ГЛАВА 3. ТЕОРИЯ АЛГОРИТМОВ

§3.1. Интуитивное понятие алгоритма и необходимость введения его точного математического понятия

Понятие алгоритма прочно вошло в жизнь математиков и людей, тесно связанных с вычислительной техникой. Зачастую мы даже не задумываемся над тем, что же такое алгоритм, а используем это слово как некое интуитивное понятие. Однако история его возникновения восходит к глубокой древности.

Термин *алгоритм* или *алгорифм* (записываемый латиницей как *algorithm*) имеет в своём составе видоизменённое географическое название *Хорезм*. Он обязан своему происхождению великому средневековому учёному Мухаммаду ибн Муссе аль Хорезми (то есть из Хорезма), написавшему обширный труд, в котором описывались процедуры арифметических действий с числами.

Первоначально под алгоритмом понимали произвольную строго определённую последовательность действий, приводящую к решению той или иной конкретной задачи. Ещё с античных времён известны алгоритм Евклида нахождения наибольшего общего делителя натуральных чисел, алгоритм деления отрезка в заданном отошении с помощью циркуля и линейки и т.п. Кстати, алгоритмы, изложенные в первых двух главах этого учебного пособия, – это тоже примеры интуитивного понимания алгоритма.

В начале XX века были сформулированы задачи нахождения единого процесса решения ряда родственных задач с параметрами. Если с задачей нахождения такого процесса было всё более или менее ясно (достаточно предъявить такой процесс), то что же делать, если процесс найти не удалось? Мы плохо искали или он не существует? Ответы на эти вопросы были особенно важны в связи с программой Д.Гильберта формализации всей математики.

Первые попытки дать математическое определение алгоритма привели приблизительно к следующим требованиям:

— *Определённость данных*: вид исходных данных строго определён.

— *Дискретность*: процесс разбивается на отдельные шаги.

— *Детерминированность*: результат каждого шага строго определён в зависимости от данных, к которым он применён.

— *Элементарность шага*: переход на один шаг прост.

— *Направленность*: что считать результатом работы алгоритма, если следующий шаг невозможен.

— *Массовость*: множество возможных исходных данных потенциально бесконечно.

Несмотря на недостатки такого определения[1] это интуитивное определение может служить для решения многих задач. Однако в современной математике и информатике активно используются алгоритмы, не удовлетворяющие такому интуитивному определению: недетерминированные вычисления, алгоритмы с оракулом, «зацикливающиеся» алгоритмы и т.п.

Для математического уточнения определения понятия алгоритма начиная с 30-х годов XX века были введены различные математические понятия алгоритма. Первыми математическими понятиями алгоритма были рекурсивные функции и машины Тьюринга.

Все математические понятия алгоритма имеют дело с конструктивными объектами.

§3.1.1. Конструктивные объекты

Под конструктивными объектами обычно понимают такие объекты, которые могут быть закодированы словами в некотором конечном алфавите в соответствии с точными инструкциями. Из известных вам объектов таковыми являются, например, пропозициональные формулы, формулы исчисления предикатов, формулы формальных теорий, графы, программа для компьютера на выбранном языке и т.п. Один из стандартных способов задания множества конструктивных объектов – *формулы Бэкуса*. По сути дела они представляют собой рекурсивные определения понятий.

Формулы Бэкуса имеют вид

<понятие> ::= <понятие1> | <понятие2> | <понятие$_i$> <понятие$_j$> | ...

где <понятие> – определяемое понятие, <понятие1>, <понятие2>, <понятие$_i$>, <понятие$_j$>, ... – ранее определённые понятия или само определяемое понятие, символ | читается как «или», символ ::= читается как «это есть».

Простейшим примером использования формул Бэкуса является определение понятия <слово в алфавите $\{a_1, \ldots, a_n\}$> (в этом примере дальше будем для краткости писать <слово>).

[1] Являются ли оба шага следующего вычисления простыми: $x := 0.25; z := \sin(y^x)$?

<буква> ::= $a_1 \mid \ldots \mid a_n$
<пустое слово> ::=
<слово> ::= <пустое слово> | <слово><буква>

Другим простым примером является понятие <пропозициональная формула> (в этом примере дальше будем для краткости писать <проп-формула>), в которой все пропозициональные переменные имеют вид $p1\ldots1$.

<проп-переменная> ::= p | <проп-переменная>1
<бинарная логическая связка> ::= $\&$ | $\vee$ | $\to$ | $\leftrightarrow$ | $\oplus$ | $|^2$ | $\downarrow$
<проп-формула> ::= <проп-переменная> | $\neg$ <проп-формула> | (<проп-формула><бинарная логическая связка><проп-формула>)

Более сложным является описание с помощью формул Бэкуса натурального числа. Более точно, само число нельзя описать, но можно описать понятие *запись натурального числа в системе счисления ...* . Начнём с унарной системы счисления, в которой каждое натуральное число – это конечная последовательность одного и того же символа. Часто в качестве этого символа берут | , т.е. «палочку», поэтому эту систему ещё называют «палочковой». В этом примере будем использовать единицу 1.

<ноль> ::=
<палочковое число> ::= <ноль> | <палочковое число>1

В языке Формальной Арифметики FA роль натуральных чисел играют постоянные термы вида $S(\cdots S(0)\cdots)$. Если вместо префиксной записи $S(x)$ использовать постфиксную запись x', то получим другую запись натурального числа $0^{\prime\ldots\prime}$. Если вместо символа $'$ писать 1, то получим унарную запись натурального числа, удобную тем, что в последовательности чисел не требуется их разделять специальным символом.

<ноль> ::= 0
<унарная запись натурального числа> ::= <ноль> | <унарная запись натурального числа>1

А вот с двоичной (восьмеричной, десятичной, ...) записью натурального числа придётся повозиться. Для её определения потребуются понятия <ноль>, <цифра>, <ненулевая цифра>, <ненулевое двоичное (восьмеричное, десятичное, ...) число>.

В качестве упражнений можно написать формулы Бэкуса для

[2]Здесь, к сожалению, символ Шеффера имеет то же обозначение, что и «или» в формулах Бэкуса.

1. двоичной записи натурального числа;

2. десятичных чисел, кратных трём;

3. двоичной записи целого числа;

4. двоичной записи рационального числа;

5. конечно-двоичной записи рационального числа (это то, что используется в компьютере с некоторыми ограничениями);

Запись вещественного (в любой системе счисления) не является конструктивным объектом и не может быть задана формулами Бэкуса.

Синтаксис одного из первых алгоритмических языков прграммирования Алгол 60 был полностью описан с помощью формул Бэкуса. Но так никогда и не был реализован в полном объёме.

§3.2. Представление о рекурсивных функциях. Тезис Чёрча.

Понятие рекурсивных функций было предложено Чёрчем и Клини.

Определение. *Простейшими называются функции натурального аргумента* S, O, I_n^m, *определяемые равенствами:* $S(x) = x+1$, $O(x) = 0$, $I_n^m(x_1, \ldots, x_n) = x_m$ *при* $1 \le m \le n$.

Определение. *Функция* f *от* $n+1$ *переменных получена из функции* g *от* n *переменных и функции* h *от* $n+2$ *переменных с помощью оператора примитивной рекурсии, если*

$$\begin{cases} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{cases}.$$

Определение. *Функция называется* **примитивно рекурсивной**, *если она может быть получена из простейших с помощью применения операторов подстановки или примитивной рекурсии.*

Следует отметить, что всякая примитивно рекурсивная функция определена для любого набора значений своих аргументов. Примитивно рекурсивными являются, например, функции, определяемые термами $x+y$, $|x-y|$, $x \cdot y$, $[\frac{x}{y}]$, x^y, $[\sqrt{x}]$, характеристические функции предикатов $=$, $<$, $>$, $\le$, $\ge$ и многие другие.

Нахождение корней функции вызывает некоторые проблемы. Так, например, μ-оператор, действие которого определяется как $g(x_1, \ldots, x_n) = \mu\, y\{f(x_1, \ldots, x_n, y) = 0\}$ (наименьшее y, для которого $f(x_1, \ldots, x_n, y) = 0$), применённый к примитивно рекурсивной функции, не всегда да-

ёт в результате примитивно рекурсивную функцию. Но применение ограниченного μ-оператора, действие которого определяется как $g(x_1, \ldots, x_n, z) = \mu\ y_{\leq z}\{f(x_1, \ldots, x_n, y) = 0\}$ (наименьшее y, не превосходящее z и для которого $f(x_1, \ldots, x_n, y) = 0$, или 0, если такой y не существует), даёт в результате примитивно рекурсивную функцию.

Определение. *Функция называется* **частично рекурсивной**, *если она может быть получена из простейших с помощью применения операторов подстановки, примитивной рекурсии или μ-оператора.*

Не все частично рекурсивные функции определены для любого набора значений своих аргументов. Но всякая примитивно рекурсивная функция является частично рекурсивной.

Если ввести обобщённый μ-оператор, определяемый как

$$\mu^* y\{f(\overline{x}, y) = 0\} = \begin{cases} \mu\ y\{f(\overline{x}, y) = 0\} & \text{если такой } y \text{ существует,} \\ 0 & \text{иначе} \end{cases},$$

то можно определить общерекурсивные (или рекурсивные) функции.

Определение. *Функция называется* **общерекурсивной**, *если она может быть получена из простейших с помощью применения операторов подстановки, примитивной рекурсии или обобщённого μ-оператора.*

Общерекурсивные функции являются всюду определёнными и всякая примитивно рекурсивная функция является общерекурсивной. Задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Тезис Чёрча. *Всякая интуитивно вычислимая функция является общерекурсивной.*

Это утверждение нельзя ни доказать, ни опровергнуть, так как в его формулировке имеется понятие «интуитивно вычислимая», которое не имеет точного математического определения.

§3.3. Машины Тьюринга. Тезис Тьюринга-Чёрча.

Машина Тьюринга является математическим, а не техническим понятием. Однако те, кому ближе «железные» объекты, могут представлять её как вычислительное устройство, имеющее потенциально бесконечную ленту, разделённую на ячейки (т.е. в любой момент работы слева или

справа к конечной ленте можно добавить ещё одну ячейку, содержащую специальный символ, называемый пустым). На этой ленте может быть записано слово в заранее заданном алфавите. По ленте может перемещаться пишущая/читающая головка, обозревающая одну из ячеек. В зависимости от состояния машины и содержимого обозреваемой ячейки машина в соответствии с программой может заменить содержимое обозреваемой ячейки, сдвинуть (или не сдвигать) головку на одну ячейку, изменить своё состояние.

С математической точки зрения машина Тьюринга задаётся тройкой $\langle A, Q, P\rangle$, где

— $A = \{a_1, \ldots, a_n\}$ – внешний алфавит (содержащий, в частности, пустой символ, который здесь будем обозначать посредством *);

— $Q = \{q_0, q_1, \ldots, q_k\}$ – внутренний алфавит, в котором выделены начальное и заключительные состояния; ниже в качестве начального состояния будет использоваться состояние q_1, а в качестве заключительного — состояние q_0,

— P – программа.

Команда машины Тьюринга имеет вид

$$q_r a_i \to q_t S a_j,$$

где $i, j = 1, \ldots, n$, $r = 1, \ldots, k$, $t = 0, 1, \ldots, k$, S – сдвиг головки влево, вправо или отсутствие сдвига $S \in \{L, R, \_\}$. Эта команда читается следующим образом: «Если машина Тьюринга находится в состоянии q_r и обозревает символ a_i, то этот символ заменяется на a_j, головка производит сдвиг S и машина переходит в состояние q_t.»

Команды называются **согласованными**, если они имеют различные левые части, или полностью совпадают.

Программой машины Тьюринга называется конечное непустое множество согласованных команд.

Конфигурацией машины Тьюринга называется слово вида

$$b_1 \ldots b_{p-1} q_r b_p \ldots b_l,$$

где

— $b_1 \ldots b_{p-1} b_p \ldots b_l$ — слово в алфавите A, записанное на ленте;

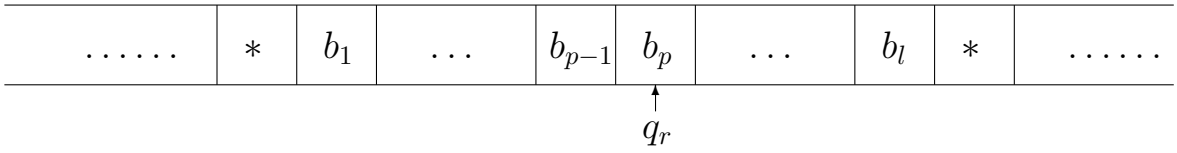
— слева и справа от этого слова на ленте находятся только пустые символы;

— машина находится в состоянии q_r и обозревает p-ый символ этого слова;

— на концах конфигурации находитя не более чем по одному пустому символу.

Машина Тьюринга всегда начинает работу в конфигурации вида q_1X, где X – исходные данные.

Конфигурация соответствует содержимому ленты и головки, изображённым ниже.



Протоколом работы машины Тьюринга называется последовательность конфигураций, первая из которых является начальной, а каждая следующая получена из предыдущей в соответствии с одной из команд.

Машина Тьюринга **заканчивает** работу над данными X, если она пришла в состояние q_0 или ни одна из команд не может быть применена к полученной конфигурации.

Тезис Тьюринга-Чёрча. *Всякая интуитивно вычислимая функция может быть вычислена на машине Тьюринга.*

Это утверждение нельзя ни доказать, ни опровергнуть, так как в его формулировке имеется понятие «интуитивно вычислимая», которое не имеет точного математического определения.

3.3.1. Теорема о композиции машин Тьюринга

Лемма 3.1. *По всякой машине Тьюринга M, которая по данным X в алфавите A вычисляет значение функции $f(X)$, можно построить машину Тьюринга M_1, которая по данным X вычисляет значение функции $f(X)$ и заканчивает работу в конфигурации $q_0f(X)$.*

Доказательство. Первым делом пометим начало слова символом, не входящим в алфавит A (например, символом $\#$) и вернёмся в начало исходных данных.

$q_1a_i \to q_1La_i$

$q_1 * \to q_2R\#$

В программе машины M каждое вхождение состояния q_i $(i = 1, \ldots, k)$ заменяем на q_{i+1}. Возможны два случая завершения работы программы над данными.

1. Машина Тьюринга завершила работу в конфигурации $\#Y'q_oY''$, где $Y'Y''$ совпадает с $f(X)$.

Заменяем в тексте программы состояние q_0 на q_{k+2} и добавляем команды

$q_{k+2}a_i \to q_{k+2}La_i$ $(a_i \in A)$

$q_{k+2}\# \to q_0 \quad R *$.

2. Машина Тьюринга завершила работу в конфигурации $\#Y'q_ta_iY'''$, где $Y'a_iY'''$ совпадает с $f(X)$ и в программе отсутствует команда с левой частью q_ta_i.

Для каждого состояния q_t и каждого символа a_i из алфавита , для которых в программе отсутствует команда с левой частью q_ta_i, добавляем команду

$q_ta_i \to q_{k+2}La_i$

и команды, указанные в п. 1) этого доказательства. ∎

Теорема 3.1. *Пусть машина Тьюринга M_1 по данным X в алфавите A_1 вычисляет значение функции $g(X)$ в алфавите A_2, машина Тьюринга M_2 по данным Y в алфавите A_2 вычисляет значение функции $f(Y)$. Тогда существует машина Тьюринга M_3, которая по данным X вычисляет значение функции $f(g(X))$.*

Доказательство. Пусть M_i $(i = 1, 2)$ имеет программу P_i и использует состояния $\{q_0, q_1, \ldots, q_{k_i}\}$. В соответствии с леммой можно считать, что M_1 заканчивает работу в конфигурации $q_0g(X)$.

В программе P_1 состояние q_0 заменяем на q_{k_1+1}, получаем программу P_1'.

В программе P_2 состояния q_i $(i = 1, \ldots, k_2)$ заменяем на q_{k_1+i}, получаем программу P_2'.

Машина с программой $P_1' \cup P_2'$ вычисляет $f(g(X))$. ∎

Аналогичную теорему можно доказать и для функций нескольких аргументов. Однако при её доказательстве потребуется лемма, которая здесь будет приведена без доказательства.

Определение. *Машина Тьюринга называется машиной Тьюринга с односторонне-ограниченной лентой, если она не использует ячейки, расположенные левее ячейки с первым символом исходных данных (левосторонне-ограниченная), или расположенные правее ячейки с последним символом исходных данных (правосторонне-ограниченная).*

Лемма 3.2. *По всякой машине Тьюринга M можно построить ма-*

шину Тьюринга с односторонне-ограниченной лентой, у которой на любых исходных данных результат её работы совпадает с результатом работы M.

Теорема 3.2. *Пусть машина Тьюринга M_1 по данным $X_1, ..., X_i, ..., X_n$ в алфавите A_1 вычисляет значение функции $f(X_1, ..., X_i, ..., X_n)$ в алфавите A_2, машина Тьюринга M_2 по данным $Y_1, ..., Y_m$ в алфавите A_2 вычисляет значение функции $g(Y_1, ..., Y_m)$. Тогда существует машина Тьюринга M_3, которая по данным $X_1, ..., X_{i-1}, X_{i+1}, ..., X_n, Y_1, ..., Y_m$ вычисляет значение функции $f(X_1, ..., X_{i-1}, g(Y_1, ..., Y_m), X_{i+1}, ..., X_n)$.*

Доказательство. Сначала по машине M_2 строим левосторонне-ограниченную машину M_3 с программой P_3.

Машина с программой P_0 с состояниями $q_0, q_1, ..., q_{k_0}$ отмечает место между X_{i-1} и X_{i+1} и подводит головку к первому символу данных $Y_1, ..., Y_m$. Заменив q_0 на q_{k_0+1} получаем программу P'_0.

В программе P_3 заменяем q_i на q_{k_0+i} при $i \neq 0$, а q_0 на q_{k_0+k3+1} и получаем программу P'_3.

Машина с программой P_4 с состояниями $q_0, q_1, ..., q_{k_4}$ вставляет результат работы программы P'_3 между X_{i-1} и X_{i+1} и подводит головку к первому символу, записанному на ленте. Заменив q_i на q_{k_0+k3+i} при $i \neq 0$, а q_0 на $q_{k_0+k3k_4+1}$ получаем программу P'_4.

В программе P_1 для машины M_1 заменяем q_i при $i \neq 0$ на $q_{k_0+k3+k_4+i}$ и получаем программу P'_1.

Машина с программой $P'_0 \cup P'_3 \cup P'_4 \cup P'_1$ вычисляет $f(X_1, ..., X_{i-1}, g(Y_1, ..., Y_m), X_{i+1}, ..., X_n)$. ∎

3.3.2. Многоленточные МТ.

Более точно, следовало бы написать k-ленточные машины Тьюринга при фиксированном k. В этой модели предполагается, что имеется k лент, на каждой из которых может быть записано свое слово. Головка обозревает одновременно по одной ячейке на каждой ленте и, в зависимости от их содержимого, может изменить или не изменять содержимое каждой из обозреваемых ячеек, сдвинуться (или не сдвигаться) на одну ячейку влево или вправо, причем на каждой ленте сдвиг может быть разным.

Команда k-ленточной машины Тьюринга имеет вид

$$q_r \begin{pmatrix} a_{i_1} \\ \vdots \\ a_{i_k} \end{pmatrix} \to q_t \begin{pmatrix} S_1 \\ \vdots \\ S_k \end{pmatrix} \begin{pmatrix} a_{j_1} \\ \vdots \\ a_{j_k} \end{pmatrix},$$

где $S_1, ..., S_k \in \{L, R, \_\}$ и обозначают соответственно сдвиги влево, вправо или отсутствие сдвига головки. Эта команда читается следующим образом: «Если машина Тьюринга находится в состоянии q_r и в обозреваемых ячейках лент записаны соответственно символы $a_{i_1}, ..., a_{i_k}$, то эти символы заменяются соответственно на $a_{j_1}, ..., a_{j_k}$, головка производит сдвиги $S_1, ..., S_k$ и машина Тьюринга переходит в состояние q_t».

Обычно предполагается, что 1-ая лента — это входная лента для записи исходных данных, k-ая лента — это выходная лента для записи результата, остальные $k-2$ ленты — это рабочие ленты. В приведённых ниже примерах это предположение не будет использовано. Однако несложно доказать, что по любой k-ленточной машине Тьюринга можно построить $k+2$-ленточную машину Тьюринга, удовлетворяющую этому условию.

На многоленточной машине Тьюринга программы для решения многих задач выглядят гораздо проще, чем соответствующие программы для одноленточной машины. Это связано с тем, что использование нескольких лент позволяет иметь одновременный доступ к нескольким (точнее, не более чем к k) различным записям.

3.3.3. Теорема о числе шагов МТ, моделирующей работу многоленточной МТ.

Теорема 3.3. *По всякой k-ленточной машине Тьюринга MTk, заканчивающей работу с исходными данными X за t шагов, можно построить одноленточную машину Тьюринга MT1, результат работы которой с исходными данными X совпадает с результатом работы исходной и число шагов которой составляет $O(t^2)$.*

Доказательство. Пусть длина записи исходных данных равна n, причём $n \leq t$. Будем считать, что 1-ая лента входная, а последняя — выходная.

На i-ом шаге ($i = 1, \ldots, t$) длина записи на j-ой ленте MTk ($j = 2, \ldots, k$) может увеличиться не более чем на единицу (т.е. стать равной i), причём запись нового символа может происходить как в середине слова, так и на одном из его концов. При этом содержимое лент имеет

вид

$$\begin{pmatrix} X \\ X_i^2 \\ \vdots \\ X_i^k \end{pmatrix}.$$

Конфигурация моделирующей машины MT1 в этот момент имеет вид

$$X * q_l X_i^2 * \ldots * X_i^k$$

при некотором l.

Для моделирования i-го шага MTk машина MT1 должна для каждого $(j = 2, \ldots, k)$

— сдвинуть головку на символ, обозреваемый MTk на j-ой ленте (не более, чем $||X_i^{j-1}|| + ||X_i^j||$ шагов);

— произвести действие, которое MTk производит со словом X_i^j (1 шаг);

— в случае необходимости переместить всё содержимое ленты правее положения головки на одну ячейку вправо (не более, чем $4\sum_{j'=j}^{k} ||X_i^{j'}||$ шагов);

— вернуться в исходное положение (не более, чем $\sum_{j'=j}^{k} ||X_i^{j'}||$ шагов).

Всего при моделировании действия MTk с одним символом на i-ом шаге MT1 совершает не более

$$\sum_{j=2}^{k} \left(||X_i^{j-1}|| + ||X_i^j|| + 1 + 4 \sum_{j'=j}^{k} ||X_i^{j'}|| + \sum_{j'=j}^{k} ||X_i^{j'}|| \right) \leq$$

$$\sum_{j=2}^{k} \left(i + i + 1 + 5 \sum_{j'=j}^{k} i \right) = \sum_{j=2}^{k} (5(k-j)i + 2i + 1) =$$

$$\frac{1}{2} 5i(k-1)(k-2) + 2i(k-1) + 2(k-1) =$$

$$\frac{3}{2} i(k-1)(3k-4) + 2(k-1).$$

Просуммировав полученное выражение по $i = 1, \ldots, t$ имеем

$$\frac{3}{2} \sum_{i=1}^{t} i \left((k-1)(3k-4) + 2(k-1) \right) =$$

$$\frac{3}{2}(k-1)(3k-4)\frac{t(t-1)}{2} + 2(k-1) = O(k^2 t^2).$$

Так как k является константой, то получили $O(t^2)$. ∎

3.3.4. Многоголовчатые МТ.

Более точно, следовало бы написать m-головчатые машины Тьюринга при фиксированном m. В этой математической модели предполагается, что имеется m головок, каждая из которых может обозревать одну ячейку ленты. В зависимости от содержимого всех обозреваемых ячеек машина может изменить или не изменять содержимое каждой из обозреваемых ячеек, сдвинуться (или не сдвигаться) на одну ячейку влево или вправо.

В этой модели возможны два варианта модификации:

– запрет на то, чтобы разные головки обозревали одну и ту же ячейку;

– головкам приписывается приоритет и в случае, если несколько головок обозревают одну и ту же ячейку, команда распространяется только на головку с наивысшим приоритетом, а остальные из этих головок пропускают свой шаг.

Команда m-головчатой машины Тьюринга имеет вид

$$q_r(a_{i_1}, ..., a_{i_m}) \to q_t(S_1, ..., S_m)(a_{j_1}, ..., a_{j_m}),$$

где $S_1, ..., S_m \in \{L, R, \_\}$ и обозначают соответственно сдвиги влево, вправо или отсутствие сдвига головки.

Многоголовчатые машины Тьюринга можно рассматривать как одну из возможных моделей параллельных вычислений, использующих общую память.

3.3.5. Недетерминированные МТ. Теорема о числе шагов МТ, моделирующей работу недетерминированной МТ.

Недетерминированные машины Тьюринга получаются, если в программе классической машины Тьюринга разрешить использование несогласованных команд. Как же следует выполнить, например, такую пару команд

$$q_1 0 \to q_1 L 1$$
$$q_1 0 \to q_2 R 0,$$

если одна предписывает изменить содержимое ячейки и в том же состоянии сдвинуться влево, а другая — не изменяя содержимого ячейки сдвинуться вправо и изменить состояние?

Для выполнения такой пары команд лента недетерминированной машины размножается, и на каждой ленте выполняется ровно одна из ко-

манд. Дальнейшие вычисления на каждой ленте продолжаются независимо.

Недетерминированные машины Тьюринга, как правило, используются для проверки истинности утверждений типа $\exists Y P(X, Y)$ (существует такой объект Y, для которого справедливо утверждение $P(X, Y)$). Для проверки истинности таких утверждений работу недетерминированной машины Тьюринга можно разбить на два этапа:

— этап угадывания, при реализации которого лента в недетерминированном режиме размножается, и на каждой из них выписывается «претендент» на решение;

— этап проверки, при реализации которого машина работает в детерминированном режиме и проверяет конкретного «претендента» на то, является ли он решением.

Вместо одного заключительного состояния q_0 обычно используют два q_Y и q_N (происходящих от слов Yes и No). Недетерминированная машина Тьюринга заканчивает работу в состоянии q_Y, если хоть на одной из лент она пришла в состояние q_Y. Если же на всех лентах недетерминированная машина Тьюринга пришла в состояние q_N, то и вся машина заканчивает работу в этом состоянии q_N.

3.3.6. Теорема о числе шагов МТ, моделирующей работу недетерминированной МТ.

Теорема 3.4. *По всякой недетерминированной машине Тьюринга, проверяющей предикат $\exists Y P(X, Y)$ и заканчивающей работу с исходными данными X за t шагов, можно построить одноленточную машину Тьюринга, результат работы которой с исходными данными X совпадает с результатом работы исходной и число шагов которой составляет $2^{O(t)}$.*

Для доказательства этой теоремы докажем две леммы.

Лемма 3.3. *Если недетерминированная машина Тьюринга, проверяющая предикат $\exists Y P(X, Y)$ заканчивает работу с исходными данными X за t шагов, то длина записи «претендента» Y не превосходит t.*

Утверждение леммы следует из того, что длина записи слова не может быть больше, чем число шагов, затраченных на его выписывание.

Лемма 3.4. *Если недетерминированная машина Тьюринга, проверяющая предикат $\exists Y P(X, Y)$ заканчивает работу с исходными данными*

X за t шагов, то количество «претендентов» Y не превосходит $2^{O(t)}$.

Доказательство. Пусть $A = \{a_1, \ldots, a_k\}$ — внешний алфавит недетерминированной машины Тьюринга. Количество «претендентов» m не превосходит количества слов в этом алфавите, длина которых не превосходит t, т.е. $m \le \sum_{i=0}^{t} k^i = \frac{k^{t+1}-1}{k-1} \le k^{t+1} = 2^{(t+1)\log k} = 2^{O(t)}$. ∎

Доказательство теоремы. Работу детерминированной машины Тьюринга организуем следующим образом:

— порождаем Y_1, проверяем $P(X, Y_1)$ $\le t$ шагов,

$\vdots$

— порождаем Y_m, проверяем $P(X, Y_m)$ $\le t$ шагов.

Общее число шагов не превосходит $m \cdot t \le t\ 2^{O(t)} = 2^{O(t+\log t)} = 2^{O(t)}$. ∎

Следствие из доказательства теоремы. *Если недетерминированная машина Тьюринга, проверяющая предикат $\exists Y P(X, Y)$ заканчивает работу с исходными данными X за t шагов, то можно построить однолентоточную машину Тьюринга, результат работы которой с исходными данными X совпадает с результатом работы исходной и число используемых ячеек которой не превосходит t.*

Доказательство. Для доказательства этого следствия достаточно в доказательстве теоремы после каждой проверки $P(X, Y_m)$ возвращать головку машины Тьюринга в исходное положение. При этом число шагов может увеличиться вдвое. ∎

§3.4. Нормальные алгоритмы Маркова.

Теория нормальных алгоритмов (или алгорифмов, как называл их создатель теории) была разработана советским математиком А. А. Марковым (1903–1979) в конце 1940-х — начале 1950-х гг. XX в.

Определение. *Марковской подстановкой называется операция над словами $P \to Q$, состоящая в следующем. В обрабатываемом слове R находят первое вхождение слова P (если таковое имеется) и, не изменяя остальных частей слова R, это вхождение заменяют в нем словом Q. Полученное слово называется результатом применения марковской подстановки к слову R. Если же вхождения P в слово R нет, то считается, что марковская подстановка $P \to Q$ не применима к слову R.*

Марковская подстановка вида $P \to \cdot\, Q$ называется заключительной.

Упорядоченный конечный список подстановок

$$\begin{cases} P_1 \ \to [\cdot] \ Q_1, \\ P_2 \ \to [\cdot] \ Q_2, \\ \quad \vdots \\ P_r \ \to [\cdot] \ Q_r, \end{cases}$$

в алфавите A называется записью нормального алгоритма в A. (Запись точки в квадратных скобках означает, что она может стоять в этом месте, а может отсутствовать.)

Применение нормального алгоритма Маркова к слову V в алфавите A состоит в нахождении первого правила, которое можно применить к V и применении его. Процесс продолжается с полученным на предыдущем шаге словом. Т.е. на каждом шаге ищется первая подстановка, которую можно применить к текущему слову.

Если на некотором шаге применено заключительное правило или не пременимо ни одно из правил, то алгоритм заканчивает работу и его результатом считается полученное на последнем шаге слово.

Заметим, что на промежуточных шагах вычисления удобно использовать расширение исходного алфавита.

Принцип нормализации Маркова. *Всякая интуитивно вычислимая функция может быть вычислена с помощью нормального алгоритма.*

Кроме того справедлива следующая теорема.

Теорема 3.5. *Следующие классы функций, заданных на натуральных числах и принимающих натуральные значения, совпадают:*

а) класс всех функций, вычислимых по Тьюрингу;

б) класс всех частично рекурсивных функций;

в) класс всех нормально вычислимых функций.

§3.5. Различие между математическими понятиями алгоритма и программами.

Широко распространено мнение, что алгоритм и программа – это одно и то же. По крайней мере всякая программа реализует некоторый алгоритм. В чём же имеется различие между программой для компьютера и математическим понятием алгоритма?

Прежде всего, всякое математическое понятие алгоритма имеет дело с потенциально бесконечным множеством конструктивно определённых

исходных данных. В то же время размер исходных данных для любой программы ограничен либо объёмом оперативной памяти, либо объёмом внешних носителей и т.п.

Существует замечательная константа 2^{202}. На первый взгляд – число как число. Студенты однажды на лекции очень быстро вычислили мне его на калькуляторе. Но что, если требуется произвести такое количество действий или использовать такой объём памяти? То есть представить его в унарной системе счисления. По сведениям астрономов и физиков это число больше, чем количество элементарных частиц в видимой части вселенной, а также больше, чем количество секунд, прошедших с момента Большого Взрыва (если он был).

В большинстве языков программирования имеются такие типы данных как *integer*, *real*, *string* и т.п. Правда ли, что компьютер действительно имеет дело с любыми целыми и вещественными числами, или с произвольными строками символов? В главе 1 мы уже вспоминали, что действия с целыми числами производятся по модулю 2^{16} или 2^{32}, а также рассмотрели, как с этим можно «бороться». Числа типа *real* – это и вовсе не вещественные числа, а рациональные, имеющие конечную десятичную (или двоичную) запись. Традиционным образом определяемые вещественные числа не являются конструктивными объектами и не могут быть алгоритмически (или программно) обработаны.

Но согласитесь, что такие математические понятия как машина Тьюринга или нормальный алгоритм Маркова отнюдь не приемлемы для практических вычислений, но являются лишь их теоретическими моделями.

В настоящее время имеется много математических понятий алгоритма, которые намного ближе к современным программам, но у них у всех исходные данные – это конструктивные объекты, множество которых потенциально бесконечно.

Одним из первых математических понятий алгоритма, которое напоминает язык программирования, является алгоритм Óливера.

Предполагается, что имеется потенциально бесконечная память, ячейки которой занумерованы. В каждую ячейку может быть записано рациональное число (пара – целое и целое положительное). Определены следующие операторы: $r_i := r_j + r_k$, $r_i := r_j - r_k$, $r_i := r_j \cdot r_k$, $r_i := r_j : r_k$, $if\ r_i = 0\ then\ gotoM$.

Другим математическим понятием алгоритма может служить базовая версия языка Pascal с той разницей, что ячейки, в которых хранятся

записи, потенциально бесконечны.

Наконец, широко распространена такая модель математического понятия алгоритма как PAM (Random Access Memory) – машина с прямым доступом, в которой разрешена операция сложения. Имеются её модификации: PAM с умножением и BOOL-PAM (разрешены поразрядные логические операции с бинарными строками).

Для всех этих математических понятий алгоритма имеют место утверждения, аналогичные Тезису Чёрча.

Пока речь идёт только о возможности построения алгоритма, решающего ту или иную задачу, все эти понятия эквивалентны. В главе 4 будет рассмотрена теория сложности алгоритмов. Для адекватной оценки времени рабты программы на компьютере подходящими из перечисленных являются лишь детерминированные модификации машины Тьюринга и PAMы.

Упражнения.

Машина Тьюринга.

Рассмотрим несколько примеров программ машин Тьюринга.

Пример 1.

Написать программу машины Тьюринга, вычисляющую сумму двух натуральных (неотрицательных целых) чисел, записанных в унарной системе счисления.

Вид исходных данных для любого алгоритма должен быть строго определен. Поэтому нельзя написать программу машины Тьюринга, вычисляющую сумму двух чисел, пока не задана система счисления. Унарная система счисления — это так называемая единиричная (или палочковая) система: каково число — столько единичек (палочек).

Начальная конфигурация машины Тьюринга будет $q_1 1...1 + 1...1$, где количество единиц в первом слагаемом равно x, а количество единиц во втором слагаемом равно y. Требуется, чтобы заключительной конфигурацией была $q_0 1...1$, где количество единиц равно $x + y$. Для этого разработаем следующий план работы машины Тьюринга.

1. Сотрем первую единицу.

2. Сдвигаем головку вправо, пока не увидим знак +, который заменим на 1.

3. Сдвигаем головку влево, пока не увидим знак * (пустой символ).

4. Сдвинем головку на один символ вправо и остановимся.

Реализация каждого пункта этого плана требует свое собственное состояние, поэтому, записав команды, реализующие один из пунктов плана, обязательно будем менять состояние машины Тьюринга.

1.1) $q_1 1 \to q_2 *$

1.2) $q_1 + \to q_0 R *$

2.1) $q_2 1 \to q_2 R 1$

2.2) $q_2 + \to q_3 1$

3.1) $q_3 1 \to q_3 L 1$

4.1) $q_3 * \to q_0 R *$

Запишем протокол работы этой машины Тьюринга, вычисляющей 2 + 3.

$q_1 11 + 111$ (по 1.1)

$q_2 1 + 111$ (по 2.1)

$1 q_2 + 111$ (по 2.2)

$1 q_3 1111$ (по 3.1)

$q_3$11111 (по 3.1)
$q_3 * 11111$ (по 4.1)
$q_0$11111

В этой машине Тьюринга был использован алфавит {∗,1,+}. Однако можно было бы использовать и алфавит {∗,1}, при этом исходные данные разделяются знаком ∗, а в команде 2.2) вместо + следует поставить ∗.

Заметим, что число шагов работы этой машины Тьюринга равно $2n+1$, где n — длина записи первого слагаемого (т.е., в частности, оно само, так как использована унарная запись натурального числа).

Пример 2.

Описать работу машины Тьюринга, вычисляющей сумму двух натуральных чисел, записанных в двоичной системе счисления.

Начальная конфигурация машины Тьюринга будет q_1X^*Y, где X и Y — двоичные записи натуральных чисел. Требуется, чтобы заключительной конфигурацией была q_0Z, где Z — двоичная запись суммы. Для этого разработаем план работы машины Тьюринга.

1. «Добежим вправо» до разделяющего аргументы пустого символа ∗.

2. «Добежим вправо» до последней непомеченной цифры числа Y. (В начальный момент все цифры не помечены.)

3. Запомним эту цифру и пометим ее. Отметим, что машина Тьюринга может запоминать что-либо только номером состояния. Пометить цифру можно, например, заменив 0 на a, а 1 на b.

4. «Добежим влево» до последней непомеченной цифры числа X, запомним эту цифру и пометим ее.

5. «Добежим влево» до первой цифры числа X и отступим на одну клетку влево.

6. «Добежим влево» до первой цифры числа, полученного сложением просмотренных частей X и Y. Отступив на одну клетку влево, запишем сумму запомненных цифр, при этом, если складывались $1+1$, то записываем 1 и запоминаем, что к сумме следующих двух цифр будет необходимо прибавить 1.

7. «Добежим вправо» до ∗, стоящей после последней цифры числа вычисленной части суммы и перейдем к выполнению п. 1 нашего плана.

8. Если одно из слов (X либо Y) оказалось короче другого, то ∗, стоящую перед соответствующим словом, будем запоминать для сложения как цифру 0.

9. Если оба аргумента полностью помечены, то сотрем помеченные цифры, переведем головку в начало результирующего слова и остановимся.

Как видно из этого плана, программа машины Тьюринга будет очень длинной и потребует большого количества состояний. Поэтому этот пример рассмотрим еще раз для другой модификации машины Тьюринга.

Однако по приведённому плану можно оценить, что число шагов работы такой машины Тьюринга с точностью до мультипликативной константы не превосходит $n \cdot m$, где n и m — длины записи X и Y соответственно, т.е. составляет $O(n \cdot m)$.

Пример 3.

Написать 3-ленточную машину Тьюринга, вычисляющую сумму двух натуральных чисел, записанных в двоичной системе счисления.

Начальная конфигурация машины Тьюринга будет $q_1 \begin{pmatrix} X \\ Y \\ * \end{pmatrix}$, где X и Y — двоичные записи натуральных чисел. Требуется, чтобы заключительной конфигурацией была $q_0 \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$, где Z — двоичная запись суммы.

Пусть символы s_1 и s_2 обозначают любую из цифр 0 или 1. Тогда программа 3-ленточной машины Тьюринга будет иметь вид.

1.1. $q_1 \begin{pmatrix} s_1 \\ s_2 \\ * \end{pmatrix} \to q_1 \begin{pmatrix} R \\ R \\ - \end{pmatrix} \begin{pmatrix} s_1 \\ s_2 \\ * \end{pmatrix}$

1.2. $q_1 \begin{pmatrix} * \\ s_2 \\ * \end{pmatrix} \to q_1 \begin{pmatrix} - \\ R \\ - \end{pmatrix} \begin{pmatrix} * \\ s_2 \\ * \end{pmatrix}$

1.3. $q_1 \begin{pmatrix} s_1 \\ * \\ * \end{pmatrix} \to q_1 \begin{pmatrix} R \\ - \\ - \end{pmatrix} \begin{pmatrix} s_1 \\ * \\ * \end{pmatrix}$

1.4. $q_1 \begin{pmatrix} * \\ * \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ L \\ - \end{pmatrix} \begin{pmatrix} * \\ * \\ * \end{pmatrix}$

В состоянии q_1 числа выравниваются по последней цифре.

2.1. $q_2 \begin{pmatrix} 0 \\ 0 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

2.2. $q_2 \begin{pmatrix} 0 \\ 1 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$

2.3. $q_2 \begin{pmatrix} 1 \\ 0 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$

2.4. $q_2 \begin{pmatrix} 1 \\ 1 \\ * \end{pmatrix} \to q_3 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$

В состоянии q_2 складываются цифры, обозреваемые на первых двух лентах. В случае сложения двух единиц машина переходит в состояние q_3.

3.1. $q_3 \begin{pmatrix} 0 \\ 0 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

3.2. $q_3 \begin{pmatrix} 0 \\ 1 \\ * \end{pmatrix} \to q_3 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$

3.3. $q_3 \begin{pmatrix} 1 \\ 0 \\ * \end{pmatrix} \to q_3 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

3.4. $q_3 \begin{pmatrix} 1 \\ 1 \\ * \end{pmatrix} \to q_3 \begin{pmatrix} L \\ L \\ L \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

В состоянии q_3 складываются цифры, обозреваемые на первых двух лентах, сложенные с единицей. В случае сложения двух нулей машина возвращается в состояние q_2.

4.1. $q_2 \begin{pmatrix} * \\ 0 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} - \\ L \\ L \end{pmatrix} \begin{pmatrix} * \\ 0 \\ 0 \end{pmatrix}$

4.2. $q_2 \begin{pmatrix} * \\ 1 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} - \\ L \\ L \end{pmatrix} \begin{pmatrix} * \\ 1 \\ 1 \end{pmatrix}$

4.3. $q_2 \begin{pmatrix} 0 \\ * \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ - \\ L \end{pmatrix} \begin{pmatrix} 0 \\ * \\ 0 \end{pmatrix}$

4.4. $q_2 \begin{pmatrix} 1 \\ * \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ - \\ L \end{pmatrix} \begin{pmatrix} 1 \\ * \\ 1 \end{pmatrix}$

4.5. $q_3 \begin{pmatrix} * \\ 0 \\ * \end{pmatrix} \to q_2 \begin{pmatrix} - \\ L \\ L \end{pmatrix} \begin{pmatrix} * \\ 0 \\ 1 \end{pmatrix}$

4.6. $q_3 \begin{pmatrix} * \\ 1 \\ * \end{pmatrix} \to q_3 \begin{pmatrix} - \\ L \\ L \end{pmatrix} \begin{pmatrix} * \\ 1 \\ 0 \end{pmatrix}$

4.7. $q_3 \begin{pmatrix} 0 \\ * \\ * \end{pmatrix} \to q_2 \begin{pmatrix} L \\ - \\ L \end{pmatrix} \begin{pmatrix} 0 \\ * \\ 1 \end{pmatrix}$

4.8. $q_3 \begin{pmatrix} 1 \\ * \\ * \end{pmatrix} \to q_3 \begin{pmatrix} L \\ - \\ L \end{pmatrix} \begin{pmatrix} 1 \\ * \\ 0 \end{pmatrix}$

Команды 4.1. — 4.8. осуществляют сложение цифр в случае, когда одно из слагаемых короче другого.

5.1. $q_2 \begin{pmatrix} * \\ * \\ * \end{pmatrix} \to q_0 \begin{pmatrix} R \\ R \\ R \end{pmatrix} \begin{pmatrix} * \\ * \\ * \end{pmatrix}$

5.2. $q_3 \begin{pmatrix} * \\ * \\ * \end{pmatrix} \to q_0 \begin{pmatrix} R \\ R \\ - \end{pmatrix} \begin{pmatrix} * \\ * \\ 1 \end{pmatrix}$

Несложно подсчитать, что эта трёхленточная машина Тьюринга заканчивает свою работу за число шагов, которое с точностью до мультипликативной константы не превосходит $\max\{n, m\}$, где n и m — длины записи X и Y соответственно, т.е. составляет $O(\max\{n, m\})$.

Пример. Перевести двоичную запись натурального числа в восьмеричную.

Для решения этой задачи можно ввести вспомогательные символы ♯ и §, которые помогут нам отделять по 3 цифры с конца слова. Сначала пометим начало слова символом ♯ (подстановка 14) и «протащим» его до конца слова (подстановки 1 и 2). Подстановка 14 записана последней, т.к. символ ♯ вставляется перед первым вхождением символа исходного

слова. Все промежуточные слова будут содержать как исходные, так и служебные символы, поэтому чтобы команды с ними выполнялись, они должны предшествовать командам без служебных символов. На конце слова заменим ♯ на §.

Три символа перед § будем заменять на соответствующую восьмеричную цифру (подстановки 4 – 10). Поскольку в двоичной записи числа нет ведущих нулей, то для двух первых цифр, соответствующих восьмеричным цифрам 2 и 3, записаны подстановки 11 и 12. Для первой восьмеричной единицы записана подстановка 13.

1. $\sharp c \to c\sharp$
2. $\sharp \to \S$
3. $000\S \to \S 0$
4. $001\S \to \S 1$
5. $010\S \to \S 2$
6. $011\S \to \S 3$
7. $100\S \to \S 4$
8. $101\S \to \S 5$
9. $110\S \to \S 6$
10. $111\S \to \S 7$
11. $10\S \to \cdot\, 2$
12. $11\S \to \cdot\, 3$
13. $1\S \to \cdot\, 1$
14. $1 \to \sharp 1$

Процесс работы этого нормального алгоритма с двоичным числом 1001110011 выглядит следующим образом (запись $\overset{n}{\longmapsto}$ означает, что переход от слова к слову осуществлён по правилу n):

$$1001110011 \overset{14}{\longmapsto} \sharp 1001110011 \overset{1}{\longmapsto} 1\sharp 001110011 \overset{1}{\longmapsto} 10\sharp 01110011\ldots$$

$$\overset{1}{\longmapsto} 100111001\sharp 1 \overset{1}{\longmapsto} 1001110011\sharp \overset{2}{\longmapsto} 1001110011\S$$

$$\overset{6}{\longmapsto} 1001110\S 3 \overset{8}{\longmapsto} 1001\S 53 \overset{4}{\longmapsto} 1\S 153 \overset{13}{\longmapsto} 1153$$

Пример. Проверить, является ли слово в алфавите A палиндромом (т.е. читается одинаково как слева направо, так и справа налево, в частности, пустое слово). В случае положительного ответа записать символ $\top$, а в случае отрицательного – символ $\bot$.

Пометим 1-ый символ слова (правило 6) и «протащим» его в конец (правило 3).

Если на конце полученного слова находятся одинаковые буквы (непомеченная и помеченная), то стираем их (правило 4). Поскольку меток нет, то начинаем проверку сначала, причём если стёрты все символы или остался только один символ, то пишем ответ $\top$ (правила 7 и 8).

Если на конце полученного слова находятся разные буквы (непомеченная и помеченная), то стираем их и ставим символ $\bot$ (правило 5). Все символы перед $\bot$ стираем (правила 1 и 2).

В приведённом ниже алгоритме использованы «макро-правила», в которых буква c использована вместо любой буквы алфавита A. При использовании в одном правиле букв c и c_1 имеется в виду, что это различные буквы алфавита A.

$$
\begin{array}{lrcl}
1. & c\bot & \to & \bot \\
2. & \bot & \to \cdot & \bot \\
3. & \sharp c\sharp c_1 & \to & c_1\sharp c\sharp \\
4. & c\sharp c\sharp & \to & \\
5. & c_1\sharp c\sharp & \to & \bot \\
6. & cc_1 & \to & c_1\sharp c\sharp \\
7. & c & \to \cdot & \top \\
8. & & \to \cdot & \top
\end{array}
$$

Процесс работы этого нормального алгоритма со словом NON выглядит следующим образом:

$$NON \overset{6}{\longmapsto} O\sharp N\sharp N \overset{3}{\longmapsto} ON\sharp N\sharp \overset{4}{\longmapsto} O \overset{7}{\longmapsto} \top$$

Процесс работы этого нормального алгоритма со словом NOT выглядит следующим образом:

$$NOT \overset{6}{\longmapsto} O\sharp N\sharp T \overset{3}{\longmapsto} ON\sharp T\sharp \overset{5}{\longmapsto} O\bot \overset{1}{\longmapsto} \bot \overset{2}{\longmapsto} \bot$$

Для нормальных алгоритмов имеет место утверждение, аналогичное тезису Чёрча и тезизу Тьюринга – Чёрча [18].

§3.6. Теоремы о невозможности построения алгоритма.

Как уже говорилось, математические понятия алгоритма появились на свет благодаря тому, что возникла необходимость доказывать, что не существует алгоритма, решающего ту или иную проблему. В этом параграфе будут доказаны теоремы для некоторых таких задач. Естественно, что такие задачи должны быть чётко сформулированы в однозначно понимаемых терминах.

3.6.1. Код алгоритма. Применимость алгоритма к данным. Универсальный алгоритм.

Поскольку каждый алгоритм (в терминах математичекого понятия алгоритма) может быть задан своей программой (или термом для рекурсивных функций), которая является словом в конечном алфавите, то это слово будем называть кодом алгоритма. Код алгоритма A будем обозначать посредством $\#A$.

Определение. *Алгоритм A называется применимым к данным P, если он заканчивает работу над данными P за конечное число шагов.*

$$!A(P)$$

Определение. *Алгоритм A называется самоприменимым если он приме́нм к собственному коду.*

$$!A(\#A)$$

Определение. *Алгоритм A называется самоанулируемым если результат его применения к собственному коду равен пустому слову (нулю, если он обробатывает числа).*

$$A(\#A) = \Lambda$$

Определение. *Алгоритм U называется универсальным, если для любого алгоритма A и исходных данных P, к которым он приме́нм, U приме́нм к $\#A$ и P и результаты их работы совпадают*

$$\forall AP(!A(P) \rightarrow !U(\#A, P)\ \&\ U(\#A, P) = A(P)).$$

В некотором смысле универсальный алгоритм является аналогом одного из видов компьютерных трансляторов, а именно интерпретатора.

Определение. *Алгоритм B называется продолжением алгоритма A ($A \subset B$), если для любых исходных данных P, к которым приме́нм алгоритм A, алгоритм B тоже приме́нм к ним и результаты их работы совпадают*

$$\forall P(!A(P) \rightarrow !B(P)\ \&\ A(P) = B(P).$$

3.6.2. Теоремы о несуществовании алгоритма

Теорема 3.6.1. *Не существует такого алгоритма B, который примени́м к кодам тех и только тех алгоритмов, которые не являются самоприменимыми*

$$\neg\exists B\forall A(!B(\#A) \leftrightarrow \neg!A(\#A)).$$

Доказательство. Предположим что такой алгоритм B_0 существует

$$\forall A(!B_0(\#A) \leftrightarrow \neg!A(\#A)).$$

Тогда при $A = B_0$ верно

$$!B_0(\#B_0) \leftrightarrow \neg!B_0(\#B_0),$$

что невозможно. ∎

Теорема 3.6.2. *Не существует такого алгоритма B, который равен нулю на кодах тех и только тех алгоритмов, которые не являются самоанулируемыми*

$$\neg\exists B\forall A(B(\#A) = \Lambda \leftrightarrow A(\#A) \neq \Lambda).$$

Доказательство. Предположим что такой алгоритм B_0 существует

$$\forall A(B_0(\#A) = \Lambda \leftrightarrow A(\#A) \neq \Lambda).$$

Тогда при $A = B_0$ верно

$$B_0(\#B_0) = \Lambda \leftrightarrow B_0(\#B_0) \neq \Lambda,$$

что невозможно. ∎

Теорема 3.6.3. *Не существует всюду применимого продолжения универсального алгоритма.*

Доказательство. Предположим что такой алгоритм B_0 существует.

Построим алгоритм C, определяемый равенством $\forall x(C(x) = B_0(x, x)||a_1)$, т.е. к результату его работы приписан символ a_1. Этот алгоритм всюду примени́м и, следовательно, примени́м к собственному коду и $C(\#C) = B_0(\#C, \#C)||a_1$.

Так как B_0 — продолжение универсального алгоритма, то верно, что $\forall AP(!A(P) \rightarrow !B_0(\#A, P)\ \&\ B_0(\#A, P) = A(P))$, в частности, при $A =$

C, $P = \#C$ $B_0(\#C, \#C) = C(\#C)$. Что противоречит полученному ранее значению для $C(\#C)$. ∎

§3.7. Массовые проблемы. Алгоритмическая разрешимость и неразрешимость.

Определение. *Массовой проблемой называется задача вида*

$$(?x)\ \varphi(x),$$

где $\varphi(x)$ – формула какого-либо формализованного языка со свободной переменной x и которая читается «при каких x верна формула $\varphi(x)$?».

Примерами массовых проблем могут служить следующие:

1. $(?a, b, c)\ \exists x(x \in \mathbf{R}\ \&\ ax^2 + bx + c = 0)$,
2. $(?m)(\forall x(x \in [0, 1] \to mx^2 + 2(m-1)x + m - 3 > 0))$,
3. $(?P)\ \exists x(x \in \mathbf{Z}\ \&\ P$– многочлен с целыми коэффициентами $\&$ $P(x) = 0)$,
4. $(?P)\ \exists \overline{x}(\overline{x} \in \mathbf{Z}^*\ \&\ P$– полином с целыми коэффициентами $\&$ $P(\overline{x}) = 0)$,
5. $(?F)$ (F– тавтологичная пропозициональная формула),
6. $(?F)$ (F– общезначимая формула исчисления предикатов).

Определение. *Массовая проблема $(?x)\varphi(x)$ называется алгоритмически разрешимой, если существует всюду применимый алгоритм B, равный пустому слову Λ на тех и только тех значениях параметра x, для которых верна формула $\varphi(x)$*

$$\exists B \forall x(B(x) = \Lambda \leftrightarrow \varphi(x)).$$

В приведённых примерах все массовые проблемы, кроме 4 и 6, являются алгоритмически разрешимыми. В 1-ой достаточно проверить знак дискриминанта. 2-ая — стандартная задача с параметром. Для проверки 3-ей достаточно разложить свободный член многочлена на сомножители и проверить, являются ли делители свободного члена (или они со знаком минус) корнями многочлена. Для решения 5-ой достаточно построить таблицу истинности.

В 4-ом примере сформулирована X проблема Гильберта. Сам Гильберт к началу XX века формулировал её как «построить алгоритм, позволяющий по произвольному полиному с целыми коэффициентами найти

все его целые корни». Эта проблема в отрицательном смысле (т.е. что такого алгоритма не существует) была решена в 1969г.

В 6-ом примере сформулирована проблема проверки общезначимости (что равносильно выводимости в исчислении предикатов) предикатной формулы. Схема доказательства её алгоритмической неразрешимости будет дана далее.

3.7.1. Теоремы об алгоритмической неразрешимости некоторых массовых проблем

Докажем алгоритмическую неразрешимость некоторых простейших массовых проблем.

Теорема 3.7.1. *Массовая проблема самоприменимости алгоритма*

$$(?A)\ !A(\#A)$$

алгоритмически неразрешима.

Доказательство (от противного). Предположим, что $(?A)\ !A(\#A)$ алгоритмически разрешима, т.е. имеется всюду применимый алгоритм B_0, такой что

$$\forall A(B_0(\#A) = \Lambda \leftrightarrow !A(\#A)).$$

Построим алгоритм C, который применим к данным x тогда и только тогда, когда $B_0(x) \neq \Lambda$. Доказательство проведём для такого математического понятия алгоритма как машина Тьюринга.

Пусть M_1 — программа машины Тьюринга, вычисляющей B_0 и имеющей состояния $q_0, q_1, ..., q_{k1}$.

Заменим в программе M_1 состояние q_0 на q_{k1+1} и добавим команды

$q_{k1+1}\ a \to q_0\ a$, где a — любой непустой символ внешнего алфавита,

$q_{k1+1}\ * \to q_{k1+2}\ *$

Эта машина Тьюринга остановится, если $B_0(P) \neq \Lambda$ (т.е. $\neg !A(\#A)$), и головка будет бесконечно стоять на ячейке, в которой записан пустой символ, если $B_0(P) = \Lambda$, (т.е. $!A(\#A)$). Но по теореме 1 такой машины Тьюринга не существует. ∎

Теорема 3.7.2. *Массовая проблема самоанулируемости алгоритма*

$$(?A)\ !A(\#A)$$

алгоритмически неразрешима.

Д о к а з а т е л ь с т в о (от противного). Предположим, что $(?A)\ A(\#A) = 0$ алгоритмически разрешима, т.е. имеется алгоритм B_0, такой что

$$\forall A(B_0(\#A) = \Lambda \leftrightarrow A(\#A) = 0).$$

Построим алгоритм C, который равен нулю на тех и только тех данных x для которых $B_0(x) \neq 0$. Доказательство проведём для такого математического понятия алгоритма как машина Тьюринга.

Пусть M_1 — программа машины Тьюринга, вычисляющей B_0 и имеющей состояния $q_0, q_1, ..., q_k$ и обрабатывающей слова в алфавите $\{a_1, \ldots, a_n, *\}$.

Заменим в программе M_1 состояние q_0 на q_{k1+1} и добавим команды

$q_{k+1} * \to q_0\ a_1$

$q_{k+1}\ a_i \to q_{k+2}\ R\ *$, где $i \in \{1, \ldots, n\}$

$q_{k+2}\ a_i \to q_{k+2}\ R\ *$, где $i \in \{1, \ldots, n\}$

$q_{k+2} * \to q_0\ *$

Эта машина Тьюринга даёт в ответе Λ, если $B_0(P) \neq \Lambda$ (т.е. $A(\#A) = \Lambda$) и непустое значение, если $B_0(P) = \Lambda$, (т.е. $A(\#A) \neq 0$). Но по теореме 2 такой машины Тьюринга не существует. ∎

Теорема 3.7.3. *Массовая проблема применимости алгоритма к данным алгоритмически неразрешима ни в одной из следующих формулировок*

1. $(?A\ P)\ !A(P)$,
2. $(?A)\ !A(P)$,
3. $(?P)\ !A(P)$.

Замечание. Во второй и третьей формулировках имеются свободные переменные P и A соответственно. По ним предполагается квантор существования. Из алгоритмической неразрешимости проблемы в третьей формулировке следует алгоритмическая неразрешимость проблемы в первой формулировке.

Кроме того, если доказана алгоритмическая неразрешимость проблемы в третьей формулировке для конкретного алгоритма, и в качестве данных P взяты те данные, для которых невозможно определить применимость к ним этого алгоритма, то тем самым будет доказана алгоритмическая неразрешимость проблемы во второй формулировке.

Поэтому докажем алгоритмическую неразрешимость проблемы в третьей формулировке для универсального алгоритма.

Лемма. *Массовая проблема применимости универсального алгоритма к данным* $(?P)\ !U(P)$ *алгоритмически неразрешима.*

Доказательство (от противного). Предположим, что $(?P)\ !U(P)$ алгоритмически разрешима, т.е. имеется алгоритм B_0, такой что

$$\forall P(B_0(P) = \Lambda \leftrightarrow !U(P)).$$

Построим алгоритм C, который для кода любого алгоритма A и исходных данных P в качестве ответа выдаёт $A(P)$, если $B_0(\#A, P) = \Lambda$, и останавливается иначе. Доказательство проведём для такого математического понятия алгоритма как машина Тьюринга.

Пусть M_1 — программа машины Тьюринга, вычисляющей B_0 и имеющей состояния $q_0, q_1, ..., q_{k1}$. Без потери общности можно считать, что при работе этой машины Тьюринга головка не сдвигается левее своего начального положения.

Пусть M_2 — программа машины Тьюринга, вычисляющей U и имеющей состояния $q_0, q_1, ..., q_{k2}$.

В качестве упражнения можно построить машину Тьюринга, которая дублирует входное слово и останавливается в начале его второго экземпляра. Пусть эта машина имеет программу M_0 и имеет состояния $q_0, q_1, ..., q_{k0}$.

Заменим в программе M_0 состояние q_0 на q_{k0+1}, в программе M_1 состояния q_i на q_{k0+i} и q_0 на $q_{k0+k1+1}$ и добавим команды

$q_{k0+k1+1}\ \Lambda \to q_{k0+k1+2}\ L\ *$

$q_{k0+k1+1}\ a \to q_0\ a$, где a — любой непустой символ внешнего алфавита

$q_{k0+k1+2}\ a \to q_{k0+k1+2}\ L\ a$, где a — любой непустой символ внешнего алфавита

$q_{k0+k1+2}\ * \to q_{k0+k1+3}\ R$

Добавим также программу M_2, в которой заменим состояния q_i $(i \neq 0)$ на $q_{k0+k1+2+i}$.

Эта машина Тьюринга является всюду применимым продолжением универсальной машины Тьюринга. Но по теореме 2 такой машины Тьюринга не существует. ∎

Теорема 3.7.4. *Массовая проблема проверки общезначимости предикатной формулы алгоритмически неразрешима.*[3]

[3]Обычно говорят, что исчисление предикатов алгоритмически неразрешимо.

С х е м а д о к а з а т е л ь с т в а. Построим по программе M универсального алгоритма и исходным данным P предикатную формулу, которая истинна тогда и только тогда, когда универсальный алгоритм применим к данным P.

Для этого достаточно рассмотреть исходные предикаты $O(i,k) \Leftrightarrow$ «на i-ом шаге машина M находится в состоянии q_k», $H(i,j) \Leftrightarrow$ «на i-ом шаге машина M обозревает j-ю ячейку», $S(i,j,h) \Leftrightarrow$ «на i-ом шаге в j-ой ячейке записан символ a_h».

С помощью этих предикатов можно описать весь процесс работы универсального алгоритма над исходными данными. Тот факт, что $!U(P)$ запишется формулой, в посылке импликации которой стоит описание работы U над P, а в заключении формула $\exists i O(i,0)$.

Если исчисление предикатов алгоритмически разрешимо, то существует алгоритм, который по каждой такой формуле проверяет, общезначима ли она (а следовательно, истинна в предложенной интерпретации) или нет. Тем самым построен алгоритм, проверяющий применимость универсального алгоритма к данным. ■

ГЛАВА 4. ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ

§4.1. Задачи, приводящие к понятию вычислительной сложности алгоритма

Пример 1. Можно ли по числу выполненных операторов программы оценить время её работы? Например, верно ли, что выполнение следующих трёх операторов займёт три единицы времени?

1. $x := z^2$;
2. $y := 5$;
3. $z := \log(\sin(xy))$;

В зависимости от того, кем написан транслятор для этого языка программирования, первый оператор либо будет представлен в кодах в виде *if «показатель степени» = 2 then* $x := z * z$; *else .....*, либо значение x будет вычисляться по формуле $e^{2*\ln z}$, при этом значения и логарифма, и экспоненты вычисляются с помощью рядов с некоторой (достаточно хорошей) точностью. В большинстве трансляторов используется первый вариант для вычисления квадрата, куба, ... А для вычисления 25-ой степени?

Второй оператор, безусловно, можно считать как выполненным за одну единицу времени.

Выполнение же третьего оператора обязательно будет использовать разложения в ряд функций log и sin.

Так сколько же здесь шагов вычисления?

Пример 2. Подсчитаем число «шагов» вычисления функции a^n при различных разрешённых элементарных операциях.

Если можно использовать операции возведения в степень, умножение и сложение, то значение функции будет вычислено за один «шаг» (возведение в степень).

Если же запретить использование операции возведения в степень (причины см. в первом примере), то используя алгоритм быстрого возведения получим, что число «шагов» (операций умножения) имеет порядок $\log n$.

Кроме того, если числа a и n достаточно велики, то как окончательный результат, так и результату многих промежуточных вычислений не поместятся в ячейку. Следовательно, необходимо выполнять действия с числами произвольной длины, а это потребует выполнения дополнитель-

ных шагов работы программы (см. главу 1 этого учебного пособия).

Пример 3. Вычисление значения $x + y$ на одноленточной машине Тьюринга при условии, что x и y заданы в унарной системе счисления.

Из исходной конфигурации $q_1 \underbrace{1\ldots1}_{x} + \underbrace{1\ldots1}_{y}$ машина в процессе вычисления за x шагов придёт в конфигурацию $\underbrace{1\ldots1}_{x-1} q_2 + \underbrace{1\ldots1}_{y}$. Двигаясь влево машина придёт в заключительную конфигурацию $q_0 \underbrace{1\ldots1}_{x+y}$ ещё за $x + 1$ шаг.

Всего $2x + 1$ шагов. Отметим, что в этой задаче число x и длина его записи совпадают ($||x|| = x$).

Пример 4. Вычисление значения $x + y$ на одноленточной машине Тьюринга при условии, что x и y заданы в двоичной системе счисления.

При переходе из исходной конфигурации $q_1 x + y$ в заключительную $q_0\{x + y\}$ (см. план работы такой машины Тьюринга в примере 2 главы 3) машине придётся многократно проходить запись на ленте для того, чтобы запомнить очередной (справа) символ записей x и y, а также записать результат их сложения (быть может, с прибавлением единицы из предыдущего результата сложения цифр). Общее число шагов составит $O(||x|| \cdot ||y||)$.

Отметим, что в этой задаче число x и длина его записи связаны соотношением ($||x|| = O(\log_2 x)$).

Пример 5. Вычисление значения $x + y$ на трёхленточной машине Тьюринга при условии, что x и y заданы в двоичной системе счисления.

Сложение чисел в этой модели происходит привычным нам способом сложения «в столбик» (см. пример 3 главы 3). Т.е. следует выровнять записи X и Y по правому краю и складывать цифры, каждый раз сдвигаясь на одну ячейку вправо.

Переход от исходной конфигурации $q_1 \begin{pmatrix} X \\ Y \\ * \end{pmatrix}$ к заключительной $q_0 \begin{pmatrix} X \\ Y \\ \{X + Y\} \end{pmatrix}$ произойдёт не более чем за $2\max\{||X||, ||Y||\} + 3$ шагов[4].

[4] Обратите внимание, что полученная оценка $O(\max\{||X||, ||Y||\})$ совпадает с оценкой числа «шагов» сложения чисел произвольной длины, полученной в главе 1.

Сравнивая число шагов примерах 3, 4, 5 может возникнуть ощущение, что самый быстрый способ вычисления представлен в примере 3. Сравним полученные оценки для $x \approx y \approx 1000$. В первом примере $||X|| = x \approx 1000$, а во втором и третьем $||X|| \approx ||Y|| \approx 10$. Число шагов вычисления в этих примерах составит соответственно

1. ≈ 2000
2. ≈ 200
3. ≈ 20.

Из этих примеров можно сделать следующее заключение.

При оценке числа шагов вычисления необходимо учитывать

- математическую модель алгоритма, с помощью которого они производятся;
- способ представления исходных данных;
- длину записи результата и промежуточных вычислений.

§4.2. Временная и ёмкостная (зональная) сложности алгоритма

Под вычислительной сложностью алгоритма понимают функцию, зависящую от ДЛИНЫ записи исходных данных и характеризующую

- *число шагов работы алгоритма над исходными данными* (временна́я сложность);
- *объём памяти, необходимой для работы алгоритма над исходными данными* (ёмкостная или зональная сложность).

Математики рассматривают также другие понятия сложности алгоритмов: алгебраическая сложность (количество арифметических операций)[5], схемная сложность (количество функциональных элементов, необходимых для построения схемы, реализующей вычисления), сложность по Колмогорову (длина записи алгоритма), скорость роста функции и многие другие. В этом курсе будут рассмотрены только временна́я и ёмкостная сложности алгоритмов.

Следующие определения относятся как к временно́й, так и к ёмкостной сложности, поэтому в определениях не будет уточняться, о какой именно сложности идёт речь.

[5] В главе 1 мы видели, что она плохо согласуется с временем вычисления.

Определение. *Сложностью $S_A(P)$ алгоритма A при работе над данными P называется число шагов или объём памяти, затраченные в процессе работы алгоритма A над данными P.*

Определение. *Верхней (нижней) оценкой сложности алгоритма A при работе над данными длины n называется*

$$S_A^U(n) = \max_{P:||P||=n}\{S_A(P)\}$$

(соответственно

$$S_A^L(n) = \min_{P:||P||=n}\{S_A(P)\}).$$

Определение. *Точной верхней оценкой сложности задачи Z с исходными данными длины n называется*

$$S_Z^U(n) = \min_{A:\ A \text{ решает } Z}\{S_A^U(n)\}.$$

Нахождение точной верхней оценки сложности задачи — довольно трудное (и к тому же неблагодарное) дело. Обычно устанавливают порядок таких оценок или их асимптотику. В дальнейшем изложении будет использована O-символика для указания на порядок роста функций сложности.

$$f(n) = O(g(n)) \quad \Longleftrightarrow \quad \exists C \forall n (f(n) \le C \cdot g(n)).$$

§4.3. Время реализации алгоритмов с различной временно́й сложностью

То, что экспонента растёт существенно быстрее, чем полином, нас учат начиная со школьных лет. Это всем известно, но таблица 1 (взятая из [4]) позволяет воочию убедиться, насколько практически неприменимы алгоритмы, имеющие экспоненциальную временну́ю сложность. Эта таблица предполагает, что скорость работы компьютера 10^6 операций в секунду.

	n					
$f(n)$	10	20	30	40	50	60
n	10^{-5} сек.	$2 \cdot 10^{-5}$ сек.	$3 \cdot 10^{-5}$ сек.	$4 \cdot 10^{-5}$ сек.	$5 \cdot 10^{-5}$ сек.	$6 \cdot 10^{-5}$ сек.
n^2	10^{-4} сек.	$4 \cdot 10^{-4}$ сек.	$9 \cdot 10^{-4}$ сек.	$16 \cdot 10^{-4}$ сек.	$25 \cdot 10^{-4}$ сек.	$36 \cdot 10^{-4}$ сек.
n^3	10^{-3} сек.	$8 \cdot 10^{-3}$ сек.	$27 \cdot 10^{-3}$ сек.	$64 \cdot 10^{-3}$ сек.	$1.25 \cdot 10^{-1}$ сек.	$2.16 \cdot 10^{-1}$ сек.
n^5	0.1 сек.	3.2 сек.	24.3 сек.	1.7 мин.	5.2 мин.	13 мин.
2^n	10^{-3} сек.	1 сек.	17.9 мин.	12.7 дней	35.7 лет	366 веков
3^n	0.059 сек.	58 мин.	6.5 лет	3955 веков	$2 \cdot 10^8$ веков	$1.3 \cdot 10^{13}$ веков

Табл. 1. Время вычисления при заданной временно́й сложности $f(n)$ с исходными данными длины n.

Существует мнение, что совершенствование компьютеров и увеличение скорости их работы позволят существенно уменьшить время работы любой программы. Таблица 2 (взятая из [4]) показывает изменение наибольшего размера исходных данных задачи, решаемой за 1 час.

Продолжить эту таблицу при других скоростях работы компьютера можно самим, используя формулу $f(Ni) \cdot 10^6 = f(Ni') \cdot 10^t$, Ni' — наибольший размер исходных данных задачи, решаемой за 1 час при скорости работы компьютера 10^t. Для полиномиальных по времени алгоритмов происходит извлечение корня k-ой степени из $Ni^k \cdot 10^{t-6}$. Поэтому увеличение размера происходит **в разы**. Для экспоненциальных по времени алгоритмов происходит логарифмирование по соответствующему основанию выражения $a^{Ni} \cdot 10^{t-6}$. Поэтому увеличение размера происходит **на единицы**.

скорость опер./сек.			
Функция временной сложности	10^6	10^8	10^9
n	N1	100· N1	1000· N1
n^2	N2	10· N2	31.6· N2
n^3	N3	4.64· N3	10· N3
n^5	N4	2.5· N4	3.98· N4
2^n	N5	N5 + 6.64	N5 + 9.97
3^n	N6	N6 + 4.19	N6 + 6.29

Табл. 2. Изменение размера задачи, решаемой за 1 час при увеличении быстродействия компьютера.

§4.4. Классы алгоритмов и задач. Классы P, NP и P-SPACE. Соотношения между этими классами.

В конце 70-х годов XX века сложилась следующая, сведённая в таблицу 3, система обозначений для некоторых классов сложности. Говорят, что задача принадлежит классу сложности **C**, если существует алгоритм из класса **C**, решающий эту задачу.

Буква **F** в начале названия класса используется для обозначения класса функций. Если её нет, то это класс предикатов.

Буквы **D** или **N** означают, что в определении класса сложности использована детерминированная или соответственно недетерминированная машина Тьюринга. Букву **D** обычно не пишут.

Функция сложности – это функция от длины записи исходных данных, ограничивающая число шагов или количество ячеек соответствующей машины Тьюринга. Так, например, **LOG** – логарифмическая функция, **LIN** – линейная функция, **QLIN** – квазилинейная функция (т.е.

Класс функций или предикатов	Детерминированная или недетерминированная	Функция сложности	Временна́я или ёмкостная
F	[D] N	LOG LIN QLIN P (Poly) EXP-LIN EXP ⋮	[TIME] SPACE

Табл. 3. Обозначения для некоторых классов сложности.

функция вида $(an + b) \log n$), **P** – полином и т.д.

Заметим, что для приведённых здесь функций временно́й сложности, начиная с **P** и ниже, не имеет значения, на какой детерминированной модели машины Тьюринга производились оценки числа шагов, так как все они моделируют друг друга за полином шагов от длины исходных данных. Однако для первых трёх функций при оценке временно́й сложности важно использование именно классической машины Тьюринга.

Например, класс **P** (полное обозначение в соответствии с приведёнными обозначениями **D-P-TIME**) – это класс предикатов, для которых существует алгоритм, который может быть реализован на детерминированной машине Тьюринга, число шагов которой не превосходит полинома от длины записи исходных данных.

Например, класс **NP** (полное обозначение в соответствии с приведёнными обозначениями **N-P-TIME**) – это класс предикатов, для которых существует алгоритм, который может быть реализован на недетерминированной машине Тьюринга, число шагов которой не превосходит полинома от длины записи исходных данных.

Например, класс **P-SPACE** (полное обозначение в соответствии с приведёнными обозначениями **D-P-SPACE**) – это класс предикатов, для которых существует алгоритм, который может быть реализован на детерминированной машине Тьюринга, число использованных ячеек которой не превосходит полинома от длины записи исходных данных.

В настоящее время известны, например, следующие соотношения между основными классами сложности: **P**$\subseteq$ **NP** $\subseteq$ **P-SPACE** $\subset$ **EXP**. Вопрос о том, строгими или нестрогими являются первые два включения, объявлен одним из труднейших вопросов математики на XXI век.

В качестве следствия теоремы 3.4 , доказанной в главе 3, можно доказать следующую теорему.

Теорема 4.1. *Если задача вида* $\exists Y\ P(X,Y)$ *принадлежит классу* **NP**, *то существует решающая её одноленточная машина Тьюринга, число шагов которой составляет* $2^{p(n)}$, *где* $p(n)$ – *полином от длины записи исходных данных* $n = ||X||$.

Как теорема 3.4, так и эта теорема доказываются с «большим запасом», т.е. предполагается, что «претенденты» на решение могут иметь длину, равную числу шагов недетерминированной машины Тьюринга, а также могут принимать в качестве значения любое слово в заданном алфавите. Последняя теорема была опубликована, например, в [4] ещё в 1979 г., но никакой более сильный результат до настоящего времени не известен. Таким образом, если задача принадлежит классу **NP**, то в настоящее время можно гарантировать только экспоненциальное от длины записи аргумента время её решения.

Дальнейшее изложение в этой главе будет посвящено, в основном, изучению класса **NP**.

§4.5. Полиномиальная сводимость и полиномиальная эквивалентность. Классы эквивалентности по отношению полиномиальной эквивалентности.

Определение. *Задача* Z_1 *вида* $\exists Y\ P_1(X,Y)$ *при* $X \in D_1$ *полиномиально сводится к задаче* Z_2 *вида* $\exists Y\ P_2(X,Y)$ *при* $X \in D_2$

$$Z_1 \propto Z_2,$$

если существует функция f, *отображающая* D_1 *в* D_2 *и такая, что*

- *существует машина Тьюринга, вычисляющая функцию* f *не более чем за полиномиальное от длины записи исходных данных число шагов (*$f \in$ **FP***);*
- *задача* Z_1 *имеет решение с исходными данными* X *тогда и только тогда, когда задача* Z_2 *имеет решение с исходными данными* $f(X)$

$$\forall X_{\in D_1}(\exists Y\ P_1(X,Y) \leftrightarrow \exists Y\ P_2(f(X),Y)).$$

Далее, если это не будет вызывать неправильного прочтения, под задачей Z_i всегда будем понимать задачу вида $\exists Y\ P_1(X,Y)$ при $X \in D_i$.

Лемма 4.1. *Отношение полиномиальной сводимости рефлексивно* $\forall Z(Z \propto Z)$ *и транзитивно* $\forall Z_1 Z_2 Z_3 (Z_1 \propto Z_2 \ \& \ Z_2 \propto Z_3 \to Z_1 \propto Z_3)$.

Доказательство непосредственно следует из того, что тождественное отображение принадлежит классу **FP**, а также сумма полиномов является полиномом.

Лемма 4.2. *Если* $Z_1 \propto Z_2$ *и* $Z_2 \in \mathbf{P}$, *то* $Z_1 \in \mathbf{P}$.

Доказательство леммы следует из того, что в качестве алгоритма решения задачи Z_1 можно взять следующий: к исходным данным X задачи Z_1 применяем функцию f, осуществляющую полиномиальную сводимость ($p_1(||X||)$ шагов), а затем решаем задачу Z_2 с данными $f(X)$ (длина записи которых не превосходит $p_1(||X||)$) за $p_2(p_1(||X||))$ шагов. Всего потребуется не более $p_1(||X||) + p_2(p_1(||X||))$ шагов. Здесь p_1 и p_2 – полиномы.

Пример полиномиальной сводимости.

Приведём пример задач, одна из которых полиномиально сводится к другой.[6]

ГАМИЛЬТОНОВ ЦИКЛ (ГЦ)	**КОМИВОЯЖЁР**
Дано: граф $G = (V, E)$	Дано: $C = \{c_1, ..., c_n\}$ – множество городов $d_{ij} \in \mathbf{Z}_+$ – расстояния между c_i и c_j $B \in \mathbf{Z}_+$
Вопрос: существует ли в G гамильтонов цикл?	Вопрос: существует ли маршрут, проходящий через все города, длина которого меньше B?
$\exists(v_{i_1}, ..., v_{i_n})(\{v_{i_1}, ..., v_{i_n}\} = V \ \&$ $\{v_{i_1}, v_{i_2}\} \in E \ \& \ , ..., \ \&$ $\{v_{i_n}, v_{i_1}\} \in E)$	$\exists(c_{i_1}, ..., c_{i_m})(\{c_{i_1}, ..., c_{i_m}\} = C \ \&$ $\sum_{j=1}^{m-1} d_{i_j i_{j+1}} + d_{i_m i_1} \leq B$

Покажем, что **ГЦ** $\propto$ **КОМИВОЯЖЁР**. Для этого предъявим полиномиальный по времени алгоритм, который по графу $G = (V, E)$ строит исходные данные C, d_{ij} и B с требуемыми свойствами.

$$C = V, \qquad B = n,$$

[6]Здесь и далее посредством $\mathbf{Z}_+$ будем обозначть множество целых положительных чисел.

$$d_{ij} = \begin{cases} 1, \text{ если } \{v_i, v_j\} \in E \\ 2, \text{ если } \{v_i, v_j\} \notin E \end{cases}.$$

В графе есть гамильтонов цикл, тогда и только тогда, когда маршрут проходит между теми городами, расстояния между которыми равно 1.

Определение. *Задача Z_1 полиномиально эквивалентна задаче Z_2 ($Z_1 \sim_p Z_2$), если $Z_1 \propto Z_2$ и $Z_2 \propto Z_1$.*

Теорема 4.2. *Отношение полиномиальной эквивалентности является отношением эквивалентности, то есть оно*
— рефлексивно $\forall Z(Z \sim_p Z)$,
— симметрично $\forall Z_1 Z_2(Z_1 \sim_p Z_2 \to Z_2 \sim_p Z_1)$ *и*
— транзитивно $\forall Z_1 Z_2 Z_3(Z_1 \sim_p Z_2 \ \& \ Z_2 \sim_p Z_3 \to Z_1 \sim_p Z_3)$.

Доказательство очевидно.

Отношение $\sim_p$ разбивает класс **NP** на классы эквивалентности. Один из них – класс **P** – самые «быстро решаемые» задачи из класса **NP**.

§4.6. NP-полные задачи.

Определение. *Задача Z называется NP-полной, если она сама принадлежит классу **NP** и любая задача из класса **NP** полиномиально сводится к ней.*

Теорема 4.3. *Класс NP-полных задач образует класс эквивалентности по отношению полиномиальной эквивалентности в классе **NP**.*

Кроме того, класс NP-полных задач — это класс самых «долго решаемых» задач из класса **NP**.

Замечание. Если в определении NP-полной задачи убрать требование её принадлежности классу **NP**, то получим определение NP-трудной задачи. В частности, если задача распознавания $\exists Y \ P(X, Y)$ является NP-полной, то соответствующая ей задача поиска $(?Y) \ P(X, Y)$ является NP-трудной.

На рис. 1 представлено взаимное расположение классов **P**, **NP** (верхняя оценка временной сложности возрастает при просмотре диаграммы снизу вверх).

§4.7. Задача ВЫПОЛНИМОСТЬ (ВЫП).Теорема Кука

Первым примером NP-полной задачи является следующая задача.

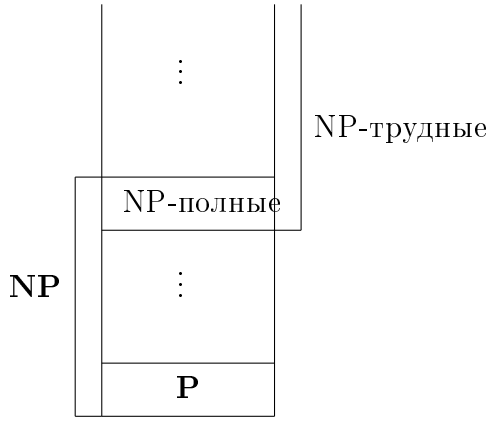


Рис. 1. Взаимное расположение классов **P**, **NP**,
NP-полных и NP-трудных задач.

ВЫПОЛНИМОСТЬ (ВЫП)

Дано: $U = \{u_1, ..., u_n\}$ – множество пропозициональных переменных,
$C = \{c_1, ..., c_m\}$ – множество предложений над U.

Вопрос: выполнимо ли множество C, т.е. существует ли набор значений

для переменных из U, для которого истинны все предложения из C?

$$\exists u_1, ..., u_n (c_1 \ \& \ ... \ \& \ c_m).$$

Список литературы

[1] Aho A.V., Hopcroft J.E., Ullman J.D.: The design and analysis of computer algorithms. Addison-Wesley Publishing Company Reading, Massfchusetts (1976)

[2] Du D.Z., Ko K.I. Theory of Computational Complexity. A Wiley-Interscience Publication. John Wiley & Sons, Inc. (2000)

[3] Forsyth R.S. Pascal at Work and Play. An introduction to computer programming in Pascal. Chapman & Hall. London, New-York (1982)

[4] Garey M.R., Johnson D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)

[5] Kosovskii N.K. Elements of mathematical logic and its applications to the theory of sub-recursive algorithms. Leningrad University Press, Leningrad (1981). (In Russian.)

[6] Kosovskiy N.K. Pspace-completeness of finite order predicate logics over finite domain // 3 International Conference "Smirnov's lectures". Moscow, 2001, p. 44. (In Russian)

[7] Kosovskaya T.M., Kosovskiy N.K. Belonging to **FP** of double-polynomial pascal-like function over subprograms from **FP**. In: Computer Tools in Education. No 3 (2010). (In Russian)

[8] Kosovskii N.K., Kosovskaya T.M.. Total algorithmic extension of an algorithm running on the bounded space. In: Vestnik of Sankt-Petersburg State University. Series 1: Mathematics, Mechanics, Astronomy. No 2 (2014). (Be published in Russian)

[9] Косовская Т.М., Косовский Н.К. О полиномиальных алгоритмах решения диофантовых систем линейных уравнений и сравнений // Материалы VIII Международного семинара «Дискретная математика и ее приложения», ч.1, М., МГУ, 2004.

[10] Схрейвер

[11] Окулов С.М. Программирование в алгоритмах. — М.: БИНОМ. Лаборатория знаний, 2007.

[12] Кнут Д. Искусство программирования на ЭВМ, т.3: Сортировка и поиск

[13] Лекции по теории графов

[14] Липский В.

[15] Новиков Ф.А.

[16] Романовский

[17] Bron C., Kerbosh J. (1973), Algorithm 457 — Finding all cliques of an undirected graph, Comm. of ACM, 16, p. 575—577.

[18] http://mathhelpplanet.com/static.php?p=normalnyye-algoritmy-markova