Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А. А. Волосевич

ОСНОВЫ ТЕОРИИ АЛГОРИТМОВ

Учебно-методическое пособие
по курсу «Теория алгоритмов»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2007

УДК 681.3.06 (075.8)
ББК 32.973.26 – 018.2 я73
В 68

Р е ц е н з е н т
зав. кафедрой МО ЭВМ БГУ,
канд. техн. наук Л. Ф. Зимянин

Волосевич, А. А.

В 68 Основы теории алгоритмов : учеб.-метод. пособие по курсу «Теория алгоритмов» для студ. спец. I-31 03 04 «Информатика» всех форм обуч. / А. А. Волосевич. – Минск : БГУИР, 2007. – 54 с. : ил.

ISBN 978-985-488-228-4

Учебно-методическое пособие составлено в соответствии с рабочей программой курса «Теория алгоритмов». В него включены базовые определения и основные результаты классической теории алгоритмов, а также теории сложности вычислений. Описаны различные математические уточнения понятия «алгоритм», приведен численный анализ сложности некоторых алгоритмов.

Пособие может быть рекомендовано студентам и магистрантам технических специальностей для изучения математических основ теории алгоритмов и сложности вычислений.

УДК 681.3.06 (075.8)
ББК 32.973.26 – 018.2 я73

ISBN 978-985-488-228-4

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данном пособии рассмотрены базовые понятия двух математических дисциплин: классической теории алгоритмов и теории сложности вычислений. Теория алгоритмов зародилась в начале XX века, когда возникла необходимость в математически точном описании интуитивного понятия алгоритма. В тридцатые годы прошлого столетия эта задача была решена, когда практически одновременно и независимо друг от друга были опубликованы работы А. Чёрча, С. К. Клини, А. М. Тьюринга и Э. Л. Поста. Хотя в техническом отношении полученные уточнения понятия алгоритма были различными, вскоре была установлена взаимная моделируемость этих понятий. Произведённое уточнение дало немедленный эффект – вскоре была доказана алгоритмическая неразрешимость многих прикладных математических задач. В пособии рассматриваются некоторые из возможных подходов к уточнению понятия алгоритма. Кроме этого, формулируются центральные результаты теории алгоритмов в виде набора теорем и утверждений.

Существование алгоритма для решения некой массовой проблемы еще не означает практической разрешимости проблемы. Это может быть обусловлено высокой временной сложностью алгоритма. Рассмотрением различных подходов к определению сложности алгоритмов и массовых проблем занимается теория сложности вычислений. Пособие содержит описание различных классов сложности, а также несколько примеров числового анализа конкретных алгоритмов.

Так как пособие имеет небольшой объем, представленный в нем материал не претендует на теоретическую полноту. Он носит конспективный характер и может являться отправной точкой для рассмотрения более серьезных математических трудов по рассматриваемым дисциплинам. Основная цель пособия – представление базовых сведений, знание которых показывает высокую математическую культуру любого современного специалиста, работающего в информационной сфере.

1. НЕФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ АЛГОРИТМА

Уже на самых ранних этапах развития математики в ней стали возникать различные вычислительные процессы чисто механического характера. С их помощью искомые величины ряда задач вычислялись из исходных величин по определённым правилам и инструкциям. Со временем все такие процессы в математике получили название *алгоритмов*. Примерами алгоритмов могут служить процесс нахождения наибольшего общего делителя двух натуральных чисел (алгоритма Евклида), процесс сложения десятичных чисел «в столбик», процесс решения системы линейных уравнений методом последовательного исключения неизвестных и т. д.

Дадим более формальное определение рассмотренным понятиям.

Определение 1 (Массовая проблема, алгоритм). *Пусть задано некоторое непустое множество X, элементы которого будем называть исходными объектами, и непустое множество Y, элементы которого будем называть искомыми объектами. Будем говорить, что сформулирована массовая проблема (или массовая задача), если требуется для любого исходного объекта указать определённый искомый объект. Если существует универсальный способ, который позволяет это сделать, то будем говорить, что заданная массовая проблема имеет решение, а указанный способ будем назвать алгоритмом решения массовой проблемы.*

Рассмотрим примеры массовых проблем.

1. Пусть в качестве множества исходных объектов выступают массивы натуральных чисел, состоящие из n элементов. В качестве искомых объектов используется то же множество. Сформулируем *массовую проблему сортировки*: по любому массиву требуется указать массив, состоящий из тех же элементов, но упорядоченных по возрастанию. Данная проблема имеет решение – алгоритм сортировки. Заметим, что существуют различные алгоритмы сортировки.

2. Пусть исходные объекты – тройки рациональных чисел (a, b, c); множество искомых объектов – {«да», «нет»}. Массовая проблема формулируется следующим образом: по тройке чисел (a, b, c) требуется выдавать ответ «да», если уравнение $ax^2 + bx + c = 0$ имеет действительные корни, и ответ «нет» в противном случае. Очевидно, что данная проблема также имеет решение.

3. Пусть множество исходных объектов – это множество программ, записанных на языке Pascal; множество искомых объектов – {«да», «нет»}. Массовая проблема: по тексту программы определить, «зацикливается» она (ответ «да») или нет (ответ «нет»). Существование или отсутствие алгоритма решения этой массовой проблемы не очевидно.

Приведённое выше определение алгоритма не является математически строгим. Однако все алгоритмы обладают некими общими чертами, или свойствами, ясно обозначающимися из предыдущих примеров и признающихся характерными для понятия алгоритма:

1. Алгоритм работает с конструктивными объектами. Под *конструктивным объектом* понимается такой объект, который может быть описан при помощи конечного слова в некотором алфавите. Примерами конструктивных объектов могут служить рациональное число, массив рациональных чисел, слово. Входные данные алгоритма, промежуточные величины и конечный результат алгоритма всегда являются такими конструктивными объектами.

2. Алгоритм – это дискретный процесс. Процесс построения результатов алгоритма идёт в дискретном времени. В начальный момент задаётся исходная конечная система объектов, а в каждый последующий момент времени (на каждом *шаге алгоритма*) новая система объектов получается по определённому закону из системы объектов, имевшихся на предыдущем шаге.

3. Алгоритм детерминирован. Система объектов, получаемых на любом (не начальном) шаге алгоритма, однозначно определяется объектами, полученными на предшествующих шагах.

4. Шаг алгоритма элементарен. Закон получения последующей системы объектов из предшествующей системы должен быть простым и локальным.

5. Алгоритм обладает свойством массовости. Начальная система объектов алгоритма может выбираться из некоторого (потенциально бесконечного) множества.

Понятие алгоритма, определяемое этими пятью свойствами, также не является строгим: в формулировках свойств имеются слова «способ», «объект», «простой», «локальный», точный математический смысл которых не установлен. В дальнейшем это нестрогое понятие алгоритма будет называться *интуитивным понятием алгоритма*.

Интуитивное понятие алгоритма хотя и не является строгим, но оно настолько ясно, что к началу XX века практически не было серьёзных случаев, когда математики разошлись бы во мнениях относительно того, является ли алгоритмом тот или иной конкретно заданный процесс. Однако в XX веке положение существенно изменилось. На первый план выдвинулись такие массовые проблемы, существование алгоритма решения которых было сомнительным. Действительно, одно дело – доказать существование алгоритма, другое – доказать отсутствие. Первое можно сделать путем фактического описания процесса, решающего задачу. В этом случае достаточно и интуитивного понятия алгоритма, чтобы удостовериться в том, что описанный процесс есть алгоритм. Доказать отсутствие алгоритма таким путём невозможно. Для этого надо математически точно знать, *что* такое алгоритм. В 1920-х годах задача точного определения понятия алгоритма стала одной из центральных математических проблем. Она была решена в 1936 году, когда практически одновременно и независимо друг от друга были опубликованы работы А. Чёрча, С. К. Клини, А. М. Тьюринга и Э. Л. Поста. Хотя в техническом отношении полученные уточнения понятия алгоритма были различными, вскоре была установлена взаимная моделируемость этих понятий. Произведённое уточнение дало немедленный эффект – вскоре была доказана алгоритмическая неразрешимость многих прикладных математических задач.

2. АРИФМЕТИЧЕСКИЕ И ИНТУИТИВНО ВЫЧИСЛИМЫЕ ФУНКЦИИ

Первым шагом на пути к математическому уточнению интуитивного понятия алгоритма является сведение решения массовой проблемы к процессу вычисления значений некоторой функции.

Сделаем следующее замечание. Начиная этого момента, под множеством натуральных чисел будем понимать целые неотрицательные числа. Таким образом, $\mathbf{N} = 0, 1, ...$

Определение 2 (Арифметическая функция). *Функцию будем называть арифметической, если ее аргументы и значение принадлежат множеству натуральных чисел. Таким образом, арифметические функции – это функции вида* $f: \mathbf{N}^n \to \mathbf{N}$.

Пусть имеется некая массовая проблема с множеством исходных объектов X и множеством искомых объектов Y. Для существования решения массовой проблемы необходимо, чтобы элементы множеств X и Y были конструктивными объектами. Следовательно, элементы этих множеств можно пронумеровать натуральными числами[1]. Пусть $x \in X$ – некий исходный объект, обозначим через $n(x)$ его номер. Если в массовой проблеме по исходному объекту x требуется получить искомый объект $y \in Y$ с номером $n(y)$, то определим арифметическую функцию $f: \mathbf{N}^n \to \mathbf{N}$ такую, что $f(n(x)) = n(y)$.

В качестве примера построения арифметических функций по массовым проблемам рассмотрим массовые проблемы из предыдущего раздела.

1. Если дан массив $[x_1, x_2, \ldots, x_n]$ натуральных чисел, то ему в соответствие можно поставить натуральное число $2^{x_1} \cdot 3^{x_2} \cdot \ldots \cdot p(n)^{x_n}$, где $p(n)$ – n-е простое число. Рассмотрим, например, массив длиной 5:

$$[x_1, x_2, x_3, x_4, x_5] \mapsto 2^{x_1}3^{x_2}5^{x_3}7^{x_4}11^{x_5}.$$

Эта нумерация определяет арифметическую функцию f (так, $f(73\,500) = f(2^2 3^1 5^3 7^2 11^0) = 2^0 3^1 5^2 7^2 11^3 = 4891425$).

2. Любое рациональное число имеет некоторый натуральный номер. Нумерация множества искомых объектов проблемы тривиальна:

$$\{\text{«да»}, \text{«нет»}\} \mapsto \{1, 0\}.$$

Для данной массовой проблемы можно построить арифметическую функцию одного аргумента, воспользовавшись приемом из предыдущего примера, а можно рассмотреть функцию трех аргументов (три номера элементов исходной тройки (a, b, c)).

3. Нумерация текстов программ может быть осуществлена следующим образом: любую программу можно воспринимать как запись числа в 256-ричной системе счисления (если для записи программы использовались символы таблицы ASCII).

---

[1] Процесс нумерации конструктивных объектов можно также рассматривать в качестве массовой проблемы.

Переход от массовой проблемы к арифметической функции позволяет свести вопрос о существовании решения массовой проблемы к вопросу существования эффективного способа вычислений значений арифметической функции по ее аргументу (аргументам).

Выделим из всех арифметических функций следующее подмножество.

Определение 3 (Вычислимая функция). *Арифметическую функцию* $f: \mathbf{N}^n \to \mathbf{N}$ *назовем вычислимой, если существует алгоритм, позволяющий для любого набора значений ее аргументов вычислить значение функции (или указать, что функция на данном наборе не определена)*.

Так как в определении вычислимой функции используется интуитивное понятие алгоритма, то часто вместо термина «вычислимая функция» используется термин «интуитивно вычислимая функция».

Таким образом, массовая проблема имеет решение, если арифметическая функция, соответствующая этой проблеме, является интуитивно вычислимой. Вместо уточнения понятия алгоритма можно рассматривать уточнение понятия «вычислимая функция». Обычно при этом действуют по следующей схеме:

1. Вводят точно определенный класс функций.

2. Убеждаются, что все функции из этого класса являются вычислимыми.

3. Принимают гипотезу (тезис) о том, что класс вычислимых функций совпадает с введенным классом функций.

3. МАШИНЫ ТЬЮРИНГА

Машины Тьюринга были предложены математиком Аланом Тьюрингом в 1936 году для уточнения интуитивного понятия алгоритма.

Машина Тьюринга состоит из следующих компонентов.

1. Внешний алфавит $A = \{a_0, a_1, a_2, \ldots, a_n\}$. Внешний алфавит состоит из конечного множества символов. Одна буква алфавита – a_0 – является особой. Это пустая буква (пробел). В дальнейшем такую пустую букву будем обозначать как λ.

2. Лента. Лента потенциально бесконечна в обе стороны и разделена на ячейки. Каждая ячейка содержит один из символов алфавита A, но только конечное множество ячеек не заполнено пустой буквой.

3. Управляющая головка. Головка способна воспринимать символ в ячейке ленты и изменять его, а также передвигаться вправо и влево. При этом в конкретный момент времени воспринимается только одна определённая ячейка, и за один такт работы машины Тьюринга головка перемещается либо на одну ячейку вправо, либо на одну ячейку влево, либо остаётся на месте.

4. Внутренний алфавит $Q = \{q_0, q_1, q_2, \ldots, q_m\}$. Внутренний алфавит служит для представления всех возможных внутренних состояний машины Тьюринга. В любой конкретный момент времени машина Тьюринга находится в одном из этих состояний. После одного такта работы состояние машины может измениться. Имеются два особых состояния машины Тьюринга: q_0 – заключительное (пассивное) состояние, в котором машина прекращает работу; q_1 – началь-

ное состояние, из которого стартует машина.

5. Программа. Программа – это таблица, которая состоит из $n + 1$ столбцов и m строк. В ячейки таблицы помещается команда вида $a_j t q_l$.

Таблица 1

Запись программы для машины Тьюринга

	a_0	a_1	…	a_i	…	a_n
q_1						
…						
q_k				$a_j t q_l$		
…						
q_m						

Если команда $a_j t q_l$ находится в строке q_k, столбце a_i, то это означает следующее: если машина Тьюринга находится в состоянии q_k, а управляющая головка при этом обозревает символ a_i, то

1) головка записывает в обозреваемую ячейку символ a_j (возможно, $a_j = a_i$);

2) головка должна переместиться в направлении $t \in \{L, S, R\}$, где L означает «влево», R – «вправо», а S – «оставаться на месте»;

3) машина должна перейти в состояние q_l.

Некоторые ячейки таблицы могут быть пустыми, то есть команд в программе не более чем $m(n + 1)$.

Заметим, что программа полностью определяет работу машины Тьюринга. Более того, если известна программа, можно восстановить машину Тьюринга, то есть внешний алфавит и множество внутренних состояний.

Точное определение машины Тьюринга выглядит следующим образом.

Определение 4 (Машина Тьюринга). *Будем называть машиной Тьюринга систему вида* $(A, a_0, Q, q_0, q_1, Sh, \tau)$, *где*

1. A – *конечное множество, внешний алфавит;*
2. $a_0 \in A$ – *пустая буква алфавита;*
3. Q – *конечное множество, внутренние состояния;*
4. $q_0 \in Q$ – *заключительное состояние;*
5. $q_1 \in Q$ – *начальное состояние;*
6. $Sh = \{L, S, R\}$ – *множество возможных сдвигов управляющей головки;*
7. τ – *программа, отображение вида*

$$\tau : A \times (Q \setminus \{q_0\}) \ni (a_i, q_k) \mapsto (a_j, t, q_l) \in A \times Sh \times Q.$$

Отметим, что машина Тьюринга представляет собой точно определённый математический объект.

Дальнейшей целью раздела будет описание класса арифметических функций, вычислимых на машинах Тьюринга.

Для представления натуральных чисел на ленте машины Тьюринга будем использовать «единичную» систему счисления. Натуральные числа в данной системе счисления записываются следующим образом: 0 = |, 1 = ||, 2 = ||| и т. п.

Рассмотрим машины Тьюринга с внешним алфавитом $A = \{\lambda, |\}$. Опишем процесс вычисления на машине Тьюринга значений некоторой арифметической функции $f(x_1, x_2, \ldots, x_n)$. Пусть дан набор значений аргументов $(a_1, a_2, \ldots, a_n)$ функции f. Запишем значения аргументов в «единичной» системе счисления на ленте машины Тьюринга, разделив их пробелом:

$$\underbrace{||\ldots|}_{a_1}\lambda\underbrace{||\ldots|}_{a_2}\lambda\ldots$$

Поместим управляющую головку под самый левый символ |. Запустим машину. Возможны следующие ситуации:

1. Машина Тьюринга через некоторое время останавливается, при этом на ленте машины Тьюринга записано некое число m в «единичной» системе счисления (и больше ничего).

2. Машина «зацикливается», то есть никогда не останавливается.

3. Машина Тьюринга останавливается, однако на ленте записано более чем одно число в единичной системе счисления.

Определение 5 (Функция, вычислимая по Тьюрингу). *Будем говорить, что машина Тьюринга вычисляет арифметическую функцию $f(x_1, x_2, \ldots, x_n)$, если для любого набора $(a_1, a_2, \ldots, a_n)$ значений аргументов функции машина либо получает в качестве ответа значение m, причём $m = f(a_1, a_2, \ldots, a_n)$, либо не останавливается, если функция f на наборе $(a_1, a_2, \ldots, a_n)$ не определена. Арифметическую функцию будем называть вычислимой по Тьюрингу, если существует машина Тьюринга, которая ее вычисляет.*

Множество всех вычислимых по Тьюрингу функций обозначим через T.

По определению, для установления того, является ли некая арифметическая функция вычислимой по Тьюрингу, достаточно предъявить машину, которая вычисляет эту функцию. Рассмотрим несколько примеров вычислимых по Тьюрингу функций.

1. $f_1(x) \equiv 0$. Покажем, что $f_1(x) \in T$. Построим машину, которая её вычисляет. Эта машина должна работать следующим образом: стереть на ленте все символы |, затем написать один символ | и остановиться. Вот программа, которая выполняет эти действия:

	λ	|
q_1	$\|Sq_0$	λRq_1

2. $f_2(x, y) = x + y$. Программа для вычисления данной функции может работать следующим образом: стереть два первых символа |, затем переместить головку вправо до пустой ячейки, записать туда символ | и остановиться.

	λ	|
q_1	–	λRq_2
q_2	λSq_0	λRq_3
q_3	$\|Sq_0$	$\|Rq_3$

Команда в ячейке λq_2 предназначена для исходного слова вида $|\lambda||$ (когда первый аргумент функции равен 0). Одна ячейка в таблице не используется.

В приведенных примерах можно было бы добавить в программы (таблицы) произвольное количество пустых строк. Значит, если существует одна машина Тьюринга для вычисления некоторой функции, то существует бесконечное множество машин для вычисления данной функции.

В качестве упражнения предлагается построить машины Тьюринга для вычисления следующих функций:

1. $f(x) = x + 3$,

2. $f(x) = 2x$,

3. $f(x) = 3x + 2$,

4. $f(x, y) = \begin{cases} 3x, & \text{если } y \geq 2; \\ x + 4, & \text{если } y < 2, \end{cases}$

5. $f(x, y) = \max(x, y)$.

Для усиления вычислительных свойств машины Тьюринга (расширения множества T) было предложено несколько модификаций машины. Рассмотрим некоторые из них.

1. Машины Тьюринга с другим внешним алфавитом. В качестве внешнего алфавита предлагалось рассматривать множества $A_1 = \{\lambda, 0, 1\}$ (для записи чисел в двоичной системе счисления), $A_2 = \{\lambda, 0, 1, ..., 9\}$ (для записи чисел в десятичной системе).

2. k-ленточные машины Тьюринга. Для записи исходных данных и проведения промежуточных расчетов предлагалось использовать несколько лент. Соответственно в ячейки таблицы-программы помещается не одна, а k команд (каждая команда для каждой ленты).

3. Одноленточная машина Тьюринга с несколькими головками. На ленте оперирует несколько головок, каждая из которых действует независимо от других (при этом нужно определить, что делать в случае, если две разные головки обозревают одну и ту же ячейку и хотят сделать в ней запись).

Было доказано, что любая из модификаций машины Тьюринга моделируется простейшей машиной, рассмотренной в начале данного параграфа. Таким образом, множество функций T является достаточно широким. В данный момент времени не найдено ни одной интуитивно вычислимой функции, для которой нельзя было бы составить вычисляющую машину Тьюринга. Это позволяет выдвинуть следующий тезис:

Тезис Тьюринга: *любая интуитивно вычислимая функция является вычислимой по Тьюрингу*.

Обратите внимание, что данный тезис нельзя доказать, так как он связывает точное понятие вычислимости по Тьюрингу с неточным понятием интуитивно вычислимой функции.

4. МАШИНЫ ШЁНФИЛДА

Рассмотрим ещё один способ уточнения интуитивного понятия алгоритма. Это способ принадлежит Джозефу Шёнфилду. Как и Тьюринг, Шёнфилд предложил рассматривать некие вычислительные машины, называемые машинами Шёнфилда. Отличительной особенностью машин Шёнфилда является то, что программирование на них напоминает написание программы на языке ассемблер. Кроме этого, использование машин Шёнфилда позволяет значительно упростить доказательства классических теорем теории алгоритмов.

Машина Шёнфилда имеет бесконечное число *регистров*, пронумерованных натуральными числами, начиная с 0. Каждый из этих регистров может содержать любое натуральное число. Для программы необходимо лишь конечное число регистров, которое нетрудно заранее определить по программе. Кроме этого, у машины есть специальная память для программы, а также *счётчик команд*, всегда содержащий некоторое натуральное число.

Программа для машины Шёнфилда состоит из конечного списка команд, последовательно пронумерованных натуральными числами начиная с 0.

Существует два типа команд:

- `INC I` – увеличивает содержимое `I`-го регистра на 1 и увеличивает содержимое счётчика команд на 1. После этого машина переходит к следующему шагу;
- `DEC I, n` – если содержимое `I`-го регистра больше нуля, то уменьшает содержимое `I`-го регистра на 1 и заносит `n` в счётчик команд. Если содержимое `I`-го регистра равно 0, то увеличивает содержимое счётчика команд на 1.

Перед запуском в машину вводится программа, в регистры заносятся начальные данные, а в счётчик команд заносится значение 0. После этого шаг за шагом осуществляется работа машины.

Шаг машины состоит в исполнении команды, номер которой указан в счётчике команд. Если такой номер команды в программе отсутствует, то машина останавливается.

Последовательность вычисления некой функции $f(x_1, \ldots, x_n)$ на машине Шёнфилда состоит в следующем: i-е значение аргумента функции заносим в `I`-й регистр и запускаем машину; если машина останавливается, то значение функции – это значение в регистре 0; в противном случае считаем, что значение функции не определено на данном наборе аргументов.

Определение 6 (Функция, вычислимая по Шёнфилду). *Арифметическая функция называется вычислимой по Шёнфилду, если существует машина Шёнфилда, которая её вычисляет.*

Множество всех функций, вычислимых по Шёнфилду, обозначим *Sh*. О связи вычислимых функций и множества *Sh* выдвигается следующий тезис:

Тезис Шёнфилда: *любая интуитивно вычислимая функция является вычислимой по Шёнфилду.*

На практике для написания программ машины Шёнфилда удобным оказывается использование макросов. Макрос – это некая совокупность команд машины, имеющая короткое обозначение. Достаточно очевидной является следующая теорема.

Теорема 1 (Об удалении макросов). *Любая программа с макросами эквивалентна некоторой программе, не содержащей макросов.*

При удалении макросов особого внимания требуют команды вида `DEC I, n`. Данные команды необходимо изменить так, чтобы переход осуществлялся на строку с соответствующим номером в программе без макросов. Вероятно, при удалении макросов потребуется замена номеров регистров, используемых в макросах, новыми номерами.

Опишем несколько полезных макросов.

Макрос `GOTO n`. Данный макрос осуществляет переход на `n`-ю строку программы. Состоит макрос из следующей пары команд:

```
0: INC 0
1: DEC 0, n
```

Макрос `ZERO I`. Это макрос, который обнуляет содержимое `I`-го регистра. Состоит такой макрос из одной команды – `DEC I, 0`.

Макрос `[i]` $\rightarrow$ `[j],(k)`, где $i, j \neq k$. Этот макрос копирует содержимое i-го регистра в j-й регистр, используя k-й регистр в качестве вспомогательного. Содержимое i-го регистра при $i \neq j$ не изменяется. Программа для макроса `[i]` $\rightarrow$ `[j],(k)` при $i \neq j$ выглядит так:

```
0: ZERO j
1: ZERO k        обнулили j-й и k-й регистры
2: GOTO 5        переход на 5-ю команду
3: INC j
4: INC k
5: DEC i, 3      если в i-м регистре не 0, увеличим i-й и k-й регистры
6: GOTO 8
7: INC i
8: DEC k, 7      копирование из k-го регистра в i-й
```

Если `i` = `j`, то можно взять любую программу, которая ничего не меняет, например

```
0: GOTO 1
```

Макрос `F([i`$_1$`],…,[i`$_k$`])` $\rightarrow$ `[j]` вычисляет значение функции F, вычислимой по Шёнфилду с помощью программы P, от содержимого регистров `[i`$_1$`]`, …,`[i`$_k$`]`, записывает это значение в j-й регистр и при этом не изменяет значения остальных регистров.

Приведём примеры некоторых машин Шенфилда. Следующая машина определяет функцию сложения $\text{sum}(x, y) = x + y$.

```
0: [2] → [0]
```

```
1: GOTO 3
2: INC 0
3: DEC 1, 2
```

Для функции умножения mult(x, y) = x * y можно написать такую машину:

```
0: ZERO 0
1: GOTO 3
2: sum([0], [2]) → [0]
3: DEC 1, 2
```

Процесс программирования машин Шенфилда существенно легче, чем аналогичный процесс для машин Тьюринга. Объясняется это тем, что машины Шенфилда представляют собой *вычислительное устройство с произвольным доступом к данным*. В любой момент времени можно работать с любым регистром машины. Машины Тьюринга – это пример *устройства с последовательным доступом к данным*, так как, чтобы работать с некоторой ячейкой ленты машины Тьюринга, нужно последовательно перейти к этой ячейке.

5. ЧАСТИЧНО ВЫЧИСЛИМЫЕ ФУНКЦИИ

Частично вычислимые функции[1] – это формализация понятия вычислимости, предложенная в 1935 году А. Чёрчем и С. Клини. В основе данной формализации лежит идея представления любой функции через комбинации неких простейших функций.

Будем называть *простейшими* следующие арифметические функции:

1. $0(x) = 0$ – константа ноль;
2. $s(x) = x + 1$ – функция следования;
3. $I_m(x_1, x_2, \ldots, x_n) = x_m$ $(1 \le m \le n)$ – функции выбора аргумента (их n).

Заметим, что любая простейшая функция является всюду определённой и вычислимой (как в интуитивном смысле, так и по Тьюрингу и Шёнфилду).

Рассмотрим следующие операции над функциями.

1. Операция суперпозиции

Рассмотрим арифметические функции $f_1(x_1, \ldots, x_n)$, ..., $f_m(x_1, \ldots, x_n)$ и $\varphi(x_1, \ldots, x_m)$. Будем говорить, что функция $\psi(x_1, \ldots, x_n)$ получена из функций $f_1, \ldots, f_m$ и φ применением операции суперпозиции, если $\psi(x_1, \ldots, x_n) = \varphi(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$.

Очевидно, что в случае, когда функции $f_1, \ldots, f_m$ и φ являются вычислимыми, то функция ψ также является вычислимой. Функция ψ является определённой на наборе $(a_1, a_2, \ldots, a_n)$, если на данном наборе определена каждая функция f_i, и функция φ является определённой на наборе $(f_1, f_2, \ldots, f_m)$.

Приведём несколько примеров функций, полученных при помощи операции суперпозиции.

[1] Исходно такие функции назывались частично рекурсивными функциями, но в 1995 году известный американский математик Роберт Соар предложил изменить терминологию, и большинство математиков мира согласилось с его предложением.

1. $0(x_1, x_2, ..., x_n) = 0(I_m(x_1, x_2, ..., x_n)) = 0$;
2. $c_1(x) = s(0(x)) = 1$, $c_2(x) = s(c_1(x)) = 2$;
3. $c_m(x_1, x_2, ..., x_n) = m$ $(m \geq 1, m \in \mathrm{N})$;
4. $s_2(x) = s(s(x)) = x + 2$, $s_{m,k}(x_1, x_2, ..., x_n) = x_m + k$.

2. Операция примитивной рекурсии

Опишем простейший случай использования операции примитивной рекурсии. Пусть дано некое натуральное число a и арифметическая функция $h(x, y)$. Определим арифметическую функцию $f(x)$ по следующей схеме:

$$\begin{cases} f(0) = a, \\ f(x+1) = h(x, f(x)). \end{cases}$$

В этом случае говорят, что функция $f(x)$ строится из константы a и функции $h(x, y)$ при помощи оператора примитивной рекурсии.

Легко видеть, что в случае, когда функция $h(x, y)$ является вычислимой, функция $f(x)$ также является вычислимой. Значения функции $f(x)$ можно получить последовательно: $f(0) = a$, $f(1) = h(0, f(0))$, $f(2) = h(1, f(1))$ и т. д.

Рассмотрим несколько примеров функции, которые могут быть получены с использованием операции примитивной рекурсии.

1. $\mathrm{sgn}(x) = \begin{cases} 0, & x = 0, \\ 1, & x \neq 0. \end{cases}$

Зададим данную функцию так:

$$\begin{cases} \mathrm{sgn}(0) = 0, \\ \mathrm{sgn}(x+1) = c_1(x, \mathrm{sgn}(x)). \end{cases}$$

2. $\overline{\mathrm{sgn}}(x) = \begin{cases} 1, & x = 0, \\ 0, & x \neq 0. \end{cases}$

Данную функцию можно задать так:

$$\begin{cases} \overline{\mathrm{sgn}}(0) = 1, \\ \overline{\mathrm{sgn}}(x+1) = 0(x, \overline{\mathrm{sgn}}(x)). \end{cases}$$

Рассмотрим теперь более общий случай операции примитивной рекурсии. Пусть даны арифметические функции $g(x_1, ..., x_n)$ и $h(x_1, ..., x_n, y, z)$. Определим арифметическую функцию $f(x_1, ..., x_n, y)$ по следующей схеме:

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

В этом случае говорят, что функция $f(x_1, ..., x_n, y)$ строится из функций $g(x_1, ..., x_n)$ и $h(x_1, ..., x_n, y, z)$ при помощи оператора примитивной рекурсии.

Рассмотрим примеры.

1. Пусть дана следующая схема:

$$\begin{cases} f(x,0) = I_1(x), \\ f(x,y+1) = s_{3,1}(x,y,f(x,y)). \end{cases}$$

Очевидно, что данная схема определяет функцию $f(x, y)$ как $x + y$. Далее для этой функции будем использовать обозначение $\mathrm{sum}(x, y) = x + y$.

2. Рассмотрим схему

$$\begin{cases} f(x,0) = 0, \\ f(x,y+1) = \mathrm{sum}(x,y,f(x,y)). \end{cases}$$

Данная схема определяет функцию $f(x, y)$ как $x * y$.

Две рассмотренные операции над простейшими функциями позволяют описать достаточно большое подмножество арифметических функций.

Определение 7 (Примитивно-рекурсивная функция). *Арифметическая функция называется примитивно-рекурсивной, если она может быть получена из простейших за конечное число применений операций суперпозиции и примитивной рекурсии.*

Множество всех примитивно-рекурсивных функций обозначим как *Pr*. Отметим важное свойство примитивно-рекурсивных функций – они являются всюду определёнными.

В качестве упражнения покажите, что следующие функции являются примитивно-рекурсивными.

1. $power(x, y) = x^y$;

2. $x - 1 = \begin{cases} x - 1, \text{ если } x > 0, \\ 0, \quad \text{если } x = 0; \end{cases}$

3. $x - \mathrm{y} = \begin{cases} x - 1, \text{если } x \geq y, \\ 0, \quad \text{если } x = 0; \end{cases}$

4. $|x - y|$.

Класс примитивно-рекурсивных функций является достаточно широким. Однако существуют вычислимые функции, которые не принадлежат этому классу. Рассмотрим, например, функцию Аккермана:

$$\begin{cases} A(0,y) = y+1, \\ A(x,0) = A(x-1,1), \\ A(x,y) = A(x-1,A(x,y-1)). \end{cases}$$

Данная функция задана при помощи рекурсии (не примитивной рекурсии!) и является всюду определённой и интуитивно вычислимой (имеется способ нахождения значений этой функции). Функция Аккермана не является примитив-

но-рекурсивной. Доказательство этого факта опирается на следующую идею: примитивно-рекурсивные функции не могут быстро расти, а функция Аккермана растёт очень быстро.

Для построения развитой теории частично вычислимых функций рассматривается ещё одна операция над функциями.

3. Операция минимизации

Будем говорить, что функция $f(x_1, ..., x_n)$ получается из функции $g(x_1, ...,x_n, y)$ при помощи операции минимизации (μ-оператора), и обозначать

$$f(x_1, ..., x_n) = \mu y[g(x_1, ..., x_n,y) = 0],$$

если выполнено условие: $f(x_1, ..., x_n)$ определена и равна y тогда и только тогда, когда $g(x_1, ...,x_n, 0)$, ..., $g(x_1, ...,x_n, y\text{-}1)$ определены и не равны 0, а $g(x_1, ...,x_n, y) = 0$. Заметим, что процесс вычисления функции $f(x_1, ..., x_n)$ сводится к последовательному перебору значений y.

Пример. Пусть $g(x, y, z) = |(y + z) – x|$ и пусть $f(x, y) = \mu z[g(x,y,z) = 0]$. Легко убедиться, что $f(x, y) = x – y$. Заметим, что эта функция не является всюду определённой.

Определение 8 (Частично вычислимая функция). *Арифметическая функция называется частично вычислимой, если она может быть получена из простейших за конечное число применений операций суперпозиции, примитивной рекурсии и минимизации.*

Множество всех частично вычислимых функций обозначим Pc.

Определение 9 (Общерекурсивная функция). *Арифметическая функция называется общерекурсивной, если она является частично вычислимой и всюду определённой.*

Множество всех общерекурсивных функций обозначим Cv .

Непосредственно из приведённых определений следует, что $Pr \subseteq Cv \subseteq Pc$. Нетрудно установить, что данные включения являются строгими: $Pr \subset Cv \subset Pc$. Таким образом, существуют общерекурсивные функции, для получения которых *необходимо* использовать операцию минимизации (например функция Аккермана).

Как и в случае машин Тьюринга и Шёнфилда, на данный момент времени не найдено ни одной интуитивно вычислимой функции, которая не принадлежала бы множеству Pc. Это позволяет выдвинуть следующий тезис:

Тезис Чёрча: *любая интуитивно вычислимая функция является частично вычислимой.*

6. КОДИРОВАНИЕ АЛГОРИТМОВ

В теории алгоритмов важное значение имеет свойство алгоритма быть представленным неким натуральным числом. При этом исчезает принципиаль-

ная разница между командами и данными, что позволяет рассматривать алгоритмы как данные для других алгоритмов.

Существует несколько способов кодирования алгоритмов. Опишем один из них. Вначале рассмотрим построение кодов для последовательностей натуральных чисел $x_1, x_2, ..., x_n$.

Определение 10 (Код последовательности). *Пусть дана некая конечная последовательность натуральных чисел $x_1, x_2, ..., x_n$. Кодом этой последовательности, обозначаемым далее как $\langle x_1, x_2, ..., x_n \rangle$, назовём число $p_1^{x_1+1} p_2^{x_2+1} ... p_n^{x_n+1}$, где p_i – i-е простое число. Таким образом*

$$\langle x_1, x_2, ..., x_n \rangle = p_1^{x_1+1} p_2^{x_2+1} ... p_n^{x_n+1}.$$

Отметим следующие особенности данного кодирования последовательностей. Не все натуральные числа являются кодами последовательностей (например таковым не будет число $2^4 3^0 5^2$). Однако разным последовательностям натуральных чисел всегда будут соответствовать разные коды. Функции, которые позволяют узнать, является ли число кодом некоторой последовательности, а также позволяют закодировать последовательность чисел, узнать по коду составляющие последовательность числа, являются частично вычислимыми.

Пусть дан алгоритм. Этому алгоритму соответствует некая интуитивно вычислимая функция, а ей (согласно тезису Шёнфилда) соответствует некая (в общем случае не единственная) машина Шёнфилда.

Закодируем машины Шёнфилда. Кодом команды `INC I` будем считать число $\langle 0, I \rangle$, кодом команды `DEC I, n` – число $\langle 1, I, n \rangle$. Кодом программы из n команд будем считать следующее число:

$$e = \langle \text{ код 1-й команды, код 2-й команды,..., код } n\text{-й команды } \rangle.$$

Таким образом, получаем отображение множества всех машин Шёнфилда во множество натуральных чисел.

В теории алгоритмов важной является теорема, утверждающая эквивалентность математических уточнений понятия алгоритма.

Теорема 2. *Множества T, Pc и Sh совпадают.*

Теорема позволяет при решении теоретических задач выбирать то из уточнений понятия алгоритма, которое является наиболее выгодным в конкретном случае. Доказательство данной теоремы является достаточно технически сложным. Опишем идею данного доказательства для случая $Pc = Sh$. Чтобы доказать приведённое равенство, необходимо показать, что $Pc \subset Sh$ и $Sh \subset Pc$. Первое включение доказывается сравнительно просто: для этого достаточно описать машины Шёнфилда, вычисляющие простейшие функции, и описать способ реализации на машинах Шёнфилда операций суперпозиции, примитивной рекурсии и минимизации.

Для доказательства включения $Sh \subset Pc$ необходимо построить по каждой машине Шёнфилда некую частично вычислимую функцию. Эту функцию мож-

но построить таким образом, что она оказывается общей для всего класса машин Шёнфилда, вычисляющих функции от n переменных. То есть построенная функция U будет зависеть не только от начальных значений $x_1, x_2, ..., x_n$, но и от кода машины e:

$$U = U(e, x_1, x_2, ..., x_n).$$

Функции значений, зависящие от (кодов) алгоритмов, далее будут рассмотрены подробнее.

7. АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ЗАДАЧИ

Целью данного раздела будет доказательство существования алгоритмически неразрешимых проблем, то есть таких массовых задач, для решения которых не существует универсального способа.

Теорема 3. *Существуют алгоритмически неразрешимые задачи.*

Доказательство. Для доказательства теоремы проведём оценку мощности множества арифметических функций и множества вычислимых функций.

Рассмотрим множество характеристических функций, то есть множество функций вида

$$\chi_P(\bar{x}) = \begin{cases} 1, \text{ если } \bar{x} \in P; \\ 0, \text{ если } \bar{x} \notin P, \end{cases}$$

где $P \subseteq \mathrm{N}^k$. Каждая характеристическая функция является арифметической. Характеристических функций столько, сколько подмножеств у N^k. Мощность множества подмножеств N^k – континуум ($|2^{\mathrm{N}^k}| = c$). Таким образом, мощность множества всех арифметических функций не менее чем континуум.

Рассмотрим множество интуитивно вычислимых функций. Каждой вычислимой функции соответствует некая машина Шёнфилда. Каждую машину Шёнфилда можно закодировать натуральным числом. Следовательно, множество интуитивных функций не более чем счётно (имеет мощность $\aleph_0$). Так как $\aleph_0 < c$, то получаем утверждение теоремы. □

Данное доказательство не является конструктивным, так как не содержит построения алгоритмически неразрешимой задачи. Приведём пример такой задачи.

Рассмотрим множество машин Шёнфилда, вычисляющих функции одного аргумента. Назовем машину Шёнфилда *самоопределённой*, если она выдаёт некое значение, будучи запущенной на своём собственном коде, и *несамоопределённой* в противном случае (машина «зацикливается» на своём коде).

Утверждение 1. *Задача определения типа машины Шёнфилда по её коду является алгоритмически неразрешимой.*

Доказательство. Допустим противоположное. Пусть данная задача является алгоритмически разрешимой. Тогда существует машина

Шёнфилда S, которая запускается на коде некой машины и выдаёт в нулевом регистре 0, если это код несамоопределённой машины, и 1, если это код самоопределённой машины.

Построим машину S_1. Для этого дополним машину S следующими командами:

```
m+1: DEC 0, m+3
m+2: GOTO m+4
m+3: GOTO m+3
```

Данные команды останавливают машину S_1, когда в регистре 0 содержится 0, и зацикливают машину, если в нулевом регистре содержится 1.

Запустим машину S_1 на собственном коде. Если S_1 – самоопределенная, то она должна зациклиться, то есть S_1 – несамоопределенная машина. Аналогично, машина S_1 не может принадлежать и классу несамоопределенных машин. Таким образом, получаем противоречие, которое доказывает алгоритмическую неразрешимость задачи определения класса машины Шёнфилда. □

Естественно, число алгоритмически неразрешимых задач не ограничивается приведённой. К таким задачам относится, например, задача определения класса (противоречивая, выполнимая, логический закон) произвольной формулы логики предикатов, что было доказано А. Чёрчем в 1936 году.

8. УНИВЕРСАЛЬНЫЕ ФУНКЦИИ

Предложенные уточнения понятия алгоритма – машины Тьюринга, частично вычислимые функции, машины Шёнфилда – обладают одной особенностью. Одна конкретная машина или функция реализует единственный алгоритм. В то же время современные компьютеры представляют собой устройства, которые могут реализовывать множество алгоритмов. Оказывается, можно описать машину или частично вычислимую функцию для моделирования совокупности машин или функций. Такая машина при решении некоторой задачи в качестве одного из параметров может получать числовой код алгоритма решения и моделировать алгоритм по его коду. По сути такая машина является неким *транслятором алгоритмов*.

Дадим строгое математическое определение описанным понятиям.

Определение 11 (Универсальная функция). *Пусть дано некое множество арифметических функций M от n аргументов. Функция $U(n, x_1, ..., x_n)$ называется универсальной функцией для множества M, если выполняются два условия:*

1) *для любого фиксированного n_0 функция $U(n_0, x_1, ..., x_n)$ от n аргументов принадлежит множеству M;*

2) *для любой функции $f(x_1, ..., x_n) \in M$ существует число n_0 (возможно, не единственное) такое, что $f(x_1, ..., x_n) \equiv U(n_0, x_1, ..., x_n)$.*

Универсальная функция ведёт себя как компьютер, в качестве программы которого выступает значение первого аргумента функции.

Лемма 1. *Для множества частично вычислимых функций от n аргументов существует универсальная функция.*

Доказательство. Доказательство данной леммы основано на доказательстве включения $Sh \subset Pc$. Напомним, что для этого строилась одна функция $U(e, x_1, ..., x_n)$, которая зависела от кода машины *e*. Функция *U* по построению была частично вычислимой. Для любой частично вычислимой функции $f(x_1, ..., x_n)$ можно указать некое число n_0, являющееся кодом машины Шёнфилда, которая вычисляет *f*. При этом по построению функции *U* будет выполняться условие $f(x_1, ..., x_n) \equiv U(n_0, x_1, ..., x_n)$. □

Покажем теперь, что не для всех множеств арифметических функций можно построить универсальную функцию.

Лемма 2. *Для множества общерекурсивных функций от n аргументов универсальной функции не существует.*

Доказательство. Рассмотрим множество общерекурсивных функций одного аргумента, которое обозначим как $C\upsilon_1$. Предположим, что $U(n, x)$ – универсальная функция для этого множества. Рассмотрим функцию $g(x) = U(x, x) + 1$. Так как $U(n, x)$ – общерекурсивная функция, то и функция $g(x)$ является общерекурсивной. Следовательно, существует число $n_0 : g(x) = U(n_0, x)$. Однако

$$U(n_0, n_0) = g(n_0) = (\text{по определению } g) = U(n_0, n_0) + 1.$$

Получили противоречие. Следовательно, для множества $C\upsilon_1$ универсальной функции не существует. Случаи, когда количество аргументов общерекурсивной функции больше единицы, рассматриваются аналогично. □

9. НЕКОТОРЫЕ ТЕОРЕМЫ ТЕОРИИ АЛГОРИТМОВ

Рассмотрим некоторые фундаментальные результаты теории алгоритмов.

Многие частично вычислимые функции не являются всюду определёнными. Например, функция $f(x, y) = x - y$ определена, только если $x \geq y$. Отдельные частично вычислимые функции можно доопределить так, чтобы они стали общерекурсивными.

Определение 12 (Доопределение). *Пусть дана некая функция* $f(x_1, ..., x_n) \in Pc$. *Доопределением данной функции назовём общерекурсивную функцию* $\tilde{f}(x_1, ..., x_n) \in C\upsilon$ *такую, что*

$$\tilde{f}(x_1, ..., x_n) = \begin{cases} f(x_1, ..., x_n), \text{ если } f(x_1, ..., x_n) \text{ определено}, \\ m \in N, \text{ если } f(x_1, ..., x_n) \text{ не определено}. \end{cases}$$

Доопределением функции $f(x,y)=x-y$ может служить функция

$$\tilde{f}(x,y)=\begin{cases}x-y, & \text{если} \quad x\geq y,\\ 0, & \text{если} \quad x<y.\end{cases}$$

Оказывается, справедлива следующая теорема.

Теорема 4 (О доопределении). *Существует частично вычислимая функция, которую нельзя доопределить.*

Доказательство. Рассмотрим универсальную функцию $U(n,x)$ для множества частично вычислимых функций одного аргумента. Пусть $d(x)=\overline{\operatorname{sgn}}U(x,x)$. Допустим, что существует доопределение функции $d(x)$, которое обозначим как $\tilde{d}(x)$. Функция $d(x)$ является общерекурсивной, а следовательно, и частично вычислимой. Значит, существует номер n_0 такой, что $\tilde{d}(x)=U(n_0,x)$. Значение $\tilde{d}(n_0)$ определено, значит, определено и значение $U(n_0,n_0)$. Следовательно, определено значение $d(n_0)=\overline{\operatorname{sgn}}\,U(n_0,n_0)$. Можно записать

$$\overline{\operatorname{sgn}}U(n_0,n_0)=d(n_0)=\tilde{d}(n_0)=U(n_0,n_0).$$

Но равенство $\overline{\operatorname{sgn}}U(n_0,n_0)=U(n_0,n_0)$ не выполняется ни для каких натуральных чисел. Следовательно, предположение о существовании доопределения функции $d(x)$ было ложным. □

Опишем случай, когда доопределение частично вычислимой функции возможно.

Определение 13 (Рекурсивное множество). *Пусть M – некое подмножество N^n. Назовём множество M рекурсивным, если характеристическая функция этого множества*

$$\chi_M(x_1,\ldots,x_n)=\begin{cases}1, & \textit{если} \quad (x_1,\ldots,x_n)\in M,\\ 0, & \textit{если} \quad (x_1,\ldots,x_n)\notin M,\end{cases}$$

является общерекурсивной.

Тот факт, что множество M является рекурсивным, означает наличие алгоритма, который по любому натуральному числу (набору натуральных чисел) может определить, принадлежит ли это число (набор) множеству M. Примерами рекурсивных множеств являются: конечное множество, множество чётных чисел. Приведём следующую теорему без доказательства.

Теорема 5. *Если областью определения частично вычислимой функции является рекурсивное множество, то существует доопределение этой функции.*

Рассмотрим частично вычислимую функцию от двух аргументов $f(x,y)$. Этой функции соответствует некое натуральное n_0, при котором выполняется условие $f(x,y)=U(n_0,x,y)$, где U – универсальная функция для частично вычислимых функций двух аргументов. Напомним, что U можно воспринимать в роли своеобразного «компьютера», выполняющего программы машин Шёнфилда. Опишем следующее преобразование произвольной программы S, вычисляющей значение функции от двух аргументов:

1. Скопировать регистр 1 в регистр 2;
2. Обнулить регистр 1, увеличить регистр 1 на единицу x раз;
3. Выполнить программу S.

Само описываемое преобразование – перенумерация команд исходной машины S, добавление в начало программы новых команд – алгоритмично. Значит, ему соответствует некая вычислимая функция τ. Эта функция τ зависит от аргумента x и от исходной программы S (номера этой программы): $\tau=\tau(n,x)$. Также заметим, что функция τ является общерекурсивной. Результат вычисления функции $U(n_0,x,y)$ на данных x и y совпадает с результатом вычисления универсальной функции двух аргументов $\widetilde{U}(\tau(n_0,x),y)$ при данном y, если к программе n_0 применить преобразование τ, зависящее от x.

Приведённые рассуждения справедливы для любого n_0. Значит,

$$U(n,x,y)\equiv\widetilde{U}(\tau(n_0,x),y). \qquad (1)$$

Имеет место более общее утверждение.

Лемма 3 (Лемма о параметризации). *Для универсальной функции множества частично вычислимых функций n аргументов* $U(n,x_1,...,x_m,x_{m+1},...,x_n)$ *существует общерекурсивная функция* $\tau(n,x_1,...,x_m)$ *такая, что*

$$U(n,x_1,...,x_m,x_{m+1},...,x_n)\equiv\widetilde{U}(\tau(n,x_1,...,x_m),x_{m+1},...,x_n),$$

где $\widetilde{U}$ *– универсальная функция множества частично вычислимых функций n – m аргументов.*

Рассмотрим теорему, играющую центральную роль в теории алгоритмов.

Теорема 6 (Теорема Клини о неподвижной точке). *Пусть* $U(n,x)$ *– универсальная функция для множества частично вычислимых функций одного аргумента,* $f(x)$ *– произвольная общерекурсивная функция. Тогда существует число* n_0 *такое, что*

$$U(n_0,x)\equiv U(f(n_0),x).$$

Доказательство. Рассмотрим следующую вспомогательную функцию:

$$g(x,y)=U(f(\tau(x,x))y),$$

где τ – функция из равенства (1). Функция g является частично вычислимой функцией от двух аргументов. Следовательно, если $U_2(m,x,y)$ – универсальная функция для множества частично вычислимых функций двух аргументов, то существует номер m такой, что

$$g(x,y) = U_2(m,x,y).$$

По лемме о параметризации можно записать $U_2(m,x,y) = U(\tau(m,x),y)$. Учитывая определение функции $g(x, y)$, получаем равенство

$$U(\tau(m,x),y) = U(f(\tau(x,x)),y)).$$

Положив в данном равенстве $x = m$ и обозначив для краткости $\tau(m,m) = n_0$, получим утверждение теоремы. □

Рассмотрим следствие из теоремы Клини.

Следствие 1. *Пусть $U(n,x)$ – универсальная функция для множества частично вычислимых функций одного аргумента. Тогда существует число p, что для любого x*

$$U(p,x) = p.$$

Доказательство. Рассмотрим множество всех частично вычислимых функций, являющихся константами. Определим следующее отображение:

$\varphi(l)$ = минимальное n, при котором $U(n,x)$ задаёт функцию-константу, равную l. Функция $\varphi(l)$ является общерекурсивной. Значит, по теореме Клини существует p:

$$U(p,x) \equiv U(\varphi(p),x).$$

По смыслу определения функции φ, $U(\varphi(p),x) = p$. Отсюда и получается требуемое утверждение. □

Данное следствие имеет следующую интересную интерпретацию: существует программа, которая при любых входных данных печатает свой текст. В качестве упражнения предлагается написать такие программы.

Рассмотрим теорему, которая позволяет получить доказательство алгоритмической неразрешимости многих проблем.

Теорема 7 (Теорема Райса). *Пусть F – некое множество частично вычислимых функций одного аргумента, причём существуют функции, принадлежащие F, и функции, не принадлежащие F. Тогда множество номеров всех функций из F не является рекурсивным.*

Доказательство. Обозначим через N_F множество номеров всех функций из F. Предположим, что N_F является рекурсивным множеством. Это означает, что его характеристическая функция χ_{N_F} является общерекурсивной. Если N_F – рекурсивное множество, то и $\overline{N}_F$ – рекурсивное множество, так как

$\chi_{\overline{N}_F}(x) = 1 - \chi_{N_F}(x)$. По определению множества F ни множество N_F, ни множество $\overline{N}_F$ не являются пустыми.

Определим следующую функцию:

$$g(x) = \begin{cases} a \in \overline{N}_F, & \text{если} \quad x \in N_F, \\ b \in N_F, & \text{если} \quad x \in \overline{N}_F. \end{cases}$$

Функция $g(x)$ является общерекурсивной, так как фактически $g(x) = a * \chi_{N_F} + b * \chi_{\overline{N}_F}$.

По теореме о неподвижной точке существует n:

$$U(n, x) = U(g(n), x).$$

Так как $N_F \cup \overline{N}_F$, то $n \in N_F$ или $n \in \overline{N}_F$. Оба случая являются невозможными.

Пусть, к примеру, $n \in N_F$. Тогда $U(n, x)$ определяет функцию, которая принадлежит F. Значит, и $U(g(n), x) \in F$. Но при $n \in N_F$ имеем $g(n) = a$, $a \notin N_F$, а значит, и $U(g(n), x) \notin F$. Получили противоречие. Единственное предположение теоремы – это предположение о рекурсивности набора N_F. Оно не верно. □

Теорема Райса допускает следующую интерпретацию: не существует программы, которая может проверять некие нетривиальные свойства *всех* программ. Например, нельзя написать программу, которая бы по тексту любой программы на Паскале устанавливала, «зацикливается» программа или нет.

Упражнение. Описать, как использовать теорему Райса для доказательства отсутствия такой программы.

10. СЛОЖНОСТЬ АЛГОРИТМОВ И МАССОВЫХ ПРОБЛЕМ

Предметом классической теории алгоритмов является уточнение понятия алгоритма и установление алгоритмической разрешимости массовых проблем. Однако существование алгоритма для задачи ещё не означает её практическую разрешимость. В качестве примера рассмотрим *задачу коммивояжёра*: имеется n городов, соединённых сетью дорог, требуется посетить все города по маршруту наименьшей протяжённости. Тривиальный алгоритм решения задачи состоит в переборе всех возможных маршрутов. Если граф, представляющий города и соединяющие их дороги, является полным, то необходимо рассмотреть $(n - 1)!$ различных вариантов. Уже при малых n данная задача не может быть решена, так как количество вариантов становится астрономически большим. В связи с этим возникает необходимость введения некоторых мер сложности алгоритмов для поиска среди возможных алгоритмов решения массовой проблемы оптимального.

Попытаемся описать сложность какого-либо определённого алгоритма. Вообще говоря, оценка сложности алгоритма зависит от *вычислительной модели*, которая используется для описания алгоритма. В качестве вариантов *меры*

сложности алгоритма можно рассматривать такие величины, как быстродействие (время работы) алгоритма, количество используемой алгоритмом памяти и другие величины.

Из рассмотренных нами вычислительных моделей – машины Тьюринга, машины Шёнфилда, частично вычислимые функции – для дальнейшего изложения фиксируем в качестве основной модели машины Шёнфилда. Будем анализировать временную сложность вычислений на машине Шёнфилда. Пусть M – машина Шёнфилда (программа), которая соответствует некоторому алгоритму (реализует алгоритм), $x_1x_2...x_n$ – слово (число), которое кодирует входные данные программы. Обозначим через $T(M, x_1x_2... x_n)$ количество шагов машины M, которые требуются для того, чтобы выполнить программу на входных данных $x_1x_2... x_n$.

Функция T неудобна для изучения, так как в общем случае она устроена достаточно неоднородно. Определим на основе функции T новую функцию $t_M(n)$:

$$t_M(n) = \max T(M, x).$$

Здесь x – множество всевозможных наборов входных данных алгоритма, причём длина наборов (количество элементов) не превосходит n. Фактически функция $t_M(n)$ – функция от длины входных данных алгоритма. Эта функция характеризует сложность алгоритма в наихудшем случае.

Итак, налицо возможность вычислять временную сложность алгоритма, реализуемого на машине Шёнфилда. Можно сказать, что определена некая функция, которая по алгоритму получает функцию $t_M(n)$.

Пусть теперь мы пытаемся установить сложность некой массовой проблемы. Очевидный подход заключается в следующем: если для решения массовой проблемы существует несколько алгоритмов, то сложность массовой проблемы – это сложность самого хорошего (быстрого) алгоритма для её решения. Как оказывается, данный подход применим не всегда.

Теорема 8 (Блюма, об ускорении). *Существует такая вычислимая функция f, что любую машину M, вычисляющую f, можно ускорить, то есть построить машину* M_1*, также вычисляющую f, и при этом*

$$t_{M_1}(n) \le \log_2 t_M(n)$$

для почти всех n.

Смысл теоремы Блюма: имеются такие массовые проблемы, алгоритмы решения которых допускают бесконечное улучшение.

В теории сложности принят подход, при котором анализ сложности массовой проблемы проводится на основе установления принадлежности данной проблемы к особым классам, называемым классами сложности.

Определение 14 (Класс сложности). *Пусть C – некоторое выделенное множество функций* $t_M(n)$*. Классом сложности A назовём множество всех*

массовых проблем, для решения каждой из которых существует такой алгоритм (машина) M, что функция $t_M(n)$ принадлежит C.

Рассмотрим и обсудим следующий пример. Определим множество:

DTIME($t(n)$) = {массовая проблема a |∃ машина M для a и $t_M(n) = O(t(n))$}.

Для произвольной функции натурального аргумента $t(n)$ при помощи DTIME($t(n)$) можно образовать класс сложности, состоящий из массовых проблем, для решения которых существуют алгоритмы со временем работы, сравнимым с $t(n)$.

Замечание. Напомним, что запись $f(x) = O(g(x))$ означает следующее:

$$f(x) = O(g(x)) \Leftrightarrow \lim_{x\to\infty}\left|\frac{f(x)}{g(x)}\right| = k > 0.$$

Обоснованием подхода, при котором функции сложности рассматриваются с точностью до мультипликативной константы, может служить тот факт, что реальная скорость выполнения программы на компьютере определяется не только количеством шагов алгоритма, но и длительностью элементарного шага (временем выполнения машинной команды).

Целью следующих разделов будет рассмотрение конкретных классов вычислительной сложности и взаимосвязей между ними.

11. КЛАССЫ СЛОЖНОСТИ P И EXP

Определим следующий важный класс сложности P:

$$\mathrm{P} = \bigcup_{k\geq 0}\mathrm{DTIME}(n^k).$$

Класс сложности P определяет массовые проблемы, для решения которых существуют алгоритмы со временем работы, сравнимым с некоторым полиномом.

Класс сложности P обладает очень важной особенностью: он замкнут относительно операции суперпозиции. Пусть имеются две массовые проблемы A и B и пусть $A \in$ P. Если удалось построить алгоритм сведения проблемы B к проблеме A, время работы которого ограничено полиномом, то можно утверждать, что $B \in$ P.

Все рассмотренные ранее вычислительные модели (машины Тьюринга, машины Шёнфилда, частично вычислимые функции) преобразуются друг в друга при помощи алгоритмов с полиномиальной сложностью. Отсюда следует возможность менее формального изложения материала, при которой в дальнейшем будут рассматриваться реализации алгоритмов не на машинах Шёнфилда, а в неком обобщённом виде, более привычном для программистов.

Пример. Пусть требуется умножить две произвольные матрицы размером $n\times n$. Тривиальный алгоритм решения данной массовой проблемы будет содержать участок кода, подобный следующему:

```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      c[i,j] := c[i,j] + a[i,k] * b[k,j];
```

Легко установить, что внутренний оператор циклов будет выполняться n^3 раз. Следовательно, общее время работы данного алгоритма сравнимо с n^3. Значит, массовая проблема умножения матриц принадлежит классу P.

Опишем ещё один важный класс сложности EXP:

$$\mathrm{EXP} = \bigcup_{k \geq 0} \mathrm{DTIME}(2^{n^k}).$$

Класс сложности EXP определяет массовые проблемы, для решения которых существуют алгоритмы со временем работы, сравнимым с некой экспоненциальной функцией.

Внимание: если задана некоторая массовая проблема *A* из P, то данная массовая проблема принадлежит и EXP. Иными словами, $\mathrm{P} \subseteq \mathrm{EXP}$. Установление данного факта тривиально. Достаточно взять любой алгоритм для решения массовой проблемы *A* и дополнить его следующим циклом, который ничего не выполняет:

```
for i := 1 to 2^n do
  begin end;
```

Верно ли обратное включение $\mathrm{EXP} \subset \mathrm{P}$ и, следовательно, равенство P = EXP? Иными словами, можно ли для решения любой массовой проблемы из EXP предложить полиномиальный алгоритм? Ответ на данный вопрос даёт следующая теорема.

Теорема 9 (Хартманис). *Существуют массовые проблемы, для решения которых невозможно предложить алгоритм из класса сложности P.*

Из теоремы Хартманиса следует, что $\mathrm{P} \subset \mathrm{EXP}$, то есть классы P и EXP существенно различаются.

12. КЛАСС СЛОЖНОСТИ NP

Рассмотрим следующую модификацию машины Шёнфилда. Добавим к машине специальный регистр с номером –1 и особый *угадывающий модуль*. Работа модифицированной машины будет происходить следующим образом. В любой момент времени машина может перейти в *недетерминированное состояние*. В этом состоянии угадывающий модуль машины записывает в регистр –1 некое случайное число из диапазона 0...*k*. Далее машина Шёнфилда переходит в нормальное состояние и продолжает свою работу. Естественно, эта работа может зависеть от содержимого регистра –1.

Недетерминированные машины Шёнфилда позволяют ввести новый класс сложности вычислений.

Определение 15 (Класс сложности NP). *Пусть для массовой проблемы существует алгоритм, моделируемый на недетерминированной машине Шён-*

филда. При этом время работы машины в детерминированном состоянии ограничено полиномом. В этом случае будем говорить, что массовая проблема принадлежит классу сложности NP.

В качестве примера массовой проблемы из класса сложности NP рассмотрим упоминавшуюся задачу коммивояжёра. Для её решения можно предложить недетерминированную машину Шёнфилда, угадывающий модуль которой записывает в регистр –1 число, кодирующее некий маршрут (последовательность городов). В детерминированном состоянии машина Шёнфилда декодирует по числу маршрут, вычисляет его длину и сравнивает с неким эталонным (кратчайшим) маршрутом. Работа машины в детерминированном состоянии ограничена полиномом. Следовательно, задача коммивояжёра принадлежит классу сложности NP.

Если предложить менее формальное определение класса сложности NP, то можно сказать, что к этому классу относятся проблемы, которые решаются перебором возможных ответов. Машина пытается угадать ответ задачи. Для этого она переходит в недетерминированное состояние, генерирует некое число, а затем за полиномиальное время проверяет, является это число ответом задачи или нет.

Из определения классов сложности P и NP следует включение $P \subseteq NP$. Является ли это включение строгим? На данный момент (2007 год) этот факт не доказан. Данная проблема известна как проблема P=NP?

Теорема 10. *Любая недетерминированная машина Шёнфилда моделируется некоторой детерминированной машиной.*

Для моделирования недетерминированной машины Шёнфилда на детерминированной машине известны алгоритмы, временная сложность которых экспоненциальная. Следовательно, имеет место утверждение

$$P \subseteq NP \subseteq EXP.$$

Так как $P \subset EXP$, то одно из включений в приведенном соотношении является строгим.

13. СВОДИМОСТЬ И NP-ПОЛНЫЕ ЗАДАЧИ

Выше было указано, что класс P является замкнутым относительно операции суперпозиции. Формализуем приведённые рассуждения при помощи понятия сводимости.

Определение 16 (Полиномиальная сводимость). *Пусть A и B – две массовые проблемы. Будем говорить, что проблема A полиноминально сводится к проблеме B, и обозначать это как $A \propto B$, если существует алгоритм с полиномиальной сложностью, который позволяет свести любую индивидуальную задачу из A к индивидуальной задаче из B с возможностью соответствующей интерпретации ответов.*

Понятие полиномиальной сводимости позволяет ввести понятие NP-полных задач.

Определение 17 (NP-полная задача). *Массовая проблема A называется NP-полной задачей, если выполняются два условия:*

1. $A \in NP$.

2. *Для любой проблемы* $B \in NP$ *справедливо* $B \propto A$.

Если для какой-либо NP-полной задачи существует полиномиальный разрешающий алгоритм, то и для любой задачи из класса сложности NP существует полиномиальный разрешающий алгоритм. Если какая-то NP-полная задача не лежит в классе P, то и все NP-полные задачи не лежат в классе P.

Рассмотрим следующую массовую проблему. Пусть булева функция $f(x_1, ..., x_n)$ задана своей КНФ. Требуется установить, является ли данная булева функция выполнимой. Исторически NP-полнота данной проблемы была доказана первой.

Теорема 11. *Проблема проверки выполнимости произвольной КНФ является NP-полной.*

В настоящее время список NP-полных задач состоит из нескольких тысяч. Под классификацию NP-полной попадает практически любая задача, где решение ищется перебором возможных вариантов.

14. АНАЛИЗ СЛОЖНОСТИ РЕКУРСИВНЫХ АЛГОРИТМОВ

Рекурсия является достаточно общим алгоритмическим приёмом. Данный приём заключается в решении исходной задачи путём разбиения её на несколько сходных подзадач. Использование рекурсии часто позволяет достаточно просто представить и записать алгоритмы. Для многих практически важных задач лучшие оценки сложности дают алгоритмы, использующие рекурсию. Рассмотрим несколько примеров.

1. Сортировка чисел

Рассмотрим последовательность натуральных чисел $x_1, x_2, ..., x_n$, которую необходимо упорядочить по возрастанию. Очевидный алгоритм решения данной задачи состоит в нахождении минимального члена исходной последовательности, постановки его на первое место в результирующей последовательности и повторении данного шага для оставшейся неотсортированной последовательности. Легко установить, что данный алгоритм требует $O(n^2)$ попарных сравнений в худшем случае.

Предложим для задачи сортировки рекурсивный алгоритм. Пусть $n = 2^k$. При $k = 1$ алгоритм упорядочивает последовательность одним сравнением. Пусть алгоритм определён для некоторого k. Тогда при $k + 1$ алгоритм работает так:

1. Последовательность $x_1, x_2, ..., x_{2^{k+1}}$ разбивается на две подпоследовательности: $x_1, x_2, ..., x_{2^k}$ и $x_{2^k+1}, x_{2^k+2}, ..., x_{2^{k+1}}$.

2. К обеим подпоследовательностям длиной 2^k применяется построенный алгоритм. Получаем две упорядоченные последовательности: $x'_1, x'_2, ..., x'_{2^k}$ и $x'_{2^k+1}, x'_{2^k+2}, ..., x'_{2^k+1}$.

3. Осуществляется слияние двух упорядоченных последовательностей сравнением их левых элементов x'_1 и x'_{2^k+1} и помещением наименьшего в начало результирующей последовательности.

Если длина исходной последовательности n не равна 2^k, последовательность дополняется нулями (или, напротив, большими числами), чтобы её длина стала степенью двойки.

Пусть $T(n)$ – число попарных сравнений, используемых в данном рекурсивном алгоритме для произвольной последовательности длиной n. Тогда получаем соотношение

$$\begin{cases} T(n) = 2T(n/2) + n - 1, \\ T(2) = 1. \end{cases} \qquad (2)$$

Утверждение 2. *Для рекуррентного соотношения (2) справедлива формула*

$$T(2^k) = 2^k k - 2^k + 1.$$

Проверка данного утверждения производится индукцией по k.

Для произвольного n справедливо

$$T(n) = O(n \log n).$$

Таким образом, можно утверждать, что приведённый рекурсивный алгоритм сортировки лучше по порядку исходного «наивного» алгоритма.

2. Возведение в степень

Пусть зафиксирован элемент a некоторой алгебраической системы с операцией умножения и натуральное число n. Требуется найти a^n. Тривиальный алгоритм требует для этого $n - 1$ умножение.

Рассмотрим рекурсивный алгоритм умножения. Пусть $u(n) = a^n$. Если $n = 1$, то $u(1) = a$. Допустим, что $n > 1$. Вычислим числа $k = [n/2]$ и $l = n \bmod 2$, где операции $[x]$ и mod обозначают соответственно деление нацело и остаток от деления. Тогда справедливо

$$\begin{cases} u(n) = u(n/2) \cdot u(n/2), & \text{если } l = 0; \\ u(n) = u([n/2]) \cdot u([n/2]) \cdot a, & \text{если } l = 1. \end{cases}$$

Легко убедиться, что в данном алгоритме используется не более $2[\log_2 n]$ умножений.

3. Умножение матриц

Рассмотрим задачу нахождения произведения двух матриц одинакового размера $n \times n$. Пусть $n = 2$ и пусть

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix},\ B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},\ C = AB = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Используем стандартный способ нахождения матрицы C:

$$c_{11} = a_{11}b_{11}+a_{12}b_{21},\quad c_{12} = a_{11}b_{12}+a_{12}b_{22},\quad c_{21} = a_{21}b_{11}+a_{22}b_{21},\quad c_{22} = a_{21}b_{12}+a_{22}b_{22}.$$

Данный способ требует 8 умножений и 4 сложения. На первый взгляд кажется невозможным улучшить этот результат. Однако Штрассеном было предложено вычислять элементы матрицы C по следующей схеме. Вначале находим промежуточные значения:

$$p_1 = (a_{11} + a_{22})(b_{11} + b_{22}),\quad p_2 = (a_{21}+ a_{22})b_{11},\quad p_3 = a_{11}(b_{12} - b_{22}),\quad p_4 = a_{22}(b_{21} - b_{11}),$$

$$p_5 = (a_{11} + a_{12})b_{22},\qquad p_6 = (a_{21} - a_{11})(b_{11} + b_{12}),\qquad p_7 = (a_{12} - a_{22})(b_{21} + b_{22}).$$

Затем вычисляем элементы матрицы C:

$$c_{11} = p_1+p_4 - p_5+p_7,\qquad c_{12} = p_3+p_5,\qquad c_{21} = p_2+p_4,\qquad c_{22} = p_1 - p_2 + p_3+p_6.$$

Алгоритм Штрассена требует 7 умножений и 18 сложений.

Пусть теперь $n = 2^{k+1}$. Тогда алгоритм Штрассена работает так: матрицы A и B размером $2^{k+1} \times 2^{k+1}$ представляются как 2×2-матрицы над кольцом матриц порядка $2^k \times 2^k$, и применяется изложенный выше алгоритм умножения матриц 2×2. Если $n \neq 2^{k+1}$, то находят такое k, что $2^k < n < 2^{k+1}$, и к матрицам добавляется нужное число нулевых строк и столбцов.

Пусть $T(2^k)$ – число арифметических операций, используемых в алгоритме Штрассена на матрицах размером $2^{k+1} \times 2^{k+1}$. Тогда справедливо соотношение

$$\begin{cases} T(2^{k+1}) = 7T(2^k) + 18(2^k)^2, \\ T(2^0) = 1. \end{cases} \qquad (3)$$

Утверждение 3. *Для соотношения (3) справедлива формула*

$$T(2^k) = 7^{k+1} - 6*2^{2k}.$$

Доказательство данного утверждения проводится по индукции.

Ясно, что выполняется

$$T(n) = O(7^{\log_2 n}) = O((2^{\log_2 7})^{\log_2 n}) = O(n^{\log_2 7}).$$

Так как $\log_2 7 \approx 2{,}807$, то алгоритм Штрассена лучше по порядку обычного алгоритма умножения матриц.

Усилиями ряда математиков (Пан, Виноград, Романи, Копперсмит) экспоненту сложности матричного умножения удалось уменьшить до $\approx 2{,}376$. Однако на данный момент не найдено алгоритма (и не доказано отсутствие такового) для экспоненты сложности, равной 2.

4. Решение рекуррентных соотношений, соответствующих рекурсивным алгоритмам

При анализе временной сложности рекурсивных алгоритмов возникает потребность в решении рекуррентных соотношений специального вида. Рассмотрим несколько видов таких соотношений и опишем подходы к их решению.

Пусть задано рекуррентное соотношение следующего вида

$$\begin{cases} T(n) = aT(\frac{n}{k}) + f(n), \\ T(1) = c. \end{cases} \quad (4)$$

Здесь a, k, c – некоторые константы, $f(n)$ – заданная функция. Запишем несколько значений искомой функции $T(n)$:

$$T(1) = c,$$
$$T(k) = aT(1) + f(k) = ac + f(k),$$
$$T(k^2) = aT(k) + f(k^2) = a^2c + af(k) + f(k^2),$$
$$T(k^3) = aT(k^2) + f(k^3) = a^3c + a^2f(k) + af(k^2) + f(k^3),$$
$$\dots$$
$$T(k^p) = a^pc + \underbrace{a^{p-1}f(k) + a^{p-2}f(k^2) + \dots + f(k^p)}_{p \text{ слагаемых}}.$$

Обратим внимание на то, что рекуррентное соотношение (4) позволяет однозначно вычислить значение функции $T(n)$ только при n, являющихся степенями числа k. Вопрос о решении рекуррентного соотношения (4) сводится к вопросу о возможности компактной записи выражения для суммы S:

$$S = a^{p-1}f(k) + a^{p-2}f(k^2) + \dots + f(k^p).$$

Рассмотрим один из случаев, когда это возможно. Пусть $f(n) = bn^t$, где b,t – некоторые константы. В этом случае сумма S имеет следующий вид:

$$S = b(a^{p-1}k^t + a^{p-2}k^{2t} + \dots + k^{pt}).$$

Выражение в скобках является геометрической прогрессией. Первым членом этой прогрессии будем считать k^{pt}, тогда последний член – $a^{p-1}k^t$, знаменатель прогрессии – $\frac{a}{k^t}$.

Если $\frac{a}{k^t} = 1$ и соответственно $a = k^t$, то все члены прогрессии одинаковы и равны k^{pt}. Тогда $S = bpk^{pt} = bpa^p$, а $T(k^p) = a^pc + bpa^p = a^p(c + bp)$.

Если $\frac{a}{k^t} \neq 1$, то по формуле для суммы геометрической прогрессии

$$S = b \cdot \frac{k^{pt}\left(\left(\frac{a}{k^t}\right)^p - 1\right)}{\frac{a}{k^t} - 1} = \frac{bk^t(a^p - k^{pt})}{a - k^t}.$$

Значит, в этом случае получаем $T(k^p) = a^p c + \frac{bk^t(a^p - k^{pt})}{a - k^t}$. Таким образом, получаем выражение $T(k^p)$:

$$T(k^p) = \begin{cases} a^p(c + bp), & \text{если } a = k^t; \\ a^p c + \frac{bk^t(a^p - k^{pt})}{a - k^t}, & \text{если } a \neq k^t. \end{cases}$$

При необходимости можно получить выражение для $T(n)$, выполнив замену $k^p = n$. Подытожим приведённые выкладки в виде леммы.

Лемма 4. *Решением рекуррентного соотношения*

$$\begin{cases} T(n) = aT(\frac{n}{k}) + bn^t, \\ T(1) = c, \end{cases}$$

где a, b, c, k, t – некоторые константы, является функция

$$T(n) = \begin{cases} n^{\log_k a}(c + b\log_k n), & \text{если} \quad a = k^t; \\ n^{\log_k a} c + \frac{bk^t(n^{\log_k a} - n^t)}{a - k^t}, & \text{если} \quad a \neq k^t. \end{cases}$$

Приведём некоторые другие рекуррентные соотношения, возникающие при анализе рекурсивных алгоритмов.

Если рекурсия вызывает аддитивное уменьшение сложности задачи, то используется соотношение

$$\begin{cases} T(n) = aT(n-k) + f(n), \\ T(0) = c. \end{cases}$$

Иногда рекурсия ведёт к «квадратичному» уменьшению сложности. В этом случае возникает соотношение

$$\begin{cases} T(n) = a\sqrt{n}T(\sqrt{n}) + f(n), \\ T(2) = c. \end{cases}$$

Для решения приведенных соотношений в некоторых частных случаях применимы соображения, использовавшиеся при решении соотношения (4).

15. ТОЧНАЯ ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ

Полученные ранее оценки сложности для алгоритмов являлись достаточно грубыми. Как правило, речь шла об установлении функции сложности «с точностью до О большое». В данном и следующих разделах основной целью будет получение точной оценки времени выполнения для некоторых конкретных алгоритмов.

Рассмотрим следующую задачу: дан массив из N чисел, требуется найти минимальное число в массиве. Для решения этой задачи можно предложить следующий алгоритм, записанный в форме, близкой к языку Pascal:

```
min := M[1];                          {1}
for i := 2 to N do                    {2}
  if M[i] < min then                  {3}
                  min := M[i];        {4}
```

Попробуем подсчитать число шагов выполнения этого алгоритма, причём сделать это как можно точнее. Первый оператор выполняется один раз. Оператор проверки условия цикла `{2}` выполнится $N - 1$ раз (в действительности в начале цикла выполняется инициализация, так что можно увеличить общее число операторов на единицу). Проверка условия – оператор `{3}` – выполняется при каждой итерации цикла, то есть $N - 1$ раз. Особого внимания заслуживает оператор `{4}`. Количество выполнений этого оператора зависит от данных исходной задачи и не выражается в явной форме. Как правило, в большинстве алгоритмов встречаются операторы, количество выполнений которых зависит от исходных данных. В связи с этим при точной оценке сложности алгоритма принят подход, согласно которому вместо всех возможных наборов входных данных проводится оценка сложности в трёх случаях – наилучшем, наихудшем и среднем.

Вернёмся к предыдущему алгоритму. Что такое «наилучший» случай для данного алгоритма? Очевидно, таковым будет случай, когда минимальный элемент является первым элементом в исходном массиве. В этом случае количество выполнений оператора `{4}` будет равно нулю. Наихудший случай соответствует ситуации, когда исходный массив отсортирован по убыванию. Тогда оператор `{4}` будет выполняться при каждой итерации цикла, то есть $N - 1$ раз.

Рассмотрим «средний» случай. Для вывода оценки количества выполнений оператора `{4}` сделаем следующее предположение о природе элементов исходного массива: будем считать, что все элементы массива различны[1]. При первой итерации цикла вероятность выполнения оператора `{4}` равна $\frac{1}{2}$, так как такова вероятность того, что добавляемый к «куче» чисел (из одного элемента) новый элемент окажется в этой «куче» наименьшим. То есть можно сказать, что при первой итерации цикла оператор `{4}` «выполнится» $\frac{1}{2}$ раз. При второй итера-

[1] Без этого предположения вывод оценки становится затруднительным.

ции цикла вероятность выполнения оператора {4} равна $\frac{1}{3}$: «куча» состоит из двух элементов; добавляем третий, и вероятность того, что он самый маленький в «куче», равна $\frac{1}{3}$. Продолжая рассуждения, получим, что в среднем случае количество выполнений оператора {4} выражается формулой

$$S = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{N}. \quad (5)$$

Рассмотрим следующую сумму, играющую важную роль при анализе алгоритмов:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$$

Данная сумма представляет собой частичную сумму гармонического ряда $(1, \frac{1}{2}, \frac{1}{3}, \ldots)$, являющегося расходящимся. Известна следующая оценка для H_n:

$$H_n = \ln n + \gamma + \frac{1}{2n} + o(\frac{1}{n}), \quad (6)$$

где $\gamma = 0{,}5772156649...$ – это *постоянная Эйлера.*

Используя данную оценку, примем выражение для суммы (5):

$$S = H_N - 1 \approx \ln N - 0{,}42 \approx \ln N .$$

Полученная оценка является весьма точной (с точностью до аддитивной константы). Таким образом, для количества выполнений оператора {4} получены оценки: $B(N) = 0$, $W(N) = N - 1$, $A(N) = \ln N$ (здесь и далее: $B(N)$ – наилучший случай, $W(N)$ – наихудший случай, $A(N)$ – средний случай). Если это необходимо, теперь легко можно подсчитать общее число выполнения операторов в нашем алгоритме. Естественно, требуется рассмотрение трёх случаев. Кроме этого, иногда для отдельных операторов вводят так называемые *стоимости выполнения*, характеризующие время выполнения отдельного оператора. Пусть, например, стоимости выполнения операторов нашего алгоритма равны c_1, c_2, c_3, c_4 соответственно. Тогда алгоритм выполняется за время

- $c_1 + (N-1)c_2 + (N-1)c_3 + 0*c_4 = c_1 + (N-1)(c_2 + c_3)$ в наилучшем случае;
- $c_1 + (N-1)c_2 + (N-1)c_3 + c_4 \ln N = c_1 + (N-1)(c_2 + c_3) + c_4 \ln N$ в среднем случае;
- $c_1 + (N-1)c_2 + (N-1)c_3 + (N-1)c_4 = c_1 + (N-1)(c_2 + c_3 + c_4)$ в наихудшем случае.

Приведённый пример показывает, что общая задача точного анализа алгоритмов является достаточно нетривиальной. При её выполнении активно используется математический аппарат, в частности теория вероятностей. В даль-

нейшем рассмотрим задачу анализа на некоторых конкретных примерах алгоритмов.

16. АЛГОРИТМЫ СОРТИРОВКИ И ИХ АНАЛИЗ

1. Алгоритм быстрой сортировки: описание

Задача сортировки – одна из наиболее часто встречающихся при обработке данных. В разд. 14 был предложен алгоритм сортировки, имеющий сложность $O(N \log_2 N)$ по числу сравнений. Рассмотрим ещё один алгоритм сортировки и проведём более точную оценку его сложности.

Идея алгоритма *быстрой сортировки* предложена Ч. Хоаром и заключается в следующем. Пусть требуется отсортировать по возрастанию массив *M,* содержащий *N* чисел. Выберем элемент массива *M,* называемый *осевым*, и переупорядочим массив так, чтобы все элементы, расположенные левее осевого, были меньше его, а элементы, расположенные правее осевого элемента, были больше его. В левой и правой части (относительно осевого элемента) массива элементы не упорядочиваются. Затем алгоритм быстрой сортировки применяется к левой и правой частям массива (т. е. вызывается рекурсивно).

Запись алгоритма быстрой сортировки может выглядеть следующим образом:

```
{предположение: M - глобальный массив}
{сортируем от индекса first до last}
procedure QuickSort(first, last)
begin
  if first < last then begin
    {находим позицию осевого элемента}
    pivot := FindPivot(first, last)
    QuickSort(first, pivot-1);        {сортируем левую часть}
    QuickSort(pivot + 1, last)        {сортируем правую часть}
  end
end;
```

В данной записи использовалась функция `FindPivot` для нахождения позиции осевого элемента. Она допускает несколько вариантов реализации. Рассмотрим один из простейших:

```
function FindPivot(first, last)
begin
  pivot_value := M[first];
  pivot := first;
  for i := first + 1 to last do
    if M[i] < pivot_value then begin      {Comp}
      inc(pivot);
      M[i] <-> M[pivot]                   {Swap}
    end;
  M[first] <-> M[pivot];
  result   := pivot
end;
```

Логика работы данного кода заключается в следующем. За значение осевого элемента принимается значение первого элемента просматриваемой части массива. Затем выполняется проход по массиву. Если обнаружен элемент, меньший осевого, то указатель осевого элемента `pivot` увеличивается на единицу, и найденный элемент переставляется с элементом с новым номером `pivot`. После очередной итерации цикла в массиве можно выделить четыре части. Одна состоит из первого – осевого – элемента массива. Вторую часть составляют элементы с индексами от `first` до `pivot` – это элементы, меньшие осевого. Третья часть массива – элементы с индексами от `pivot+1` до `i` – элементы, большие осевого. Наконец, четвёртая часть – это непросмотренные элементы массива.

2. Алгоритм быстрой сортировки: анализ среднего случая

После обсуждения алгоритма быстрой сортировки проведём его анализ. Будем анализировать число сравнений элементов сортируемого массива (оператор `{Comp}`).

При вызове функции `FindPivot` для участка массива длиной N она выполняет $N - 1$ сравнение. Таким образом, общее число сравнений в алгоритме быстрой сортировки зависит от того, сколько раз вызывается функция `FindPivot` и для участков какой длины она вызывается.

Начнём анализ алгоритма быстрой сортировки со среднего случая. Пусть $A(N)$ – количество сравнений в среднем для массива длиной N. Так как быстрая сортировка является рекурсивным алгоритмом, найдём рекуррентное соотношение для $A(N)$. Очевидно, что $A(0) = 0$, $A(1) = 0$. Пусть при очередном вызове функция `FindPivot` вернула значение P. Тогда

$$A(N) = (N-1) + A(P-1) + A(N-P) \text{ для } N \geq 2.$$

Число P может с равной вероятностью принять любое значение от 1 до N. Таким образом, среднее значение $A(N)$ для $N \geq 2$ выражается как

$$A(N) = (N-1) + \frac{1}{N}\sum_{i=1}^{N} A(i-1) + A(N-i)).$$

В записанной сумме аргумент первого слагаемого пробегает значения от 0 до $N - 1$, а аргумент второго слагаемого – те же значения, но в обратном порядке. Следовательно,

$$A(N) = (N-1) + \frac{2}{N}\sum_{i=0}^{N-1} A(i) \qquad \text{для } N \geq 2, \tag{7}$$

$$A(1) = A(0) = 0.$$

Для упрощения рекуррентного соотношения (7) избавимся от дроби $\frac{1}{N}$, умножая соотношение на *N*. Кроме этого, запишем явно выражение для $A(N-1)$:

$$A(N)\cdot N = (N-1)N + 2\sum_{i=0}^{N-1} A(i),$$

$$A(N)\cdot N = (N-1)N + 2A(N-1) + 2\sum_{i=0}^{N-2} A(i), \qquad (8)$$

$$A(N-1)\cdot (N-1) = (N-2)(N-1) + 2\sum_{i=0}^{N-2} A(i). \qquad (9)$$

Вычитая из равенства (8) равенство (9), получим

$$A(N)\cdot N - A(N-1)\cdot (N-1) = (N-1)N + 2A(N-1) - (N-2)(N-1) =$$
$$= 2A(N-1) + 2N - 2.$$

Это позволяет записать окончательное рекуррентное соотношение, не содержащее суммы:

$$A(N) = \frac{(N+1)A(N-1) + 2N - 2}{N}. \qquad (10)$$

Будем решать рекуррентное соотношение (10). Разделим обе части соотношения на *N+1:*

$$\frac{A(N)}{N+1} = \frac{(N+1)A(N-1) + 2N - 2}{N(N+1)} = \frac{A(N-1)}{N} + \frac{2N-2}{N(N+1)}.$$

Сделаем замену $D(N) = \frac{A(N)}{N+1}$ (очевидно, $D(1) = \frac{A(1)}{2} = 0$):

$$D(N) = D(N-1) + \frac{2N-2}{N(N+1)}.$$

Для функции $D(N)$ легко получить выражение через сумму:

$$D(N) = \sum_{k=1}^{N} \frac{2k-2}{k(k+1)} = 2\left[\sum_{k=1}^{N} \frac{k-1}{k(k+1)}\right] = 2\left[\sum_{k=1}^{N} \frac{1}{k+1} - \sum_{k=1}^{N} \frac{1}{k(k+1)}\right] =$$
$$= 2\left[\sum_{k=1}^{N} \frac{1}{k+1} - \sum_{k=1}^{N} \frac{1}{k} - \frac{1}{k+1}\right] = 2\left[H_{N+1} - 1 - \left(1 - \frac{1}{N+1}\right)\right].$$

Теперь запишем выражение для $A(N)$*:*

$$A(N) = (N+1)D(N) = 2(N+1)H_{N+1} - 4(N+1) + 2 \approx 2(N+1)\ln(N+1) - (4-2\gamma)N.$$

Сознательно огрубив оценку для $A(N)$, можем записать

$$A(N) \approx 2(N+1)\ln(N+1).$$

Рассмотрение среднего случая для числа сравнений закончено.

3. Асимптотически оптимальный алгоритм сортировки

Прежде чем выполнять оценку алгоритма быстрой сортировки в наилучшем и наихудшем случае, проанализируем вопрос: насколько оптимальным является данный алгоритм по числу сравнений?

Пусть имеется некий алгоритм, выполняющий сортировку последовательности чисел $x_1, x_2, ..., x_N$ путём сравнения элементов последовательности. Этому алгоритму можно поставить в соответствие *дерево сортировки,* имеющее следующую структуру. Узлы дерева – точки разветвления – содержат сравнение неких элементов последовательности x_i и x_j. Листья дерева – концевые точки – представляют собой перестановку исходной последовательности, соответствующую отсортированной последовательности. Количество сравнений, которое должен выполнить алгоритм в конкретном случае, равняется количеству рёбер, ведущих от корня дерева сортировки к некоторому его листу. Лучшему случаю соответствует самый короткий путь от корня к листу, худшему – самый длинный, а для подсчёта среднего случая надо разделить суммарную длину всех путей от корня к листьям на количество листьев.

Количество возможных листьев дерева сортировки равно $N!$, где N – число элементов сортируемой последовательности. Пусть K – определённый уровень дерева сортировки, считая от корня (корень – это нулевой уровень). Тогда количество узлов на K-ом уровне равно 2^K. В случае, когда дерево сортировки является идеально сбалансированным, выполняется неравенство

$$2^{K+1} \geq N!$$

Логарифмируя данное неравенство и пользуясь *формулой Стирлинга* $N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$, получим следующую оценку для числа K:

$$K > (N + \frac{1}{2})\log_2 N - 1{,}4427N.$$

Таким образом, нижняя оценка числа сравнений для любого алгоритма сортировки, использующего сравнения, удовлетворяет асимптотике $O(N\log_2 N)$. Следовательно, рассмотренный алгоритм быстрой сортировки является асимптотически оптимальным по числу сравнений. Заметим, что в разд. 14 был рассмотрен алгоритм сортировки слиянием с оценкой сложности $O(N\log_2 N)$. Значит, этот алгоритм также является асимптотически оптимальным.

Используя идею дерева сортировки, легко понять, какие ситуации при быстрой сортировке соответствуют наилучшему и наихудшему случаю. Очевидно, что наименьшее число сравнений будет достигаться, когда в результате очередного вызова `FindPivot` осевой элемент оказывается ровно в середине сортируемого участка. Таким образом,

$$B(N) = (N-1) + B\left(\frac{N}{2}\right) + B\left(\frac{N}{2}\right).$$

Приходим к рекуррентному соотношению

$$\begin{cases} B(N) = (N-1) + 2B\left(\frac{N}{2}\right), \\ B(2) = 1. \end{cases}$$

Его решение было рассмотрено в параграфе, посвященном рекурсивным алгоритмам:

$$B(N) = N\log_2 N - N + 1.$$

Наихудший случай для числа сравнений получается, если каждый вызов `FindPivot` разбивает массив на максимально неравные участки, то есть когда осевой элемент оказывается либо на первой, либо на последней позиции. В этом случае выполняется рекуррентное соотношение

$$\begin{cases} W(N) = (N-1) + W(N-1) + W(1), \\ W(1) = 0. \end{cases}$$

Решение данного соотношения выражается формулой

$$W(N) = \frac{N(N-1)}{2}.$$

Таким образом, в худшем случае алгоритм быстрой сортировки отнюдь не является «быстрым». Он работает с таким же порядком скорости, как, например, алгоритм пузырьковой сортировки.

4. Сортировка подсчетом

Выше было доказано, что любой алгоритм сортировки, основанный на сравнении элементов сортируемой последовательности, в лучшем случае выполняет $O(N\log_2 n)$ сравнений, а значит, и работает не менее $O(N\log_2 n)$ шагов.

Рассмотрим алгоритм, называемый *сортировка подсчётом,* который не использует сравнения элементов и в определённых ситуациях позволяет выполнять сортировку массива длиной N за время, сравнимое с $O(N)$.

Пусть M – сортируемый массив длиной N. Предположим, что элементами массива M являются целые числа, лежащие в диапазоне от 1 до K. Заве-

дём специальный массив подсчётов *C* из *K* элементов. Перед началом сортировки обнулим элементы массива *C*. Сделаем первый проход по массиву *M*, подсчитывая количество различных элементов и запоминая результат в массиве *C*. После подсчёта элементов заполним массив *M*, используя информацию из массива C.

Приведённый алгоритм иллюстрируется следующей программой:

```
procedure CountSort
begin
  for i := 1 to K do
    C[i] := 0;              {1}
  for i := 1 to N do
    inc(C[M[i]]);           {2}
  j := 1;
  for i := 1 to K do
    for t := 1 to C[i] do begin
      M[j] := i;            {3}
      inc(j);               {4}
    end;
end;
```

Проведём анализ данной программы. Оператор `{1}` выполняется *K* раз, оператор `{2}` – *N* раз. Легко понять, что и операторы, отмеченные как `{3}` и `{4}`, в сумме выполнятся ровно *N* раз. Таким образом, время работы алгоритма можно оценить как $O(N+K)$. Следовательно, сортировка подсчётом выполняется за *линейное* (относительно числа элементов исходного массива) время. Платой за подобную «резвость» является наличие дополнительного массива и необходимость операций с ним. Если, например, сортируется массив из 100 чисел, каждое из которых имеет тип `integer`, потребуется дополнительный массив из 4 294 967 296 элементов. В этом случае ни о каком преимуществе по скорости (а тем более по используемой памяти) перед традиционными алгоритмами сортировки говорить не приходится.

17. АЛГОРИТМЫ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

1. Вычисление значения многочлена в точке

Работа с многочленами – традиционная задача вычислительной математики. Она интересна сама по себе, а также как составная часть более сложных задач. Напомним некоторые определения.

Определение 18. *Пусть зафиксировано некое поле F. Выражение*

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_{n-1} x^{n-1} = \sum_{j=0}^{n-1} a_j x^j ,$$

где $a_j, x \in F$, *будем называть многочленом от переменной x над полем F степени не выше n – 1 (если* $a_{n-1} \neq 0$, *то говорят, что степень многочлена равна n – 1).*

В дальнейшем будем считать, что многочлены рассматриваются над полем *C* комплексных чисел . Из определения следует, что многочлен однозначно задаётся набором $(a_0, a_1, ..., a_{n-1})$ своих *коэффициентов.*

Рассмотрим следующую задачу: требуется по известному многочлену $A(x)$ и значению переменной x_0 вычислить значение многочлена $y = A(x_0)$.

Приведём «наивный» алгоритм решения данной задачи:

```
y := a[0];
for j := 1 to n-1 do
  y := y + a[j]*x^j;      {x^j означает возведение в степень j}
```

Нетрудно установить, что в данном алгоритме выполняется $A(n) = n - 1$ сложений и $M(n) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$ умножений. Также очевидно, что данный алгоритм не является оптимальным. Рассмотрим следующую его модификацию, которая позволяет значительно сократить число умножений:

```
y := a[0];
p := x;
for j := 1 to n-1 do
  begin
    y := y + a[j]*p;
    p := p*x
  end;
```

В приведённой программе на каждой итерации цикла выполняется одно сложение и два умножения. Следовательно, общее количество сложений $A(n) = n - 1$, а количество умножений $M(n) = 2(n-1)$.

Результат предыдущего алгоритма по числу умножений также можно улучшить. Запишем многочлен $A(x)$ в следующем виде:

$$A(x) = a_0 + x(a_1 + x(a_2 + ...x(a_{n-2} + xa_{n-1})...)).$$

Начнём вычисление значения многочлена с внутренних скобок, постепенно продвигаясь «наружу». На каждом шаге требуется одно умножение и одно сложение:

```
y := 0;
for j := n-1 downto 0 do
  y := y*x + a[j]
```

Вычисление значения многочлена подобным образом называется *вычислением по схеме Горнера.* Такой алгоритм требует $A(n) = n$ сложений и $M(n) = n$ умножений.

Алгоритм вычисления по схеме Горнера также допускает улучшение. Оно достигается путём предварительной обработки исходного многочлена. Данную обработку можно выполнить как вручную, так и реализовать в виде подпрограммы. Рассмотрим алгоритм обработки подробнее.

Предположим следующее:

1. Исходный многочлен является *унимодальным,* то есть коэффициент при старшей степени многочлена равен 1;

2. Степень многочлена *n*–1 на единицу меньше некоторой степени двойки[1] $(n-1=2^k-1,\ n=2^k)$.

Пусть многочлен $A(x)$ удовлетворяет данным предположения. Запишем $A(x)$ в следующем виде:

$$A(x)=(x^j+b)B(x)+C(x).$$

Здесь показатель степени *j* равен 2^{k-1}, а число *b* равно $a_{j-1}-1$. Подобный выбор показателя степени и константы *b* приводит к тому, что многочлены $B(x)$ и $C(x)$ будут унимодальными и иметь степень $2^{k-1}-1$. Описанное преобразование затем применяется к многочленам $B(x)$ и $C(x)$.

Приведём пример подобного преобразования многочлена:

$$A(x)=x^7+4x^6-8x^4+6x^3+9x^2+2x-3,$$

$$A(x)=(x^4+5)(x^3+4x^2+8)+(x^3-11x^2+2x-37),$$

$$A(x)=(x^4+5)\left[(x^2-1)(x+4)+(x+12)\right]+\left[(x^2+1)(x-11)+(x-26)\right].$$

Подсчитаем, сколько действий требуется для вычисления значения преобразованного многочлена $A(x)$. Количество сложений равно 10, а количество умножений – 3. Кроме этого, необходимы ещё два умножения: одно для того, чтобы получить x^2 из *x*, а другое – для получения x^4 из x^2. Следовательно, всего необходимо 5 умножений.

Нетрудно установить (составив соответствующие рекуррентные соотношения), что в общем случае количество умножений выражается числом $M(N)=\frac{N}{2}+\log_2 N$, а количество сложений – числом $A(N)=\frac{3N-1}{2}$, где *N* – степень многочлена.

Подытожим приведенные способы вычисления значения многочленов в точке x_0 в виде табл. 2 (*N* – степень многочлена).

[1] В некоторых случаях для использования приводимого далее способа вычисления значения многочлена в точке выгодно преобразовать многочлен (добавить нулевые коэффициенты и одночлен старшей степени) для выполнения данных условий.

Таблица 2

Способы вычисления значения многочленов

Способ	**Умножения**	**Сложения**
Модифицированный «наивный»	$2N$	N
Схема Горнера	$N+1$	$N+1$
Предварительная обработка	$\frac{N}{2}+\log_2 N$	$\frac{3N-1}{2}$

2. Умножение многочленов

Рассмотрим задачу умножения двух многочленов. Пусть $A(x)=\sum_{j=0}^{n-1} a_j x^j$ и $B(x)=\sum_{j=0}^{n-1} b_j x^j$ – многочлены степени $n-1$. Их произведением будет многочлен степени $2n-2$. Коэффициенты многочлена $C(x)$ можно вычислить по формуле

$$c_j=\sum_{j=0}^{n-1} a_k b_{j-k}\,.$$

Использование данной формулы предполагает, что векторы коэффициентов многочленов $A(x)$ и $B(x)$, изначально имеющие длину n, дополнены нулевыми элементами до длины $2n-1$ ($a_i=b_i=0$ для $i=n,\ldots,2n-1$).

Запишем алгоритм для вычисления произведения многочленов:

```
for j:= 0 to 2*n - 2 do
  begin
    c[j] := 0;
    for к := 0 to j do
      c[j] := c[j] + a[k]*b[j-k]
end;
```

Данный «наивный» алгоритм требует $1+2+\ldots+2n-1=n(2n-1)=O(n^2)$ умножений. Попытаемся улучшить этот показатель.

Прежде чем описать модифицированный алгоритм умножения многочленов, напомним некоторые сведения из алгебры. Пусть $A(x)$ – произвольный многочлен степени $n-1$. Рассмотрим следующий набор из n точек:

$$\{(x_0,y_0),(x_1,y_1),\ldots,(x_{n-1},y_{n-1})\}.$$

Если в данном наборе все значения x_i различны, то он однозначно определяет некий многочлен степени не выше $n-1$. В свою очередь многочлен $A(x)$ также однозначно определяет некий набор точек, если положить $y_i=A(x_i)$.

Очевидно, что переход от коэффициентов многочлена $A(x)$ к набору его значений требует времени не более, чем $O(n^2)$: при помощи схемы Горнера значение многочлена считается в одной точке за время $O(n)$, а общее число точек равно *n*.

Получение коэффициентов многочлена по набору его значений в точках *(интерполяция)* может быть выполнено по *формуле Лагранжа*:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}.$$

Решение этой задачи также требует времени $O(n^2)$.

Как связаны рассмотренные вопросы с задачей умножения многочленов? Дело в том, что умножение двух многочленов можно выполнить следующим образом. Вначале вычисляются значения многочлена $A(x)$ степени $n - 1$ в $2n - 1$ точках $x_0, x_1, ..., x_{2n-2}$. Получается набор

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{2n-2}, y_{2n-2})\}.$$

Затем вычисляется значение многочлена $B(x)$ степени $n - 1$ в тех же точках $x_0, x_1, ..., x_{2n-2}$, и получается набор

$$\{(x_0, z_0), (x_1, z_1), ..., (x_{2n-2}, z_{2n-2})\}.$$

После этого вторые компоненты каждой пары набора перемножаются (это требует $O(n)$ операций):

$$\{(x_0, y_0 z_0), (x_1, y_1 z_1), ..., (x_{2n-2}, y_{2n-2} z_{2n-2})\}.$$

Этот набор описывает значения многочлена $C(x) = A(x) \cdot B(x)$ в точках $x_0, x_1, ..., x_{2n-2}$. Для получения коэффициентов многочлена $C(x)$ достаточно по набору точек провести интерполяцию.

Количество операций, необходимое для получения произведения многочленов, можно уменьшить, если научиться эффективно вычислять значения многочленов и производить интерпретацию в определённом наборе точек. Алгоритм, который будет рассмотрен далее, строит набор значений многочлена в точках, являющихся комплексными корнями из единицы.

Определение 19. *Комплексное число $\omega \in C$ называется корнем n-й степени из единицы, если выполняется равенство*

$$\omega^n = 1.$$

Значение $\omega_n = e^{\frac{2\pi i}{n}}$ называется главным корнем n-й степени из единицы.

Корни *i*-й степени из единицы обладают следующими свойствами, приводимыми без доказательств:

1. $\omega_n^k = \omega_{dn}^{dk}$.

2. Если *n* – чётное число, то $\omega_n^{n/2} = \omega_2 = -1$.

3. Если *k* не кратно *n,* то $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$.

4. Если *n* – чётное число, то, возведя в квадрат все *n* корней *n*-й степени из единицы, получим $\frac{n}{2}$ корней $\frac{n}{2}$-й степени из единицы, причём каждый корень будет получен два раза.

Определение 20. *Пусть* $A(x)$ *– многочлен,* $a = (a_0, a_1, ..., a_{n-1})$ *– вектор его коэффициентов. Вектор* $y = (y_0, y_1, ..., y_{n-1})$*, элементы которого вычисляются по формуле*

$$y_k = A(\omega_n^k),$$

где ω_n *– главный корень n-й степени из единицы, называется дискретным преобразованием Фурье вектора a.*

В дальнейшем для дискретного преобразования Фурье вектора *a* будет использоваться запись

$$y = DFT_n(a).$$

Использовав описанную выше идею вычисления произведения многочленов $A(x)$ и $B(x)$, можно записать, что

$$c = DFT_{2n-1}^{-1}(DFT_{sn-1}(a) \cdot DFT_{sn-1}(b)).$$

В данной формуле *a, b, c* – это векторы коэффициентов многочленов $A(x)$, $B(x)$ и $C(x)$ соответственно, DFT^{-1} – преобразование, обратное дискретному преобразованию Фурье, точка обозначает поэлементное умножение двух векторов. Таким образом, для эффективного умножения многочленов требуется эффективный алгоритм, выполняющий дискретное преобразование Фурье.

Опишем алгоритм, называемый *быстрым преобразованием Фурье.* Данный алгоритм применим, если длина вектора, подвергаемого преобразованию, является степенью двойки[1]. Этот алгоритм относится к классу алгоритмов «разделяй и властвуй». Заметим, что для любого многочлена $A(x)$ справедливо тождество

$$y = A(x) = A^0(x^2) + x \cdot A^1(x^2),$$

[1] В противном случае вектор дополняется нулевыми элементами.

где $A(x) = a_0 + a_2 x + a_4 x^2 + ...a_{n-2} x^{n-2}$, $A^1(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{n-1}$.

В алгоритме быстрого преобразования Фурье в исходном массиве *a* выделяются два подмассива с чётными и нечётными элементами, затем алгоритм рекурсивно вызывается для них, после этого результаты двух вызовов комбинируются в окончательный ответ.

```
function RFFT(a);
// входной параметр - вектор a, результат - вектор y=DFT(a)
n := length(a)          // считаем количество элементов в a
if n = 1 then return a // DFT вектора длиной 1 - это сам вектор
else begin
  // разделяем массив a
  a0 := [a[0], a[2], ..., a[n-2]];
  a1 := [a[1], a[3], ..., a[n-1]];

  // рекурсивный вызов RFFT
  y0 := RFFT(a0);  {*}
  y1 := RFFT(a1);

  // объединяем в окончательный результат
  w := 1;
  wn := e^((2*Pi*i)/n);     // wn - главный комплексный корень
  for k := 0 to n-1 do
    begin
      y[к] := y0[к] + w * y1[k];
      w := w * wn
    end
end;
```

Как видим, оператор, помеченный `{*}`, вычисляет вектор y^0, у которого $y_k^0 = A^0(\omega_{\frac{n}{2}}^k)$. Однако, используя свойство комплексных корней из единицы, можно записать

$$y_k^0 = A^0(\omega_{\frac{n}{2}}^k) = A^0(\omega_n^{2k}).$$

Аналогичные рассуждения применяются к вектору y^1. Количество итераций в цикле построения вектора *y* можно уменьшить в два раза, если использовать свойство комплексных корней $\omega_n^k = -\omega_n^{k+\frac{n}{2}}$ (справедливое при чётных *n*):

```
for k := 0 to (n/2)-1 do
  begin
    y[k] := y0[k] + w * y1[k];
    y[k+(n/2)] := y0[k] - w * y1[k];
    w := w * wn
  end
```

Для алгоритма *RFFT* справедливо следующее рекуррентное соотношение, выражающее общее число операций:

$$\begin{cases} T(n) = 2T(n/2) + bn, \\ T(1) = c. \end{cases}$$

Здесь b и c – некоторые константы. Как было показано в разд. 14, решение данного соотношения удовлетворяет оценке $T(n) = O(n \log_2 n)$. Таким образом, алгоритм *RFFT* позволяет провести вычисление значений многочлена в n точках, являющихся комплексными корнями n-й степени из единицы, за время $T(n) = O(n \log_2 n)$.

Покажем, что задача интерполяции многочлена по вычисленным значениям также имеет сложность $O(n\log_2 n)$. Запишем дискретное преобразование Фурье в матричной форме:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Квадратную матрицу в правой части данного равенства обозначим как V_n. Если считать, что индексы элементов данной матрицы изменяются от 0 до $n - 1$, то элемент υ_{ij} равен ω_n^{ij}.

Обратное преобразование Фурье, которое позволит найти коэффициенты многочлена по коэффициентам Фурье, может быть записано в матричной форме как

$$a = V_n^{-1}(y).$$

Теорема 12. *Элемент с индексом* (j, i) *матрицы* V_n^{-1} *равен* $\dfrac{\omega_n^{-ij}}{n}$.

Доказательство. Пусть утверждение теоремы выполняется. Рассмотрим матричное произведение $V_n^{-1} \cdot V_n$ и элемент полученной матрицы с индексом (j, j'):

$$[V_n^{-1} \cdot V_n]_{jj'} = \sum_{i=0}^{n-1} \left(\frac{\omega_n^{-ij'}}{n} \right) \omega_n^{ij'} = \sum_{i=0}^{n-1} \frac{\omega_n^{i(j'-j)}}{n}.$$

Если $j = j'$, то последняя сумма равна 1. Если $j \neq j'$, то по свойству комплексных корней n-й степени из единицы данная сумма равна 0. Таким образом, матрица $V_n^{-1} \cdot V_n$ является единичной матрицей. Значит, матрица V_n^{-1}, эле-

мент которой с индексом (j, i) равен $\frac{\omega_n^{-ij}}{n}$, действительно является обратной матрицей к матрице V_n. □

Зная структуру матрицы V_n^{-1}, легко записать формулу для получения элемента вектора a:

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{i(j'-j)} .$$

Для вычисления вектора a можно использовать алгоритм *RFFT* с небольшими модификациями: заменить ω_n на ω_n^{-1}, в конце разделить элементы полученного вектора на n. Следовательно, в данном случае задача интерполяции многочлена может быть выполнена за время $O(n\log_2 n)$.

Подводя итог, запишем схему алгоритма, выполняющего умножение двух многочленов.

1. Исходные данные: вектор A длиной K коэффициентов первого многочлена, вектор B длиной M коэффициентов второго многочлена. Требуемый результат: вектор C длиной $K + M - 1$ коэффициентов произведения многочленов.

2. Найти число N такое, что $N = 2^P \geq K+M-1$. Дополнить векторы A и B нулевыми элементами до длины N.

3. Выполнить преобразование $\widetilde{A} = DFT_N(A)$, $\widetilde{B} = DFT_N(B)$.

4. Найти вектор C, выполнив почленное умножение векторов A и B.

5. Выполнить преобразование $C = DFT_N^{-1}(\widetilde{C})$.

6. При необходимости «урезать» вектор C до длины $K + M - 1$, убрав старшие коэффициенты.

В настоящее время используются алгоритмы, позволяющие получить дискретное преобразование Фурье и не применяющие рекурсию (они имеют лучшую константу в асимптотике $O(n\log_2 n)$, а также алгоритмы для преобразования Фурье в кольце вычетов по модулю некоторого простого числа (их применение оправдано, если коэффициенты исходного многочлена – целые числа)).

3. Алгоритм Винограда для умножения матриц

В разд. 14 был рассмотрен алгоритм Штрассена умножения матриц, имеющий сложность $O(N^{\log_2 7})$. Однако использование данного алгоритма выдвигает довольно жёсткие требования к исходным матрицам (квадратные, размер является степенью двойки), а также требует довольно сложной программной реализации.

Математик Виноград предложил свой алгоритм умножения матриц. Как будет показано далее, этот алгоритм работает за время $O(N^3)$, но он выполняется быстрее, чем стандартные алгоритмы умножения.

Идею алгоритма Винограда рассмотрим на примере. Пусть требуется умножить матрицу размером 2 на 4 на матрицу размером 4 на 3:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{pmatrix}.$$

Запишем выражение для вычисления элемента c_{11} в явной форме:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}.$$

Выполним преобразование данного выражения следующим образом:

$$c_{11} = (a_{11} + b_{21})(b_{11} + a_{12}) + (a_{13} + b_{41})(b_{31} + a_{41}) - a_{11}a_{12} - b_{11}b_{21} - a_{13}a_{14} - b_{31}b_{41}.$$

На первый взгляд кажется, что число арифметических операций при таком способе вычисления c_{11} только возросло. Однако новый способ позволяет выполнить предварительно некоторые операции, запомнить их результат, а затем использовать в дальнейшем. Например, выражение для c_{12}:

$$c_{12} = (a_{11} + b_{22})(b_{12} + a_{12}) + (a_{13} + b_{42})(b_{31} + a_{14}) - a_{11}a_{12} - b_{11}b_{22} - a_{13}a_{14} - b_{32}b_{42}$$

содержит слагаемые $-a_{11}a_{12} - a_{13}a_{14}$, которые уже использовались при вычислении c_{11}.

Использовав данную идею, можно предложить следующий алгоритм, оформленный в виде программы:

```
// Умножение A[1..N,1..K] на B[1..K,1..M]
L := K div 2;

// считаем множители для A
for i := 1 to N do
  RF[i] := A[i,1] * A[i,2];
  for j := 2 to L do
    RF[i] := RF[i] + A[i,2j-1] * A[i,2j1];

// считаем множители для B
for i := 1 to M do
  CF[i] := B[1,i] * B[2,i];
  for j := 2 to L do
    CF[i] := CF[i] + B[2j-1,i] * B[2j,i];

// вычисляем C = A * B
for i := 1 to N do
  for j := 1 to M do
    C[i,j] := - (RF[i] + CF[j]);
    for p := 1 to L do
    C[i,j] := C[i,j]+(A[i,2p-1]+B[2p,j])*(A[i,2p] + B[2p-1,j])
```

```
// если K - нечетное, требуется добавить слагаемые
if K mod 2 = 1 then
                  for i := 1 to N do
                    for j := 1 to M do
                      C[i,j] := C[i,j] + A[i,K]*B[K,j])
```

Подсчитаем количество арифметических операций, выполняемых в данном алгоритме в случае чётной общей размерности перемножаемых матриц.

Таблица 3

Количество арифметических операций в алгоритме Винограда

Действие	**Умножения**	**Сложения**
Предварительная обработка A	NL	$N(L-1)$
Предварительная обработка B	ML	$M(L-1)$
Вычисление C	NML	$NM(3L+2)$
Общая сумма	$\frac{(NM+N+M)K}{2}$	$\frac{(N+M)(K-2)+NM(3K+4)}{2}$

Как видно из табл. 3, в алгоритме Винограда выполняется почти в два раза меньше умножений, чем в стандартном алгоритме умножения матриц. Правда, в алгоритме увеличивается количество сложений и требуется дополнительная память для хранения промежуточных результатов. Однако, как показывает практика, применение алгоритма Винограда оправдано в подавляющем большинстве случаев.

ЛИТЕРАТУРА

1. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман ; пер. с англ. – М. : Мир, 1979. – 535 с.

2. Верещагин, Н. К. Вычислимые функции / Н. К. Верещагин, А. Шень. – М. : МЦНМО, 1999. – 176 с.

3. Грин, Д. Математические методы анализа алгоритмов / Д. Грин, Д. Кнут ; пер. с англ. – М. : Мир, 1987. – 120 с.

4. Кнут, Д. Искусство программирования. В 3 т. / Д. Кнут ; пер. с англ. – М. : Вильямс, 2000. – Т. 1 – 720 с. ; Т. 2 – 832 с. ; Т. 3 – 832 с.

5. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест ; пер. с англ. – М. : Вильямс, 2005. – 1290 с.

6. Непейвода, Н. Н. Прикладная логика / Н. Н. Непейвода. – Новосибирск : НГУ, 2000. – 491 с.

Св. план 2007, поз. 67

Учебное издание

Волосевич Алексей Александрович

ОСНОВЫ ТЕОРИИ АЛГОРИТМОВ

Учебно-методическое пособие
по курсу «Теория алгоритмов»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Редактор Т. П. Андрейченко
Корректор М. В. Тезина

Подписано в печать 12.12.2007. Формат 60×84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая Усл. печ. л. 3,37.
Уч.-изд. л. 3,0. Тираж 125 экз. Заказ 425.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
ЛИ № 02330/0056964 от 01.04.2004. ЛП № 0233/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6