгоритма.

Пусть

$A = (A_1, \ldots, A_m)$ – последовательность операторов, по которым в соответствии с некоторыми алгоритмами входные слова перерабатываются в выходные,

$B(\cdot, \cdot)$ – оператор передачи управления, который на каждой паре $(l_1, l_2) \in \{1, 2, \ldots, m\}$ действует по следующей схеме.

Если $Q = (q_1, \ldots, q_p)$ – последовательность из $\{l_1, l_2\}^p$, то в результате k-го ($k \le p$) выполнения оператора $B(l_1, l_2)$ управление передается

оператору A_{l_1} в случае, если $q_k = l_1$,

или оператору A_{l_2} в случае, если $q_k = l_2$.

При $k > p$ вычисления прекращаются.

Итак, последовательности операторов A и индексов Q ставится в соответствие обычный (детерминированный) алгоритм. Этот алгоритм представляет собой пару $A_Q = <A, Q>$, в которой A – недетерминированный алгоритм, а Q – угадывающая последовательность.

Пусть A – недетерминированный алгоритм, с помощью которого решается массовая задача Z. Устройство, выдающее для каждой задачи $z \in Z$ соответствующую угадывающую последовательность, называется *оракулом*.

Таким образом, отличие недетерминированного алгоритма от детерминированного состоит в наличии оракула.

В силу тезиса Чёрча любой недетерминированный алгоритм можно интерпретировать в виде недетерминированной машины Тьюринга, где алгоритмы A_1, …, A_m интерпретируются обычными, детерминированными машинами Тьюринга, а оператор передачи управления в зависимости от угадывающей последовательности запускает соответствующие машины.

Говорят, что недетерминированный алгоритм A решает распознавательную задачу за полиномиальное время, если найдется полином $P(n)$, для которого выполняется следующее условие: *каждое входное слово s для данной задачи имеет ответ «да» тогда и только тогда, когда для s существует такая угадывающая последовательность Q, что алгоритм A_Q, будучи примененным к этому входу, останавливается, сообщив «да», и время его работы не превосходит P(n)*.

Заметим, что согласно этому определению каждому входу с ответом «да» ставится в соответствие алгоритм A_Q.

Массовая задача называется **недерминированно разрешимой за полиномиальное время, или переборной**, если существует недетерминированный алгоритм, решающий эту задачу за полиномиальное время.

Класс всех переборных задач обозначается через NP.

Нетрудно заметить, что любая эффективно решаемая задача является переборной и, следовательно, $N \subseteq NP$.

Класс *NP* чрезвычайно широк – большинство «естественных» массовых задач входят в него.

В настоящее время до сих пор не решен вопрос о совпадении классов *P* и *NP*.

В теории алгоритмов вопрос о равенстве классов сложности *P* и *NP* является одной из центральных открытых проблем уже более 40 лет. Если на него будет дан утвердительный ответ, это будет означать, что теоретически возможно решать многие задачи существенно быстрее, чем сейчас.