IPR MEDIA
ИЗДАТЕЛЬСТВО

8-800-555-22-35

izdat@iprmedia.ru

В учебном пособии излагаются некоторые аспекты вычислительной сложности при работе с целыми числами и графами, а также описаны основные понятия теории алгоритмов и некоторые классы сложности алгоритмов. Приводятся алгоритмы работы с «длинными» целыми числами, которые не помещаются в одну ячейку компьютера, доказываются оценки числа шагов работы этих алгоритмов. Анализируется число шагов решения некоторых задач на графах при разных способах их задания. Отдельная глава посвящена описанию трёх математических понятий алгоритма: рекурсивных функций, машин Тьюринга и их модификаций, нормальных алгоритмов Маркова. Доказываются теоремы о невозможности построения некоторых алгоритмов и об алгоритмической неразрешимости некоторых массовых проблем. Изложены основные понятия вычислительной сложности алгоритмов, даны сведения о современном делении алгоритмов на классы сложности. Учебное пособие предназначено для студентов, обучающихся по направлениям подготовки, связанным с технологиями программирования и искусственным интеллектом, и изучающих дисциплины «Теория алгоритмов», «Теория вычислительной сложности алгоритмов», «Анализ алгоритмов».

9 785449 718556

УЧЕБНОЕ ПОСОБИЕ

Алгоритмы и анализ их сложности

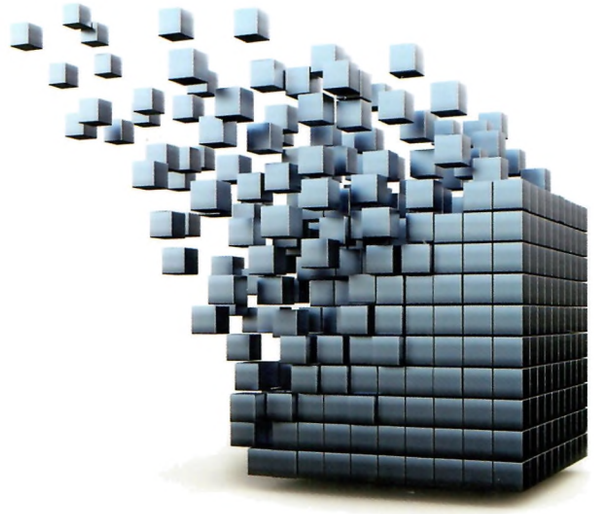
Т.М. Косовская

ФГОС ВО

Т.М. Косовская

Алгоритмы и анализ их сложности

Учебное пособие



IPR MEDIA
ИЗДАТЕЛЬСТВО

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Т.М. Косовская

АЛГОРИТМЫ И АНАЛИЗ ИХ СЛОЖНОСТИ

УЧЕБНОЕ ПОСОБИЕ

Москва
Ай Пи Ар Медиа
2023

УДК 510.5
ББК 22.127
К71

Автор:
Косовская Т.М. — д-р физ.-мат. наук, проф.,
и.о. зав. кафедрой информатики
Санкт-Петербургского государственного университета

Рецензенты:
Поздняков С.Н. — д-р пед. наук, зав. кафедрой
алгоритмической математики Санкт-Петербургского
государственного электротехнического университета «ЛЭТИ»;
Новиков Ф.А. — д-р техн. наук, проф. Санкт-Петербургского
политехнического университета им. Петра Великого

Косовская, Татьяна Матвеевна.
К71 Алгоритмы и анализ их сложности : учебное пособие /
Т.М. Косовская. — Москва : Ай Пи Ар Медиа, 2023. — 116 с.
ISBN 978-5-4497-1855-6

В учебном пособии излагаются некоторые аспекты вычислительной сложности при работе с целыми числами и графами, а также описаны основные понятия теории алгоритмов и некоторые классы сложности алгоритмов. Приводятся алгоритмы работы с «длинными» целыми числами, которые не помещаются в одну ячейку компьютера, доказываются оценки числа шагов работы этих алгоритмов. Анализируется число шагов решения некоторых задач на графах при разных способах их задания. Отдельная глава посвящена описанию трёх математических понятий алгоритма: рекурсивных функций, машин Тьюринга и их модификаций, нормальных алгоритмов Маркова. Доказываются теоремы о невозможности построения некоторых алгоритмов и об алгоритмической неразрешимости некоторых массовых проблем. Изложены основные понятия вычислительной сложности алгоритмов, даны сведения о современном делении алгоритмов на классы сложности.

Подготовлено с учётом требований Федерального государственного образовательного стандарта высшего образования.

Учебное пособие предназначено для студентов, обучающихся по направлениям подготовки, связанным с технологиями программирования и искусственным интеллектом, и изучающих дисциплины «Теория алгоритмов», «Теория вычислительной сложности алгоритмов», «Анализ алгоритмов».

ISBN 978-5-4497-1855-6

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 6

ГЛАВА 1. ПРОСТЕЙШИЕ ДИСКРЕТНЫЕ АЛГОРИТМЫ И ОЦЕНКИ ЧИСЛА ИХ ШАГОВ 8

1.1. Различие между числом шагов алгоритма и вычислительной сложностью алгоритма 8

1.2. Метод Гаусса для матриц с целыми коэффициентами. Анализ роста коэффициентов 10

1.3. Арифметика многоразрядных чисел 15

1.3.1. Представление неотрицательного многоразрядного числа как числа в системе счисления по заданному модулю 15

1.3.2. Запись многоразрядного числа из файла. Оценка числа шагов 16

1.3.3. Вывод многоразрядного числа в файл. Оценка числа шагов 18

1.3.4. Сложение двух неотрицательных многоразрядных чисел. Оценка числа шагов 19

1.3.5. Предикаты равенства и неравенств многоразрядных чисел. Оценка числа шагов 19

1.3.6. Вычитание двух положительных многоразрядных чисел. Оценка числа шагов 20

1.3.7. Умножение многоразрядного числа на макроцифру. Оценка числа шагов 21

1.3.8. Умножение многоразрядных чисел. Оценка числа шагов 22

1.3.9. Деление многоразрядных чисел. Оценка числа шагов 23

1.4. Сортировки и оценки числа шагов 26

1.4.1. «Пузырёк» 27

1.4.2. Сортировка слияниями фон Неймана 28

1.4.3. Сортировка подсчётом 29

ГЛАВА 2. АЛГОРИТМЫ НА ГРАФАХ 30

2.1. Различные способы представления графа в компьютере и оценки числа шагов решения простейших задач 30

2.1.1. Матрица смежности 31

2.1.2. Списки смежности 32

2.1.3. Матрица инцидентности 32
2.1.4. Массивы V и E 33
2.2. Оценки числа шагов некоторых стандартных алгоритмов 34
2.2.1. Алгоритм Дейкстры поиска кратчайшего пути во взвешенном ориентированном графе. Оценки числа шагов 34
2.2.2. Алгоритм Р. Прима нахождения остова минимального веса. Оценки числа шагов 35
2.2.3. Нахождение Гамильтонова цикла. Оценки числа шагов 36
2.2.4. Связь между понятиями «независимое множество», «вершинное покрытие» и «КЛИКА» 36
2.2.5. Алгоритм построения максимальной клики 37

ГЛАВА 3. ТЕОРИЯ АЛГОРИТМОВ 39
3.1. Интуитивное понятие алгоритма и необходимость введения его точного математического понятия 39
3.2. Представление о рекурсивных функциях. Тезис Чёрча 41
3.3. Машины Тьюринга. Тезис Тьюринга — Чёрча 43
3.3.1. Примеры программ машин Тьюринга 45
3.3.2. Теорема о композиции машин Тьюринга 48
3.3.3. Многоленточные машины Тьюринга 50
3.3.4. Теорема о числе шагов машины Тьюринга, моделирующей работу многоленточной машины Тьюринга 55
3.3.5. Многоголовчатые машины Тьюринга 56
3.3.6. Недетерминированные машины Тьюринга 57
3.3.7. Теорема о числе шагов машины Тьюринга, моделирующей работу недетерминированной машины Тьюринга 58
3.4. Нормальные алгоритмы Маркова 59
3.5. Конструктивные объекты 63
3.6. Различие между математическими понятиями алгоритма и программами 65
3.7. Теоремы о невозможности построения алгоритма 69
3.7.1. Код алгоритма. Применимость алгоритма к данным. Универсальный алгоритм 69
3.7.2. Теоремы о несуществовании алгоритма 70
3.8. Массовые проблемы. Алгоритмическая разрешимость и неразрешимость 71

ГЛАВА 4. ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ 76
4.1. Задачи, приводящие к понятию вычислительной сложности алгоритма 76
4.2. Временна́я и ёмкостная (зональная) сложности алгоритма 79
4.3. Время реализации алгоритмов с различной временно́й сложностью 80

ГЛАВА 5. КЛАССЫ СЛОЖНОСТИ АЛГОРИТМОВ 83
5.1. Классы P, NP и P-SPACE. Соотношения между этими классами 83
5.2. Полиномиальная сводимость и полиномиальная эквивалентность 85
5.3. NP-полные задачи 87
5.4. Задача ВЫПОЛНИМОСТЬ (ВЫП). Теорема Кука 88
5.5. Основные NP-полные задачи 90
5.6. Методы доказательства NP-полноты 92
5.7. Метод сужения доказательства NP-полноты 97
5.8. Анализ подзадач 98
5.9. Задачи с числовыми параметрами. Псевдополиномиальные задачи 99
5.10. «Похожие» задачи с разной вычислительной сложностью 102
5.11. Задачи с числовыми параметрами и сильная NP-полнота 104

СПИСОК ЛИТЕРАТУРЫ 109

ВВЕДЕНИЕ

Учебное пособие написано по результатам чтения лекций на математико-механическом факультете Санкт-Петербургского университета. В него вошли материалы, излагавшиеся в курсах «Алгоритмы и анализ сложности», «Анализ алгоритмов», «Математическая логика» (раздел «Теория алгоритмов») для студентов бакалавриата и «Дополнительные главы математической логики и теории алгоритмов» для студентов магистратуры.

Идея включения в учебное пособие материала, изложенного в гл. 1, возникла достаточно давно, после того, как Н.К. Косовский [1] обратил внимание на то, что при работе метода Гаусса с целыми числами (используются только операции умножения и сложения/вычитания) длина записи элементов матрицы растёт экспоненциально относительно номера итерации и, следовательно, относительно длины записи исходных данных. Однако, во-первых, во всех учебниках написано, что это полиномиальный по времени алгоритм, а во-вторых, при экспоненциальном росте длины записи результата алгоритм не может быть полиномиальным по времени. Обращение к специалистам по алгебре не дало положительного разрешения этой проблемы: в алгебре принято считать количество произведённых арифметических операций, не обращая внимания на рост длины записи результата. Люди же, занимающиеся приложениями, прекрасно знают о возникающих переполнениях и «борются» с ними разработкой приближённых методов.

Кроме того, при общении со студентами выяснилось, что во многих книгах (к счастью, не во всех) основные алгоритмы на графах излагаются при представлении графа матрицей смежности. Это часто объясняют тем, что работа с матрицей смежности очень проста и наглядна даже для начинающего программиста. В связи с этим возникло желание наглядно для студентов сравнить «время», точнее, число шагов работы программ для разных способов представления графа.

Рассмотрению таких вопросов посвящены гл. 1 и 2 учебного пособия.

Для многих студентов и даже работающих программистов понятия «алгоритм» и «программа» одинаковы. Чтобы избавить их от этого заблуждения, в гл. 3 учебного пособия даны начальные сведения о теории алгоритмов, в частности представление о рекурсивных функциях, и описаны такие важные математические понятия алгоритма, как ма-

шина Тьюринга, недетерминированная машина Тьюринга и нормальные алгоритмы Маркова.

Для любого программиста важными параметрами используемого или разрабатываемого алгоритма являются время его работы и объём используемой памяти. Определению того, что в настоящее время понимают под вычислительной сложностью алгоритма или вычислительной сложностью задачи, посвящено начало гл. 4.

В настоящее время широко известна проблема, равны или не равны классы ***P*** и ***NP***. Словами ***P*** — ***NP*** любят пощеголять даже те, кто понятия не имеет, о чём же, собственно, идёт речь. В связи с этим в гл. 4 излагаются (в несколько адаптированном по сравнению с [2] виде) основы современной теории сложности алгоритмов. В частности, рассматриваются классы ***P***, ***NP*** и ***P-SPACE***. Описываются некоторые основные методы доказательства *NP*-полноты задачи.

В издании изложены сведения, позволяющие приобрести знания о вычислительной сложности алгоритмов, а также умения анализировать поставленную задачу с точки зрения её вычислительной сложности и выбора метода решения.

Учебное пособие содержит упражнения, позволяющие студентам закрепить изложенный материал и приобрести навыки оценки вычислительной сложности алгоритмов.

ГЛАВА 1. ПРОСТЕЙШИЕ ДИСКРЕТНЫЕ АЛГОРИТМЫ И ОЦЕНКИ ЧИСЛА ИХ ШАГОВ

1.1. РАЗЛИЧИЕ МЕЖДУ ЧИСЛОМ ШАГОВ АЛГОРИТМА И ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТЬЮ АЛГОРИТМА

При работе с числами очень часто говорят о числе шагов алгоритма как о числе арифметических операций (сложения, вычитания, умножения или деления). В этом разделе будет показано, что такие «шаги» не всегда адекватны числу операций на компьютере, выполняемых за одинаковое или по крайней мере сравнимое время.

Здесь и далее запись $||A||$ будет использоваться для обозначения длины записи A, чтобы не путать с $|A|$ — «абсолютное значение числа A», «мощность множеств A» и т.п.

Пример 1. Число «шагов» вычисления функции f, где $f(n) = 2^n$.

Известен алгоритм «быстрого возведения в степень», вычисляющий 2^n за $\log_p n$ умножений, где p — основание системы счисления.

Длина записи натурального числа n приблизительно равна $||n|| \approx \approx \log_p n$. Следовательно, $||2^n|| \approx n \log_p 2$.

Верно ли, что за $\log_p n$ «шагов» возможно выписать число длиной n?

Что здесь понималось под словом «шаг»?

Пример 2. Вычисление n-го числа Фибоначчи.

Числа Фибоначчи определяются рекурсивно: $F_0 = 0, F_1 = 1, \ldots, F_{n+2} = = F_n F_{n+1}, \ldots$ Для их вычисления известен следующий алгоритм.

Пусть матрица P имеет вид

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Тогда $(F_{n-2}\ F_{n-1}) \cdot P = (F_{n-1}\ F_n)$, т.е. $(0\ 1)P^n = (F_n\ F_{n+1})$.

Следовательно, F_{n+1} вычисляется за n умножений матрицы P на себя и одно умножение на строку (0 1).

Сложность вычисления n-го числа Фибоначчи F_n линейна относительно его номера n?

Известно, что скорость роста этих чисел экспоненциальна. Точнее,

$$F_n = \left[\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n\right]$$

и $||F_n|| = O(n)$.

Получаем, что по исходному данному n, длина записи которого $||n|| \approx \log_p n$, выписывается результат, длина записи которого $||F_n|| = O(n)$.

В обоих этих примерах под **«шагом»** понималось выполнение умножения без учёта роста длины записи результата.

Следует различать вычислительную сложность алгоритма, зависящую от ДЛИНЫ записи исходных данных, и «число шагов» работы алгоритма.

Это различие следует главным образом из того, что понимается под словом «шаг».

Замечание.

В этих примерах использована (и ниже часто будет применяться) символика $O(f(n))$. Поэтому здесь приведём определение того, что понимается под записью $g(n) = O(f(n))$.

Для функций f и g натурального аргумента n записываем $g(n) = O(f(n))$ тогда и только тогда, когда существует положительная константа C такая, что для всех натуральных чисел n верно $g(n) \leqslant C \cdot f(n)$.

$$\exists C \forall n (g(n) \leqslant C \cdot f(n)).$$

Классическое определение вычислительной сложности алгоритма даётся в терминах машины Тьюринга. Ниже будет показано, что зачастую для определения вычислительной сложности алгоритма можно использовать, например, число операций, выполняемых программой для компьютера, написанной на языке Паскаль. Но при этом обязательно необходимо учитывать рост длины записи промежуточных результатов.

В 70-е гг. XX в. студентов учили, что для оценки числа шагов вычисления значения многочлена можно учитывать только количество умно-

жений, так как время выполнения на компьютере сложения существенно меньше времени выполнения умножения.

В современных компьютерах за счёт наличия в них сопроцессоров, выполняющих арифметические операции, можно считать, что арифметические операции над целыми числами, если результат такой операции помещается в одну ячейку, выполняются за примерно одно и то же время. Поэтому далее будем учитывать количество выполнения любой из арифметических операций.

1.2. МЕТОД ГАУССА ДЛЯ МАТРИЦ С ЦЕЛЫМИ КОЭФФИЦИЕНТАМИ. АНАЛИЗ РОСТА КОЭФФИЦИЕНТОВ

Хорошо известны следующие свойства метода Гаусса решения систем линейных уравнений (в том числе и нахождения обратной матрицы): метод является точным; метод неустойчив. При его применении к матрицам с элементами типа *real* разработаны различные методы «борьбы» с неустойчивостью метода, обусловленной тем, что операция деления в компьютере выполняется приближённо (более того, младшие разряды числа обрезаются, а не округляются).

На первом курсе всех студентов учат применению метода Гаусса к матрицам с целыми коэффициентами, в котором отсутствует деление, а следовательно, отсутствует округление промежуточных результатов. Казалось бы, все элементы каждой строки массива типа *real* можно умножить на общий знаменатель элементов строки и получить целочисленную матрицу коэффициентов равносильной системы, после этого производить вычисления без использования деления.

Почему так не поступают? Всё дело в большой скорости роста элементов матрицы в процессе применения метода Гаусса. Посмотрим, насколько же быстро они растут.

В литературе можно найти утверждения (см., например, [3]) о том, что метод Гаусса завершает работу не более, чем за полиномиальное (кубическое) от размерности матрицы число арифметических операций. Но сколько же реально операций выполняет компьютер?

Пусть задана целочисленная матрица размером $n \times m$, причём длина записи каждого из её элементов не превосходит M ($||a_{ij}|| \leqslant M$, $i = 1, \ldots, m$, $j = 1, \ldots, n$). После первой итерации имеем матрицу вида:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & & & \\ \vdots & & B^1 & \\ 0 & & & \end{pmatrix},$$

где элементы матрицы B^1 с индексами элементов $i = 2, \dots, m, j = 2, \dots, n$ вычислены по формуле:

$$b_{ij}^1 = a_{ij}a_{11} - a_{1j}a_{j1}.$$

Учитывая то, что при умножении целых чисел их длины складываются (может быть, минус единица), а при сложении (вычитании) длина записи результата не превосходит максимума их длин плюс 1, имеем, что

$$\|b_{ij}^1\| \leqslant 2M + 1,$$

$i = 2, \dots, m, j = 2, \dots, n$.

После k-й итерации имеем матрицу вида:

$$\begin{matrix} a_{11} & a_{12} & \dots & a_{1k} & \dots & a_{1n} \\ 0 & b_{22}^1 & \dots & b_{2k}^1 & \dots & b_{2n}^1 \\ 0 & 0 & \dots & b_{kk}^{k-1} & \dots & b_{kn}^{k-1}, \\ 0 & 0 & \dots & 0 & & \\ & & \vdots & & B^k & \\ 0 & 0 & \dots & 0 & & \end{matrix}$$

где элементы матрицы B^k с индексами элементов $i = k + 1, \dots, m, j = k + 1, \dots, n$ вычислены по формуле:

$$b_{ij}^k = b_{ij}^{k-1}b_{kk}^{k-1} - b_{kj}^{k-1}b_{ik}^{k-1}. \qquad (1)$$

Здесь, конечно, необходимо оговаривать случаи, когда $b_{kk}^{k-1} = 0$, $b_{ki}^{k-1} = 0$ при всех $i = k, \dots, m$, но в «худшем» (с точки зрения числа выполненных операций и скорости роста длин коэффициентов) случае имеем:

$$\|b_{ij}^2\| \leq 2(2M + 1) + 1 = 4(M + 1) - 1;$$

$$\|b_{ij}^3\| \leq 2(4M + 3) + 1 = 8(M + 1) - 1;$$

$$\|b_{ij}^4\| \leq 2(8M + 7) + 1 = 16(M + 1) - 1.$$

По индукции с учётом оговорённых случаев несложно доказать, что после прямого прохода метода Гаусса для матрицы ранга r:

$$\left\|b_{ij}^{r}\right\| \le 2^{r}(M+1)-1. \qquad (2)$$

Приведённые выше примеры показали, что никакой алгоритм, длина записи промежуточных данных которого экспоненциально зависит от длины записи исходных данных, не может завершать свою работу за полиномиальное от длины записи исходных данных число шагов. Получается, что метод Гаусса не полиномиален?

Можно возразить, что если при вычислении очередной строки при применении метода Гаусса **находить** наибольший общий делитель значений её элементов, то можно уменьшить рост длины их записей. Но это сильно увеличит время.

Но можно воспользоваться сформулированной в [3] теоремой Сильвестра, согласно которой для всех $k \geqslant 2$ каждый элемент матрицы B^k (т.е. b_{ij}^k при $i = k+1, \ldots, m$, $j = k+1, \ldots, n$) делится нацело на $b_{(k-1)(k-1)}^{k-2}$ (здесь при $k = 2$ элемент $b_{(k-1)(k-1)}^{k-2}$ следует воспринимать как a_{11}).

Таким образом, при вычислении элементов b_{ij}^k при $k \geqslant 2$ можно, оставаясь в рамках целых чисел, пользоваться формулой:

$$b_{ij}^{k} = \frac{b_{ij}^{k-1}b_{kk}^{k-1} - b_{kj}^{k-1}b_{jk}^{k-1}}{b_{(k-1)(k-1)}^{k-2}}. \qquad (3)$$

При этом длина записи b_{ij}^k уменьшится по крайней мере на длину записи $b_{(k-1)(k-1)}^{k-2}$ минус единица.

Посмотрим, как же изменится оценка (2) длины записи элементов после прямого прохода метода Гаусса для матрицы ранга r.

Для $k \geqslant 2$, $i = k+1, \ldots, m$, $j = k+1, \ldots, n$ имеем

$$\|b_{ij}^{k}\| \le 2\,\|b_{ij}^{k-1}\| + 1 - (\|b_{(k-1)(k-1)}^{k-2}\| - 1) =$$
$$= 2\,\|b_{ij}^{k-1}\| - \|b_{(k-1)(k-1)}^{k-2}\| + 2.$$

Учитывая то, что $\|a_{ij}\| \leqslant M$, получаем:

$$\|b_{ij}^{2}\| \le 2(2M+1) - M + 2 = 3M + 4;$$
$$\|b_{ij}^{3}\| \le 2(3M+4) - (2M+1) + 2 = 4M + 9;$$
$$\|b_{ij}^{4}\| \le 2(4M+9) - (3M+4) + 2 = 5M + 16;$$
$$\|b_{ij}^{5}\| \le 2(5M+16) - (4M+9) + 2 = 6M + 25.$$

По индукции с учётом оговорённых случаев несложно доказать, что после прямого прохода метода Гаусса для матрицы ранга r:

$$\left\|b_{ij}^{r}\right\| \le (r+1)M + r^2. \tag{4}$$

Согласитесь, что коэффициент $r + 1$ при M существенно меньше, чем 2^r. Так, например, если $M = 16$, а ранг матрицы $r = 10$, то в случае оценки (2) без использования теоремы Сильвестра гарантированные длины записи коэффициентов не превосходят $2^{10} \cdot 16 - 1 = 16\,383 \approx \approx 1\,024 \cdot 16$. С использованием этой теоремы согласно оценке (4) их гарантированные длины записи не превосходят $11 \cdot 16 + 10^2 = 276 = 17{,}25 \cdot 16$.

Однако это всё равно намного более длинное число, чем то, которое можно записать в компьютере с описанием типа *integer* (длина двоичной записи не превосходит 16) или *longinteger* (длина двоичной записи не превосходит 32).

Как же поступает большинство современных компьютеров, если результат арифметической операции с числами типа *integer* не помещается в ячейку? В действительности арифметические операции в компьютере выполняются по модулю 2^{16} и 2^{32}. При этом числа принадлежат отрезкам $[-2^{15}, 2^{15} - 1]$ и $[-2^{31}, 2^{31} - 1]$ для чисел типа *integer* или *longinteger* соответственно.

Так, например, пусть в языке С у нас есть 32-битная ячейка памяти. Если мы воспринимаем её содержимое как целое число со знаком, то это будет число из отрезка $[-2^{31}, 2^{31} - 1]$. Если процессор прибавит к $2^{31} - 1$ единицу (выполнит операцию *inc*), получим:

$$inc(2^{31} - 1) = -2^{31}.$$

Соответственно, вычитание единицы из -2^{31} (выполнение операции *dec*) даёт:

$$dec(-2^{31}) = 2^{31} - 1.$$

Рассмотрим очень простой и наглядный пример решения системы линейных уравнений с целыми коэффициентами на вычислительном устройстве, которое работает с целыми десятичными числами длины 2 (и числом −100), т.е. с целыми числами из отрезка [−100, 99] по модулю 200.

Требуется решить систему линейных уравнений с расширенной матрицей:

$$\left(\begin{array}{cc|c} 4 & -7 & 47 \\ 3 & -5 & 48 \end{array}\right).$$

Коэффициенты системы подобраны так, чтобы ее решением была пара чисел (101, 51).

После выполнения первого шага метода Гаусса получим расширенную матрицу:

$$\left(\begin{array}{cc|c} 4 & -7 & 47 \\ 12-12 & -20+21 & 48\cdot 4-47\cdot 3 \end{array}\right).$$

При вычислении «на бумажке» или на устройстве, позволяющем производить вычисления с числами бóльшей длины, получим:

$$\left(\begin{array}{cc|c} 4 & -7 & 47 \\ 0 & 1 & 51 \end{array}\right).$$

Из чего следует, что $y = 51$, $x = \dfrac{47+7\cdot 51}{4} = \dfrac{404}{4} = 101$, что и является решением системы.

На нашем гипотетическом вычислительном устройстве выражение $48 \cdot 4 - 47 \cdot 3$ будет вычисляться следующим образом:

$48 \cdot 4 = 192 \pmod{200} = -8$;

$47 \cdot 3 = 141 \pmod{200} = -59$;

$-8 - (-59) = 51 \pmod{200} = 51$.

В результате применения первой итерации метода Гаусса получаем расширенную матрицу:

$$\left(\begin{array}{cc|c} 4 & -7 & 47 \\ 0 & 1 & 51 \end{array}\right).$$

Очевидно, что число $y = 51$ является решением этой системы и значение для y получено верно.

Обратным ходом метода Гаусса имеем матрицу:

$$\left(\begin{array}{cc|c} 4 & 0 & 47-(-7)\cdot 51 \\ 0 & 1 & 51 \end{array}\right).$$

При этом: $47 - (-7) \cdot 51 = 47 + 357 = (\text{mod } 200)\ 47 + (-43) = 4$.

То есть первая строка имеет вид (4 0 | 4), и получаем $x = 1$, что не является решением системы.

1.3. АРИФМЕТИКА МНОГОРАЗРЯДНЫХ ЧИСЕЛ

Обсудим, как можно в компьютере производить вычисления с целыми числами произвольной разрядности. Алгоритмы этого параграфа взяты главным образом из [4] и приводятся для чисел, представленных массивами или списками. Коды процедур желающие могут посмотреть в [4] или написать самостоятельно в качестве упражнения.

1.3.1. Представление неотрицательного многоразрядного числа как числа в системе счисления по заданному модулю

Из курса алгебры известно, что всякое целое неотрицательное число x может быть представлено в m-ичной системе счисления (при $m \geqslant 2$) в виде $x = m^{k-1}x_0 + m^{k-2}x_1 + \ldots + mx_{k-2} + x_{k-1}$. При этом k — длина записи m-ичного представления числа x, $0 \leqslant x_i \leqslant m - 1$ при $i = 0, \ldots, k - 1$.

Ниже числа $x_0, \ldots, x_{k-1}$ будем называть **макроцифрами.**

Если m — некоторое целое число, которое может быть записано в одну ячейку, то представление числа в виде $(k, x_{k-1}, \ldots, x_0)$ удобно для хранения и осуществления операций с числами произвольной разрядности. То, что сначала расположены младшие разряды, а затем старшие, обусловлено тем, что арифметические операции, как правило, выполняются начиная с младших разрядов.

Другое соображение по поводу выбора значения числа m заключается в том, что, во-первых, на вход поступают десятичные числа, во-вторых, с этими числами придётся производить по крайней мере операцию сложения. Желательно, чтобы результат сложения двух макроцифр также помещался в одну ячейку. Поэтому разумно в качестве основания системы счисления при работе с длинными числами выбрать степень десятки.

Например, если вычисления производятся с числами типа *integer*, то $m = 10\,000 = 10^4$ — максимальная степень десятки, которую можно запи-

сать в одну ячейку ($10\,000 < 65\,536 = 2^{16}$). При вычислениях с числами типа *longinteger* $m = 1\,000\,000\,000 = 10^9$ — максимальная степень десятки, которую можно записать в одну ячейку ($1\,000\,000\,000 < 4\,294\,967\,296 = 2^{32}$).

1.3.2. Запись многоразрядного числа из файла. Оценка числа шагов

Пусть в файле записано десятичное число, заданное словом $a_1 \ldots a_n$ ($0 \leqslant a_i \leqslant 9$). Требуется представить его динамическим массивом (или списком), как это было определено в предыдущем разделе.

Рассмотрим пример такого представления (см. табл. 1.1) для числа 49 583 при $m = 100$ в массиве $A[0..j]$. Изначально $j = 0$. Число из файла считывается в переменную *ch*. В примечании описан этап работы алгоритма.

Таблица 1.1

Пример процесса записи числа из файла

A[0]	*A*[1]	*A*[2]	*A*[3]	*ch*	Примечание
0				4	Считывание одной цифры. 1 шаг
1	4			9	Запись цифры в массив и считывание цифры. 2 шага
1	49			5	Запись числа из двух цифр и считывание цифры. 2 шага
2	95	4		8	Перенос 1-й цифры в *A*[2], запись числа их двух цифр и считывание цифры. 3 шага
2	58	49		3	Перенос 1-й цифры в *A*[2], запись двух чисел из двух цифр и считывание цифры. 4 шага
3	83	95	4		Перенос 1-й цифры из $A[i]$ в $A[i + 1]$ ($i = 1, 2$), запись двух чисел из двух цифр и считывание цифры. 5 шагов

В этой таблице под «шагом» понимается одна из следующих операций: считывание цифры из файла, запись цифры в целочисленный массив, выделение первой цифры многозначного числа и её удаление из него, приписывание цифры в конец числа. Заметим, что эти «шаги» не равнозначны, так как последние два требуют нахождения остатка от деления на 10, а также умножения на 10 и сложения. Однако последние две операции не являются настоящими операциями деления или умножения, а выполняются с помощью сдвигов и конкатенации двух слов.

При считывании i-й цифры ($i = 1, \ldots, n$) количество «шагов» увеличивается по мере увеличения количества цифр, которые нужно перенести в следующие элементы массива, т.е. по мере увеличения содержимого $A[j]$. При этом параметры i и j связаны следующим образом:

$$j = A[0] = \left\lfloor \frac{i+1}{\|m-1\|} \right\rfloor.$$

При $j = 1, \ldots, \left\lfloor \frac{n+1}{\|m-1\|} \right\rfloor$ выполняются следующие операции:

```
while A[j] < m − 1 do
  { READ(ch);
  for l = 1..j do
      { if l = 1 then ch1 := ch else ch1 := ⌊A[l] : 10^(||m||−1)⌋;
      A[l] := A[l] · 10 + ch1 }
      }
  }
```

В теле второго цикла 4 операции. В условном операторе производится две операции: одно сравнение и одна операция либо присваивания, либо выделения последней цифры из числа. Второе присваивание требует ещё две операции: умножения на 10 (сдвиг) и сложения.

Второй цикл выполняется j раз, в его заголовке при каждом выполнении производится одна операция сравнения l с j и одна операция увеличения j на единицу.

Всего количество операций для каждого j составит $\|m - 1\| (1 + 4j)$.

Первый цикл выполняется $\|m - 1\|$ раз, в его заголовке при каждом выполнении производится одна операция сравнения.

Пусть $M = \left\lfloor \frac{n+1}{\|m-1\|} \right\rfloor$. Учитывая то, что $j = 1, \ldots, M$, получаем оценку числа «шагов» алгоритма:

$$\sum_{j=1}^{M} \|m-1\|(1+4j) = \|m-1\|(2M(M-1)+M) =$$

$$= \|m-1\|(2M^2 - M) \le (n+1)\left(2\frac{n+1}{\|m-1\|}\right) = O\left(\frac{n^2}{\|m-1\|}\right).$$

Таким образом, с учётом того, что число m является константой, алгоритм выполняется за квадратичное от длины записи исходного числа в файле количество «шагов».

Замечание. Здесь стоит отметить, что для того, чтобы записать целое неотрицательное десятичное число длины μ_1 как число типа *integer* или длины μ_2 как число типа *longinteger*, их длины должны удовлетворять неравенствам $m_1 = 10^{\mu_1} \leqslant 2^{16} = 65\,536$ и $m_2 = 10^{\mu_2} \leqslant 2^{32} = 4\,294\,967\,296$ соответственно. Максимальные значения для μ_1 и μ_2 будут соответственно $\mu_1 = 4$ и $\mu_2 = 9$. При этом значения для макроцифр будут из отрезков [0, 9 999] и [0, 999 999 999] соответственно, что значительно меньше числа, которое можно записать в одну ячейку. Это позволит в качестве промежуточных действий с макроцифрами выполнять их сложение.

1.3.3. Вывод многоразрядного числа в файл. Оценка числа шагов

При выводе числа необходимо помнить, что в каждом элементе массива, в котором хранится многоразрядное число, записана не последовательность цифр, а число, записанное этими цифрами. Поэтому число, десятичная запись которого меньше, чем длина записи выбранного нами основания m, требуется дополнить ведущими нулями.

Так, например, при выводе числа 1 000 034 506, представленного в массиве в виде

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
5	6	45	3	0	10

,

следует последовательно выводить 5 строк цифр 10 00 03 45 06. То есть при выводе каждой макроцифры требуется проверить длину её запи-

си и в случае необходимости дополнить её ведущими нулями (не более чем $||m - 1||$ штук).

В этом разделе под «шагом» будем понимать одну из следующих операций: запись «макроцифры» в символьную переменную, сравнение длины записи «макроцифры» с $||m - 1||$, дополнение строки ведущим нулём.

Таким образом, вывод каждой «макроцифры» потребует $O(||m - 1||)$ «шагов». Общее число шагов составит $O(||m - 1||k) = O(n)$.

1.3.4. Сложение двух неотрицательных многоразрядных чисел. Оценка числа шагов

Чтобы сложить два неотрицательных многоразрядных числа, записанных в массивы A и B, достаточно последовательно складывать по модулю m числа, записанные в $A[i]$, $B[i]$ и d для $i = 1, ..., \max\{A[0], B[0]\}$, где изначально $d = 0$, затем d — это 1 (если $A[i - 1] + B[i - 1] + d > m$[1]) или 0 в противном случае.

При подсчёте числа шагов в этом разделе под «шагом» понимается одна из следующих операций: вычисление $A[i - 1] + B[i - 1] + d \pmod m$, проверка условия $A[i - 1] + B[i - 1] + d > m$ и вычисление d.

Общее число «шагов» при сложении двух неотрицательных чисел не превосходит $3\max\{A[0], B[0]\} + 1$, т.е. составляет $O(\max\{A[0], B[0]\})$.

Замечание. Отметим, что так как для чисел типа *integer* $A[i - 1] < 10^5$, $B[i - 1] < 10^5$, $d < 2$, то $A[i - 1] + B[i - 1] + d < 2 \cdot 10^5 + 2 < 65\,536$, то результат вычисления может быть записан в одну ячейку. Аналогично для чисел типа *longinteger* $A[i - 1] + B[i - 1] + d < 2 \cdot 10^9 + 2 < 4\,294\,967\,296$.

1.3.5. Предикаты равенства и неравенств многоразрядных чисел. Оценка числа шагов

Прежде чем определить отрицательные многоразрядные числа и действия с ними (в частности, вычитание положительных многоразрядных чисел), рассмотрим алгоритмы вычисления значений предикатов $x = y$, $x \neq y$, $x < y$, $x \leqslant y$. В действительности, так как $x \neq y \Leftrightarrow \neg(x = y)$,

[1] Напомним, что сумма двух макроцифр помещается в одну ячейку и, следовательно, такую сумму можно иметь в качестве результата промежуточного действия.

а $x \leqslant y \Leftrightarrow \neg(y < x)$ или $x \leqslant y \Leftrightarrow (x < y) \vee (x = y)$, то достаточно уметь вычислять значения предикатов $x = y$ и $x < y$.

Очевидно, что если $A[0] \neq B[0]$, то $x = y \Leftrightarrow false$. Если дополнительно $A[0] < B[0]$, то $x < y \Leftrightarrow true$ (если $A\{0\} > B[0]$, то $x > y \Leftrightarrow true$).

Поэтому основные проблемы при вычислении упомянутых предикатов возникают, когда многоразрядные числа имеют одинаковую длину записи, т.е. $A[0] = B[0]$.

Оценим число шагов вычисления значений предикатов $x = y$ и $x < y$ для случая, когда $A[0] = B[0]$.

Начиная со старшего разряда (т.е. с $A[A[0]]$ и $B[B[0]]$) сравниваем значения чисел в $A[i]$ и $B[i]$, пока они совпадают. Если для некоторого i_0 $A[i_0] \neq B[i_0]$, то $x \neq y$. Если при этом $A[i_0] < B[i_0]$, то $x < y$, если $A[i_0] > B[i_0]$, то $x > y$.

Если под «шагом» понимать количество сравнений «макроцифр», то общее число «шагов» такой процедуры не превосходит $A[0]$.

В общем случае число «шагов» вычисления каждого из четырёх предикатов не превосходит $\min\{A[0], B[0]\}$, хотя стоит помнить, что эта оценка достигается только в случае, если $A[0] = B[0]$. В противном случае число «шагов» равно 1.

1.3.6. Вычитание двух положительных многоразрядных чисел. Оценка числа шагов

Рассмотрим сначала процедуру вычисления $x - y$ для случая, когда $x \geqslant y$. Процедура аналогична процедуре вычисления суммы, но если $A[i] < B[i]$, то следует «занять» единицу из $A[i + 1]$. Как и в случае сложения многоразрядных чисел, введём переменную d, значение которой изначально равно 0. Последовательно для $i = 1, \ldots, \max\{A[0], B[0]\}$ вычисляем $A[i] - d - B[i] \pmod m$, где d — это 1 (если $A[i - 1] - B[i - 1] - d < 0$) или 0 в противном случае.

Прежде чем заняться вычитанием чисел $x - y$ для случая, когда $x < y$, следует определить запись отрицательного числа. Это можно сделать, например, отведя дополнительный элемент массива $A[-1]$ или $A[A[0] + 1]$, в котором будет храниться 0, если число положительное, или 1, если число отрицательное (или какой-то другой признак положительности или отрицательности). Можно знак числа хранить в $A[0]$, но при этом во всех случаях, описанных выше, следует писать $|A[0]|$. В любом случае будем считать, что $A[0] > 0$ и имеется признак знака числа.

Если $x < y$, то можно, например, вычислять $y - x$, а затем результату присваивать признак отрицательного числа.

При подсчёте числа шагов в этом разделе под «шагом» понимается одна из следующих операций: вычисление $A[i-1] - d - B[i-1] \pmod m$, проверка условия $A[i-1] - d - B[i-1] > 0$ и вычисление d. Кроме того, предварительно проверяется условие $x \geqslant y$.

Общее число «шагов» при вычитании двух положительных чисел не превосходит $4\max\{A[0], B[0]\} + 1$, т.е. составляет $O(\max\{A[0], B[0]\})$.

1.3.7. Умножение многоразрядного числа на макроцифру. Оценка числа шагов

Обозначим макроцифру посредством C. Если $C = 0$, то результат равен нулю. В противном случае $d := 0$ и последовательно для $i = 1, \ldots, A[0]$ вычисляем $b := A[i]C + d, A[i] := b \pmod m$ и $d := [b/m]$. Если при $i = A[0]$ выполнено $d \neq 0$, то $A[0] := A[0] + 1, A[A[0]] := d$.

```
if C = 0 then return {0}
else { d := 0;
       for i := 1.. A[0] do
          { b := A[i] · C + d;
            A[i] := b (mod m);
            d := b/m;
          }
       };
if d ≠ 0 then
          { A[0] := A[0] + 1;
            A[A[0]] := d;
          }
```

Здесь под «шагом» будем понимать одну из следующих операций: умножение макроцифр, сложение макроцифр, вычисление неполного частного и остатка от деления результата предыдущих операций на m.

В условном операторе после *else* выполняется одно присваивание и оператор цикла, в котором (помимо двух операций, необходимых для организации цикла) производятся умножение, сложение остатка от деления на m, вычисление неполного частного. Всего в операторе цикла 6 «шагов».

Общее число операций не превосходит $\max\{1,2 + 6A[0] + \max\{1,3\}\} = 5 + 6A[0]$, т.е. составляет $O(A[0])$.

Замечание. Умножение двух макроцифр (для вычисления числа b) — это не одна операция, так как по правилам обычного вычисления умножения и сложения целых чисел в компьютере результаты промежуточных вычислений записываются по модулю m. В качестве упражнения читателю предлагается разработать алгоритм вычисления произведения двух макроцифр $b = A[i] \cdot C$ и $d := [b/m]$ и проверить, что число шагов этого алгоритма — константа. При этом оценка имеет вид $C'A[0] + 1$ и общая оценка $O(A[0])$ не изменится.

1.3.8. Умножение многоразрядных чисел. Оценка числа шагов

Пусть требуется перемножить два многоразрядных числа, записанных в массивах A и B. Результат будет записан в массив C, в котором изначально записано число 0.

Последовательно при $i = 1, \ldots, B[0]$ производим следующие действия:

1. Умножаем число, записанное в A, на $B[i]$ (результат в массиве $D1$) и записываем со сдвигом во вспомогательный массив D начиная с $D[i - 1]$ (в первые $i - 2$ элемента массива записываем 0).

В соответствии с п. 1.2.7 умножение потребует не более $C'A[0] + 1$ «шагов» при некоторой константе C', но, учитывая запись нулей, нужно не более $C'A[0] + i$ «шагов». Значение $D[0]$ не превосходит $A[0] + i$.

2. Складываем числа, записанные в массивах C и D ($C[0]$ и $D[0]$ не превосходят $A[0] + i$).

```
C[0] := 1; C[1] := 0;
for i = 1..B[0] do
   { D1 := A · B[i]; \* Здесь применяется процедура
                умножения многоразрядного числа на макроцифру.
for j = 0..i–2 do D[j] := 0;
for j = i–3..D1[0] + i–3 do D[j] := D1[j –(i + 2)];
D := C + D; \* Здесь применяется процедура
                сложения многоразрядных чисел.
}
```

В соответствии с п. 1.2.4 число «шагов» при сложении двух положительных чисел не превосходит $3\max\{C[0], D[0]\} + 1 \leqslant 3\,(A[0] + i) + 1 = 3A[0] + 3i + 1$.

Просуммировав оценки в п. 1 и 2, получаем:

$$\sum_{i=1}^{B[0]}((C'A[0]+i)+(3A[0]+3i+1))=\sum_{i=1}^{B[0]}((C'+3)A[0]+4i+1)=$$

$$=(C'+3)A[0]B[0]+2B[0](B[0]-1)+B[0]=O(A[0]B[0]+B[0]^2).$$

В предположении, что $B[0] \leqslant A[0]$ (это условие проверяется за один «шаг» и в противном случае можно умножать B на A), получаем оценку:

$$O(A[0]B[0]).$$

1.3.9. Деление многоразрядных чисел. Оценка числа шагов

Будем подбирать неполное частное от деления чисел x и y, записанных в массивах A и B, делением промежутка, в котором оно может находиться, пополам. Пусть L и U — нижняя и верхняя границы промежутка соответственно, M — целая часть середины промежутка, $z = y \cdot M$ — число, которое будем сравнивать с делимым.

При этом будем предполагать, что число, записанное в A, больше числа, записанного в B (в противном случае неполное частное равно 0, а остаток совпадает с делимым).

0. Выбор начального приближения L и U к отрезку, в котором находится неполное частное.

1. Вычисление середины M отрезка $[L, U]$. Длина этого числа не превосходит $A[0] - B[0]$.

2. Умножение значения середины отрезка на число y, $z = y \cdot M$.

3. Сравнение полученного числа с x $(z < x) \vee (z > x)$.

Для иллюстрации приведём пример деления чисел x = 45 973 на y = 261 при записи по основанию системы счисления 100 (см. табл. 1.2). Очевидно, что при делении трёхзначного числа на двузначное (имеются в виду макроцифры), будет однозначное или двузначное. Поэтому минимальным и максимальным числами интервала для значения можно выбрать соответственно 99 и 9 999.

В приведённом примере полужирным шрифтом выделено значение M, дважды появившееся в таблице и равное 176, которое и является неполным частным. Чтобы этого не происходило, достаточно вычислять разность $x - z$, и если она попала в промежуток $[0, y)$, то число M является целой частью от деления x на y. При этом значение $x - z$ равно остатку от деления x на y.

Таблица 1.2

Пример деления многоразрядных чисел с начальными приближениями $L_0 = 99$, $U_0 = 9\,999$

L	U	M	$z = yM$	$(z < x) \vee (z > x)$	Примечание
99	9 999	5 049	1 317 789	1 317 789 > 45 973	$[L, M]$
99	5 049	2 574	671 814	671 814 > 45 973	$[L, M]$
99	2 574	1 336	348 696	348 696 > 45 973	$[L, M]$
99	1 336	717	187 137	187 137 > 45 973	$[L, M]$
99	717	408	106 488	106 488 > 45 973	$[L, M]$
99	408	253	66 033	66 033 > 45 973	$[L, M]$
99	253	**176**	45 936	45 936 > 45 973	$[M, U]$
176	253	214	55 874	55 874 > 45 973	$[L, M]$
176	214	195	50 895	50 895 > 45 973	$[L, M]$
176	195	185	48 285	48 285 > 45 973	$[L, M]$
176	185	180	46 980	46 980 > 45 973	$[L, M]$
176	180	178	46 458	46 458 > 45 973	$[L, M]$
176	178	177	46 197	46 197 > 45 973	$[L, M]$
176	177	**176**	45 936	45 936 < 45 973	$[M, U]$

Кроме того, очевидно, что промежуток $[L, U]$ можно с самого начала сузить до $[99, 999]$.

Подсчитаем число «шагов» выполнения операций нахождения целой части от деления и остатка от деления двух многоразрядных чисел. В этом разделе под «шагом» понимается любая из операций, выполнение которой считалось «шагом» в предыдущих разделах.

Прежде всего заметим, что если C — целая часть от деления x на y, то $x = Cy + r$, $||x|| = ||C|| + ||y|| + d$ и $||C|| = ||x|| - ||y|| - d$, где $d \in \{0, 1\}$. Следовательно, значение C заведомо принадлежит отрезку $[10^{||x|| - ||y|| - 1}, 10^{||x|| - ||y||}]$ (если рассматривается десятичная система счисления и $||a||$ — длина десятичной записи числа a) или $[m^{||x|| - ||y|| - 1}, m^{||x|| - ||y||}]$ (если рассматривается m-ичная система счисления и $||a||$ — длина m-ичной записи числа a). Длина такого отрезка не превосходит $m^{||x|| - ||y||} - m^{||x|| - ||y|| - 1} = m^{||x|| - ||y|| - 1}(m - 1)$.

Количество делений отрезка пополам не превосходит $\log_2(U_0 - L_0)$, где U_0 и L_0 — первоначальные значения границ отрезка, причём $U_0 - L_0 \leqslant m^{A[0]-B[0]-1}(m-1)$. Всего количество делений отрезка пополам $\log_2(U_0 - L_0) \leqslant \log_2(m^{A[0]-B[0]-1}(m-1)) = (A[0] - B[0] - 1)\log_2 m + \log_2(m-1) \leqslant (A[0] - B[0])\log_2 m$.

Для каждого отрезка производятся операции, описанные ниже:

1. Вычисление середины отрезка.

Длина этого числа не превосходит $A[0] - B[0]$. Сложение двух чисел длины не более, чем $A[0] - B[0]$, в соответствии с п. 1.2.4 совершается не более чем за $3(A[0] - B[0]) + 1$ «шаг». На вычисление его половины потребуется ещё $(A[0] - B[0]) + 1$ «шаг». Всего не более $4(A[0] - B[0]) + 2$ «шага».

2. Умножение значения середины отрезка на число *y*.

В соответствии с п. 1.2.8 умножение чисел длины $A[0] - B[0]$ и $B[0]$ потребует $7(A[0] - B[0])\,B[0] + 2B[0]\,(B[0] - 1) + B[0] = 7A[0]\,B[0] + 9B[0]^2 + B[0]$.

3. Сравнение полученного числа с *x*.

В соответствии с п. 1.2.5 сравнение таких чисел производится не более чем за $A[0]$ «шагов».

Сложив полученные оценки числа «шагов» и умножив результат на максимальное количество повторов, получим:

$$((4(A[0] - B[0] + 2) + (7A[0]\,B[0] + 9B[0]^2 + B[0]) + A[0])) \cdot \log_2(U_0 - L_0) =$$

$$= O(A[0]\,B[0]\log_2(U_0 - L_0)) = O(A[0]\,B[0] \cdot (A[0] - B[0])).$$

В окончательной оценке отсутствуют параметры L_0 и U_0, но очевидно, что в зависимости от начальных приближений будет меняться и число шагов работы алгоритма. Это отражено в предпоследнем выражении полученной оценки.

Проделаем рассмотренный пример с другими, более аккуратно вычисленными приближениями.

Пусть требуется вычислить неполное частное от деления числа $x = 45\,973$ на $y = 261$ при записи по основанию системы счисления 100. Эти числа лежат в промежутках $[4 \cdot 10^4, 5 \cdot 10^4]$ и $[2 \cdot 10^2, 3 \cdot 10^2]$ соответственно. Следовательно, $\frac{4 \cdot 10^4}{3 \cdot 10^2} \leq \left\lfloor \frac{x}{y} \right\rfloor \leq \frac{5 \cdot 10^4}{2 \cdot 10^2}$ и в качестве приближений (см. табл. 1.3) можно взять числа $L_0 = 133$ и $U_0 = 250$.

Таблица 1.3

Пример деления многоразрядных чисел с начальными приближениями $L_0 = 135$, $U_0 = 250$

L	U	M	$z = y \cdot M$	$(z < x) \vee (z > x)$	Примечание
135	250	192	50 112	50 112 > 45 973	$[L, M]$
135	192	163	42 543	42 543 < 45 973	$[M, U]$
163	192	177	46 197	46 197 > 45 973	$[L, M]$
163	177	170	44 370	44 370 < 45 973	$[M, U]$
170	177	173	45 153	45 153 < 45 973	$[M, U]$
173	177	175	45 675	45 675 < 45 973	$[M, U]$
175	177	**176**	45 936	45 936 < 45 973	$[M, U]$
176	177	**176**	45 936	45 936 < 45 973	$[M, U]$

Упражнения.

1. Разработайте алгоритм вычисления произведения двух макроцифр, результатом работы которого будут две макроцифры: значение этого произведения по модулю m и целая часть от деления произведения на m. Оцените число его шагов.

2. Выразите границы начального приближения частного от деления чисел x и y, записанных в массивах A и B через числа $A[0]$, $B[0]$, $||m - 1||$ (где m — основание выбранной системы счисления для представления многоразрядных чисел), $||A[A[0]]||$ и $||B[B[0]]||$.

3. Подсчитайте число «шагов» вычисления значения полинома по схеме Горнера, если все коэффициенты — это макроцифры, а значение переменной представлено как многоразрядное число.

4. Оцените число «шагов» вычисления наименьшего общего кратного двух многоразрядных чисел.

5. Оцените число «шагов» вычисления наибольшего общего делителя двух многоразрядных чисел.

6. Разработайте алгоритм разложения многоразрядного числа на простые сомножители и оцените число его «шагов».

1.4. СОРТИРОВКИ И ОЦЕНКИ ЧИСЛА ШАГОВ

В этом параграфе не будут подробно описываться алгоритмы сортировки, которые хорошо известны даже начинающим студентам-про-

граммистам. Упор будет сделан на подсчёт оценок числа шагов некоторых алгоритмов, на то, что в этих оценках имеется в виду под словом «шаг» и каковы параметры этих оценок. Подробное изложение различных видов сортировок имеется, например, в [5].

В этом параграфе будем считать, что n чисел размещены в массиве $a[1..n]$ и их следует расположить в порядке возрастания.

1.4.1. «Пузырёк»

Самый простой и легко программируемый алгоритм сортировки обычно известен под названием «пузырёк». В этом алгоритме имеются два вложенных цикла по $i = 2, ..., n$ и $j = 1, ..., i$. Тем самым количество выполнений тела циклов равно:

$$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}.$$

В теле циклов сравниваются значения $a[i]$ и $a[j]$. В случае необходимости содержания элементов массива меняются местами. Обмен значениями переменных x и y можно осуществить с помощью трёх операторов присваивания с использованием вспомогательной переменной z: $z := x$; $x := y$; $y := z$[2]. Таким образом, в теле цикла каждый раз выполняется не более четырёх операций.

Если под «шагом» понимать выполнение операции сравнения двух чисел или операции присваивания, то всего «пузырёк» завершает работу за число «шагов», не превосходящее $2(n + 2)(n - 1)$, что составляет $O(n^2)$.

Именно такая оценка числа «шагов» этого алгоритма имеется практически в любой литературе, где описываются алгоритмы сортировки.

Следует отметить, что мы не учли количество операторов увеличения переменной цикла на 1 и оператора сравнения значения переменной цикла с границей цикла. Это повлияет на мультипликативный сомножитель, но оценка всё равно останется $O(n^2)$.

Посмотрим, как изменится полученная оценка, если в массиве хранятся многоразрядные числа. Пусть k — максимальное количество макроцифр в сортируемых многоразрядных числах. Тогда под «шагом» следует понимать соответствующие операции с макроцифрами, и как операция сравнения, так и операция присваивания многоразрядных

[2] Можно не использовать вспомогательную переменную, но при этом требуется выполнить сложение двух чисел и дважды вычитание двух чисел: $x := x + y$; $y := x - y$; $x := x - y$.

чисел осуществляется не более чем за k «шагов», суммарное число «шагов» не превосходит $2k(n+2)(n-1)$, что составляет $O(n^2k)$.

Отметим, что аналогичная оценка имеет место и в случае, если следует упорядочить по алфавиту массив из слов произвольной длины.

1.4.2. Сортировка слияниями фон Неймана

Одним из самых эффективных по времени алгоритмов сортировки произвольных массивов является ***алгоритм фон Неймана (сортировка слияниями).*** Будем предполагать, что длина массива $n = 2^t$. Если это не так, то дополним его очень большими числами. Вот его описание.

1. Разобьём массив на пары $(a[2i-1], a[2i])$ $(i = 1, \ldots, 2^{t-1})$ и упорядочим элементы в каждой паре.

Это потребует $\frac{n}{2} = 2^{t-1}$ сравнений чисел и не более чем такое же количество обменов значений элементов массива. Всего не более чем $4 \cdot 2^{t-1}$ «шагов».

2. В цикле по $j = 1, \ldots, t-1$ «сливаем» уже упорядоченные наборы из 2^j элементов (первый со вторым, третий с четвёртым, $(2^{t-j}-1)$-й с 2^{t-j}-м). В результате получим 2^{t-j-1} упорядоченных набора из 2^{j+1} элементов каждый.

«Слияние» двух упорядоченных наборов длины 2^j в один упорядоченный набор длины 2^{j+1} может быть произведено следующим образом:

```
m := 1; k := 1; l := 1; n := 2^(j+1)                    4 «шага»
      следующие операции выполняются 2^(j+1) раз
while m < n do                                          1 «шаг»
  { if A[k] < B[l] then                                 1 «шаг»
      { C[m] := A[k]; k := k+1; }                       2 «шага»
    else                                              или 2 «шага»
  { C[m] := B[l]; l := l+1; }
    m := m+1;                                           1 «шаг»
  }
```

Всего $4 + 5 \cdot 2^{j+1}$ «шагов». Таких слияний при j-й итерации производится $\frac{n}{2 \cdot 2^j} = \frac{2^t}{2^{j+1}} = 2^{t-j-1}$. То есть при j-й итерации выполняется $2^{t-j-1}(4 + 2^{j+1} \cdot 5) = 4 \cdot 2^{t-j-1} + 5 \cdot 2^t = O(2^t) = O(n)$ «шагов».

Количество итераций равно t. Учитывая, что $t = \log_2 n$, получаем $O(n \log_2 n)$ «шагов».

Учитывая замечание к алгоритму сортировки «пузырёк» при сортировке многоразрядных чисел или слов в заданном алфавите в этой оценке требуется дописать мультипликативный сомножитель k, где k — максимальное количество макроцифр в многоразрядных числах или максимальная длина сортируемых чисел, т.е. оценка имеет вид $O(kn \log_2 n)$.

1.4.3. Сортировка подсчетом

Во многих учебниках и монографиях (см., например, [5]) утверждается, что никакой алгоритм сортировки не может иметь оценку числа шагов ме́ньшую, чем $O(n \log_2 n)$. Однако эта сортировка имеет оценку $O(N)$. В чём дело и какая разница между параметрами n и N? Разберёмся в этом алгоритме.

Пусть требуется отсортировать массив целых положительных чисел, наибольшее из которых равно N. Заведём вспомогательный массив $c[1..N]$, значения элементов которого первоначально равны 0.

В цикле по $i = 1, \ldots, n$ значение $c[a[i]]$ увеличиваем на единицу (n «шагов»).

В цикле по $j = 1, \ldots, N$, если $c[j] \neq 0$, то очередным $c[j]$ элементам массива a присваиваем j.

```
for (i = 0; i < n; i++)
    c[a[i]]++;
k = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < c[i]; j++)
    a[k++] = i;
```

Очевидно, что следует просмотреть два массива длиной n и N и либо увеличить значение элемента на 1, либо произвести присваивание. Всего $O(n + N)$ «шагов».

Естественно, что при сортировке многоразрядных чисел эту оценку следует умножить на k: $O(k(n + N))$.

Следует заметить, что число n — это количество элементов в массиве (длина массива), а N — это значение самого элемента, причём максимального. В общем случае N может экспоненциально зависеть от n. Но если N намного меньше n или разность $\Delta = \max_i a[i] - \min_i a[i]$ имеет тот же порядок, что и n, а границами массива c являются числа $\max_i a[i]$ и $\min_i a[i]$, то количество «шагов» работы алгоритма действительно линейно относительно n.

ГЛАВА 2. АЛГОРИТМЫ НА ГРАФАХ

2.1. РАЗЛИЧНЫЕ СПОСОБЫ ПРЕДСТАВЛЕНИЯ ГРАФА В КОМПЬЮТЕРЕ И ОЦЕНКИ ЧИСЛА ШАГОВ РЕШЕНИЯ ПРОСТЕЙШИХ ЗАДАЧ

В этом учебном пособии не будут излагаться ни теория графов, ни алгоритмы на графах, так как этот материал изучается на младших курсах специальностей, связанных с программированием или информатикой. Желающие могут освежить этот материал, например, в [6, 7, 8].

Ниже будут использованы следующие обозначения:

V — произвольное конечное множество;

E — подмножество множества двухэлементных подмножеств множества V;

$G = (V, E)$ — граф с множеством вершин V и множеством рёбер E;

A — подмножество множества упорядоченных пар множества V;

$G = (V, A)$ — орграф с множеством вершин V и множеством дуг A;

n — количество вершин в графе;

m — количество рёбер в графе;

$N(v)$ — окружение вершины v, т.е. множество вершин, смежных с v;

$OUT(v)$ — множество вершин орграфа, непосредственно достижимых из v;

$IN(v)$ — множество вершин орграфа, из которых v непосредственно достижима;

$deg(v)$ — степень вершины v, т.е. количество вершин в окружении;

w_{ij} — вес ребра $\{v_i, v_j\}$ или ребра (v_i, v_j) во взвешенном графе.

Будут получены оценки числа шагов алгоритмов, основанных на обходе графа, алгоритма Дейкстры нахождения кратчайшего пути, ал-

горитма Прима нахождения остова минимального веса, алгоритма нахождения Гамильтонова цикла и алгоритма генерации всех независимых множеств. К числу алгоритмов, основанных на обходе графа, относятся, например, выделение компонент связности, проверка графа на двудольность и выделение долей, нахождение остова графа, нахождение цикла в графе и многие другие.

При оценке числа шагов решения задач, связанных с обходом графа, большое значение имеет способ представления графа. Рассмотрим некоторые из широко распространённых способов представления графа и оценим число шагов нахождения всех вершин, смежных с заданной.

2.1.1. Матрица смежности

Матрица смежности графа — это квадратная матрица $A_{n \times n}$, элементы которой определены так:

$$a_{ij} = \begin{cases} 1, & \text{если } \{v_i, v_j\} \in E, \\ 0 & \text{иначе.} \end{cases}$$

При обходе графа в глубину или в ширину для каждой вершины необходимо проверить все (при обходе в глубину — постепенно, а при обходе в ширину — сразу) вершины, смежные с данной. При использовании матрицы смежности для одной вершины это можно сделать за n проверок того, следует ли помещать вершины в стек или в очередь. Эта процедура выполняется для каждой из n вершин.

Этим объясняется то, что алгоритмы, основанные на обходе графа в глубину или в ширину при представлении графа матрицей смежности, имеют оценку числа шагов вида $O(n^2)$.

Для орграфа элементы матрицы смежности определяются так:

$$a_{ij} = \begin{cases} 1, & \text{если } \{v_i, v_j\} \in A, \\ 0 & \text{иначе.} \end{cases}$$

Рассуждениями, аналогичными таковым для неориентированного графа, получаем оценку числа шагов вида $O(n^2)$.

2.1.2. Списки смежности

Списки смежности — это одномерный массив, i-м элементом которого является список вершин, смежных с v_i, т.е. окружение вершины v_i.

Очевидно, что для нахождения всех вершин, смежных с v, требуется $deg(v)$ проверок. Просуммировав эту величину по всем v, получаем $\Sigma_{v \in V} deg(v) = 2m$[3].

Этим объясняется то, что алгоритмы, основанные на обходе графа в глубину или в ширину при представлении графа списками смежности, имеют оценку числа шагов вида $O(n + m)$.

Для графов с разными свойствами эта оценка может быть видоизменена.

Если граф является деревом или лесом, то $m < n$ и оценка принимает вид $O(n)$.

Если граф полный, то $m = \frac{n(n-1)}{2}$ и оценка принимает вид $O(n^2)$.

Если степени всех вершин графа не превосходят некоторой константы C (существенно меньшей, чем n), то $m \leqslant Cn$ и оценка принимает вид $O(n)$.

Если граф связен и степени вершин произвольны, то $m \geqslant n - 1$ и оценка принимает вид $O(m)$.

Для орграфа список смежности для вершины v состоит из вершин, входящих в $OUT(v)$ (или в $IN(v)$). Поскольку $\Sigma_{v \in V} ||OUT(v)|| = \Sigma_{v \in V} ||IN(v)|| = m$, то рассуждениями, аналогичными таковым для неориентированного графа, получаем оценку числа шагов вида $O(n + m)$.

2.1.3. Матрица инцидентности

Матрица инцидентности графа — это матрица $B_{n \times m}$, элементы которой определены так:

$$b_{ij} = \begin{cases} 1, & \text{если } v_i \in e_j, \\ 0 & \text{иначе.} \end{cases}$$

Основными свойствами такой матрицы являются следующие два: $\sum_{i=1}^{n} b_{ij} = 2$, $\sum_{j=1}^{m} b_{ij} = deg(v_i)$.

[3] Здесь использована лемма о рукопожатиях: «Сумма степеней вершин графа равна удвоенному количеству рёбер».

Если для вершины необходимо проверить все вершины, смежные с данной, то придётся в строке, соответствующей этой вершине, найти все столбцы, на пересечении с которыми стоит 1 (m проверок), и в каждом из этих столбцов найти строку, на пересечении с которой стоит 1 (n проверок). Всего не более чем $deg(v_i)n + (m - deg(v_i))$.

Поскольку эти операции следует проделать для каждой вершины, то всего имеем:

$$\sum_{i=1}^{n}(deg(v_i)n + (m - deg(v_i)) = n\sum_{i=1}^{n}(deg(v_i) + nm +$$

$$+ \sum_{i=1}^{n} deg(v_i) = 2nm + nm - 2m = 3nm - 2m.$$

Этим объясняется то, что алгоритмы, основанные на обходе графа в глубину или в ширину при представлении графа матрицей инцидентности, имеют оценку числа шагов вида $O(nm)$.

2.1.4. Массивы V и E

Конечно, вряд ли найдётся программист, который выбирает способ задания графа на основании его определения, т.е. с помощью одномерного массива V, в котором хранятся имена (или номера) вершин, и двумерного массива E, в котором хранятся пары номеров смежных вершин. Однако в гл. 4 нам придётся иметь дело именно с таким способом задания графа, поэтому следует показать, что он достаточно эффективен.

Прежде всего по массиву E построим массив E', в котором вместе с каждой парой $\{v_i, v_j\}$ присутствует пара $\{v_j, v_i\}$ (новый массив вдвое длиннее исходного), и отсортируем его по номеру первого элемента пары. Вместе с элементом v_i массива V будем хранить номер строки массива E', в которой впервые встречается пара вида $\{v_i, u\}$ для некоторой вершины u.

Фактически получено представление графа списками смежности, которые записаны в массив E' длины $2m$. Для его записи потребуется $2m$ присваиваний и $O(m \log m)$ сравнений и присваиваний при сортировке.

Сложив оценку числа предварительных шагов и оценку числа шагов алгоритмов, основанных на обходе графа в глубину или в ширину, получим $O(m \log m) + O(n + m) = O(n + m \log m)$. Все комментарии к этой оценке, написанные в разд. 2.1.2, остаются справедливыми.

2.2. ОЦЕНКИ ЧИСЛА ШАГОВ НЕКОТОРЫХ СТАНДАРТНЫХ АЛГОРИТМОВ

2.2.1. Алгоритм Дейкстры поиска кратчайшего пути во взвешенном ориентированном графе. Оценки числа шагов

Задан взвешенный орграф $G = (V, A)$ (все веса w_{ij} положительны) и две выделенные вершины s — старт и f — финиш. Требуется найти кратчайший путь из s в f.

Вводятся два вспомогательных массива *pre* и *d*, в которых хранятся предыдущая вершина в кратчайшем (на данный момент) пути из s в v_i и (известная на данный момент) величина этого кратчайшего расстояния.

1. $pre := \infty$, вершину s помечаем.

2. Для всех вершин $u \in OUT(s)$ делаем присваивания $pre(u) := s$; $d(u) := w_{su}$.

3. Среди непомеченных вершин u, у которых $pre(u) \neq \infty$ находим вершину $v = \arg(\min_u d(u))$. Помечаем вершину v.

4. Пересчитываем значения массивов *pre* и *d* для непомеченных вершин u из $OUT(v)$.

Если $d(u) > d(v) + w_{vu}$, то $\{pre(u) := v; d(u) := d(v) + w_{vu}\}$.

5. Если $v \neq f$, то возвращаемся к выполнению п. 3. Иначе алгоритм заканчивает работу и кратчайшим путём из s в f является $s, \ldots, pre(pre(f)), pre(f), f$.

В п. 1 совершается n присваиваний.

В п. 2 совершается $2\,\|OUT(s)\|$ присваиваний.

Выполнение п. 3, 4 и 5 производится не более чем $n - 1$ раз.

При i-м выполнении п. 3 количество непомеченных вершин не превосходит $n - i$, и, следовательно, нахождение вершины, доставляющей $\max_u d(u)$, требует не более $n - i - 1$ сравнений.

В п. 4 выполняется не более $\|OUT(v)\|$ сложений, $\|OUT(v)\|$ сравнений и $2\,\|OUT(v)\|$ присваиваний.

В п. 5 выполняется одно сравнение.

Просуммировав полученные оценки, имеем:

$$n + 2\,\|OUT(s)\| + \sum_{i=1}^{n-1} (n - i - 1 + 4\,\|OUT(v_i)\| + 1) =$$

$$= n + 2\,\|OUT(s)\| + n(n-1) - \frac{(n-1)(n-2)}{2} + 4m \le$$

$$\le \frac{n(n-1)}{2} + 4m - 1 = O(n^2 + m) = O(n^2).$$

2.2.2. Алгоритм Р. Прима нахождения остова минимального веса. Оценки числа шагов

Задан взвешенный граф $G = (V, E)$, все веса w_{ij} положительны. Требуется найти остов минимального веса.

Не умаляя общности, будем считать, что граф связен и ищется остовное дерево минимального веса.

Вводятся два вспомогательных массива m и d, в которых $m[i]$ и $d[i]$ — это номер вершины, смежной с v_i в остове, и вес ребра $\{v_i, m[i]\}$ соответственно.

1. $d := \infty$. Выбираем произвольно вершину v_0, помечаем её как вошедшую в остов.

2. Для всех вершин $u \in N(v_0)$ делаем присваивания $m[u] := v_0$; $d[u] := w_{v_0u}$.

3. Среди непомеченных вершин u, у которых $d[u] \ne \infty$, находим вершину $v = \arg(\min_u d[u])$. Помечаем вершину v как вошедшую в остов.

4. Пересчитываем значения массивов m и d для непомеченных вершин u из $N(v)$.

Если $d[u] > w_{vu}$, то $\{m[u] := v; d[u] := w_{vu}\}$.

5. Если имеются непомеченные вершины, то возвращаемся к выполнению п. 3. Иначе алгоритм заканчивает работу и в остов входят все рёбра вида $\{u, m[u]\}$ $(u \ne v_0)$, причём вес остова равен $\Sigma_{u \ne v_0} d[u]$.

Оценки числа шагов работы этого алгоритма аналогичны оценкам числа шагов алгоритма Дейкстры за исключением того, что в п. 4 не выполняется операция сложения. В результате получаем:

$$n + 2deg(v_0) + \sum_{i=1}^{n-1} (n - i - 1 + 3deg(v_i) + 1) =$$

$$= n + 2deg(v_0) + n(n-1) - \frac{(n-1)(n-2)}{2} + 3m \le$$

$$\le \frac{n(n-1)}{2} + 3m - 1 = O(n^2 + m) = O(n^2).$$

2.2.3. Нахождение Гамильтонова цикла. Оценки числа шагов

Алгоритм нахождения Гамильтонова цикла на первый взгляд очень напоминает обход графа в глубину. Однако при удалении вершины из стека алгоритм «забывает», что эту вершину уже посещали. Алгоритм заканчивает работу, если в стеке находятся все n вершин и крайние вершины в стеке смежны.

В результате если выписать в виде дерева все возможные содержания стека, то каждая вершина v исходного графа может оказаться в этом дереве не более чем $deg(v)$ раз. И каждое появление вершины v в дереве имеет такую же степень вершины, что и v. Кроме того, высота такого дерева не превосходит n.

Суммарное количество посещений вершин не превосходит $d^n - 1$, где $d = \max_i(deg(v_i) - 1)$, и не меньше n, если Гамильтонов цикл был найден при первом обходе графа.

Таким образом, предложенный здесь алгоритм нахождения Гамильтонова цикла в графе имеет экспоненциальную относительно количества вершин в графе оценку числа шагов.

Можно возразить, что это не единственный (и не самый эффективный) алгоритм нахождения Гамильтонова цикла. Ниже, в гл. 4 будет отмечено, что в настоящее время не существует алгоритма, который бы по заданному графу за полиномиальное от длины его записи число шагов отвечал на вопрос, имеет ли граф Гамильтонов цикл. И тем более в настоящее время не существует алгоритма, который бы по заданному графу за полиномиальное от длины его записи число шагов строил такой цикл.

2.2.4. Связь между понятиями «независимое множество», «вершинное покрытие» и «КЛИКА»

Определение. Множество V' называется **вершинным покрытием графа** $G = (V, E)$, если всякое ребро графа инцидентно некоторой вершине из V'.

$$\forall uv(\{u, v\} \in E \rightarrow (u \in V' \vee v \in V')).$$

Определение. Множество V' называется **независимым множеством графа** $G = (V, E)$, если никакие две вершины из V' не смежны.

$$\forall uv((u \in V' \,\&\, v \in V') \to \{u, v\} \notin E).$$

Определение. Множество V' называется **кликой в графе** $G = (V, E)$, если любые две вершины из V' смежны.

$$\forall uv((u \in V' \,\&\, v \in V') \to \{u, v\} \in E).$$

Теорема 2.2.1. Следующие утверждения равносильны:

1. V' является вершинным покрытием в $G = (V, E)$.
2. $V \setminus V'$ является независимым множеством в $G = (V, E)$.
3. $V \setminus V'$ является кликой в $\overline{G} = (V \setminus V', \overline{E})$.

Для доказательства запишем формулы, определяющие соответствующие понятия, для каждого из утверждений теоремы:

1. $\forall uv(\{u, v\} \in E \to (u \in V' \vee v \in V'))$.
2. $\forall uv((u \in V \setminus V' \,\&\, v \in V \setminus V') \to \{u, v\} \notin E) \Leftrightarrow$
   $\forall uv(\{u, v\} \in E \to \neg(u \in V \setminus V' \,\&\, v \in V \setminus V')) \Leftrightarrow$
   $\forall uv(\{u, v\} \in E \to (u \notin V \setminus V' \vee v \notin V \setminus V')) \Leftrightarrow$
   $\forall uv(\{u, v\} \in E \to (u \in V' \vee v \in V'))$.
3. $\forall uv((u \in V \setminus V' \,\&\, v \in V \setminus V') \to \{u, v\} \in \overline{E}) \Leftrightarrow$
   $\forall uv(\{u, v\} \notin \overline{E} \to \neg(u \in V \setminus V' \,\&\, v \in V \setminus V')) \Leftrightarrow$
   $\forall uv(\{u, v\} \in E \to (u \notin V \setminus V' \vee v \notin V \setminus V')) \Leftrightarrow$
   $\forall uv(\{u, v\} \in E \to (u \in V' \vee v \in V'))$.

Таким образом, наличие алгоритма нахождения одного из этих множеств вершин в графе влечёт наличие алгоритма с такой же вычислительной сложностью нахождения оставшихся двух множеств.

2.2.5. Алгоритм построения максимальной клики

Определение. Клика в графе $G = (V, E)$ называется **максимальной**, если любая вершина графа, не входящая в неё, не смежна хотя бы с одной вершиной этой клики.

Самый интуитивно простой алгоритм построения максимальной клики заключается в полном переборе всех возможных подграфов размера k с проверкой того, является ли хотя бы один из них полным. Этот алгоритм неэффективен, поскольку число подграфов с k вершинами в графе с n вершинами равно значению биномиального коэффициента:

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Всего требуется проверить на полноту $\sum_1^n C_n^k = 2^n - 1$ графов.

Другой алгоритм работает так: две клики размера k_1 и k_2 «склеиваются» в бóльшую клику размера $k_1 + k_2$, причём кликой размера 1 полагается отдельная вершина графа. Алгоритм завершается, как только ни одного слияния больше произвести нельзя. Время работы данного алгоритма линейно, однако он является эвристическим, поскольку не всегда приводит к нахождению клики максимального размера. В качестве примера неудачного завершения можно привести случай, когда вершины, принадлежащие максимальной клике, оказываются разделены и находятся в кликах меньшего размера, причём последние уже не могут быть «склеены» между собой.

Одним из самых эффективных алгоритмов поиска клик является ***алгоритм Брона — Кербоша***: метод ветвей и границ для поиска всех клик.

Алгоритм оперирует тремя множествами вершин графа и является рекурсивной процедурой *extend (candidates, not)*.

Множество *compsub* — множество, содержащее на каждом шаге рекурсии полный подграф для данного шага. Строится рекурсивно.

Множество *candidates* — множество вершин, которые могут увеличить *compsub*.

Множество *not* — множество вершин, которые уже использовались для расширения *compsub* на предыдущих шагах алгоритма.

ПОКА *candidates* НЕ пусто И *not* НЕ содержит вершины, СМЕЖНОЙ СО ВСЕМИ вершинами из *candidates*,

{ Выбираем вершину *v* из *candidates* и добавляем её в *compsub*;

Формируем *new-candidates* и *new-not*, удаляя из *candidates* и *not* вершины, НЕ СМЕЖНЫЕ с *v*;

ЕСЛИ *new-candidates* и *new-not* пусты;

ТО *ompsub* — клика;

ИНАЧЕ *extend(new-candidates, new-not)*;

Удаляем *v* из *compsub* и *candidates* и помещаем в *not*

}

Вычислительная сложность алгоритма линейна относительно количества клик в графе. В худшем случае алгоритм работает за $O(3^{n/3})$ шагов [9].

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 своим происхождением великому средневековому учёному Мухаммаду ибн Муссе аль Хорезми (т.е. 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 отношении 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 в. были сформулированы задачи нахождения единого процесса решения ряда родственных задач с параметрами. Если с задачей нахождения такого процесса было всё более или менее ясно (доста-

точно 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 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.

Относительно пункта *«определённость данных»* можно заметить, что описанные в предыдущем разделе алгоритмы на графах не «привязаны» к способу задания графа.

Самые большие сложности возникают в пункте *«элементарность шага»*. Являются ли все шаги следующего вычисления элементарными:

$$x := 0{,}25;$$
$$y := x^3;$$
$$z := \sin(y^x)?$$

Вычисление x^3 в большинстве компьютеров происходит как $x \cdot x \cdot x$. А если вычислить x^{15}? Придётся находить $e^{15\ln x}$, причём $u = \ln x$ и e^{15u} будут определены с помощью разложения соответствующей функции в ряд. При вычислении значения $\sin(y^x)$ без разложений в ряд вообще не обойтись.

Сложности есть и с пунктом *«массовость»*. Так, например, первое доказательство великой теоремы Ферма: «Не существует натуральных чисел x, y, z и $n > 2$, таких что $x^n + y^n = z^n$», было произведено на компью-

тере. К тому времени была известна константа C, такая что если такие числа существуют, то они не превосходят этой константы. Была написана программа, содержащая четыре вложенных цикла по x, y, z и $n > 2$ с верхними границами C, в теле которых стояла проверка условия $x^n + y^n = z^n$. Программа была написана аккуратно, с сохранением промежуточных вычислений на случай сбоев работы. Она работала около года и выдала результат: «Таких чисел не существует». Является ли эта программа реализацией алгоритма? Ведь у неё не то что не бесконечное множество исходных данных, у этой программы исходных данных нет.

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

3.2. ПРЕДСТАВЛЕНИЕ О РЕКУРСИВНЫХ ФУНКЦИЯХ. ТЕЗИС ЧЁРЧА

Понятие рекурсивных функций было предложено независимо Чёрчем и Клини. Каждое из определений содержало недостатки, поэтому в результате совместного обсуждения получилось то, что сейчас вошло во многие учебники, например в [10].

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 $1 \leqslant m \leqslant n$[4].

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 и/или примитивной рекурсии.

Всякая примитивно рекурсивная функция определена для любого набора значений своих аргументов. Примитивно рекурсивными явля-

[4] Обратите внимание, что первые две функции напоминают функции из сигнатуры аксиоматической теории чисел.

ются, например, функции, определяемые термами $x + y$, $|x - y|$, $x \cdot y$, $[\frac{x}{y}]$, x^y, $[\sqrt{x}]$, характеристические функции предикатов $=$, $<$, $>$, $\leqslant$, $\geqslant$ и многие другие.

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

Но применение ограниченного μ-оператора, действие которого определяется как $g(x_1, \ldots, x_n, z) = \mu_{y \leqslant z}\{f(x_1, \ldots, x_n, y) = 0\}$ (наименьший y, не превосходящий z и для которого $f(x_1, \ldots, x_n, y) = 0$, или 0, если такого 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 и/или μ-оператора.

Не все частично рекурсивные функции определены для любого набора значений своих аргументов. Так, например, функция $\mu y\{x + y = 5\}$ определена только для x, равных 0, 1, 2, 3, 4 и 5. 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 и/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.

Понятие рекурсивной функции над числами было распространено на функции, обрабатывающие слова. При этом простейшая функция S заменяется на функции, приписывающие один из символов алфавита к слову. Рекурсия ведётся по длине обрабатываемого слова, μ-оператор находит наименьшее по длине слово.

Тезис Чёрча. Всякая интуитивно вычислимая функция является общерекурсивной.

Это утверждение называется «тезис», а не «теорема», так как его нельзя ни доказать, ни опровергнуть. В его формулировке имеется понятие «интуитивно вычислимая функция», которое не имеет точного математического определения. Каждый человек может иметь своё мнение, относительно того, можно ли значение этой функции вычислить при ВСЕХ наборах значений переменных. Но это его субъективное мнение. Его оппонент может предложить некоторое значение аргумента, для которого предложенный интуитивный метод вычисления не даёт результата. И так может продолжаться сколь угодно долго.

3.3. МАШИНЫ ТЬЮРИНГА. ТЕЗИС ТЬЮРИНГА — ЧЁРЧА

Машина Тьюринга (МТ) [11] 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 — это следующая структура:

$$\langle A; Q, Q_I, Q_E; P \rangle.$$

Здесь $A = \{a_1, \ldots, a_n\}$ — внешний алфавит, или алфавит символов, которые могут быть записаны на ленте, содержащий, в частности, пустой (или бланковый) символ; $Q = \{q_0, q_1, \ldots, q_k\}$ — внутренний алфавит, или алфавит состояний, в которых может находиться машина, содержащий два выделенных подмножества: Q_I — множество начальных состояний и Q_E — множество заключительных состояний; P — программа.

При дальнейшем изложении пустой символ будет обозначаться посредством *, в качестве начального состояния будет использоваться состояние q_1, в качестве заключительного (если не оговорено особо) — состояние q_0.

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.

Примером несогласованных команд могут служить следующие две команды:

$$q_1| \to q_1 R1$$
$$q_1| \to q_1 R0.$$

Эти команды предполагают, что в одной и той же ситуации (машина Тьюринга находится в состоянии q_1 и обозревает ячейку, в которой записан символ |) нужно заменить | и на 0, и на 1.

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-й символ этого слова; на концах конфигурации находится не более чем по одному пустому символу.

Машина Тьюринга всегда начинает работу в конфигурации вида $q_1 X$, где X — исходные данные. Конфигурация соответствует содержимому ленты и головки, изображённым ниже.

...	*	b_1	...	b_{p-1}	b_p	...	b_l	*	...
					↑				
					q_r				

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.

Однако доказана равносильность тезиса Чёрча и тезиса Тьюринга — Чёрча. В качестве несложного упражнения можно доказать, что функции S, O и I_n^m могут быть вычислены на машине Тьюринга. Возможность вычисления на машине Тьюринга суперпозиции двух функций будет доказана для частного случая ниже.

3.3.1. Примеры 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 единичная (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. Будем сдвигать головку вправо, пока не увидим знак +, который заменим на 1.

3. Будем сдвигать 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 применить 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 $q_1 X * Y$, где X и Y — двоичные записи натуральных чисел. Требуется, чтобы заключительной конфигурацией была $q_0 Z$, 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 0 и запоминаем, что к сумме следующих двух цифр будет необходимо прибавить 1.

Если складывать необходимо запомненные цифры +1, то в качестве суммы пишем 0 + 0 как 1 и «освобождаемся от запоминания»; вместо 0 + 1 или 1 + 0 пишем 0, вместо 1 + 1 пишем 1 и запоминаем, что к сумме следующих двух цифр будет необходимо прибавить 1.

7. «Добежим вправо» до символа *, стоящего после последней цифры числа вычисленной части суммы, и перейдем к выполнению п. 1 нашего плана. При этом если в п. 6 было запомнено, что в следующий раз к сумме цифр требуется прибавить 1, то все команды в п. 1–6 нужно продублировать с новыми состояниями, соответствующими тому, что «1 на ум пошло».

8. Если одно из слов (X либо Y) оказалось короче другого, то символ *, стоящий 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 чисел X и Y соответственно, т.е. составляет $O(n \cdot m)$.

3.3.2. 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 $q_0 f(X)$.

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

$$q_1 a_i \to q_1 L a_i;$$
$$q_1 * \to q_2 R \#.$$

В программе машины M каждое вхождение состояния q_i ($i = 1, \ldots, k$) заменяем на q_{i+1}.

Кроме того, необходимо «сдвигать» символ # влево, если головка его обозревает. С этой целью для каждой команды вида

$$q_r a_i \to q_t L a_j$$

добавим две команды:

$$q_t \# \to q_{t'} L *;$$
$$q_{t'} * \to q_t R \#.$$

Возможны два случая завершения работы программы над данными: $\# * \ldots * Y' q_0 Y''$, где $Y'Y''$ совпадает с $f(X)$, и $\# * \ldots * Y' q_t a_i Y'''$, где $Y' a_i Y'''$ совпадает с $f(X)$ и в программе отсутствует команда с левой частью $q_t a_i$.

В первом случае заменяем в тексте программы состояние q_0 на q_{k+2} и добавляем команды:

$$q_{k+2} a_i \to q_{k+2} L a_i \ (a_i \in A);$$
$$q_{k+2} \# \to q_{k+3} R *;$$
$$q_{k+3} * \to q_{k+3} R *;$$
$$q_{k+3} a_i \to q_0 L a_i \ (a_i \in A \setminus \{*\}).$$

Во втором случае для каждого состояния q_t и каждого символа a_i из алфавита A, для которых в программе отсутствует команда с левой частью $q_t a_i$, добавляем команду

$$q_t a_i \to q_{k+2} L a_i$$

и команды, указанные для первого случая.

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 $\{q_0, q_1, ..., q_{k_i}\}$. В соответствии с леммой можно считать, что M_1 заканчивает работу в конфигурации $q_0 g(X)$.

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

В программе P_2 состояния q_i ($i = 1, ..., k_2$) 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 машину 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 $f(X_1, ..., X_{i-1}, g(Y_1, ..., Y_m), X_{i+1}, ..., X_n g(X))$.

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

Машина с программой P_0 с состояниями $q_0, q_1, \ldots, q_{k0}$ отмечает место между X_{i-1} и X_{i+1} и подводит головку к первому символу данных $Y_1, \ldots, Y_m$. 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 $q_{k_0+k_3+1}$ и получаем программу P_3'.

Машина с программой P_4 с состояниями $q_0, q_1, \ldots, q_{k_4}$ 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 $q_{k_0+k_3+i}$ при $i \neq 0$, а q_0 на $q_{k_0+k_3k_4+1}$, получаем программу P_4'.

В программе P_1 для машины M_1 заменяем q_i при $i \neq 0$ на $q_{k_0+k_3+k_4+i}$ и получаем программу P_1'.

Машина с программой $P_0' \cup P_3' \cup P_4' \cup P_1'$ вычисляет $f(X_1, \ldots, X_{i-1}, g(Y_1, \ldots, Y_m), X_{i+1}, \ldots, X_n g(X))$.

3.3.3. Многоленточные машины Тьюринга

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.

Обычно предполагается, что 1-я лента — это входная лента для записи исходных данных, k-я 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. Ниже в примерах это предположение не будет применяться, но всегда будут явно указываться исходная и заключительная конфигурации.

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 $S_1, \ldots, S_k \in \{L, R, \_\}$ 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 $a_{i_1}, \ldots, a_{i_k}$, то эти символы заменяются соответственно на $a_{j_1}, \ldots, a_{j_k}$, головка производит сдвиги $S_1, \ldots, S_k$ и машина Тьюринга переходит в состояние q_t».

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.

Вернемся к задаче сложения двух чисел, записанных в двоичной системе счисления, для решения которой на одноленточной машине Тьюринга был разработан план программы, но сама программа не была приведена из-за её чрезмерной громоздкости.

Пример 3. Написать трёхленточную 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 трёхленточной 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 $O(\max\{n, m\})$.

Отметим, что эта оценка совпадает с оценкой числа «шагов» при сложении многоразрядных чисел. Более того, оценки числа шагов многоленточной машины Тьюринга более адекватны оценкам числа «шагов» компьютера, решающего ту же задачу, чем оценки числа «шагов» одноленточной машины Тьюринга для этой задачи.

В качестве упражнения можно написать программу многоленточной машины Тьюринга, вычисляющую произведение двух чисел (или хотя бы разработать план для такой программы). Можно использовать пятиленточную машину Тьюринга, в которой на первые две ленты записываются сомножители (исходные данные), на последнюю — результат работы, а 3-я и 4-я ленты служат для выполнения промежуточных вычислений (аналогично массивам $D1$ и D при умножении многоразрядных чисел).

Оценив число шагов этой машины Тьюринга, можно убедиться, что она совпадает с оценкой числа «шагов» при умножении многоразрядных чисел.

3.3.4. Теорема о числе шагов машины Тьюринга, моделирующей работу многоленточной машины Тьюринга

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 $n \leqslant t$. Будем считать, что 1-я лента входная, а последняя — выходная.

На i-м шаге ($i = 1, \ldots, t$) длина записи на j-й 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-й 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 $4\sum_{j'=j}^{k} \left\| X_i^{j'} \right\|$ шагов);
- вернуться в исходное положение (не более чем $\sum_{j'=j}^{k} \left\| X_i^{j'} \right\|$ шагов).

Всего при моделировании действия MTk с одним символом на i-м шаге число шагов $MT1$ не превосходит

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

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

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

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

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

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

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

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

3.3.5. Многоголовчатые машины Тьюринга

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:

1) запрет на то, чтобы разные головки обозревали одну и ту же ячейку, точнее, требование, чтобы головка с большим номером всегда обозревала ячейку, которая находится правее, чем ячейка, обозреваемая головкой с меньшим номером;

2) 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:

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

где $S_1, \ldots, S_m \in \{L, R, \_\}$ 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.

3.3.6. Недетерминированные машины Тьюринга

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?

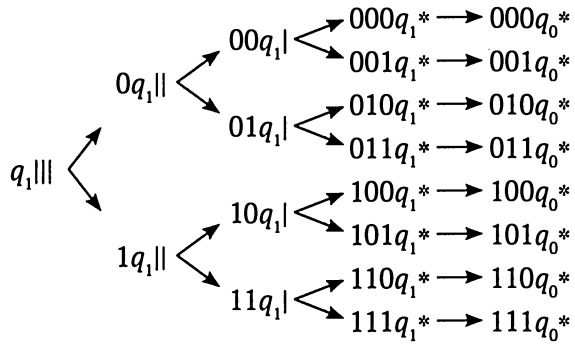
Для выполнения такой пары команд лента недетерминированной машины Тьюринга размножается, и на каждой ленте выполняется ровно одна из команд. Дальнейшие вычисления на каждой ленте продолжаются независимо в соответствии с общей программой.

Продемонстрируем протокол работы над исходными данными ||| следующей программы:

$$q_1 | \to q_1 R 1$$

$$q_1 | \to q_1 R 0$$

$$q_1 * \to q_0 *.$$



Несложно увидеть, что эта программа выписывает 2^n наборов из нулей и единиц длины n, причём в качестве исходных данных взято число n, записанное в унарной системе счисления.

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:***

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

2) 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 при этом используют два q_Y и q_N (происходящих от слов Yes и No). Недетерминированная машина Тьюринга заканчивает работу в состоянии q_Y, если хотя бы 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.

3.3.7. Теорема о числе шагов машины Тьюринга, моделирующей работу недетерминированной машины Тьюринга

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.

$$m \le \sum_{i=0}^{t} k^i = \frac{k^{t+1}-1}{k-1} \le k^{t+1} = 2^{(t+1)\log k} = 2^{O(t)},$$

так как k является константой.

Доказательство теоремы. Работу детерминированной машины Тьюринга организуем следующим образом:

- порождаем Y_1, проверяем $P(X, Y_1) \leqslant t$ шагов;

⋮

- порождаем Y_m, проверяем $P(X, Y_m) \leqslant t$ шагов.

Общее число шагов не превосходит $m \cdot t \leqslant t2^{O(t)} = 2^{O(t+\log t)} = 2^{O(t)}$.

3.4. НОРМАЛЬНЫЕ АЛГОРИТМЫ МАРКОВА

Теория нормальных алгоритмов (или алгорифмов, как называл их создатель теории) [12] была разработана советским математиком А.А. Марковым (1903–1979) в конце 1940-х — начале 1950-х гг.

Определение. Марковской подстановкой $P \to Q$ называется операция над словами P и 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 нём словом Q. Полученное слово называется результатом применения мар-

ковской 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.**

Результатом применения подстановки $P \to Q$ к слову R является слово, полученное из R в результате замены первого вхождения в него слова P на слово Q. Так, например, результатом применения подстановки $m \to p$ к слову *papa* является слово *tapa*.

Подстановка $P \to Q$ называется **неприменимой к слову** R, если слово P не входит в слово R. Так, например, подстановка $s \to p$ не применима к слову *papa*.

Упорядоченный конечный список подстановок

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

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. То есть на каждом шаге ищется первая подстановка, которую можно применить к текущему слову.

Если на некотором шаге применено заключительное правило или не применено 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.

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, так как 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 § (подстановка 2).

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. $\#c \to c\#$
2. $\# \to \S$
3. $000\S \to \S0$
4. $001\S \to \S1$
5. $010\S \to \S2$
6. $011\S \to \S3$
7. $100\S \to \S4$
8. $101\S \to \S5$
9. $110\S \to \S6$
10. $111\S \to \S7$
11. $10\S \to \cdot 2$
12. $11\S \to \cdot 3$
13. $1\S \to \cdot 1$
14. $1 \to \#1$

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

$$1001110011 \xrightarrow{14} \#1001110011 \xrightarrow{1} 1\#001110011 \xrightarrow{1} 10\#01110011\ldots$$
$$\xrightarrow{1} 100111001\#1 \xrightarrow{1} 1001110011\# \xrightarrow{2} 1001110011\S$$
$$\xrightarrow{6} 100111\S3 \xrightarrow{8} 1001\S53 \xrightarrow{4} 1\S153 \xrightarrow{13} 1153$$

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-й 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 применены «макроправила», 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 любые, может быть, различные буквы алфавита A.

1. $c\bot \to \bot$
2. $\bot \to\cdot \bot$
3. $\#c\#c_1 \to c_1\#c\#$
4. $c\#c\# \to$
5. $c_1\#c\# \to \bot$
6. $cc_1 \to c_1\#c\#$
7. $c \to\cdot \top$
8. $\to\cdot \top$

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

$$NON \xrightarrow{6} O\#N\#N \xrightarrow{3} ON\#N\# \xrightarrow{4} O \xrightarrow{7} \top.$$

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

$$NOT \xrightarrow{6} O\#N\#T \xrightarrow{3} ON\#T\# \xrightarrow{5} O\bot \xrightarrow{1} \bot \xrightarrow{2} \bot.$$

Для нормальных алгоритмов имеет место утверждение, аналогичное тезису Чёрча и тезису Тьюринга — Чёрча [12].

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

Как и в случае тезиса Чёрча и тезиса Тьюринга — Чёрча, это утверждение нельзя ни доказать, ни опровергнуть, так как в формулировке утверждения присутствует неформализованное понятие «интуитивно вычислимая функция».

Но справедлива теорема.

Теорема 3.5. Следующие классы функций совпадают:

а) класс всех функций, вычислимых по Тьюрингу;

б) класс всех частично рекурсивных функций;

в) класс всех нормально вычислимых функций.

В частности, легко можно доказать, что по всякой программе машины Тьюринга можно построить нормальный алгоритм, вычисляющий ту же функцию.

Действительно, каждой команде машины Тьюринга поставим в соответствие следующие замены (см. табл. 3.1).

Таблица 3.1

Соответствие замен в нормальном алгоритме командам машины Тьюринга

Команда МТ	Замена	Комментарий
$q_j a_i \to q_r a_t$	$q_j a_i \to q_r a_t$	$j, r \in \{1, ..., k\}, i, t \in \{1, ..., n\}$
$q_j a_i \to q_r L a_t$	$a_l q_j a_i \to q_r a_l a_t$	$j, r \in \{1, ..., k\}, i, t, l \in \{1, ..., n\}$
$q_j a_i \to q_r R a_t$	$q_j a_i \to a_t q_r$	$j, r \in \{1, ..., k\}, i, t \in \{1, ..., n\}$
$q_j a_i \to q_0 a_t$	$q_j a_i \to \cdot\, a_t$	$j \in \{1, ..., k\}, i, t \in \{1, ..., n\}$
$q_j a_i \to q_0 L a_t$	$a_l q_j a_i \to \cdot\, a_l a_t$	$j \in \{1, ..., k\}, i, t, l \in \{1, ..., n\}$
$q_j a_i \to q_0 R a_t$	$q_j a_i \to \cdot\, a_t$	$j \in \{1, ..., k\}, i, t \in \{1, ..., n\}$

При этом последними в записи нормального алгоритма должны быть замены вида $a_i \to q_1 a_i$, где a_i — символ рабочего алфавита машины Тьюринга, присутствующий в командах, в левой части которых имеется состояние q_1.

3.5. КОНСТРУКТИВНЫЕ ОБЪЕКТЫ

Заметим, что во всех приведённых выше математических понятиях алгоритма они обрабатывают слова в некотором алфавите. Может показаться, что это не так в определении рекурсивных функций. Но по сути в рекурсивных функциях числа представлены в унарной системе счисления: имеются функции O и S, в качестве натуральных чисел выступают (как и в аксиоматической теории чисел) постоянные термы вида $S(... (S(0)) ...)$. В некоторой литературе вместо функции S исполь-

зуют штрих. Так и получается унарная запись числа, начинающаяся с нуля: $0, 0', 0'', \ldots, 0^{\overbrace{|\ldots|}^{n}}, \ldots$.

Определение. Конструктивные объекты — это объекты, которые могут быть построены из конечного числа исходных объектов с помощью применения к ним конечного числа строго определённых операций.

Обычно конструктивные объекты (языки) задаются с помощью формальных грамматик. В этом учебном пособии не будут излагаться формальные грамматики, а будет продемонстрирован способ задания конструктивных объектов с помощью формул Бэкуса.

Формулы Бэкуса имеют вид:

$$\langle \textit{Понятие} \rangle ::= \langle \textit{Понятие 1} \rangle \mid \ldots \mid \langle \textit{Понятие n} \rangle \mid \langle \textit{Понятие i} \rangle * \langle \textit{Понятие j} \rangle \mid \ldots \mid \langle \textit{Понятие } i_1 \rangle \# \ldots \% \langle \textit{Понятие } i_k \rangle.$$

Здесь ⟨*Понятие*⟩ — это имя определяемого объекта; ⟨*Понятие i*⟩ — имя ранее определённого, или исходного, или определяемого объекта; *, #, % — какие-либо из допустимых операций; знак ::= читается как «это есть»; знак | читается как «или».

Самым простым конструктивным объектом является слово в заданном алфавите $A = \{a_1, \ldots, a_n\}$.

⟨*буква*⟩ ::= $a_1 \mid \ldots \mid a_n$

⟨*слово*⟩ ::= ⟨*буква*⟩ | ⟨*слово*⟩ ⟨*буква*⟩

При таком определении пустое слово не является словом, что не очень удобно при решении многих задач. Чтобы исправить этот недостаток, можно ввести понятие пустого слова и изменить определение слова.

⟨*пустое слово*⟩ ::=

⟨*буква*⟩ ::= $a_1 \mid \ldots \mid a_n$

⟨*слово*⟩ ::= ⟨*пустое слово*⟩ | ⟨*слово*⟩ ⟨*буква*⟩

Более сложно определяемым конструктивным объектом является двоичная (или десятичная, или *m*-ичная) запись натурального числа. Казалось бы, что это слово в алфавите {0, 1}. Но проблема в том, что запись натурального числа, отличного от нуля, не начинается с 0[5]. В связи с этим придётся ввести понятие непустого слова в алфавите {0, 1} и целого положительного числа.

[5] В математической логике и теории алгоритмов число 0 принято считать натуральным, в отличие от алгебры, где натуральные числа начинаются с единицы.

⟨*цифра*⟩ ::= 0 | 1

⟨(0 – 1) *слово*⟩ ::= ⟨*цифра*⟩ | ⟨(0 – 1) *слово*⟩ ⟨*цифра*⟩

⟨*целое* **положительное**⟩ ::= 1⟨(0 – 1) *слово*⟩

⟨*натуральное*⟩ ::= ⟨*целое положительное*⟩ | 0

Объектами, само определение которых наталкивает на мысль об их определении с помощью формул Бэкуса, являются пропозициональные формулы. Проблема состоит в том, что в их определении присутствует понятие пропозициональной переменной. Есть несколько выходов из этого положения. Первый (плохой) — считать, что понятие пропозициональной переменной является исходным, но тогда у нас в исходных объектах появляется бесконечное подмножество. Второй — считать, что пропозициональная переменная — это слово в заданном алфавите, начинающееся с определённой (или одной из фиксированного множества) буквы. Ниже приведён не самый удобный для использования, но коротко записываемый способ определения:

⟨*пропозициональная переменная*⟩ ::= *p* | ⟨*пропозициональная переменная*⟩1

⟨*логическая связка*⟩ ::= & | ∨ | → | ↔ | ⊕ |

⟨*пропозициональная формула*⟩ ::= ⟨*пропозициональная переменная* | ¬⟨*пропозициональная формула*⟩ | (⟨*пропозициональная формула*⟩ ⟨*логическая связка*⟩ ⟨*пропозициональная формула*⟩)

3.6. 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 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, т.е. 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 гл. 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 всех них исходные данные — это конструктивные объекты, множество которых потенциально бесконечно.

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

Предполагается, что имеется потенциально бесконечная память, ячейки которой занумерованы. В каждой 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 go to *M*.

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 RAM (Random Access Memory) — машина с прямым доступом, в которой разрешена операция сложения. Имеются её модификации: RAM с умножением и BOOL-RAM (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 гл. 4 будет рассмотрена теория сложности алгоритмов. Для адекватной оценки времени работы программы на компьютере подходящими из перечисленных являются лишь детерминированные модификации машины Тьюринга, нормальные алгоритмы и RAMы.

Упражнения.

Написать программы машины Тьюринга и нормального алгоритма, решающие следующие задачи. Написать протокол работы с заданными исходными данными.

1. Предикат равенства двух слов в алфавите $A = \{a_1, \ldots, a_n\}$. Исходные данные: а) дом, дом; б) дом, дам.

2. Вычисление предиката «X кратно трем», где X задано в десятичной системе счисления. Исходные данные: а) 242; б) 243.

3. Поменять местами первую и последнюю буквы слова в алфавите $A = \{a_1, \ldots, a_n\}$. Исходные данные: дом.

4. Заменить i-ю цифру числа X на цифру a (i задано в унарной системе счисления, X и a — десятичные). Исходные данные: $i = 3$, $X = 331$, $a = 2$.

5. Вычислить $MAX(X, Y)$, где X, Y заданы в двоичной системе счисления. Исходные данные: а) $X = 2$, $Y = 3$; б) $X = 6$, $Y = 3$.

6. Вычислить усеченную разность $X \dot{-} Y$, где X, Y заданы в двоичной системе счисления.

$$X \dot{-} Y = \begin{cases} X - Y, \text{ если } X \geq Y, \\ 0 \qquad \text{иначе.} \end{cases}$$

Исходные данные: а) $X = 3$, $Y = 2$; б) $X = 2$, $Y = 3$.

7. Вычислить значение постоянной пропозициональной формулы. Исходные данные: $((\neg t \vee f) \rightarrow \neg t)\ \&\ t$. Здесь t — обозначает *true*, f — обозначает *false*.

8. Вычислить значение пропозициональной формулы в ДНФ с двумя переменными на заданном наборе значений. Исходные данные: $x = true$, $y = false$, $\neg x \,\&\, y \vee \neg y$.

9. Вычислить значение пропозициональной формулы в КНФ с двумя переменными на заданном наборе значений. Исходные данные: $x = false$, $y = true$, $(\neg x \vee y) \,\&\, \neg y$.

10. Вычислить целую часть от деления x на 2^n, где x — двоичная запись натурального числа, n — унарная запись натурального числа. Исходные данные: $x = 18_{10}$, $n = 2_{10}$.

11. Вычислить остаток от деления x на 2^n, где x — двоичная запись натурального числа, n — унарная запись натурального числа. Исходные данные: $x = 20_{10}$, $n = 2_{10}$.

12. Вычислить произведение x на 2^n, где x — двоичная запись натурального числа, n — унарная запись натурального числа. Исходные данные: $x = 20_{10}$, $n = 2_{10}$.

13. В слове из алфавита $\{a_1, ..., a_k, b_1, ..., b_n\}$ все буквы a_i $(i = 1, ..., k)$ заменить словами $0\underbrace{I\,I\,...\,I}_{i \text{ раз}}0$. Исходные данные: $k = 3$, $n = 4$, слово $a_3b_2a_1b_4$.

14. Вычислить предикат делимости числа x на y, заданных в унарной системе счисления. Исходные данные: $x = 9_{10}$, $y = 2_{10}$.

15. Инверсия слова в алфавите $\{a_1, ..., a_n\}$. Исходные данные: $n = 5_{10}$, слово: КОТ.

В качестве упражнений на определение конструктивных объектов можно предложить следующие задачи:

1. Записать формулы Бэкуса, определяющие:

- целые числа;
- рациональные числа (определяемые как пара чисел);
- рациональные числа (определяемые как конечная или бесконечная периодическая запись);
- десятичную запись натуральных чисел, кратных трём.

2. Доказать, что десятичная (или двоичная) запись вещественных чисел не является конструктивным объектом.

3. Записать формулы Бэкуса, определяющие:

- терм;
- предикатную формулу;
- команду машины Тьюринга;
- марковскую подстановку.

3.7. 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.

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

Поскольку каждый алгоритм (в терминах математического 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 применим к собственному коду:

$$!A(\#A).$$

Определение. Алгоритм A называется **самоанулируемым**, если результат его применения к собственному коду равен пустому слову:

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

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

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

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 применим алгоритм A, алгоритм B тоже применим и результаты их работы совпадают.

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

3.7.2. Теоремы о несуществовании алгоритма

Теорема 3.6.1. Не существует такого алгоритма B, который применим 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:

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

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 $\forall x(C(x) = B_0(x, x)||a_1)$[6]. Этот алгоритм всюду применим и, следовательно, применим к собственному коду и $C(\#C) = B_0(\#C, \#C)||a_1$.

[6] $P||Q$ означает конкатенацию (приписывание друг к другу) слов P и Q, a_1 — первый символ выходного алфавита.

Так как B_0 — продолжение универсального алгоритма, то верно, что $\forall AP(!A(P) \to !B_0(\#A, P)\ \&\ B_0(\#A, P) = A(P))$, в частности, при $A = C$, $P = \#C$ $B_0(\#C, \#C) = C(\#C)$. Это противоречит полученному ранее значению для $C(\#C)$.

3.8. 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 задача читается как «при каких x верна формула $\varphi(x)$?».

Примерами массовых проблем могут служить следующие:

1. При каких значениях коэффициентов квадратный трёхчлен имеет вещественный корень?

$$(?a, b, c)\ \exists x(x \in \boldsymbol{R}\ \&\ ax^2 + bx + c = 0)$$

2. При каких значениях параметра m значение квадратного трёхчлена больше нуля для всех x из отрезка $[0, 1]$?

$$(?m)(\forall x(x \in [0, 1] \to mx^2 + 2(m - 1)x + m - 3 > 0))$$

3. У каких многочленов с целыми коэффициентами P имеются целые корни?

$$(?P)\ \exists x(x \in \boldsymbol{Z}\ \&\ P(x) = 0)$$

4. У каких полиномов P от нескольких переменных с целыми коэффициентами имеются целые корни?

$$(?P)\ \exists \overline{x}(\overline{x} \in \boldsymbol{Z}^{*}\ \&\ P(\overline{x}) = 0)$$

5. Какие пропозициональные формулы тавтологичны (т.е. истинны для любого набора пропозициональных переменных)?

$(?F)$ (F — тавтологичная пропозициональная формула)

6. Какие предикатные формулы общезначимы (т.е. в любой интерпретации при любых наборах значений предметных переменных формула истинна)[7]?

(?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 (?x) $\varphi(x)$.

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

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

В 4-м примере сформулирована X проблема Гильберта. Сам Гильберт к началу XX в. 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 1969 г.

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

Теоремы об алгоритмической неразрешимости

Докажем алгоритмическую неразрешимость некоторых простейших массовых проблем.

---

[7] Примером общезначимой формулы является $\forall x A \leftrightarrow \neg\exists x \neg A$ для любой формулы A. Примером необщезначимой формулы служит $\exists x P(x)$, так как если $P(x)$ интерпретируется как «студент x был на лекции, на которой были слушатели», то эта формула истинна. Если же $P(x)$ интерпретируется как «x — натуральное число и $x + 1 = 0$», то эта формула ложна.

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 $q_0, q_1, \ldots, q_{k1}$.

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

$q_{k1+1} * \to q_{k1+2} *$;
$q_{k1+1} a \to q_0 a$, где a — любой непустой символ внешнего алфавита;
$q_{k1+2} a \to q_{k1+2} a$, где a — любой символ внешнего алфавита.

Эта машина Тьюринга остановится, если $B_0(P) \neq \Lambda$ (т.е. $\neg !A(\#A)$), и головка будет бесконечно двигаться вправо, если $B_0(P) = \Lambda$ (т.е. $!A(\#A)$). Но по теореме 3.6.1 такой машины Тьюринга не существует.

Теорема 3.7.2. Массовая проблема самоанулируемости алгоритма (?A) $A(\#A) = \Lambda$ алгоритмически неразрешима.

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

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

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

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

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

$q_{k1+1} * \to q_0\ 1$;
$q_{k1+1}\ a \to q_{k1+2}\ 0$, где a — любой непустой символ внешнего алфавита.

Эта машина Тьюринга даёт в ответе 0, если $B_0(P) \neq \Lambda$ (т.е. $A(\#A) = 0$), и ненулевое значение, если $B_0(P) = \Lambda$, (т.е. $A(\#A) \neq 0$). Но по теореме 3.6.2 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 в противном случае. Доказательство проведём для такого математического понятия алгоритма, как машина Тьюринга.

Пусть M_1 — программа машины Тьюринга, вычисляющей B_0 и имеющей состояния $q_0, q_1, \ldots, q_{k1}$. 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 $q_0, q_1, \ldots, q_{k2}$.

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, $q_1, \ldots, q_{k0}$.

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;

$q_{k0+k1+2}\,a \to q_{k0+k1+2}\,La$, 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[8].

Схема доказательства. 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-м шаге машина M находится в состоянии q_k», $H(i, j) \Leftrightarrow$ «на i-м шаге машина M обозревает j-ю ячейку», $S(i, j, h) \Leftrightarrow$ «на i-м шаге в j-й 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.

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

ГЛАВА 4. ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ

4.1. ЗАДАЧИ, ПРИВОДЯЩИЕ К ПОНЯТИЮ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ АЛГОРИТМА

Первые 2 примера уже рассматривались выше, но здесь они будут приведены ещё раз.

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. $z := \log(\sin(x \cdot y))$?

В зависимости от того, как написан транслятор для этого языка программирования, либо первый оператор 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 и т.д. А для вычисления 25-й степени?

Второй оператор, безусловно, можно считать 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 в степень, 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 результаты многих промежуточных вычислений не поместятся в ячейку. Следовательно, необходимо выполнять действия с числами произвольной длины, а это потребует выполнения дополнительных шагов работы программы (см. гл. 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 $2x + 1$ шаг. 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 гл. 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 (может быть, 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 суммы двух чисел $x + y$ 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 гл. 3). То есть следует выровнять записи X и Y по правому краю и складывать цифры, каждый раз сдвигаясь на одну ячейку вправо.

Переход от

$$q_1\begin{pmatrix} X \\ Y \\ * \end{pmatrix}$$ (исходная конфигурация)

к

$$q_0\begin{pmatrix} X \\ Y \\ \{X+Y\} \end{pmatrix}$$ (заключительная конфигурация)

произойдёт не более чем за $2\max\{||X||, ||Y||\} + 3$ шага[9].

При сравнении числа шагов в примерах 3, 4, 5 может возникнуть ощущение, что самый быстрый способ вычисления представлен в примере 3, в котором производится сложение чисел в унарной системе счисления. Ведь там число шагов линейно относительно длины записи исходных данных. Сравним полученные оценки для $x \approx y \approx 1\,000$. В первом примере $||X|| = x \approx 1\,000$, а во втором и третьем $||X|| \approx ||Y|| \approx 10$.

Число шагов вычисления в этих примерах составит соответственно:

1. ≈2 000.
2. ≈200.
3. ≈20.

Из этих примеров можно сделать следующее заключение.

При оценке числа шагов вычисления необходимо учитывать:

1. Математическую модель алгоритма, с помощью которого они производятся.
2. Способ представления исходных данных.
3. Длину записи результата и промежуточных вычислений.

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

4.2. ВРЕМЕННА́Я 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).

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

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

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

Взаимное расположение нижней и верхней сложностей алгоритма A при работе над данными длины n и над некоторыми данными P с длиной записи n представлено на рис. 4.1.

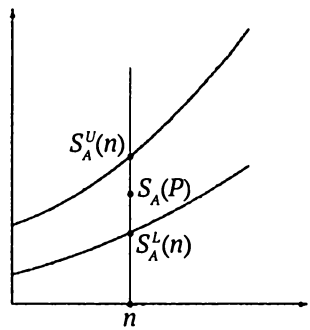


Рис. 4.1. Взаимное расположение нижней и верхней оценок сложности алгоритма A над данными P с длиной записи n

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

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

Пример случая, когда для задачи Z имеется только 2 алгоритма A_1 и A_2, представлен на рис. 4.2. При этом точная верхняя оценка сложности задачи Z совпадает с $S_{A_1}^U(n)$ при $0 \leqslant n \leqslant n_1$, с $S_{A_2}^U(n)$ при $n_1 \leqslant n \leqslant n_2$ и с $S_{A_1}^U(n)$ при $n \geqslant n_2$.

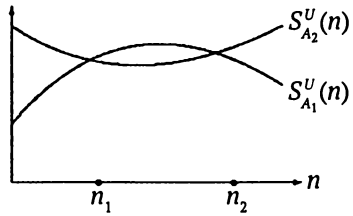


Рис. 4.2. Пример точной верхней и нижней точной оценки сложности при существовании только двух алгоритмов, решающих задачу Z

Нахождение точной верхней оценки сложности задачи — довольно трудное (и к тому же неблагодарное) дело. Обычно устанавливают порядок таких оценок или их асимптотику:

$$f(n) = O(g(n)) \Leftrightarrow \exists C \forall n (f(n) \leq C \cdot g(n)).$$

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 табл. 4.1 (взятая из [2]) позволяет воочию убедиться, насколько практически неприменимы алгоритмы, имеющие экспоненциальную временну́ю сложность. Эта таблица предполагает, что скорость работы компьютера 10^6 операций в секунду.

Таблицу можно заполнить самостоятельно, взяв в руки калькулятор и воспользовавшись тем, что время работы $T(n)$ выражается через

функцию сложности $f(n)$ (количество операций с данными длиной n) и быстродействие компьютера S (количество секунд, требуемых на выполнение одной операции) формулой $T(n) = f(n) \cdot S$.

Так, при заполнении строки таблицы с $f(n) = 2^n$ и $S = 10^{-6}$ получаем:

$T(10) = 2^{10} \cdot 10^{-6} = 1\,024 \cdot 10^{-6} = 0{,}001024$ с.

$T(20) = 2^{20} \cdot 10^{-6} = 1\,024 \cdot 1\,024 \cdot 10^{-6} = 1\,024 \cdot 0{,}001024 \approx 1{,}048576$ с.

$T(30) = 2^{10} \cdot T(20) \approx 1\,024 \cdot 1{,}048576$ с $= 1\,073{,}741824$ с $\approx$ $\approx 17{,}8956970666667$ мин.

$T(40) = 2^{10} \cdot T(30) \approx 1\,024 \cdot 17{,}8956970666667$ мин $\approx$ $\approx 18\,325{,}19379626667$ мин $\approx 305{,}4198964044444$ ч $\approx 12{,}72582902518519$ дней.

$T(50) = 2^{10} \cdot T(40) \approx 1\,024 \cdot 12{,}72582902518519$ дней $=$ $= 13\,031{,}24892178963$ дней $= 35{,}70205184051953$ лет.

$T(60) = 2^{10} \cdot T(50) \approx 1\,024 \cdot 35{,}70205184051953$ лет $= 36\,558{,}901084692$ лет $\approx$ ≈ 366 веков.

Желающие могут повторить эти вычисления для быстродействия $S = 10^{-14}$, которое мне на лекциях обычно сообщают студенты.

Таблица 4.1

Время вычисления при заданной временной сложности $f(n)$ с исходными данными длины n

	n					
$f(n)$	**10**	**20**	**30**	**40**	**50**	**60**
n	10^{-5} с	$2 \cdot 10^{-5}$ с	$3 \cdot 10^{-5}$ с	$4 \cdot 10^{-5}$ с	$5 \cdot 10^{-5}$ с	$6 \cdot 10^{-5}$ с
n^2	10^{-4} с	$4 \cdot 10^{-4}$ с	$9 \cdot 10^{-4}$ с	$16 \cdot 10^{-4}$ с	$25 \cdot 10^{-4}$ с	$36 \cdot 10^{-4}$ с
n^3	10^{-3} с	$8 \cdot 10^{-3}$ с	$27 \cdot 10^{-3}$ с	$64 \cdot 10^{-3}$ с	$1{,}25 \cdot 10^{-1}$ с	$2{,}16 \cdot 10^{-1}$ с
n^5	0,1 с	3,2 с	24,3 с	1,7 мин	5,2 мин	13 мин
2^n	10^{-3} с	1 с	17,9 мин	12,7 дней	35,7 лет	366 веков
3^n	0,059 с	58 мин	6,5 лет	3 955 веков	$2 \cdot 10^8$ веков	$1{,}3 \cdot 10^{13}$ веков

Существует мнение, что совершенствование компьютеров и увеличение скорости их работы позволит существенно уменьшить время работы любой программы. Таблица 4.2 (взятая из [2]) показывает изменение наибольшего размера исходных данных задачи, решаемой за 1 ч.

Здесь N_1, N_2, N_3, N_4, N_5, N_6 — наибольшая длина записи исходных данных, при которых задача гарантированно может быть решена за 1 ч при условии, что верхняя оценка её временно́й сложности составляет n, n^2, n^3, n^5, 2^n, 3^n соответственно.

Таблица 4.2

Изменение размера задачи, решаемой за 1 ч при увеличении быстродействия компьютера

Скорость опер./с			
Функция временной сложности	**10^6**	**10^8**	**10^9**
n	N_1	$100 \cdot N_1$	$1\,000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31{,}6 \cdot N_2$
n^3	N_3	$4{,}64 \cdot N_3$	$10 \cdot N_3$
n^5	N_4	$2{,}5 \cdot N_4$	$3{,}98 \cdot N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Продолжить эту таблицу при других скоростях работы компьютера можно самим, используя формулу:

$$f(N_i) \cdot 10^6 = f(N_i') \cdot 10^t,$$

где N_i' — наибольший размер исходных данных задачи, решаемой за 1 ч при скорости работы компьютера 10^t; f — соответствующая функция временно́й сложности.

Для полиномиальных по времени алгоритмов происходит извлечение корня k-й степени (k = 1, 2, 3, 5 для первых четырёх строк таблицы) из $N_i^k \cdot 10^{t-6}$. Поэтому увеличение размера происходит **в разы**. Точнее, в $10^{\frac{t-6}{k}}$ раз.

Для экспоненциальных по времени алгоритмов происходит логарифмирование по основанию a (a = 2, 3 для двух последних строк таблицы) выражения $a^{N_i} \cdot 10^{t-6}$. Поскольку $\log_a(a^{N_i} \cdot 10^{t-6}) = N_i + (t-6)\log_a 10$, то увеличение размера происходит **на единицы**. Точнее, на $(t-6)\log_a 10$.

ГЛАВА 5. КЛАССЫ СЛОЖНОСТИ АЛГОРИТМОВ

5.1. КЛАССЫ P, NP И P-SPACE. СООТНОШЕНИЯ МЕЖДУ ЭТИМИ КЛАССАМИ

В конце 70-х гг. XX в. сложилась следующая сведённая в табл. 5.1 система обозначений для некоторых классов сложности. Говорят, что задача принадлежит классу сложности ***C***, если существует алгоритм из класса ***C***, решающий эту задачу.

Таблица 5.1

Обозначения для некоторых классов сложности

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*

Буква ***F*** в начале названия класса используется для обозначения класса функций. Если её нет, то это класс предикатов, т.е. задачи, имеющие в ответе «ДА» или «НЕТ».

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. 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.

Приведём точные определения основных (в рамках нашего курса) классов сложности: класса ***P*** (полное обозначение в соответствии с приведёнными обозначениями ***D-P-TIME***), класса ***NP*** (полное обозначение в соответствии с приведёнными обозначениями ***N-P-TIME***) и класса ***P-SPACE*** (полное обозначение в соответствии с приведёнными обозначениями ***D-P-SPACE***).

Определение. Класс ***P*** — 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.

Определение. Класс ***NP*** — 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.

Определение. Класс ***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 следующие соотношения между основными классами сложности: $\boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{P\text{-}SPACE} \subset \boldsymbol{EXP}$. Вопрос о том, строгими или нестрогими являются первые два включения, объявлен одним из труднейших вопросов математики на XXI в.

В качестве следствия теоремы 3.4, доказанной в гл. 3, можно привести следующую теорему.

Теорема 4.1. Если предикат вида $\exists Y\, P(X, Y)$ принадлежит классу ***NP***, то существует проверяющая его 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 записи, равную числу шагов недетерминированной машины Тьюринга, а также могут принимать в качестве значения любое слово этой длины в заданном алфавите. Последняя теорема была опубликована, например, в [2] 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***.

5.2. ПОЛИНОМИАЛЬНАЯ СВОДИМОСТЬ И ПОЛИНОМИАЛЬНАЯ ЭКВИВАЛЕНТНОСТЬ

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 ($f \in \boldsymbol{FP}$);

- 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 неоднозначного прочтения, под задачей Z_i всегда будем понимать задачу вида $\exists Y P_i(X_i, Y)$ при $X_i \in D_i$.

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

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 $Z_2 \in \boldsymbol{P}$, то $Z_1 \in \boldsymbol{P}$.

Доказательство леммы следует из того, что в качестве алгоритма решения задачи Z_1 можно взять следующий: к исходным данным X за-

дачи 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 **ГАМИЛЬТОНОВ ЦИКЛ (ГЦ)** и **КОМИВОЯЖЁР**, одна из которых полиномиально сводится к другой[10].

Условия этих задач приведены в табл. 5.2.

Таблица 5.2

Условия задач ГЦ и КОМИВОЯЖЁР

ГЦ	КОМИВОЯЖЁР
Дано: граф $G = (V, E)$. $(\|\|V\|\| = n)$	Дано: $C = \{c_1, \ldots, c_n\}$ — множество городов, $d_{ij} \in Z_+$ — расстояния между c_i и c_j, $B \in Z_+$.
Вопрос: существует ли в G гамильтонов цикл?	Вопрос: существует ли маршрут, проходящий через все города, длина которого не больше B?
$\exists(v_{i_1}, \ldots, v_{i_n})(\{v_{i_1}, \ldots, v_{i_n}\} = V \,\&$ $\{v_{i_1}, v_{i_2}\} \in E \,\&\, , \ldots, \&$ $\{v_{i_n}, v_{i_1}\} \in E)$	$\exists(c_{i_1}, \ldots, c_{i_m})(\{c_{i_1}, \ldots, c_{i_m}\} = C \,\&$ $\sum_{j=1}^{m-1} d_{i_j i_{j+1}} + d_{i_m i_1} \le B$

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:

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

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

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

[10] Здесь и далее посредством Z_+ будем обозначать множество целых положительных чисел.

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, т.е. 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***.

5.3. NP-ПОЛНЫЕ ЗАДАЧИ

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

Из этого определения непосредственно следует теорема.

Теорема 4.3. Класс *NP*-полных задач образует класс эквивалентности по отношению к 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 $\exists Y\, P(X, Y)$ «существует ли Y (зависящий от X), удовлетворяющий условию $P(X, Y)$» является *NP*-полной, то соответствующая ей задача поиска $(?Y)\, P(X, Y)$ «при каких Y (зависящих от X), выполняется условие $P(X, Y)$» является *NP*-трудной. Очевидно, что задача поиска может оказаться намного сложнее задачи распознавания. Например, ответ на вопрос, имеет ли многочлен 15-й степени корень, очевиден, и таким ответом является «ДА». Задача «найти корень многочлена 15-й степени» гораздо сложнее.

На рис. 5.1 представлено взаимное расположение классов ***P*** и ***NP*** в предположении, что $\boldsymbol{P \neq NP}$ (верхняя оценка временно́й сложности возрастает при просмотре диаграммы снизу вверх). Буквами ***NPc*** и ***NPh*** обозначены *NP*-полные и *NP*-трудные задачи.

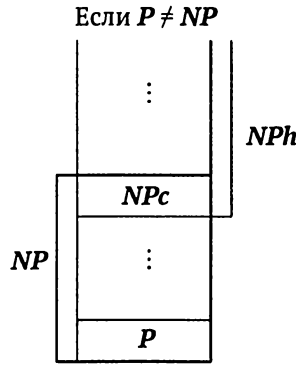


Рис. 5.1. Взаимное расположение классов *P*, *NP*, *NP*-полных и *NP*-трудных задач

Заметим, что если $\boldsymbol{P \neq NP}$, то между классом эквивалентности ***P*** и классом *NP*-полных задач есть ещё классы. В настоящее время к числу так называемых «открытых», или «висячих», задач, для которых не известен полиномиальный по времени решающий алгоритм и не доказана их *NP*-полнота, относится, например, задача ИЗОМОРФИЗМ ГРАФОВ (ИГ) или в английском написании GRAPH ISOMORPHISM (GI).

ИЗОМОРФИЗМ ГРАФОВ (ИГ)
Дано: графы $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$.
Вопрос: изоморфны ли $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$?

$$\exists f(f: V_1 \to V_2 \,\&\, f - \text{биекция} \,\&$$
$$\& \,\forall uv(u \in V_1 \,\&\, v \in V_1 \to (\{u, v\} \in E_1 \leftrightarrow (\{f(u), f(v)\} \in E_2)))$$

Многие комбинаторные задачи полиномиально эквивалентны задаче ИГ. Имеется даже термин «GI-полные задачи». В 2017 г. Ласло Бабай [13] предложил квазиполиномиальный алгоритм её решения. Полученная им оценка временно́й сложности $2^{O(\log^3 n)} = O(n^{\log^2 n})$ до сих пор (2023 г.) проверяется.

5.4. ЗАДАЧА ВЫПОЛНИМОСТЬ (ВЫП). ТЕОРЕМА КУКА

После того как дано определение какого-либо объекта или понятия, встаёт вопрос о том, а существуют ли такие объекты. Ведь если та-

ких объектов не существует, то всё, что мы говорим о них, является истинным.

Первым примером *NP*-полной задачи является следующая задача.

ВЫПОЛНИМОСТЬ (ВЫП)[11]

Дано: $U = \{u_1, \ldots, u_n\}$ — множество пропозициональных переменных; $C = \{c_1, \ldots, c_m\}$ — 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?

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

Теорема Кука. Задача **ВЫП** *NP*-полна.

Доказательство этой теоремы имеется в [2]. Здесь будет приведена только схема её доказательства.

Во-первых, необходимо доказать, что эта задача принадлежит классу ***NP***.

Действительно, порождение всех возможных наборов значений переменных $u_1, \ldots, u_n$ может быть осуществлено на недетерминированной машине Тьюринга за n шагов (такая недетерминированная машина Тьюринга, которая в качестве исходных данных имеет строку $\underbrace{|\ldots|}_{n}$ была приведена в качестве примера в гл. 3).

На каждой из полученных в недетерминированном режиме 2^n лент вычисляется c_1 & ... & c_m для найденных значений переменных. На двухленточной машине Тьюринга это можно сделать за $O(n||C||)$ шагов. Следовательно, на одноленточной — за $O(n^2||C||^2)$, т.е. за полином от $||U||$ и $||C||$.

Для доказательства того, что всякая задача из класса ***NP*** полиномиально сводится к задаче ВЫП, воспользуемся тем, что каждая такая задача может быть задана программой M для недетерминированной машины Тьюринга, заканчивающей работу не более чем за полином шагов от длины записи исходных данных.

Затем по каждой индивидуальной задаче из класса ***NP*** (т.е. программе M над алфавитом $S = \{s_0, \ldots, s_v\}$ с состояниями $\{q_1, \ldots, q_{r-2}, q_Y, q_N\}$ и исходным данным X длины n), проверяемой не более чем за $p(n)$ шагов, построим индивидуальную задачу из ВЫП таким образом, что они имеют решение одновременно.

[11] Эту задачу также называют задачей о КНФ или задачей *SAT* от слова SATISFIABILITY.

Для этого введём пропозициональные переменные:

$Q_{i,k}$ — «На i-м шаге программа M находится в состоянии q_k» ($0 \leqslant i \leqslant p(n)$, $1 \leqslant k \leqslant r$);

$H_{i,j}$ — «На i-м шаге читающая/пишущая головка обозревает j-ю ячейку» ($0 \leqslant i \leqslant p(n)$, $-p(n) \leqslant j \leqslant p(n)$);

$S_{i,j,k}$ — «На i-м шаге в j-й ячейке записан символ s_k» ($0 \leqslant i \leqslant p(n)$, $-p(n) \leqslant j \leqslant p(n)$, $0 \leqslant k \leqslant v$).

Очевидно, что количество этих переменных полиномиально от длины записи индивидуальной задачи. Следовательно, длина их записи не превосходит значения этого полинома, умноженного на его логарифм, и тоже полиномиальна.

После этого записывается $6p(n)(p(n) + 1)(r + 1)(v + 1)$ предложений (т.е. элементарных дизъюнкций, вид которых см. в [2]), истинных в том и только том случае, когда недетерминированная машина Тьюринга работает в соответствии с заданной программой над данными X и заканчивает работу в состоянии q_Y не более чем за $p(n)$ шагов.

- В каждый момент времени i ($0 \leqslant i \leqslant p(n)$) машина с программой M находится ровно в одном состоянии.

- В каждый момент времени i ($0 \leqslant i \leqslant p(n)$) читающая/пишущая головка обозревает ровно одну ячейку.

- В каждый момент времени i ($0 \leqslant i \leqslant p(n)$) каждая ячейка содержит ровно один символ из S.

- В момент времени 0 вычисление находится в исходной конфигурации стадии проверки при входе X.

- Не позднее, чем через $p(n)$ шагов, M переходит в состояние q_Y и, следовательно, принимает X.

- В каждый момент времени i ($0 \leqslant i \leqslant p(n)$) конфигурация программы M в момент времени $i + 1$ получается из конфигурации в момент времени i одноразовым применением команды программы M.

Длина каждого предложения не превосходит $2p(n)$. Таким образом, на выписывание этих предложений требуется $O(p(n)^3rv)$ шагов, т.е. сводимость полиномиальна.

5.5. ОСНОВНЫЕ NP-ПОЛНЫЕ ЗАДАЧИ

Другими примерами *NP*-полных задач являются следующие.

3-ВЫПОЛНИМОСТЬ (3-ВЫП)

Дано: $U = \{u_1, \ldots, u_n\}$ — множество пропозициональных переменных;

$C = \{c_1, ..., c_m\}$ — множество предложений над U с тремя переменными каждое.

Вопрос: выполнимо ли множество C, т.е. существует ли набор значений для переменных из U, для которого истинны все предложения из C?

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

ТРЕХМЕРНОЕ СОЧЕТАНИЕ[12]

Дано: множество $M \subseteq X \times Y \times Z$,

где $|X| = |Y| = |Z| = q$, $X \cap Y = Y \cap Z = Z \cap X = \varnothing$.

Вопрос: существует ли $M' \subseteq M$ мощности q где никакие два разных элемента из M' не имеют ни одной равной координаты?

$$\exists M'(M' \subseteq M \,\&\, |M'| = q \,\&\, \forall xyzuvw([x, y, z] \in M' \,\&\, \{u, v, w\} \in$$
$$\in M' \,\&\, \{x, y, z\} \neq \{u, v, w\} \rightarrow x \neq u \,\&\, y \neq v \,\&\, z \neq w)).$$

НЕЗАВИСИМОЕ МНОЖЕСТВО (НМ)

Дано: граф $G = (V, E)$;

$J \in Z_+, J \leqslant |V|$.

Вопрос: имеется ли в G независимое множество не менее чем из J элементов?

$$\exists V'(V' \subseteq V \,\&\, |V'| \geqslant J \,\&\, \forall uv((u \in V' \,\&\, v \in V') \rightarrow \{u, v\} \notin E)).$$

ВЕРШИННОЕ ПОКРЫТИЕ (ВП)

Дано: граф $G = (V, E)$;

$K \in Z_+, K \leqslant |V|$.

Вопрос: имеется ли в G вершинное покрытие не более чем из K элементов?

$$\exists V'(V' \subseteq V \,\&\, |V'| \leqslant K \,\&\, \forall uv(\{u, v\} \in E \rightarrow (u \in V' \vee v \in V')).$$

КЛИКА

Дано: граф $G = (V, E)$;

$J \in Z_+, J \leqslant |V|$.

Вопрос: содержит ли G клику не менее чем из J вершин?

$$\exists V'(V' \subseteq V \,\&\, |V'| \geqslant J \,\&\, \forall uv((u \in V' \,\&\, v \in V') \rightarrow \{u, v\} \in E)).$$

[12] Здесь и ниже обозначение $|A|$ использовано для мощности множества A.

ГАМИЛЬТОНОВ ЦИКЛ

Дано: граф $G = (V, E)$.

Вопрос: содержит ли G Гамильтонов цикл?

$$\exists(v_{i_1}, \ldots, v_{i_n})(\{v_{i_1}, \ldots, v_{i_n}\} = V \,\&\, \{v_{i_1}, v_{i_2}\} \in E \,\&, \ldots, \&\, \{v_{i_n}, v_{i_1}\} \in E).$$

РАЗБИЕНИЕ

Дано: конечное множество A,

для каждого $a \in A$ его «вес» $s(a) \in Z_+$.

Вопрос: существует ли разбиение множества A на два подмножества одинакового веса?

$$\exists A'\left(A' \subseteq A \,\&\, \sum_{a \in A'} s(a) = \sum_{a \in (A \setminus A')} s(a)\right).$$

У последней задачи может быть и другая формулировка.

РЕШЕНИЕ ЛИНЕЙНОГО ОДНОРОДНОГО УРАВНЕНИЯ В ЧИСЛАХ ИЗ {–1, 1}

Дано: $\{s_1, \ldots, s_n\}$ при $s_i \in Z_+$ $(i = 1, \ldots, n)$.

Вопрос: существует ли решение линейного однородного уравнения с коэффициентами $\{s_1, \ldots, s_n\}$ в числах из $\{-1, 1\}$?

$$\exists x_1 \ldots x_n(\&_{i=1}^{n} \;(x_i \in \{-1, 1\}) \,\&\, s_1 x_1 + \ldots + s_n x_n = 0).$$

5.6. МЕТОДЫ ДОКАЗАТЕЛЬСТВА NP-ПОЛНОТЫ

Как мы видели в схеме доказательства *NP*-полноты задачи ВЫП, возможно доказательство *NP*-полноты любой задачи непосредственно с помощью полиномиального сведения любой задачи из класса ***NP*** к исследуемой. Такие доказательства достаточно редки.

Доказательство *NP*-полноты многих задач основано на полиномиальной сводимости задачи, для которой уже известна её *NP*-полнота, к исследуемой.

Теорема 4.4. Если для задачи Z и *NP*-полной задачи Z_1 выполнены условия:

$$Z \in \boldsymbol{NP}, \;\; Z_1 \propto Z,$$

то Z *NP*-полна.

Доказательство основано на транзитивности отношения полиномиальной сводимости.

Замечание. Для доказательства *NP*-полноты задачи Z недостаточно доказать только, что $Z_1 \propto Z$. На рис. 5.2 показан случай, когда $Z_1 \propto Z$, но Z не *NP*-полна (она *NP*-трудна). На этом рисунке области D_Z, D_{Z_1} и $f(D_{Z_1})$ соответствуют множествам исходных данных для задач Z и Z_1 и образ множества данных для задачи Z_1 при отображении f, обеспечивающем полиномиальную сводимость. Буквами ***NPc*** и ***NPh*** обозначены *NP*-полные и *NP*-трудные задачи.

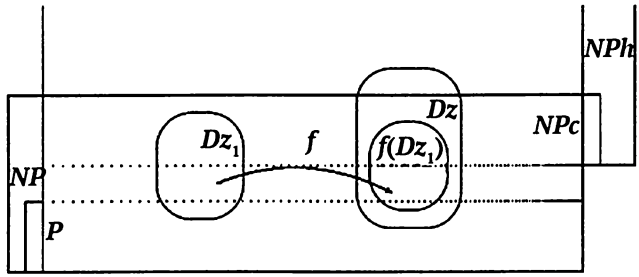


Рис. 5.2. Иллюстрация того, что для доказательства *NP*-полноты задачи Z требуется доказательство того, что $Z \in NP$

С помощью этой теоремы докажем *NP*-полноту задач 3-ВЫП и ВП.

Теорема 4.5. Задача 3-ВЫП *NP*-полна.

Доказательство. То, что эта задача принадлежит классу ***NP***, доказывается аналогично принадлежности классу ***NP*** задачи ВЫП.

Покажем, что задача ВЫП полиномиально сводится к задаче 3-ВЫП. Для этого по исходным данным U и C для задачи ВЫП построим такие исходные данные U' и C', что рассматриваемые задачи на этих исходных данных имеют решение одновременно.

Множество переменных U включим в множество U'.

Пусть $c_j = z_j^1 \vee z_j^2 \vee \ldots \vee z_j^k \in C$. Здесь $z_j^1, z_j^2, \ldots, z_j^k$ — литералы (переменные из U или их отрицания). Рассмотрим по отдельности случаи $k = 1$, $k = 2$, $k = 3$ и $k \geqslant 4$.

При $k = 3$ предложение c_j заносим в C'.

При $k = 2$

$$c_j = z_j^1 \vee z_j^2 \Leftrightarrow (z_j^1 \vee z_j^2 \vee y_j) \;\&\; (z_j^1 \vee z_j^2 \vee \neg y_j) = c_j^1 \;\&\; c_j^2.$$

Переменную y_j заносим в U', предложения c_j^1, c_j^2 заносим в C'.

При $k = 1$

$$c_j = z_j^1 \Leftrightarrow (z_j^1 \vee y_j^1 \vee y_j^2) \,\&\, (z_j^1 \vee \neg y_j^1 \vee y_j^2) \,\&\, (z_j^1 \vee y_j^1 \vee \neg y_j^2) \,\& \\ \& \,(z_j^1 \vee \neg y_j^1 \vee \neg y_j^2) = c_j^1 \,\&\, c_j^2 \,\&\, c_j^3 \,\&\, c_j^4.$$

Переменные y_j^1 и y_j^2 заносим в U', предложения c_j^1, c_j^2, c_j^3, c_j^4 заносим в C'.

Построим предложения, выполнимость конъюнкции которых равносильна выполнимости исходного предложения $c_j = z_j^1 \vee z_j^2 \vee \ldots \ldots \vee z_j^{k-1} \vee z_j^k$ при $k \geqslant 4$.

$$c_j^1 = z_j^1 \vee z_j^2 \vee y_j^1,$$

$$c_j^1 = \neg y_j^1 \vee z_j^3 \vee y_j^2,$$

$$\vdots$$

$$c_j^{l-1} = \neg y_j^{l-2} \vee z_j^l \vee y_j^{l-1},$$

$$\vdots$$

$$c_j^{k-3} = \neg y_j^{k-4} \vee z_j^{k-2} \vee y_j^{k-3},$$

$$c_j^{k-2} = \neg y_j^{k-3} \vee z_j^{k-1} \vee z_j^k.$$

Действительно, если исходное предложение выполнимо, например, при $z_j^l = true$, то $y_j^1, y_j^2, \ldots, y_j^{l-2}$ присвоим значения *true*, а $y_j^{l-1}, \ldots, y_j^{k-3}$ — значения *false*. Эти значения обеспечивают истинность построенных предложений.

Если выполнена конъюнкция построенных предложений, то хоть при одном l значение z_j^l равно *true*. В противном случае, просматривая предложения сверху вниз до предпоследнего включительно, получаем, что значения $y_j^1, y_j^2, \ldots, y_j^{k-3}$ равны *true*. При этом в последнем предложении стоит дизъюнкция ложных литералов.

Переменные $y_j^1, y_j^2, \ldots, y_j^{k-3}$ добавляем в U'. Предложения $c_j^1, c_j^2, \ldots, c_j^{k-2}$ добавляем в C'.

Покажем, что построение U' и C' по заданным U и C полиномиально от $||U||$ и $||C||$. Оценку числа шагов будем производить очень грубо, так как нам важна только полиномиальность полученной оценки. Пусть K — максимальное число литералов в предложениях, $K \leqslant ||C||$.

Количество новых переменных в U' не превосходит $\max\{2, K-3\} = O(K)$, общее количество переменных в U' составляет $O(||U|| + ||C||)$. Переменные можно записать так, что длина записи имени переменной не

превосходит $\log(||U'||) = O(\log(||U|| + ||C||)) = O(||U|| + ||C||)$. Таким образом, запись множества U' может быть реализована за $O((||U|| + ||C||)^2)$ шагов машины Тьюринга.

Количество предложений в C' не превосходит $||C|| \times \max\{4, K-2\} = O(||C||^2)$. Каждое предложение в C' содержит 3 литерала, поэтому длина его записи составляет $O(3(||U|| + ||C||))$. Таким образом, запись множества C' может быть реализована за $O(||C||^2(||U|| + ||C||))$ шагов машины Тьюринга.

Сложив полученные оценки и учитывая, что $||U|| \leqslant ||C||$, получаем полиномиальность от $||U||$ и $||C||$ построения U' и C' по заданным U и C:

$$O((||U|| + ||C||)^2) + ||C||^2(||U|| + ||C||) = O(||C||^3).$$

Теорема 4.6. Задача ВП *NP*-полна.

Доказательство. То, что эта задача принадлежит классу ***NP***, следует из того, что:

- во-первых, в недетерминированном режиме породить все подмножества множества V мощности не более чем K можно за $K \log K$ шагов на 2^K лентах ($K \leqslant ||V||$);

- во-вторых, в детерминированном режиме для конкретного подмножества V' проверка $\forall uv(\{u, v\} \in E \to (u \in V' \vee v \in V')$ на машине с прямым доступом возможна за $O(||E|| \cdot ||V'||) = O(||E||) \cdot ||V||)$.

Учитывая то, что детерминированная машина Тьюринга моделирует работу РАМ за число её шагов в кубе, получаем полиномиальное от длины записи графа число шагов недетерминированной машины Тьюринга, решающей задачу ВП.

Для того чтобы доказать, что 3-ВЫП полиномиально сводится к ВП, покажем, что по исходным данным любой индивидуальной задачи из 3-ВЫП за полином шагов можно построить исходные данные для задачи ВП так, чтобы эти индивидуальные задачи имели (или не имели) решение одновременно.

Пусть $U = \{u_1, \ldots, u_n\}$ и $C = \{c_1, \ldots, c_m\}$ — исходные данные для 3-ВЫП. Каждой переменной u_i ($i = 1, \ldots, n$) поставим с соответствие две вершины графа u_i и $\overline{u}_i$ ($2||U||$ шагов) и ребро $\{u_i, \overline{u}_i\}$ ($2||U||$ шагов).

Каждому предложению c_j поставим в соответствие три вершины графа a_j^1, a_j^2, a_j^3 ($3||C||$ шагов) и три ребра $\{a_j^1, a_j^2\}$, $\{a_j^2, a_j^3\}$, $\{a_j^3, a_j^1\}$ ($6||C||$ шагов).

Если k-м ($k = 1, 2, 3$) членом предложения c_j является u_i, то добавим ребро $\{u_i, a_j^k\}$, если же k-м ($k = 1, 2, 3$) членом предложения c_j является $\neg u_i$, то добавим ребро $\{\overline{u}_i, a_j^k\}$ ($6||C||$ шагов).

Так как покрытие каждого из рёбер вида $\{u_i, \overline{u}_i\}$ не может содержать менее n вершин, а покрытие каждого из рёбер треугольника $\{a_j^1, a_j^2\}$, $\{a_j^2, a_j^3\}$, $\{a_j^3, a_j^1\}$ не может содержать менее $2m$ вершин, то положим $K = n + 2m$ ($\max\{||n||, ||2m||\}$, что не превышает $\max\{||U||, ||C||\}$ шагов).

На построение этих исходных данных требуется $4||U|| + 12||C|| + \max\{||U||, ||C||\}$ на машине с прямым доступом. Следовательно, исходные данные для ВП строятся за полином шагов от длины записи исходных данных задачи 3-ВЫП.

Проиллюстрируем это сведéние на примере. Пусть $U = \{u_1, u_2, u_3, u_4\}$, $C = \{u_1 \vee \neg u_2 \vee u_3, \neg u_1 \vee u_3 \vee \neg u_4, u_2 \vee \neg u_3 \vee u_4\}$.

В этом примере $K = 10$.

Соответствующий граф изображён на рис. 5.3.

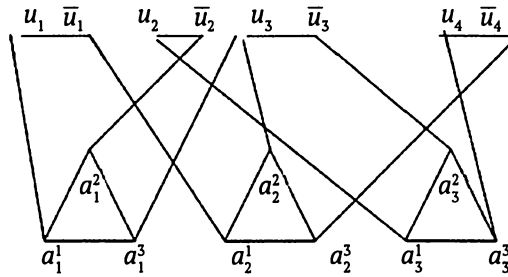


Рис. 5.3. Граф, соответствующий исходным данным в примере

Чтобы не загромождать рассуждение индексами, покажем на этом примере, что по набору констант, выполняющих множество C, можно построить вершинное покрытие для графа, и наоборот.

В этом примере очевидно, что набор констант ($t\ t\ f\ f$) выполняет C. Включим в вершинное покрытие графа вершины u_1, u_2, $\overline{u}_3$, $\overline{u}_4$. При этом наклонные рёбра, ведущие от них к a_1^1, a_3^1, a_1^3, a_2^3 будут покрыты. Исключим из вершинного покрытия любые 3 из них, входящие в разные «треугольники», например a_3^1, a_1^3, a_2^3. Остальные вершины «треугольников» включим в покрытие. Получили множество $V' = \{u_1, u_2, \overline{u}_3, \overline{u}_4, a_1^1, a_1^2, a_2^1, a_2^2, a_3^2, a_3^3\}$.

В этом примере можно подобрать другое вершинное покрытие, например $V' = \{\overline{u}_1, \overline{u}_2, u_3, u_4, a_1^1, a_1^3, a_2^1, a_2^3, a_3^1, a_3^2\}$. В качестве набора констант, выполняющих множество C, следует брать ($f\ f\ t\ t$).

5.7. МЕТОД СУЖЕНИЯ ДОКАЗАТЕЛЬСТВА NP-ПОЛНОТЫ

Определение. Задача Z_1 является сужением на множество D_1 задачи Z_2 с исходными данными из множества D_2, если $D_1 \subseteq D_2$ и $\forall X(X \in D_1 \rightarrow (Z_1 \leftrightarrow Z_2))$.

Если задача принадлежит классу ***NP***, а её подзадача *NP*-полна, то и исходная задача *NP*-полна, так как подзадача полиномиально сводится к исходной задаче с помощью тождественного отображения.

На этом основан метод сужения доказательства *NP*-полноты. То есть для доказательства *NP*-полноты задачи Z достаточно:

- доказать, что $Z \in$ ***NP***;
- среди известных *NP*-полных задач найти такую задачу Z_1, которая является сужением задачи Z.

Пример. Доказать *NP*-полноту следующей задачи.

МНОЖЕСТВО ПРЕДСТАВИТЕЛЕЙ

Дано: множество S, семейство C подмножеств множества S, $K \in \boldsymbol{Z}^+$.

Вопрос: содержит ли S множество представителей для C мощности, не превосходящей K, т.е. существует ли $S' \subseteq S$ такое, что $||S'|| \leqslant K$ и S' содержит по крайней мере один элемент из каждого множества семейства C?

$$\exists S'(S' \subseteq S \;\&\; ||S'|| \leqslant K \;\&\; \forall c(c \in C \rightarrow \exists s(s \in S' \;\&\; s \in c)))$$

Доказательство.

1. Задача МНОЖЕСТВО ПРЕДСТАВИТЕЛЕЙ принадлежит ***NP***, так как если предъявлено подмножество S' множества S мощности не более K, то проверка того, что S' содержит по крайней мере один элемент из каждого множества семейства C, может быть осуществлена не более чем за $||S'|| \cdot ||C|| \leqslant ||S|| \cdot ||C||$ шагов на двухленточной машине Тьюринга и, следовательно, не более чем за полином шагов от длины записи исходных данных на классической машине Тьюринга. Процесс порождения подмножества S' занимает не более $||S||$ шагов.

2. Если рассмотреть сужение этой задачи, в котором C — семейство двухэлементных подмножеств множества S, то получим граф $G = (S, C)$. Сама же задача МНОЖЕСТВО ПРЕДСТАВИТЕЛЕЙ превратится в задачу ВП.

5.8. АНАЛИЗ ПОДЗАДАЧ

Следующее определение носит терминологический характер. Очевидно, что термин «подзадача» взаимозаменим с термином «сужение».

Определение. Задача Z_1 с исходными данными из множества D_1 является подзадачей задачи Z_2 с исходными данными из множества D_2 ($Z_1 \subseteq Z_2$), если $D_1 \subseteq D_2$ и $\forall X(X \in D_1 \rightarrow (Z_1 \leftrightarrow Z_2))$.

Очевидными примерами подзадач являются 3-ВЫП $\subseteq$ ВЫП. При этом обе эти задачи *NP*-полны. Однако задача ВЫП имеет в качестве подзадачи, например, задачу 2-ВЫП, в условии которой каждое предложение содержит ровно 2 дизъюнктивных члена. Для решения 2-ВЫП известен полиномиальный алгоритм.

Если посредством

$$\circ, \odot \text{ и } \bullet$$

обозначить соответственно полиномиальные по времени, открытые[13] и *NP*-полные задачи, то возможно только следующее соотношение между ними:

$$\circ \subset \ldots \circ \subset \odot \subset \ldots \odot \subset \bullet \subset \ldots \bullet.$$

Так, например, для задачи ВЫП с длиной предложений n имеем следующие подзадачи:

n	1	2	3	...	произвольно
	$\circ$	$\circ$	$\bullet$	...	$\bullet$

Примером задачи, имеющей существенно различные с точки зрения теории сложности подзадачи, является следующая задача.

РАСПИСАНИЕ С ОТНОШЕНИЕМ ПРЕДШЕСТВОВАНИЯ

Дано: T — множество «заданий» длительностью 1;

$<\cdot$ — частичный порядок на T;

$m \in \boldsymbol{Z}_+$ — число «процессоров»;

$D \in \boldsymbol{Z}_+$ — директивный срок.

[13] Задачи из класса ***NP***, для решения которых не известен полиномиальный алгоритм и не доказана их *NP*-полнота.

Вопрос: существует ли такое расписание $\sigma : T \to \{0, 1, ..., D\}$, что для каждого $i \in \{0, 1, ..., D\}$ $||\{t : t \in T \,\&\, \sigma(t) = i\}|| \leqslant m$ и если $t <\cdot\, t_1$, то $\sigma(t) < \sigma(t_1)$.

Для этой задачи на рис. 5.4 изображена диаграмма современного состояния знаний о семействе её подзадач [2] (кружок, расположенный левее или ниже данного, изображает его подзадачу). Пустой кружок означает, что сужение задачи имеет полиномиальный алгоритм. Кружок с точкой внутри показывает, что соответствующее сужение является «открытой» задачей, т.е. для неё неизвестен полиномиальный алгоритм и не доказана *NP*-полнота. Тёмный кружок означает, что задача *NP*-полна.

	$m \leq 1$	$m \leq 2$	$m \leq 3$	...	$m \leq 100$	...	m
$<\cdot$ произвольно	○	○	⊙	...	⊙	...	●
$<\cdot$ дерево	○	○	○	...	○	...	○
$<\cdot$ пусто	○	○	○	...	○	...	○

Рис. 5.4. Современное состояние знаний о семействе подзадач задачи РАСПИСАНИЕ С ОТНОШЕНИЕМ ПРЕДШЕСТВОВАНИЯ

Как видно из этой диаграммы, *NP*-полнота доказана, только если частичный порядок и количество процессоров произвольны. При отсутствии частичного порядка или если он задаётся деревом, задача принадлежит классу ***P***.

Вывод. Прежде чем отказываться от программирования для многократного использования (для исходных данных большого размера) алгоритма, решающего *NP*-полную задачу, проверьте, не поставлена ли перед вами её подзадача, имеющая полиномиальный алгоритм.

5.9. ЗАДАЧИ С ЧИСЛОВЫМИ ПАРАМЕТРАМИ. ПСЕВДОПОЛИНОМИАЛЬНЫЕ ЗАДАЧИ

Рассмотрим алгоритм, решающий задачу **РАЗБИЕНИЕ**, и оценим число шагов его работы.

Напомним, что в условии задачи дано множество положительных чисел $\{s_1, ..., s_n\}$ и его требуется разбить на два подмножества с одинаковой суммой.

1. Вычислим $s_1 + \ldots + s_n$. Если число нечётное, то задача решения не имеет. В противном случае определим $B = ½(s_1 + \ldots + s_n)$.

2. Определим таблицу с элементами t_{ij} при $i = 1, \ldots, n, j = 0, 1, \ldots, B$. $t_{ij} \Leftrightarrow$ «в множестве $\{s_1, \ldots, s_i\}$ есть подмножество веса j».

3. Заполним таблицу, используя свойства t_{ij}:

$t_{i0} = T$ для всех i,

если $t_{ij} = T$, то $t_{(i+1)j} = T$,

если $t_{ij} = T$, то $t_{(i+1)(j+s_{i+1})} = T$.

4. Если в столбце с номером B появилось значение T, то задача имеет решение. Если же после заполнения таблицы ни в одной строке в последнем столбце нет значения T, то задача решения не имеет.

Пример. Пусть веса равны $s_1 = 1$, $s_2 = 9$, $s_3 = 5$, $s_4 = 3$, $s_5 = 8$. $s_1 + \ldots + s_5 = 26$ чётно, $B = 13$.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2	T	T								T	T			
3	T	T				T	T			T	T			
4	T	T		T	T	T	T		T	T	T		T	T
5	T	T		T	T	T	T		T	T	T	T	T	T

В последнем столбце имеется значение T, следовательно, задача имеет решение.

Эта таблица позволяет решить не только задачу распознавания, но и задачу поиска: разбить множество чисел на два подмножества с одинаковой суммой.

Впервые значение T появилось в последнем столбце в 4-й строке в результате прибавления $s_4 = 3$ к подмножеству веса 10. В 10-м столбце значение T впервые появилось во 2-й строке в результате прибавления $s_2 = 9$ к подмножеству веса 1. В 1-м столбце значение T впервые появилось в 1-й строке в результате прибавления $s_1 = 1$ к нулю. Вес подмножества $\{s_4, s_2, s_1\}$ равен $s_4 + s_2 + s_1 = 3 + 9 + 1 = 13$.

Получили, что $s_4 + s_2 + s_1 = s_5 + s_3$.

Учитывая свойства элементов таблицы, процесс её заполнения потребует не более $n(B + 1)$ шагов (если рассматривать традиционную машину Тьюринга, то это выражение следует возвести в куб или в четвёртую степень).

Где же экспонента, обещанная для *NP*-полных задач? Неужели мы доказали, что $\boldsymbol{P = NP}$?

Вспомним, что в определении класса ***NP*** речь идёт о полиноме от **ДЛИНЫ ЗАПИСИ** исходных данных.

Параметр n характеризует длину записи, но параметр B — это сумма самих числовых исходных. Длина записи числа имеет тот же порядок, что и логарифм этого числа, т.е. $B = 2^{\log B} \approx 2^{\|B\|}$. Вот и появилась экспонента.

Определение. Задача с числовыми параметрами называется **псевдополиномиальной**, если число шагов решающей её машины Тьюринга не превосходит полином от этих числовых параметров и длины записи остальных исходных данных, при этом значения числовых параметров не могут быть ограничены полиномом от длины записи остальных исходных данных.

Вывод. Нельзя отказываться от написания программы для многократного решения *NP*-полной или *NP*-трудной задачи, если она псевдополиномиальна и величина числовых исходных данных не слишком велика.

К такого рода задачам относятся, например, многие задачи составления расписаний, появление в которых слишком больших чисел маловероятно, поскольку задания должны быть выполнены в приемлемое время. Псевдополиномиальные алгоритмы будут работать «экспоненциально долго» только для тех исходных данных, которые содержат «экспоненциально большие» числа. Такие числа используются в криптографии, но в большинстве практически решаемых задачах отсутствуют.

Кроме того, если числовые параметры псевдополиномиальной задачи заданы в унарной системе счисления, то длина записи каждого такого параметра совпадает с его значением и задача становится полиномиальной. На этом иногда основываются «доказательства» того, что $\boldsymbol{P = NP}$, периодически появляющиеся в Интернете. Безусловно, они не являются доказательствами, так как рассматриваются разные задачи, точнее, разное представление исходных данных.

5.10. «ПОХОЖИЕ» ЗАДАЧИ С РАЗНОЙ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТЬЮ

<table>
<tr><th>P</th><th>NP-полные</th></tr>
<tr><td>КРАТЧАЙШИЙ ПУТЬ
МЕЖДУ ДВУМЯ ВЕРШИНАМИ</td><td>ДЛИННЕЙШИЙ ПУТЬ
МЕЖДУ ДВУМЯ ВЕРШИНАМИ</td></tr>
<tr><td colspan="2">Дано: Граф $G = (V, E)$, «длины» $l(e) \in Z_+$ $(e \in E)$,
«длины» $l(e) \in Z_+$ $(e \in E)$, выделенные вершины $s, f \in V$, число $B \in Z_+$.
Вопрос: Существует ли в G простой путь из s в f длины</td></tr>
<tr><td>$\leq B$</td><td>$\geq B$</td></tr>
<tr><td>РЁБЕРНОЕ ПОКРЫТИЕ</td><td>ВЕРШИННОЕ ПОКРЫТИЕ</td></tr>
<tr><td colspan="2">Дано: Граф $G = (V, E)$, число $K \in Z_+$.
Вопрос: Существует ли подмножество</td></tr>
<tr><td>$E' \subseteq E, \|E'\| \leq K$
$\forall v_{\in V} \exists e_{\in E'}$</td><td>$V' \subseteq V, \|V'\| \leq K$
$\forall e_{\in E} \exists v_{\in V'}$</td></tr>
<tr><td colspan="2">$v \in e$</td></tr>
<tr><td>ТРАНЗИТИВНАЯ
РЕДУКЦИЯ</td><td>МИНИМАЛЬНЫЙ
ЭКВИВАЛЕНТНЫЙ ОРГРАФ</td></tr>
<tr><td colspan="2">Дано: Орграф $G = (V, A)$, число $K \in Z_+$.
Вопрос: Существует ли подмножество</td></tr>
<tr><td>$A' \subseteq V \times V$</td><td>$A' \subseteq A$</td></tr>
<tr><td colspan="2">такое, что $\|A'\| \leq K$
и для всех $u, v \in V$ граф $G' = (V, A')$ содержит путь из u в v
тогда и только тогда граф G содержит такой путь.</td></tr>
<tr><td>РАСПИСАНИЕ ДЛЯ ПРЯМОГО
ДЕРЕВА ЗАДАНИЙ</td><td>РАСПИСАНИЕ ДЛЯ ОБРАТНОГО
ДЕРЕВА ЗАДАНИЙ</td></tr>
<tr><td colspan="2">Дано: Множество T заданий с единичной длительностью,
директивный срок $d(t) \in Z_+$ $(t \in T)$, число $m \in Z_+$
частичный порядок $<\cdot$ на T, относительно которого
каждое задание имеет не более одного непосредственно</td></tr>
<tr><td>следующего за ним</td><td>предшествующего ему</td></tr>
<tr><td colspan="2">Вопрос: Можно ли составить для множества T расписание
на m процессорах, на каждом из которых задания выполняются
согласно заданному частичному порядку, так что
все директивные сроки выполнены.</td></tr>
</table>

Упражнения.

Используя метод сужения, доказать *NP*-полноту следующих задач.

Замечание. При доказательстве *NP*-полноты задачи МНОЖЕСТВО ПРЕДСТАВИТЕЛЕЙ формально число шагов порождения множества S' можно было ограничить числовым параметром K, но сразу было учтено, что $||S'|| \leqslant ||S||$.

Ниже в упражнениях для некоторых числовых параметров явно указано ограничение на их значения через длину записи нечисловых параметров, а в некоторых случаях это не сделано.

1. РАСПИСАНИЕ ДЛЯ МУЛЬТИПРОЦЕССОРНОЙ СИСТЕМЫ.

Дано: конечное множество «заданий» A;

«длительности» $l(a) \in Z_+$ для всех $a \in A$;

число «процессоров» $m \in Z_+$;

«директивный срок» $D \in Z_+$.

Вопрос: существует ли разбиение $A = A_1 \cup \ldots \cup A_m$ множества A на m непересекающихся подмножеств такое, что для всех i $(1 \leqslant i \leqslant m)$ $\Sigma_{a \in A_i} l(a) \leqslant D$?

2. ОСТОВНОЕ ДЕРЕВО ОГРАНИЧЕННОЙ СТЕПЕНИ.

Дано: граф $G = (V, E)$ и $K \in Z_+$.

Вопрос: существует ли в G остовное дерево, в котором все вершины имеют степень не более K?

3. ИЗОМОРФИЗМ ПОДГРАФУ.

Дано: два графа $G = (V_1, E_1)$ и $H = (V_2, E_2)$.

Вопрос: содержит ли граф G подграф, изоморфный H?

То есть существуют ли такие подмножества $V \subset V_1$ и $E \subset E_1$ и биекция $f: V_2 \to V$, что $|V| = |V_2|$ и $\forall uv(u, v \in E_2 \leftrightarrow f(u), f(v) \in E)$?

4. САМЫЙ ДЛИННЫЙ ПУТЬ.

Дано: граф $G = (V, E)$ и $K \in \boldsymbol{Z}_+$, $K \leqslant |V|$.

Вопрос: имеется ли в G простой путь (т.е. путь, не проходящий дважды через одну вершину), содержащий не менее чем K рёбер?

5. НАИБОЛЬШИЙ ОБЩИЙ ПОДГРАФ.

Дано: два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$, $K \in \boldsymbol{Z}_+$.

Вопрос: существуют ли такие подмножества $E_1' \in E_1$ и $E_2' \in E_2$, что $|E_1'| = |E_2'| \geqslant K$ и подграфы $G_1' = (V_1, E_1')$ и $G_2' = (V_2, E_2')$ изоморфны?

6. РЮКЗАК.
Дано: конечное множество U;
«размеры» $s(u) \in \boldsymbol{Z}_+$;
«стоимости» $v(u) \in \boldsymbol{Z}_+$;
ограничение на «размеры» $B \in \boldsymbol{Z}_+$;
ограничение на «стоимости» $K \in \boldsymbol{Z}_+$.
Вопрос: существует ли подмножество $U' \subseteq U$, такое что

$$\sum_{u \in U'} s(u) \leq B \;\&\; \sum_{u \in U'} v(u) \geq K ?$$

7. РАЗБИЕНИЕ НА ГАМИЛЬТОНОВЫ ПОДГРАФЫ.
Дано: граф $G = (V, E)$ и $K \in \boldsymbol{Z}_+, K \leqslant |V|$.
Вопрос: можно ли множество вершин графа G разбить на k ($k \leqslant K$) непересекающихся подмножеств $V_1, \ldots, V_k$ так, чтобы для всех i ($1 \leqslant i \leqslant k$) каждый подграф, индуцированный множеством V_i, содержал Гамильтонов цикл?

8. МИНИМУМ СУММЫ КВАДРАТОВ.
Дано: конечное множество A;
«размеры» $s(a) \in \boldsymbol{Z}_+$ для всех $a \in A$;
$K \in \boldsymbol{Z}_+, J \in \boldsymbol{Z}_+$.
Вопрос: может ли множество A быть разбито на K непересекающихся подмножеств $A_1, \ldots, A_K$ так, что

$$\sum_{i=1}^{K} \left(\sum_{a \in A_i} s(a) \right)^2 \leq J ?$$

5.11. ЗАДАЧИ С ЧИСЛОВЫМИ ПАРАМЕТРАМИ И СИЛЬНАЯ NP-ПОЛНОТА

Пусть D_Z — формализованная запись задачи Z. Введём две целочисленные функции *Length*: $D_Z \to \boldsymbol{Z}_+$ и *Max*: $D_Z \to \boldsymbol{Z}_+$. Для каждой индивидуальной задачи I задачи Z эти функции характеризуют соответственно число символов, используемых в записи задачи I, и величину максимального числа в записи I. Теоретически эти две функции могут быть произвольными, так же как метод формализации.

Так, например, для задачи РАЗБИЕНИЕ они могут быть определены следующим образом:

$$Length(I) = |A| + \sum_{a \in A} \lceil \log_2 s(a) \rceil; \quad (1)$$

$$Length(I) = |A| + \max_{a \in A} \lceil \log_2 s(a) \rceil; \quad (2)$$

$$Length(I) = |A| \cdot \left\lceil \log_2 \sum_{a \in A} s(a) \right\rceil; \quad (3)$$

$$Max(I) = \max_{a \in A} s(a); \quad (4)$$

$$Max(I) = \sum_{a \in A} s(a); \quad (5)$$

$$Max(I) = \left\lceil \left(\sum_{a \in A} s(a) \right) / \|A\| \right\rceil. \quad (6)$$

Функция *Length* по формуле (1) определена как длина записи исходных данных. Формула (2) даёт гораздо меньшее значение для функции *Length*. Формула (3), наоборот, даёт гораздо большее значение для функции *Length*.

Функция *Max* по формуле (4) определена как максимальное значение числового параметра. Формула (5) даёт гораздо большее значение для функции *Max*. Формула (6), наоборот, даёт гораздо меньшее значение для функции *Max*, а именно некое «среднее значение» длины записи числовых параметров.

Определение. Функции *Length* и *Length'* называются **полиномиально эквивалентными**, если существуют такие полиномы p и p', что для всякой индивидуальной задачи $I \in D_Z$ выполнены соотношения:

$$Length(I) \leqslant p'(Length'(I));$$
$$Length'(I) \leqslant p(Length(I)).$$

Именно в связи с таким определением при доказательстве принадлежности задачи классу ***NP*** мы достаточно «вольно» обращались со словами «*длина записи множества*» и «*мощность множества*».

Определение. Пары функций (*Length*, *Max*) и (*Length'*, *Max'*) называются **полиномиально эквивалентными**, если существуют такие по-

линомы от двух переменных q и q', что для всякой индивидуальной задачи $I \in D_z$ выполнены соотношения:

$$Max(I) \leqslant q'(Length'(I), Max'(I));$$
$$Max'(I) \leqslant q(Length(I), Max(I)).$$

От функций *Length* и *Max* требуется также, чтобы они принадлежали классу ***FP***, т.е. двоичные записи их значений вычислялись не более чем за полином от длины записи их исходных данных.

Упражнения.

1. Доказать, что любая из девяти перечисленных выше пар функций (*Length*, *Max*) полиномиально эквивалента любой другой паре функций из перечисленных выше.

2. Доказать, что в любом из определений *Length* (1)–(3) и в определении (6) функции *Max* можно заменить $|A|$ на $||A||$ и наоборот. При этом получаются полиномиально эквивалентные функции.

Подходящая функция *Length* зависит от понятия «разумная схема кодирования задачи», которая уточняется при описании условия задачи. Вспомним, что, например, при описании условия в задаче РАЗБИЕНИЕ:

- можно задавать множество объектов и их «веса» и проверять возможность разбиения этого множества на два подмножества одинакового «веса»;
- можно задавать линейное однородное уравнение с целыми коэффициентами и проверять существование его решения в числах из {0, 1}.

Подходящая функция *Max* возникает, если в качестве исходных данных задачи выступают числа. Как правило, это целые положительные числа. Более сложные числа (целые, рациональные, алгебраические) рассматриваются как некоторые структуры, полученные из целых положительных чисел. Это обусловлено тем, что математические понятия алгоритма оперируют с конструктивными объектами.

Определение. Алгоритм решения задачи Z называется **псевдополиномиальным по времени** алгоритмом (или просто псевдополиномиальным алгоритмом), если число шагов его работы не превосходит полинома от двух аргументов *Length*(I) и *Max*(I).

По определению всякий полиномиальный по времени алгоритм является псевдополиномиальным по времени алгоритмом, так как число его шагов ограничено полиномом от одного аргумента *Length*(I), но не наоборот (смотри, например, задачу РАЗБИЕНИЕ).

Например, в задачах ВЕРШИННОЕ ПОКРЫТИЕ, НЕЗАВИСИМОЕ МНОЖЕСТВ, КЛИКА имеется числовой параметр. При доказательстве того, что задача принадлежит классу ***NP***, мы получали оценку числа шагов, в которой фигурировал этот параметр. Однако в этих задачах он не превосходит количество вершин в графе.

Определение. Задачу Z будем называть **задачей с числовыми параметрами**, если не существует такого полинома p, что для всех индивидуальных задач $I \in D_Z$ выполняется $Max(I) \leqslant p(Length(I))$.

Из этого определения следует утверждение.

Утверждение. Если $\boldsymbol{P \neq NP}$, то никакая *NP*-полная задача без числовых параметров не может быть решена псевдополиномиальным алгоритмом.

Среди шести основных *NP*-полных задач единственной задачей с числовыми параметрами является задача РАЗБИЕНИЕ.

Таким образом, если $\boldsymbol{P \neq NP}$, то *NP*-полные задачи, для которых имеется потенциальная возможность их решения псевдополиномиальным по времени алгоритмом, — это задачи с числовыми параметрами.

Для произвольной задачи распознавания Z и полинома с целыми коэффициентами p посредством Z_p обозначим её подзадачу, в которой для всех её индивидуальных задач I верно $Max(I) \leqslant p(Length(I))$.

По определению задачи с числовыми параметрами задача Z_p не является задачей с числовыми параметрами. Следовательно, если она разрешима псевдополиномиальным алгоритмом, то она разрешима полиномиальным алгоритмом.

Определение. Задача Z называется ***NP*-полной в сильном смысле**, если:

- Z принадлежит ***NP***;
- существует такой полином p с целыми коэффициентами, что Z_p является *NP*-полной.

Следствие. Если Z *NP*-полна и не является задачей с числовыми параметрами, то она *NP*-полна в сильном смысле.

Утверждение. Если $\boldsymbol{P \neq NP}$, то никакая *NP*-полная в сильном смысле задача не может быть решена псевдополиномиальным алгоритмом.

Из этого следует, что задача РАЗБИЕНИЕ не может быть *NP*-полной в сильном смысле, так как она может быть решена псевдополиномиальным алгоритмом.

Пример *NP*-полной в сильном смысле задачи.

Задача КОММИВОЯЖЁР (КМ) является *NP*-полной, поскольку она полиномиально эквивалентна задаче ГЦ (см. доказательство в 4.5). Кро-

ме того, она является задачей с числовыми параметрами, так как ни расстояния между городами $d(i, j)$, ни ограничение на длину маршрута B в ней не фиксированы (могут быть произвольными).

При доказательстве сведения к ней задачи ГЦ были получены только такие индивидуальные задачи, в которых расстояния $d(i, j)$ равны 1 или 2, а граница B совпадает с числом городов.

Пусть

$$Max(I) = \max\{B, \max_{i,j} d(i, j)\},$$

$$Length(I) = m + [\log_2 B] + \sum_{i,j} [\log_2 d(i, j)].$$

Тогда все индивидуальные задачи, получаемые в описанной сводимости (в частности, ГЦ), удовлетворяют ограничению $Max(I) \leqslant p(Length(I))$.

Следовательно, задача $КМ_{Length(I)}$, подзадачей которой является ГЦ, *NP*-полна. Из этого следует сильная *NP*-полнота задачи КМ.

В качестве упражнений были рассмотрены задачи РЮКЗАК и РАСПИСАНИЕ ДЛЯ МУЛЬТИПРОЦЕССОРНОЙ СИСТЕМЫ. Их *NP*-полнота доказывалась сужением этих задач до задачи РАЗБИЕНИЕ. Однако эти сужения оставляют открытым вопрос о возможности решения этих задач псевдополиномиальными алгоритмами.

Задачу РЮКЗАК можно решить за псевдополиномиальное время методом динамического программирования. Поэтому эта задача, хотя и *NP*-полна и имеет числовые параметры, не является *NP*-полной в сильном смысле.

Для задачи РАСПИСАНИЕ ДЛЯ МУЛЬТИПРОЦЕССОРНОЙ СИСТЕМЫ имеется доказательство её *NP*-полноты в сильном смысле.

СПИСОК ЛИТЕРАТУРЫ

1. *Косовская, Т.М.* О полиномиальных алгоритмах решения диофантовых систем линейных уравнений и сравнений / Т.М. Косовская, Н.К. Косовский // Материалы VIII Международного семинара «Дискретная математика и ее приложения». Ч. 1. — Москва : Изд-во МГУ, 2004.
2. *Гэри, М.* Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. — Москва : Мир, 1982.
3. *Схрейвер, А.* Теория линейного и целочисленного программирования / А. Схрейвер. — Москва : Мир, 1991.
4. *Окулов, С.М.* Программирование в алгоритмах / С.М. Окулов. — Москва : БИНОМ. Лаборатория знаний, 2007.
5. *Кнут, Д.* Искусство программирования на ЭВМ. Том 3. Сортировка и поиск / Д. Кнут. — Москва : ИД Вильямс, 2007.
6. *Емеличев, В.А.* Лекции по теории графов / В.А. Емеличев, О.И. Мельников. — Москва : Наука, 1990.
7. *Липский, В.* Комбинаторика для программистов / В. Липский. — Москва : Мир, 1988.
8. *Новиков, Ф.А.* Дискретная математика для программистов : учебник для вузов / Ф.А. Новиков. — Санкт-Петербург : Питер, 2005.
9. *Bron, C.* Algorithm 457: Finding all cliques of an undirected graph / C. Bron, J. Kerbosh // Communications of the ACM. — 1973. — № 16. — P. 575–577.
10. *Мальцев, А.И.* Алгоритмы и рекурсивные функцию / А.И. Мальцев. — Москва : Наука, 1986.
11. *Яблонский, С.В.* Введение в дискретную математику : учебное пособие / С.В. Яблонский. — Москва : Высшая школа, 2001.
12. *Корухова, Л.С.* Введение в алгоритмы : учебное пособие для студентов 1 курса / Л.С. Корухова, М.Р. Шура-Бура. — Москва : Изд-во МГУ, 2010.
13. *Babai, L.* Graph Isomorphism in Quasipolynomial Time / L. Babai. — URL: https://arxiv.org/abs/1512.03547 (дата обращения: 08.06.2023).

ПРОФОБРАЗОВАНИЕ

Издательство «Профобразование» – издательство учебной и профильной литературы для среднего профессионального образования. Издания включены в ПОП в качестве основной и дополнительной литературы (более 700 рекомендаций ФУМО СПО)

ПАРТНЕРСКИЕ ПРОЕКТЫ

БИБЛИОТЕКА ПРОФЕССИОНАЛА №1 – одна из крупнейших в России специализированных библиотек для учреждений СПО. В цифровой библиотеке доступны к подключению более 8 000 учебных изданий, журналы, онлайн-курсы, тесты, аудио- и видеоматериалы, а также учебники издательства «Просвещение», входящие в Федеральный перечень учебников.

ВКР СМАРТ – современное решение для проверок работ на объем заимствований и банк электронных портфолио обучающихся. Безопасный и надежный репозиторий данных, а также работ студентов и преподавателей.

Заинтересованы в сотрудничестве с нами?

8 800 511 14 70
Звонок бесплатный по всей России

office@profspo.ru
Ответим на все Ваши вопросы

Следите за новостями в социальных сетях

@profobrazovanie_official

@profobrexp

Рекомендуемые новинки

Купить книгу

8 800 555 22 35 (доб. 214)

books@iprmedia.ru

Учебное издание

Косовская Татьяна Матвеевна

АЛГОРИТМЫ И АНАЛИЗ
ИХ СЛОЖНОСТИ

Редактор *Н.Г. Шиндина*
Технический редактор *М.В. Половникова*
Корректор *Е.В. Савенкова*
Компьютерная верстка *В.В. Калинин*
Обложка *С.С. Сизиумова*, фотобанк «Лори»

По вопросам приобретения экземпляров издания:
ООО Компания «Ай Пи Ар Медиа»
8-800-555-22-35 (бесплатный звонок по России)
доб. 214, 208, 222
E-mail: izdat@iprmedia.ru, books@iprmedia.ru

Наши книги также представлены:
Читай-город (chitai-gorod.ru)
Ozon (ozon.ru)
Wildberries (wildberries.ru)
Яндекс Маркет (market.yandex.ru)

Подписано в печать 22.06.2023. Бумага офсетная.
Формат 60x90/16. Гарнитура «PT Serif».
Печать цифровая. Печ. л. 7,25.
Тираж 500 экз. (1-й з-д 1–35 экз.).

Общество с ограниченной ответственностью Компания «Ай Пи Ар Медиа»,
410012, г. Саратов, а/я 916

Акционерное Общество «Т 8 Издательские Технологии» (АО «Т 8»),
109316, г. Москва, Волгоградский пр-т, д. 42, корп. 5