ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САХАЛИНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИЗБРАННЫЕ ВОПРОСЫ ТЕОРИИ АЛГОРИТМОВ

Учебно-методическое пособие

Составители:
О. О. Меркулова, А. Б. Никитина, О. А. Фёдоров

Рекомендовано Дальневосточным региональным учебно-методическим центром (ДВ РУМЦ) в качестве учебного пособия для студентов направлений подготовки 01.03.02 «Прикладная математика и информатика», 44.03.05 «Педагогическое образование (с двумя профилями подготовки)», профиль «Математика и физика» вузов региона.

Южно-Сахалинск
СахГУ
2018

УДК 510
ББК 22.12
И328

Печатается по решению учебно-методического совета Сахалинского государственного университета, 2015 г.

Рецензенты:
А. В. Доманский, *доктор физ.-мат. наук ИМГиГ ДВО РАН;*
А. Ф. Гулевская, *канд. пед. наук, доцент кафедры математики СахГУ.*

И328 **Избранные вопросы теории алгоритмов :** учебно-методическое пособие / сост.: О. О. Меркулова, А. Б. Никитина, О. А. Фёдоров. – Южно-Сахалинск : СахГУ, 2018. – 112 с.
ISBN 978-5-88811-577-0

Настоящее учебно-методическое пособие предназначено для студентов образовательных организаций высшего образования, изучающих в том или ином объеме теорию алгоритмов. Пособие рекомендовано для математических и педагогических направлений, в частности для направлений 01.03.02 «Прикладная математика и информатика», 44.03.05 «Педагогическое образование (с двумя профилями подготовки)».

Данное учебно-методическое пособие является оптимальным справочным материалом, с помощью него студенты смогут самостоятельно изучить избранные вопросы теории алгоритмов и подготовиться к сдаче зачета или экзамена.

УДК 510
ББК 22.12

ISBN 978-5-88811-577-0

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В учебно-методическом пособии «Избранные вопросы теории алгоритмов» представлены разделы, традиционно изучаемые в курсе теории алгоритмов: машины Тьюринга, нормальные алгорифмы Маркова, рекурсивные функции. Рассмотрены вопросы интуитивного и формального определения алгоритмов. Даются некоторые избранные вопросы: изучения сложности и нумерации алгоритмов, рассмотрения алгоритмически неразрешимых проблем, конструирования машин Поста и машин с неограниченными регистрами. Части книги взаимосвязаны друг с другом и снабжены большим количеством примеров, помогающих усвоить и закрепить излагаемый материал.

Учебно-методическое пособие включает лекционный и практический материал и может служить основой для проведения занятий по теории алгоритмов на направлениях «Прикладная математика и информатика», «Педагогическое образование (с двумя профилями – математика и физика)». Пособие предназначено для организации аудиторной и внеаудиторной (самостоятельной) работы студентов. Изложение теоретического материала по всем частям сопровождается рассмотрением большого количества примеров и ведется на доступном по возможности строгом языке.

Существуют различные учебные пособия по теории алгоритмов, но большинство из них не обладают системным подходом к изучению материала и достаточной доступностью изложения. Данное издание – одна из попыток создания оптимального пособия, которое должно помочь студенту в самостоятельной и более углубленной работе по теории алгоритмов. Авторы надеются, что данное пособие будет полезно не только студентам, но и преподавателям.

Цель пособия – на конкретных примерах просто и доходчиво рассказать об основах теории алгоритмов.

Опираясь на опыт работы со студентами специальностей и направлений «Прикладная математика и информатика», «Педагогическое образование (с двумя профилями)», старший преподаватель кафедры математики Меркулова Ольга Олеговна систематизировала теоретический материал и предложила различные варианты практических и самостоятельных заданий, которые нашли отражение в частях I–IV.

В работе над пособием также принимали участие кандидат педагогических наук Федоров Олег Анатольевич и кандидат педагогических наук Никитина Алла Борисовна, с их помощью была написана часть V.

ЧАСТЬ I. ОСНОВНЫЕ ПОНЯТИЯ: АЛГОРИТМЫ

Лекционное занятие № 1 «Алгоритмы»

Теория алгоритмов – это наука, изучающая общие свойства и закономерности алгоритмов, формальные модели их представления.

План занятия:

1. ***Алгоритмы в нашей жизни.***
2. ***Интуитивное понятие алгоритма, характерные черты.***
3. ***Виды алгоритмов.***
4. ***Способы описания алгоритмов.***
5. ***Формализация понятия алгоритма.***
6. ***Современное состояние теории алгоритмов.***

1. Алгоритмы в нашей жизни

Понятие алгоритма стихийно сформировалось в древнейшие времена в Египте, Вавилоне, Греции. Различные вычислительные процессы чисто механического характера, с помощью которых искомые величины ряда задач вычислялись последовательно из данных исходных величин по определенным правилам и инструкциям, получили название алгоритмов или – по другим написаниям – алгорифмов.

Рис. 1.

Слово «алгоритм» происходит от имени узбекского математика Мухаммеда аль-Хорезми (783–ок. 850 гг., рис. 1). В латинских переводах с арабского его арифметического трактата «ALgoritmi de numero Indorum» («Алгоритми о счете индийском»), где он излагал правила действия над числами в десятичной системе счисления, имя аль-Хорезми транскрибировалось как algorismi.

В 1684 году Готфрид Лейбниц в сочинении «Nova Methodvs pro maximis et minimis, itemque tangentibus…» впервые использовал слово «алгоритм» («algorithmo») в широком смысле: как систематический способ решения проблем дифференциального исчисления.

В XVIII веке в одном из германских математических словарей, изданном в г. Лейпциге в 1747 году, термин «algorithmus» был объяснен как понятие о четырех арифметических операциях. Но такое объяснение было не единственным, в часности выражение «algorithmus infinitesimalis» применялось к способам выполнения действий с бесконечно малыми величинами.

Если говорить о том, в каком веке русские ученые познакомились с понятием «алгоритм», то следует отметить, что ни в знаменитом словаре В. И. Даля, ни, спустя сто лет, в «Толковом словаре русского языка» под редакцией Д. Н. Ушакова (1935 г.) слово «алгоритм» не встречается. Зато его можно найти в популярном дореволюционном «Энциклопедическом словаре братьев Гранат» и в первом издании Большой советской энциклопедии, изданной в

1926 году. И там, и там оно трактуется одинаково: как правило, по которому выполняется то или иное арифметическое действие в десятичной системе счисления.

В третьем издании Большой советской энциклопедии (1969 г.) «алгоритм» уже характеризуется как одна из основных категорий математики, «не обладающих формальным определением в терминах более простых понятий и абстрагируемых непосредственно из опыта». Одновременно с развитием понятия «алгоритма» постепенно происходила и его экспансия из чистой математики в другие сферы. В 1985 году алгоритм вошел во все учебники информатики и обрел новую жизнь.

Многочисленные и разнообразные алгоритмы окружают нас буквально во всех сферах жизни и деятельности. Многие наши действия доведены до бессознательного автоматизма, мы порой и не осознаем, что они регламентированы определенным алгоритмом – четкой системой инструкций.

Пример 1: рецепт приготовления манной каши (к 0,2 литра холодного молока добавить две столовые ложки манной крупы, помешивая, довести до кипения и варить десять минут). Автоматизм выполнения данного действия зачастую не позволяет молодым родителям осознавать его алгоритмическую сущность.

Пример 2: прием таблеток по рецепту врача (три таблетки в день, по одной перед приемом пищи).

Пример 3: огромное количество алгоритмов встречается при изучении математики буквально с первых классов школы – это, прежде всего, алгоритмы выполнения четырех алгоритмических действий над различными числами – N, Z, Q, R, C.

Пример 4: что нужно сделать, чтобы из одной десятичной дроби вычесть другую?

Алгоритм решения:

1) уравнять число знаков после запятой в уменьшаемом и вычитаемом;

2) записать вычитаемое под уменьшаемым так, чтобы запятая оказалась под запятой;

3) произвести вычитание так, как вычитают N;

4) поставить в полученной разности запятую под запятыми в уменьшаемом и вычитаемом.

Примерами алгоритмов также являются:

1) правило отыскания наибольшего общего делителя;
2) правило извлечения квадратного корня;
3) правило отыскания решений квадратного уравнения;
4) алгоритмы геометрических построений с помощью циркуля и линейки (деление пополам отрезка и угла, опускание и восстановление перпендикуляра, проведение параллельных прямых);
5) алгоритмы вычислений площадей и объемов различных геометрических фигур и тел;
6) правила вычисления определителей различных порядков, рангов матриц и др.

2. Интуитивное понятие алгоритма, характерные черты

Одной из причин интуитивного объяснения понятия алгоритма является разнообразие объектов, с которыми работают алгоритмы. В вычислительных алгоритмах объектами являются числа, в алгоритме шахматной игры – фигуры и их позиции на шахматной доске, в алгоритме формирования текста – слова некоторого языка и правила переноса слов.

Определение 2.1 (интуитивное): алгоритм – это точное предписание о выполнении в определенном порядке некоторой системы операций для решения всех задач одного и того же типа.

Алгоритм предполагает наличие **начальных**, или **исходных, данных** и в результате применения приводит к получению определенного **результата**.

Определение 2.2 (по А. Н. Колмогорову): алгоритм – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 2.3 (по А. А. Маркову): алгоритм – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Определение 2.4: исполнитель алгоритма – это некоторая абстрактная или реальная (техническая, биологическая, биотехническая или иная) система, способная выполнить действия, предписываемые алгоритмом.

Исполнителя алгоритма характеризуют:

1) среда (например, бесконечное клеточное поле);
2) элементарные действия (некоторые шаги);
3) система команд (выполняются команды из некоторого строго заданного списка);
4) отказы (состояния, возникающие при недопустимом состоянии среды).

Зачастую исполнитель ничего не знает о цели алгоритма и выполняет все полученные команды, не задавая лишних вопросов «зачем» и «почему».

Характерные черты алгоритма:

1. Дискретность – применение каждого алгоритма осуществляется путем выполнения дискретной цепочки (последовательности) неких элементарных действий. Эти действия называются шагами, а процесс их выполнения называют алгоритмическим процессом.

2. Детерминированность (определенность) – на каждом шаге алгоритма однозначно указывается, какое действие необходимо произвести. Предписания алгоритмов должны быть точны и отчетливы, чтобы не допустить двусмысленных толкований и произвола со стороны исполнителя, их с равным успехом должны выполнять совершенно разные исполнители.

3. Элементарность – закон получения последующей системы величин из предшествующей системы должен быть простым и локальным.

4. Массовость – каждый алгоритм предназначен для решения большого числа однотипных задач.

Алгоритм, решающий массовую проблему, – это общий метод, позволяющий решить любую задачу этой проблемы. Понятие массовой проблемы является центральным понятием теории алгоритмов. Например, задача нахождения среднего арифметического чисел 4 и 6 есть единичная проблема, а задача нахождения среднего арифметического произвольных натуральных чисел m и n – уже проблема массовая.

5. Реализуемость (потенциальная осуществимость) – алгоритм предполагает наличие механизма реализации, который по описанию алгоритма порождает процесс вычисления на основе исходных данных.

6. Результативность – последовательный процесс построения величин должен быть конечным и давать результат.

Говорят, **что алгоритм применим к допустимым исходным данным**, если с его помощью можно получить искомый результат. Если результат получить нельзя, хотя исходные данные допустимы, то **алгоритм к ним не применим.**

Неприменимость алгоритма к допустимым начальным данным заключается в том, что алгоритмический процесс либо никогда не закончится (при этом говорят, что процесс бесконечен), либо его выполнение безрезультативно оборвется.

Пример 5: бесконечный алгоритмический процесс – 20 : 3 = 6, 66666…. .

Пример 6: безрезультативно обрывающийся процесс – 7 : 0 – деление невозможно, процесс натолкнулся на препятствие и безрезультативно оборвался.

Алгоритм – это единственный метод решений определенного класса однотипных задач, обладающих вышеперечисленными свойствами и оперирующих с конструктивными объектами. Исходные объекты, промежуточные и окончательные результаты должны быть конструктивными объектами.

Определение 2.5 (интуитивное): конструктивный объект – это объект, который построен (сконструирован) из некоторых элементарных неделимых элементов фиксированного точно очерченного конечного множества по определенным правилам, таким образом, что построение объекта может быть полностью описано некоторым текстом на подходящем языке.

Например, N – это конструктивный объект, любое N можно закодировать с помощью слов в алфавите $A = \{0, 1, 2, \dots, 9\}$, Q – пример неконструктивного объекта (при описании такого числа может потребоваться бесконечное число знаков после запятой).

3. Виды алгоритмов

Алгоритмы в зависимости от цели, начальных условий задачи, путей ее решения, определения действий исполнителя подразделяются на:

- **механические алгоритмы**, или иначе **детерминированные, жесткие** (например, алгоритм работы некой машины, двигателя и т. п.). Механический алгоритм задает определенные действия, обозначая их в единственной и достоверной последовательности, обеспечивая тем самым однозначный требуемый или искомый результат;
- **гибкие алгоритмы**, например **стохастические,** то есть вероятностные и эвристические. Гибкий алгоритм дает программу решения задачи несколькими путями или способами, приводящими к вероятному достижению результата. В эвристическом алгоритме достижение конечного результата программы действий однозначно не предопределено, также как не обозначена вся последовательность действий, не выявлены все действия исполнителя. К эвристическим и вероятностным алгоритмам часто относят различные инструкции и предписания.

Иногда исходный алгоритм можно разбить на отдельные связанные составляющие, называемые шагами, или **частными алгоритмами.**

Различают четыре основных типа частных алгоритмов:

- линейный алгоритм;
- алгоритм с ветвлением;
- циклический алгоритм;
- вспомогательный, или подчиненный, алгоритм.

Линейный алгоритм – набор команд (указаний), выполняемых последовательно друг за другом.

Алгоритм с ветвлением – алгоритм, содержащий хотя бы одно условие, в результате проверки которого может осуществляться разделение на несколько параллельных ветвей алгоритма.

Циклический алгоритм – алгоритм, предусматривающий многократное повторение одного и того же действия (одних и тех же операций) над новыми исходными данными.

Вспомогательный (или **подчиненный) алгоритм** – алгоритм, ранее разработанный и целиком используемый при алгоритмизации конкретной задачи.

Особняком в литературе по теории алгоритмов стоят **рекурсивные алгоритмы**, вызывающие сами себя до тех пор, пока не будет достигнуто некоторое условие-возвращение.

Определение 3.6. Рекурсия – метод определения функции через ее предыдущие и ранее определенные значения, а также способ организации вычислений, при котором функция вызывает сама себя с другим аргументом.

4. Способы описания алгоритмов

Форма записи алгоритма зависит, прежде всего, от назначения (природы) самого алгоритма, а также от того, кто (что) будет исполнителем алгоритма.

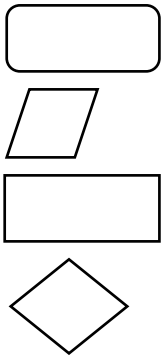
Формы записи алгоритма:

- словесная или вербальная (языковая, формульно-словесная);
- схематическая – графическая (блок-схемы) и структурограммы (схемы Насси–Шнейдермана);
- программная (в том числе псевдокод).

Словесный способ описания алгоритмов – это, по существу, обычный язык, но с тщательным отбором слов и фраз, не допускающий излишеств, двусмысленностей и повторений (обычно дополняется математическими обозначениями и некоторыми специальными соглашениями).

Графический способ описания алгоритмов – это способ представления алгоритма с помощью общепринятых графических фигур, каждая из которых описывает один или несколько шагов алгоритма. Графическое изображение алгоритма весьма широко используется ввиду того, что зрительное восприятие облегчает процесс написания программы, ее корректировки при возможных ошибках, осмысливание процесса обработки информации.

Структурная блок-схема, граф-схема алгоритма – графическое изображение алгоритма в виде схемы связанных между собой с помощью стрелок (линий перехода) блоков – графических символов, каждый из которых соответствует одному шагу алгоритма. Внутри блока дается описание соответствующего действия. В блок-схеме можно использовать строго определенные типы блоков, представленные на рисунке 2.



– начало или конец описания алгоритмов;

– ввод исходных данных или вывод результатов;

– блок арифметических или других действий;

– блок проверки условий, от которых зависит выбор направления алгоритма.

Рис. 2.

Диаграммы Насси–Шнейдермана (N–S-диаграммы) – это графический способ представления структурированных алгоритмов и программ, разработанный в 1972 году американскими аспирантами Беном Шнейдерманом и Айзеком Насси. В Германии N–S-диаграммы применяют при документировании программ, это обусловлено требованиями государственного стандарта этой страны.

N–S-диаграммы имеют ряд преимуществ перед блок-схемами: запись является более компактной (в первую очередь за счет отсутствия стрелок между элементами), соблюдены принципы структурного программирования (при использовании блок-схем можно случайно получить неструктурированный алгоритм). Диаграммы строятся по принципу пошаговой детализации – изначально диаграмма представляет собой один прямоугольник (исходная задача), затем в нем рисуется некоторая структура управления, в которой имеется несколько прямоугольников (подзадач исходной задачи, рис. 3).

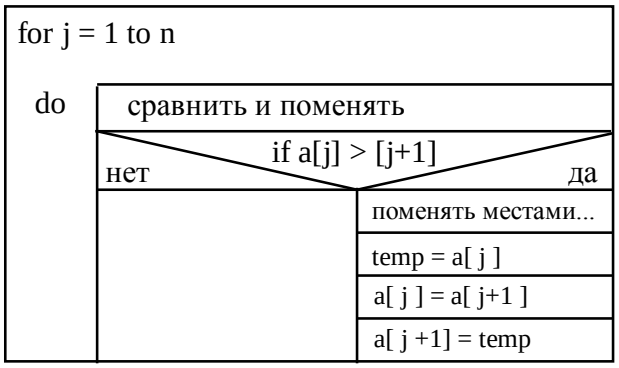


Рис. 3.

Описание алгоритмов с помощью программ – алгоритм, записанный на языке программирования, называется **программой.**

Псевдокод – компактный язык описания алгоритмов, опускающий несущественные подробности и специфический синтаксис. Главная цель использования псевдокода – обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования.

В настоящее время в мире существует несколько сотен реально используемых языков программирования. Для каждого из них существует своя область применения.

В зависимости от степени детализации предписаний обычно определяется уровень языка программирования – чем меньше детализация, тем выше уровень языка. По этому критерию можно выделить следующие уровни языков программирования:

- машинные;
- машинно-ориентированные (ассемблеры);
- машинно-независимые (языки высокого уровня).

Машинные и машинно-ориентированные языки – это языки низкого уровня, требующие указания мелких деталей процесса обработки данных.

Языки высокого уровня имитируют естественные языки, используя некоторые слова разговорного языка и общепринятые математические символы. Языки высокого уровня обычно подразделяются на:

- **процедурные (алгоритмические)** (Basic, Pascal, C и др.), которые предназначены для однозначного описания алгоритмов, для решения задачи процедурные языки требуют в той или иной форме явно записать процедуру ее решения;
- **логические** (Prolog, Lisp и др.), которые ориентированы не на разработку алгоритма решения задачи, а на систематическое и формализованное описание задачи с тем, чтобы решение следовало из составленного описания;

- **объектно-ориентированные** (Object Pascal, C++, Java и др.), в основе которых лежит понятие объекта, сочетающего в себе данные и действия над ними. Программа на объектно-ориентированном языке, решая некоторую задачу, описывает часть мира, относящуюся к этой задаче.

Любой алгоритмический язык образуют три его составляющие:

– **алфавит** – это фиксированный для данного языка набор основных символов, то есть «букв алфавита», из которых должен состоять любой текст на этом языке, никакие другие символы в тексте не допускаются;

– **синтаксис** – это правила построения фраз, позволяющие определить, правильно или неправильно написана та или иная фраза. Синтаксис языка представляет собой набор правил, устанавливающих, какие комбинации символов являются осмысленными предложениями на этом языке;

– **семантика** определяет смысловое значение предложений языка. Являясь системой правил истолкования отдельных языковых конструкций, семантика устанавливает, какие последовательности действий описываются теми или иными фразами языка и в конечном итоге какой алгоритм определен данным текстом на алгоритмическом языке.

5. Формализация понятия алгоритма

До конца первой трети XX века математики довольствовались интуитивным представлением о понятии «алгоритм», употребляя термин лишь в связи с тем или иным конкретным действием. Именно тогда были сформулированы алгоритмические проблемы, положительное решение которых представлялось маловероятным.

Пример 7: указать способ, согласно которому для любой предикатной формулы за конечное число действий можно выяснить, является ли она тождественно истинной или нет.

Пример 8: разрешимо ли в целых числах любое диафантово уравнение (например, $3xy + 4y + 5z^2 - 6 = 0$)?

Для решения этих задач найти алгоритмы не удалось, математики предположили, что такие алгоритмы не существуют. В связи с этим возникло естественное желание доказать, что таких алгоритмов действительно нет. Тогда встал вопрос: отсутствие чего надо доказать? Ведь для того, чтобы доказать отсутствие алгоритма, надо иметь общее определение понятия алгоритма, пригодное для формулировки решений большого числа разнообразных задач, а такого определения в математике не было.

В середине 30-х годов XX века сразу несколько определений понятия «алгоритм» было предложено рядом математиков почти одновременно.

В подходе к определению понятия «алгоритм» можно выделить три основных направления.

Первое направление связано с машинной математикой. Здесь сущность понятия алгоритма раскрывалось путем рассмотрения процессов, осуществляемых в машине. Впервые это было сделано А. Тьюрингом, который предложил самую общую и вместе с тем самую простую концепцию вычислительной машины. Ее описание было дано в 1937 году. А. Тьюринг исходил из общей идеи работы машины как работы вычислителя, оперирующего в соответствии с некоторым строгим предписанием.

Второе направление связано с уточнением понятия эффективно вычислимой функции. Этим направлением занимались А. Чёрч, К. Гедель, К. Клини и др. В результате их работы

был выделен класс так называемых частично рекурсивных функций, имеющих строгое математическое определение. Анализ идей, приведших к этому классу функций, дал ученым возможность высказать гипотезу о том, что класс эффективно вычислимых функций совпадает с классом частично рекурсивных функций.

Третье направление связано с понятием нормальных алгорифмов, введенным и разработанным советским математиком А. А. Марковым. Нормальные алгорифмы, по Маркову, представляют собой некоторые правила по переработке слов в каком-либо алфавите.

Впоследствии было доказано, что различные математические определения алгоритма в некотором смысле эквивалентны: вычисляют одно и то же множество функций.

Формализация понятия алгоритма – это описание стандартной универсальной формы записи информации и ограниченного четко определенного набора простейших элементарных действий человека по ее переработке, необходимых и достаточных для реализации любого содержательно описанного алгоритма.

Тезис формализации: любой содержательно описанный алгоритм может быть формализован в рамках, используемых в теории алгоритмов, строгих математических определений понятия алгоритма.

Любой из известных математике алгоритмов, а таких алгоритмов за многие годы существования математики накопилось достаточно много, может быть формализован.

Первоначально теория алгоритмов возникла в связи с внутренними потребностями математики: математическая логика, алгебра, геометрия и математический анализ остаются и сегодня одной из основных областей приложения теории алгоритмов. Однако со временем теория алгоритмов оказалась тесно связанной с рядом областей лингвистики, экономики, физиологией мозга, психологией и др. Так, достаточно старые вопросы о том, является ли мозг сложной машиной, как он работает, в чем состоит различие между трудом творческим и механическим, теперь часто формализуют следующим образом: можно ли построить машину Тьюринга, перерабатывающую поступающую информацию так же, как мозг какого-либо животного; можно ли отождествлять труд механический с трудом согласно заданному алгоритму.

4. Современное состояние теории алгоритмов

В настоящее время теория алгоритмов – это раздел математики, который изучает общие свойства алгоритмов. Различают качественную и метрическую теорию алгоритмов.

Основной проблемой **качественной теории алгоритмов** является проблема построения алгоритма, обладающего заданными свойствами. Такую проблему называют алгоритмической.

Метрическая теория алгоритмов исследует алгоритм с точки зрения их сложности. Этот раздел теории алгоритмов известен также как алгоритмическая теория сложности.

Сегодня теория алгоритмов развивается, главным образом, по трем направлениям:

- **классическая теория алгоритмов** изучает проблемы формулировки задач в терминах формальных языков, проводит классификацию задач по классам сложности (P, NP и др.);
- **теория асимптотического анализа** алгоритмов рассматривает методы получения асимптотических оценок ресурсоемкости или времени выполнения алгоритмов, в частности, для рекурсивных алгоритмов. Асимптотический анализ позволяет оценить рост потребности алгоритма в ресурсах (например, времени выполнения) с увеличением объема входных данных;

- **теория практического анализа** вычислительных алгоритмов решает задачи поиска практических критериев качества алгоритмов, разработки методики выбора рациональных алгоритмов.

Практическое применение алгоритмов чрезвычайно широко, о чем могут свидетельствовать, например, приведенные ниже сведения:

- Целью проекта по расшифровке генома человека является идентификация всех 100000 генов, входящих в состав его ДНК, определение последовательностей, образуемых 3 000 000 000 базовых пар, из которых состоит ДНК, сортировка этой информации в базах данных и разработка инструментов для ее анализа. Для реализации всех перечисленных этапов нужны сложные алгоритмы.
- Интернет позволяет пользователям в любой точке мира быстро получать доступ к информации и извлекать ее в больших объемах. Управление и манипуляция этими данными осуществляются с помощью хитроумных алгоритмов. В число актуальных задач теории алгоритмов входят: определение оптимальных маршрутов, по которым перемещаются данные, быстрый поиск страниц, на которых находится та или иная информация, и др.
- Электронная коммерция позволяет заключать сделки и предоставлять товары и услуги с помощью электронных технических средств. Для того чтобы она получила широкое распространение, важно иметь возможность защищать такую информацию, как номера кредитных карт, пароли и банковские счета. В число базовых технологий в этой области входят криптография с открытым ключом и цифровые подписи, основанные на численных алгоритмах и теории чисел.

С уверенностью можно говорить о том, что в настоящее время алгоритмы широко распространены не только в математике, но и в обычной жизни и требуют к себе внимательного отношения и тщательного изучения.

Практическое занятие № 2 «Алгоритмы»

1. Напишите алгоритм вычисления частного случая диафантова уравнения с одним неизвестным, позволяющий найти все его целочисленные решения.

2. Запишите алгоритм нахождения минимального числа x среди n чисел $a_1, a_2, \dots, a_n$.

3. Запишите алгоритм Евклида нахождения наибольшего общего делителя двух заданных натуральных чисел a, b (НОД (a, b)). Обоснуйте его.

4. Дано предписание:

(1) Если a $>$ 3, то перейти к 3, иначе к 2.

(2) Присвоить a: $=$ a $+$ 3, перейти к 4.

(3) Присвоить a: $=$ a $-$ 3, перейти к 5.

(4) Если a $<$ 4, то перейти к 2, иначе к 6.

(5) Если a $>$ 4, то перейти к 3, иначе к 6.

(6) Записать полученное значение a, перейти к 7.

(7) Процесс окончен.

Выполнить алгоритм, заданный этим предписанием, для a = – 3; 3; 5.

5. Имеется 15 предметов. В игре участвуют двое: начинающий и противник. Каждый игрок по очереди берет по одному, два или три предмета. Выигрывает тот, кто берет последний предмет. Какой стратегии должен придерживаться начинающий, чтобы всегда выиграть?

6. Найти наименьшее число в следующей последовательности чисел: 51, 25, 35, 79, 13, 26, 65, проиллюстрируйте решение задачи с помощью N–S-диаграммы.

Лекционное занятие № 3 «Алгоритмическая теория множеств»

План занятия:

1. ***Разрешимые множества.***
2. ***Полуразрешимые множества.***
3. ***Перечислимые множества.***
4. ***Равнообъемность понятий полуразрешимости и перечислимости.***
5. ***Теорема о графике.***

1. Разрешимые множества

Теория множеств служит основанием современной математики, поэтому не удивительно, что теория алгоритмов использует понятия множества явно и неявно. Однако это использование существенно различается в математике и в программировании, так как те операции с множествами, которые математик воспринимает как элементарные, для программиста таковыми не являются и часто требуют дополнительных пояснений.

Пример 1: пусть требуется описать алгоритм вычисления функции, определенной на всех натуральных числах n:

$$f(n) = \begin{cases} n + 1, \text{ если n-четно,} \\ n - 1, \text{ в противном случае.} \end{cases}$$

Математики справедливо заметят, что само определение функции $f(n)$ фактически и есть описание искомого алгоритма: получив число n в качестве исходного данного, необходимо проверить, принадлежит ли оно множеству четных чисел, и в зависимости от результата проверки выполнить $n + 1$ или $n - 1$. Программист вынужден проанализировать определение функции более детально и задать существенный дополнительный вопрос: «Как проверять принадлежность числа n множеству всех четных чисел?». Ответ в данном случае прост – надо запрограммировать отдельно функцию вычисления остатка от деления на 2.

Показательность примера не в трудности дополнительного вопроса, а в его необходимости.

Определение 1.1. Пусть M – подмножество типа конструктивных объектов X. **Характеристической функцией** множества M называется функция $\chi_M : X \to \{0,1\}$ с областью определения $D(\chi_M) = X$, заданная следующим образом: $\chi_M(x) = \begin{cases} 1, \text{ если } x \in M, \\ 0, \text{ если } x \in X \backslash M. \end{cases}$

Вопрос 1: все ли подмножества натурального ряда имеют характеристические функции?

Ответ: унарная запись натуральных чисел (число n записывается словом из n единиц) позволяет рассматривать множество всех натуральных чисел как словарное пространство $X = \{1\}^*$. Определение 1.1 сопоставляет каждому подмножеству типа конструктивных объектов его характеристическую функцию. Таким образом, каждое подмножество натурального ряда имеет характеристическую функцию.

Вопрос 2: сколько характеристических функций имеет данное подмножество натурального ряда? Могут ли характеристические функции двух различных подмножеств совпадать, то есть принимать одинаковые значения для всех значений аргумента?

Ответ: определение 1.1 для данного множества M задает его характеристическую функцию единственным образом. Если M_1 и M_2 – два различных подмножества X, то найдется элемент $x \in X$, которым они различаются. Например, $x \in M_1$, но $x \notin M_2$. Тогда значения их характеристических функций на аргументе x также будут различны: $\chi_{M_1}(x) = 1,\ \chi_{M_2}(x) = 0$.

Вопрос 3: можно ли произвольную функцию γ, заданную на всех элементах типа конструктивных объектов X со значениями в множестве $\{0,1\}$, считать характеристической функцией некоторого подмножества X? Как восстановить это подмножество?

Ответ: можно. Достаточно положить $M = \{x \in X | \chi(x) = 1\}$. Тогда χ_M совпадает с γ.

Таким образом, определение характеристической функции устанавливает взаимнооднозначное соответствие между подмножествами типа конструктивных объектов X и булевозначными функциями, определенными на элементах X. Можно увидеть, что при этом соответствии теоретико-множественные операции объединения, пересечения и дополнения переходят в логические операции дизъюнкции, конъюнкции и отрицания, применимые к значениям характеристических функций:

$$\begin{aligned} \chi_{A\cup B}(x) &= \chi_A(x) \vee \chi_B(x), \\ \chi_{A\cap B}(x) &= \chi_A(x) \wedge \chi_B(x), \\ \chi_{X\setminus A}(x) &= \neg\chi_A(x). \end{aligned} \qquad (1)$$

Определение 1.2. Подмножество M типа конструктивных объектов X называется **разрешимым**, если его характеристическая функция вычислима. Алгоритм вычисления характеристической функции часто называют **алгоритмом разрешения**, или **разрешающим алгоритмом** для множества M. Подмножества типа конструктивных объектов, которые не являются разрешимыми, называются **неразрешимыми.**

Замечание: следует отметить, что понятие разрешимости относительно, то есть требует указания того типа конструктивных объектов, о подмножестве которого идет речь.

Вопрос 4: разрешимо ли пустое множество?

Ответ: да. Каждое конечное множество, состоящее из элементов некоторого типа конструктивных объектов X_0, разрешимо как подмножество любого типа конструктивных объектов X, его содержащего. В случае пустого множества в качестве X можно взять любой тип конструктивных объектов, а его характеристическая функция принимает значение 0 для всех $x \in X$. Такую функцию можно вычислить алгоритмом, не читающим входные данные и всегда возвращающим 0 в качестве результата.

Вопрос 5: разрешимо ли множество $\{1, 2, 3\}$?

Ответ: для множества $\{1, 2, 3\}$ естественным типом конструктивных объектов будет словарное пространство $N = \{1\}^*$. Алгоритм вычисления характеристической функции $\chi_{\{1,2,3\}}(x)$ на входе $x \in N$ последовательно сравнивает x с числами 1, 2, 3. Он выдает результат 1, если обнаружит совпадение x с одним из этих чисел, и результат 0, если совпадение не обнаружено. Множество натуральных чисел $\{1, 2, 3\}$, представимых своими двоичными записями как подмножество словарного пространства $\{0,1\}^*$, также разрешимо.

Вопрос 6: существуют ли бесконечные разрешимые множества?

Ответ: да. Например, множество всех четных чисел. В качестве разрешающего алгоритма следует взять алгоритм вычисления остатка от деления на 2. Другие примеры: множество всех простых чисел, множество всех полных квадратов и др.

Теорема 1.1. Пусть A и B – разрешимые подмножества типа конструктивных объектов X, тогда их объединение, пересечение и дополнение также разрешимы.

Доказательство: согласно условию, характеристические функции χ_A и χ_B вычислимы. Булевые операции дизъюнкции, конъюнкции и отрицания вычислимы по таблице истинности. Поэтому формулы (1) задают алгоритмы вычисления характеристических функций $\chi_{A\cup B}(x)$, $\chi_{A\cap B}(x)$, $\chi_{X\setminus A}(x)$.

Вопрос 7 (тривиальный): а может быть, все множества разрешимы?

Ответ: конечно, нет! Примеры множеств, не являющихся разрешимыми, порождают существенное ограничение, лежащее в основании самого понятия алгоритма – исходные данные должны принадлежать некоторому типу конструктивных объектов. Вследствие этого несчетные бесконечные множества не могут обладать вычислимыми характеристическими функциями, то есть не являются разрешимыми. Таким образом, например, не являются разрешимыми множество всех положительных действительных чисел, множество всех точек внутри единичного круга $x^2 + y^2 = 1$ на плоскости и все другие множества мощности континуума.

Вопрос 8 (нетривиальный): существуют ли неразрешимые множества словарного пространства?

Ответ: да, существуют! Построение явных примеров неразрешимых подмножеств словарного пространства является весьма трудной задачей. Однако можно предположить, что каждый алгоритм можно точно описать конечным текстом на естественном языке и набрать этот текст на компьютере в текстовом режиме. Результирующий текст будет представлять собой конечное слово в фиксированном алфавите (из 2^8 букв). Разным алгоритмам соответствуют разные слова, поэтому мощность множества всех алгоритмов не превосходит мощности множества всех слов в указанном алфавите, то есть не более чем счетная. В то же время семейство всех подмножеств фиксированного словарного пространства несчетно. Поэтому алгоритмов не хватает для вычисления характеристических функций всех подмножеств словарного пространства.

2. Полуразрешимые множества

Еще один способ алгоритмического описания подмножеств типа конструктивных объектов X основан на их представлении в виде областей определения вычислимых функций. Пусть $f: X \to Y$ вычислимая функция и область ее определения $D(f)$ совпадает с подмножеством $M \subseteq X$. Вопрос о принадлежности произвольного элемента $x \in X$ множеству M теоретически может быть решен следующим образом: подать x в качестве входного данного для алгоритма вычисления функции f и запустить процесс вычисления значений $f(x)$. Если процесс вычисления закончится, то значение $f(x)$ определено и $x \in M$, в противном случае значение $f(x)$ не определено, то есть $x \notin X$.

Естественный алгоритм, частично реализующий указанный метод, состоит в моделировании процесса вычисления значения $f(x)$ до его завершения, после чего выдается ответ 1 («да, принадлежит»). Этот алгоритм вычисляет так называемую **полухарактеристическую функцию** $\pi_M(x)$ множества M.

$$\pi_M(x) = \begin{cases} 1, \text{ если } x \in M, \\ \text{не определено, если } x \notin M. \end{cases}$$

Определение 2.3. Множество $M \subseteq X$ называется **полуразрешимым**, если его полухарактеристическая функция $\pi_M(x)$ вычислима.

Вопрос 9: верно ли, что область определения каждой вычислимой функции $f: X \to Y$ является полуразрешимым подмножеством типа конструктивных объектов X?

Ответ: да, верно.

Вопрос 10: верно ли, что каждое полуразрешимое подмножество типа конструктивных объектов X можно представить в виде области определения некой вычислимой функции?

Ответ: да, верно. Одно из таких представлений $M = D(\pi_M)$.

Таким образом, области определения вычислимых функций и полуразрешимые множества – это одно и то же. На практике полуразрешимые множества часто возникают в связи с задачами, требующими полного перебора заранее не ограниченной совокупности вариантов.

Пример 2. Дано натуральное число n. Подобрать натуральные числа x, y, z для которых $x^n + y^n = z^n$.

Очевидный прямой алгоритм решения этой задачи состоит в переборе всевозможных троек натуральных чисел (x, y, z) и проверке условия $x^n + y^n = z^n$ для каждой из них. Перебор прерывается, когда тройка чисел, удовлетворяющая указанному условию, будет обнаружена. Если соответствующей тройки чисел не существует, то алгоритм будет работать бесконечно долго. Исходными данными для алгоритма служит натуральное число n, а результат работы – найденная тройка чисел (x, y, z), то есть этот алгоритм вычисляет некоторую функцию типа $N \to N^3$. Область определения этой функции образует полуразрешимое множество, состоящее в точности из трех натуральных чисел n, для которых поставленная задача имеет решение.

Обсуждаемый пример связан с известной Великой теоремой Ферма, которая утверждает, что при $n > 2$ таких натуральных чисел (x, y, z) не существует. Этот факт был объявлен французским математиком Пьером Ферма (1937 г.), однако доказать его удалось лишь три с половиной века спустя (1994 г.). Завершивший это доказательство английский, а впоследствии американский математик Эндрю Уайлс получил многочисленные награды и был посвящен в рыцари Британской империи. Только после опубликования его работы удается утверждать, что рассматриваемое в примере множество на самом деле является конечным множеством, а потому разрешимо.

Теорема 2.2 (теорема Чёрча-Поста). Пусть X – тип конструктивных объектов.

1. Каждое разрешимое множество X – полуразрешимо.
2. Если у полуразрешимого множества $M \subseteq X$ дополнение $X \backslash M$ также полуразрешимо, то M – разрешимо.

Замечание: условие полуразрешимости дополнения в теореме существенно (!) и не может быть опущено.

3. Перечислимые множества

Рассмотренные ранее способы алгоритмического описания подмножеств типа конструктивных объектов X так или иначе связаны с задачей распознавания принадлежности множеству: алгоритмы вычисления характеристических и полухарактеристических функций напрямую связаны с решением вопросов о принадлежности произвольных элементов $x \in X$

данному множеству $M \subseteq X$. Рассматриваемое в этом пункте понятие перечислимого множества основано на идее – порождение всех элементов множества с помощью единой алгоритмической процедуры. Так как все подмножества типа конструктивных объектов не более чем счетные, то в качестве процедуры порождения достаточно использовать алгоритмическое перечисление.

Определение 3.4. Бесконечная последовательность $x_0, x_1, \dots, x_n, \dots$ элементов, принадлежащих типу конструктивных объектов X, называется вычислимой последовательностью, если функция $f(n) = x_n$ является вычислимой функцией типа $N \to X$.

Вопрос 11: какова область определения данной функции?

Ответ: номера членов последовательности пробегают весь натуральный ряд, поэтому $D(f) = N$.

Определение 3.5. Подмножество M типа конструктивных объектов X называется **перечислимым** (рекурсивно перечислимое множество, вычислимо перечислимое множество), если либо M пусто, либо M имеет вид $M = \{m_0, m_1, \dots, m_n, \dots\}$ для некоторой вычислимой последовательности $\{m_0, m_1, \dots, m_n, \dots\}$. Саму последовательность в этом случае называют **вычислимым пересчетом множества M** (члены вычислимой последовательности могут идти в произвольном порядке, а также повторяться).

Пример 3. Всякое конечное подмножество типа конструктивных объектов X является перечислимым. Если множество M – пусто, то оно перечислимо по определению. Для непустого конечного множества $M = \{a_0, a_1, \dots, a_{k-1}\} \subseteq X$ его вычислимый пересчет задается периодической последовательностью $m_n = a_{n\ mod k}$, где $n\ mod\ k$ обозначает остаток от деления n на k.

4. Равнообъемность понятий полуразрешимости и перечислимости

Теорема 4.3. Каждое перечислимое подмножество типа конструктивных объектов X является полуразрешимым.

Теорема 4.4. Область применимости каждого алгоритма является перечислимым подмножеством области его возможных исходных данных.

Следствие 4.1. Область определения каждой вычислимой функции перечислима.

Верно и обратное: каждое перечислимое множество можно представить в виде области определения подходящей вычислимой функции.

Перечислимые множества – это области определения всевозможных вычислимых функций, и только они.

Следствие 4.2. Область значений каждой вычислимой функции перечислима.

Верно и обратное: каждое перечислимое множество можно представить в виде области значений подходящей вычислимой функции.

Перечислимые множества – это области значений всевозможных вычислимых функций, и только они.

5. Теорема о графике

В теории множеств функции отождествляются с их графиками. Однако алгоритмические описания этих объектов различаются. Пусть X и Y – типы конструктивных объектов, $f: X \to Y$ – частичная функция.

Определение 5.6. Графиком функции f условимся называть «плоское множество»:

$$\Gamma_f = \{(x, y) | x \in D(f), y = f(x)\} \subseteq X \times Y.$$

Алгоритмическим описанием функции f является алгоритм ее вычисления.

Теорема 5.5 (о графике). Функция $f: X \to Y$ вычислима тогда и только тогда, когда ее график Γ_f является перечислимым подмножеством $X \times Y$.

Необходимо отметить многообразие характеризаций класса всех перечислимых множеств (рис. 4).

M – **перечислимо**	⇔	M – полуразрешимо
		M – область определения подходящей вычислимой функции
		M – область значения подходящей вычислимой функции
		M – проекция подходящего «плоского» разрешимого множества

Рис. 4.

В терминах перечислимости удается охарактеризовать и другие основные понятия теории алгоритмов (рис. 5).

Множество $M \subseteq X$ разрешимо	⇔	M и $X \backslash M$ перечислимы
Функция f вычислима	⇔	Ее график Γ_f перечислим

Рис. 5.

Отметим, что все установленные до сих пор факты имели следующую форму: если существуют одни алгоритмы, то существуют и другие. Доказательство состояло в построении желаемых алгоритмов из имеющихся алгоритмов, по существу, в программировании. Однако центральная задача теории алгоритмов несколько иная – изучить границы возможности программирования, то есть выяснить, для каких задач не существует и не может существовать алгоритмов их решения. Одна из таких границ – тривиальная: нельзя запрограммировать функцию, у которой область определения или область значений не содержится ни в одном типе конструктивных объектов, например несчетная. Но есть и более глубокие ограничения, актуальные уже для счетных областей: как построить нетривиальный явный пример невычислимой функции? Как построить пример перечислимого, но неразрешимого множества?

Для решения этих вопросов требуются средства доказательства отрицательных фактов, то есть того, что такая-то функция не является вычислимой. Для этого оказывается важным иметь возможность анализировать не только абстрактные алгоритмы, но и их программы.

Какой язык программирования выбрать? Программист и математик выбирают язык поразному: программист – чтобы было удобно программировать, а математик, чтобы было удобно анализировать. Для анализа удобен самый примитивный язык, который все же обладает свойством универсальности, то есть принципиально позволяет реализовать все алгоритмы.

Заметим, что использование для тех же целей реальных языков программирования затруднительно ввиду заведомо более сложного устройства множества всех программ. Постро-

енные примитивные языки оказываются в той же степени не приспособленными для реального программирования, они для этого не предназначены.

Лекционные занятия № 4–5 «Список алгоритмов»

План занятия:

1. ***Комбинаторные алгоритмы.***
2. ***Алгоритмы сжатия данных.***
3. ***Вычислительная геометрия.***
4. ***Компьютерная графика.***
5. ***Компьютерное зрение.***
6. ***Криптографические алгоритмы.***
7. ***Разработка программного обеспечения.***
8. ***Медицинские алгоритмы.***
9. ***Вычислительная алгебра.***
10. ***Теоретико-числовые алгоритмы.***
11. ***Численные алгоритмы.***
12. ***Алгоритмы оптимизации.***

1. Комбинаторные алгоритмы

Комбинаторика – это раздел математики, в котором изучаются вопросы о том, сколько различных комбинаций, подчиненных тем или иным условиям, можно составить из заданных объектов. Основы комбинаторики очень важны для оценки вероятностей случайных событий, так как именно они позволяют подсчитать принципиально возможное количество различных вариантов развития событий.

1.1. Общие комбинаторные алгоритмы

1.1.1. Алгоритм Флойда для нахождения циклов – находит цикл в итерациях.

1.1.2. Генераторы псевдослучайных чисел – алгоритм, порождающий последовательность чисел, элементы которой почти независимы друг от друга и подчиняются заданному распределению (обычно равномерному).

В языках программирования обычно предусмотрены функции, позволяющие генерировать случайные числа в определенном по умолчанию диапазоне. На самом деле генерируются не случайные, а так называемые псевдослучайные числа; они выглядят случайно, но вычисляются по вполне конкретной формуле.

1.1.2.1. Алгоритм Блюма-Блюма-Шуба получил широкое распространение как алгоритм генерации псевдослучайных чисел, называемый BBS (от фамилий авторов), или генератором с квадратичным остатком. Для целей криптографии этот метод предложен в 1986 году. Он заключается в следующем: выбираются два больших простых числа p и q, сравнимые с 3 по модулю 4, то есть при делении p и q на 4 должен получаться одинаковый остаток 3 (4 – модуль сравнения). Далее вычисляется число $M = p \cdot q$, называемое целым числом Блюма. Затем выбирается другое случайное целое число x, взаимно простое (то есть не имеющее общих делителей, кроме 1) с M. Вычисляют $x_0 = x^2 mod M$, x_0 называется стартовым числом

генератора. На каждом *n*-м шаге работы генератора вычисляется $x_{n+1} = x_n^2 mod M$. Результатом *n*-го шага является один (обычно младший) бит числа x_{n+1}. Иногда в качестве результата принимают бит четности, то есть количество единиц в двоичном представлении элемента. Если количество единиц в записи числа четное, бит четности принимается равным 0, нечетное – бит четности принимается равным 1.

Пример 1: пусть $p = 11, q = 19$ (убеждаемся, что *11 mod 4 = 3, 19 mod 4 = 3*). Тогда $M = p \cdot q = 11 \cdot 19 = 209$. Выберем *x*, взаимно простое с *M*: пусть *x = 3*. Вычислим стартовое число генератора x_0: $x_0 = x^2 \bmod M = 3^2 \bmod 209 = 9 \bmod 209 = 9$.

Рассмотрим сравнения вида: $x \equiv \frac{a}{b}(mod\ m)$, где *a, b, m* – целые положительные числа. Выявим целочисленные решения дроби: $x = \frac{n \cdot m + a}{в}$, где целое *n* выберем из интервала: $\frac{-b \cdot m - a}{m} \leq n \leq \frac{b \cdot m - a}{m}$ (если записываем это двойное неравенство в десятичном виде, то дробная часть отбрасывается). В результате получим два целочисленных значения *x*, из которых принимается только положительное:

$$x_0 = \frac{n \cdot 209 + 9}{1}$$

$$\frac{-1 \cdot 209 - 9}{209} \leq n \leq \frac{1 \cdot 209 - 9}{200}$$

$$-1{,}04 \leq n \leq 0{,}96$$

$$n = -1, x_0 = -200$$

$$n = 0, x_0 = 9$$

Вычислим первые десять чисел x_i по алгоритму BBS. В качестве случайных бит будем брать младший бит в двоичной записи числа x_i:

$x_1 = 9^2 \bmod 209 = 81 \bmod 209 = 81$	младший бит.	1
$x_2 = 81^2 \bmod 209 = 6561 \bmod 209 = 82$	младший бит.	0
$x_3 = 82^2 \bmod 209 = 6724 \bmod 209 = 36$	младший бит.	0
$x_4 = 36^2 \bmod 209 = 1296 \bmod 209 = 42$	младший бит.	0
$x_5 = 42^2 \bmod 209 = 1764 \bmod 209 = 92$	младший бит.	0
$x_6 = 92^2 \bmod 209 = 8464 \bmod 209 = 104$	младший бит.	0
$x_7 = 104^2 \bmod 209 = 10816 \bmod 209 = 157$	младший бит.	1
$x_8 = 157^2 \bmod 209 = 24649 \bmod 209 = 196$	младший бит.	0
$X_9 = 196^2 \bmod 209 = 38416 \bmod 209 = 169$	младший бит.	1
$x_{10} = 169^2 \bmod 209 = 28561 \bmod 209 = 137$	младший бит.	1

Самым интересным для практических целей свойством этого метода является то, что для получения *n*-го числа последовательности не нужно вычислять все предыдущие *n* чисел x_i. Оказывается, x_n можно сразу получить по формуле $x_n = {x_0}^{2^n mod((p-1)(q-1))} mod\ M$.

Например, вычислим x_{10} сразу из x_0:

$$x_{10} = {x_0}^{2^{10} \bmod (11-1)(19-1)} \bmod 209 = {x_0}^{1024 \bmod 180} \bmod 209 =$$

$$9^{124} \bmod 209 = \left(9^4\right)^{31} \bmod 209 = 82^{31} \bmod 209 =$$

$$\left(82^{15} \bmod 209\right)\left(82^{16} \bmod 209\right) = \left(\left(82^3\right)^{15} \bmod 209\right)\left(\left(82^4\right)^4 \bmod 209\right) =$$

$$\left(26^5 \bmod 209\right)\left(42^4 \bmod 209\right) = (144 \cdot 104) \bmod 209 = 14976 \bmod 209 = 137.$$

В результате, действительно, получили такое же значение, как и при последовательном вычислении, – 137. Вычисления кажутся достаточно сложными, однако на самом деле их легко оформить в виде небольшой процедуры или программы и использовать при необходимости. Возможность «прямого» получения x_n позволяет использовать алгоритм BBS при потоковой шифрации, например, для файлов с произвольным доступом или фрагментов файлов с записями базы данных.

Безопасность алгоритма BBS основана на сложности разложения большого числа *M* на множители. Если *M* достаточно велико, его можно даже не держать в секрете: до тех пор, пока оно не разложено на множители, никто не сможет предсказать выход генератора псевдослучайных чисел. Кроме того, можно доказать, что злоумышленник, зная некоторую последовательность, сгенерированную генератором BBS, не сможет определить ни предыдущие до нее биты, ни следующие. Генератор BBS непредсказуем в левом направлении и в правом направлении. Это свойство очень полезно для целей криптографии, и оно также связано с особенностями разложения числа *M* на множители. Самым существенным недостатком алгоритма является то, что он недостаточно быстр, что не позволяет использовать его во многих областях, например, при вычислениях в реальном времени, а также, к сожалению, и при потоковом шифровании. Зато этот алгоритм выдает действительно хорошую последовательность псевдослучайных чисел с большим периодом (при соответствующем выборе исходных параметров), что позволяет использовать его для криптографических целей при генерации ключей для шифрования.

Алгоритмы получения псевдослучайных чисел еще недостаточно исследованы, но при вычислениях по методу статистических испытаний отдается предпочтение именно им, так как свойства последовательности псевдослучайных чисел можно исследовать путем пробных вычислений, а экспериментальные устройства дают новые последовательности случайных чисел при каждом их использовании.

Любой генератор псевдослучайных чисел с ограниченными ресурсами рано или поздно зацикливается – начинает повторять одну и ту же последовательность чисел. Длина циклов генератора псевдослучайных чисел зависит от самого генератора и составляет около $2^{n/2}$, где n – размер внутреннего состояния в битах, хотя линейные конгруэнтные и LFSR-генераторы обладают максимальными циклами порядка 2^n.

1.1.2.2. Вихрь Мерсенна – генератор псевдослучайных чисел, разработан в 1997 году японскими учеными Макото Мацумото и Такудзи Нисимура, основывается на свойствах простых чисел Мерсенна (числа вида: $M_n = 2^n - 1$, где n – натуральное число, названы в честь французского математика Марена Мерсенна, последовательность чисел Мерсенна начинается так: 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023…) и обеспечивает быструю генерацию высококачественных псевдослучайных чисел. Вихрь Мерсенна лишен таких недостатков, как предсказуемость, легко выявляемая статистическая зависимость, имеет огромный период, равный числу Мерсенна $2^{19937} - 1$, что более чем достаточно для многих практических приложений.

1.1.2.3. Метод Фибоначчи с запаздываниями – один из методов генерации псевдослучайных чисел. Он позволяет получить высокое «качество» псевдослучайных чисел, по способу формирования напоминают ряд фибоначчи, каждый член равен сумме двух предыдущих, слагаемые которого далеко разнесены, и прежде, чем выдавать псевдослучайные числа, нужно «засеять» буфер из натуральных чисел.

1.1.2.4. Линейный конгруэнтный метод – один из алгоритмов генерации псевдослучайных чисел. Применяется в простых случаях, входит в стандартные библиотеки различных компиляторов и не обладает криптографической стойкостью. Метод заключается в вычислении членов линейной рекуррентной последовательности по модулю некоторого натурального числа m, задаваемой следующей формулой: $x_{k+1} = (aX_k + c) mod\ m$, где a и c – некоторые целочисленные коэффициенты.

1.2. Алгоритмы на графах

Граф – это совокупность непустого множества вершин и множества пар вершин (связей между вершинами). Объекты представляются как вершины, или узлы графа, а связи – как дуги, или ребра. Для разных областей применения виды графов могут различаться направленностью, ограничениями на количество связей и дополнительными данными о вершинах или ребрах.

1.2.1. Алгоритм Беллмана-Форда – вычисляет кратчайший путь во взвешенном графе (некоторые веса ребер могут быть отрицательны).

1.2.2. Алгоритм Борувки – находит минимальное остовное дерево в графе (рис. 6). Впервые был опубликован в 1926 году Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии (Чехия). Несколько раз был переоткрыт (К. Флореком, Д. Перкалом, Ж. Соллином).

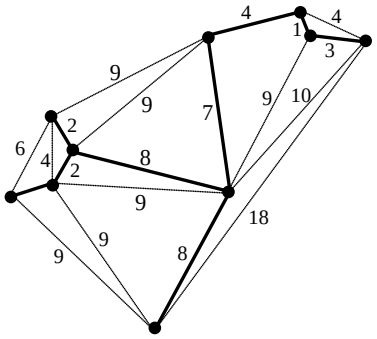


Рис. 6.

Пример 2: имеется n городов, которые нужно объединить в единую электрическую сеть. Для этого достаточно проложить *n-1* линий между городами. Как соединить города так, чтобы суммарная стоимость соединений (кабеля) была минимальна?

1.2.3. Алгоритм Дейкстры – алгоритм на графах, изобретенный нидерландским ученым Э. Дейкстрой в 1959 году. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без ребер отрицательного веса, имеет широкое применение в программировании.

Пример 3: рассмотрим выполнение алгоритма Дейкстры на примере графа, показанного на рисунке 7. Пусть требуется найти кратчайшие расстояния от первой вершины до всех остальных.

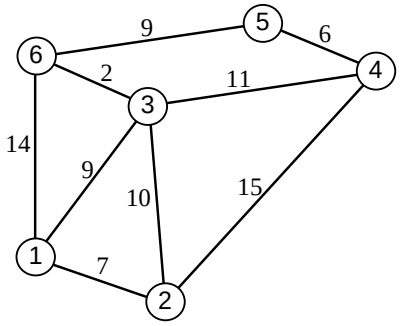


Рис. 7.

Кружками обозначены вершины, линиями – пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначена их «цена» – длина пути. Рядом с каждой вершиной серым обозначена метка – длина кратчайшего пути в эту вершину из вершины один (рис. 8).

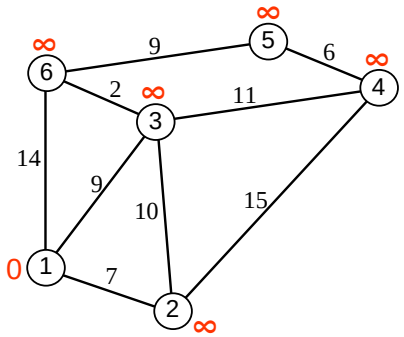


Рис. 8.

Первый шаг. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Ее соседями являются вершины 2, 3 и 6 (рис. 9).

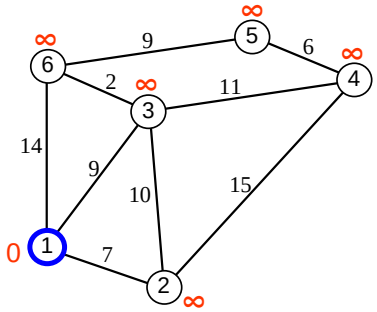


Рис. 9.

Первый по очереди сосед вершины 1 – вершина 2, потому что длина пути до нее минимальна. Длина пути в нее через вершину 1 равна сумме кратчайшего расстояния до вершины 1, значению ее метки и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7 (рис. 10).

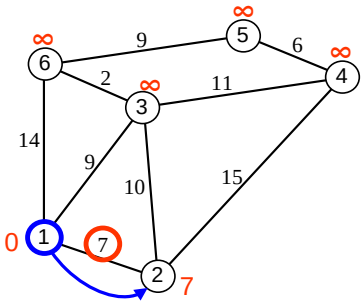


Рис. 10.

Аналогичную операцию проделываем с двумя другими соседями 1-й вершины – 3-й и 6-й (рис. 11).

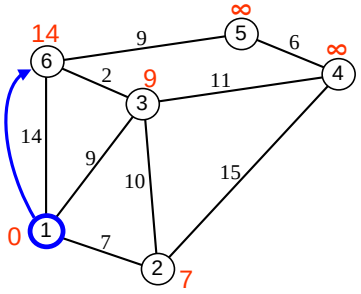


Рис. 11.

Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Э. Дейкстра). Вычеркнем ее из графа, чтобы отметить, что эта вершина посещена (рис. 12).

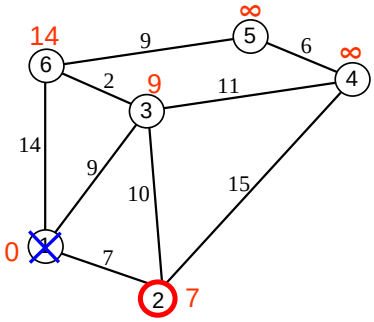


Рис. 12.

Второй шаг. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7 (рис. 13).

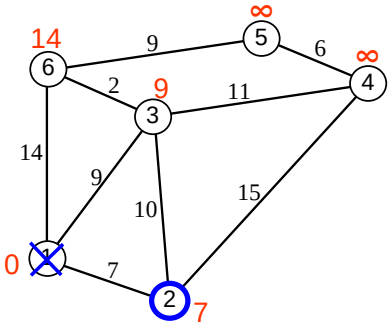


Рис. 13.

Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Первый (по порядку) сосед вершины 2 – вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

Следующий сосед вершины 2 – вершина 3, так как имеет минимальную метку из вершин, отмеченных как непосещенные. Если идти в нее через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9 < 17, поэтому метка не меняется (рис.14).

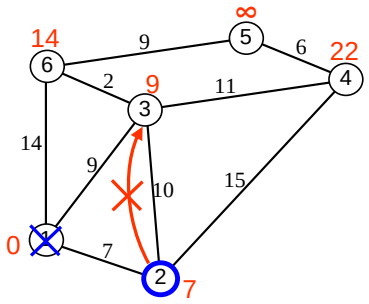


Рис. 14.

Еще один сосед вершины 2 – вершина 4. Если идти в нее через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 (7 + 15 = 22). Поскольку 22 < ∞, устанавливаем метку вершины 4, равной 22 (рис. 15).

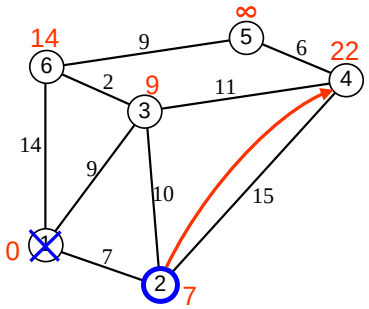


Рис. 15.

Все соседи вершины 2 просмотрены, замораживаем расстояние до нее и помечаем ее как посещенную (рис. 16).

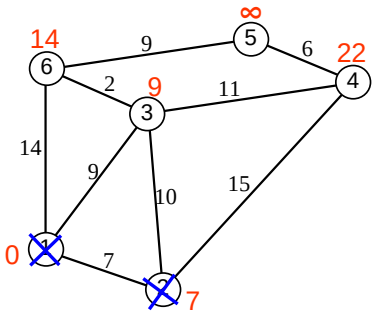


Рис. 16.

Третий шаг. Повторяем шаг алгоритма, выбрав вершину 3. После ее «обработки» получим результаты, указанные на рисунке 17.

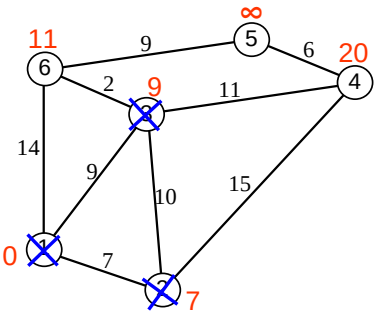


Рис. 17.

Дальнейшие шаги. Повторяем шаг алгоритма для оставшихся вершин. Это будут вершины 6, 4 и 5, соответственно порядку (рис. 18).

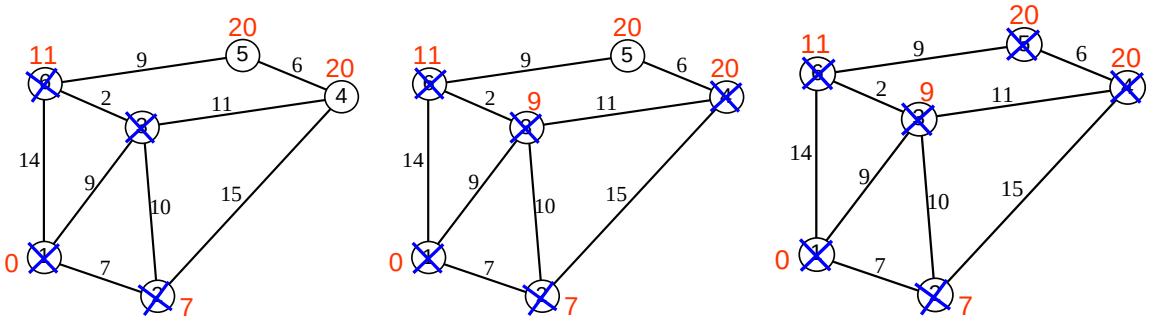


Рис. 18.

Завершение выполнения алгоритма. Алгоритм заканчивает работу, когда нельзя больше обработать ни одной вершины. В данном примере все вершины зачеркнуты, однако ошибочно полагать, что так будет в любом примере – некоторые вершины могут остаться незачеркнутыми, если до них нельзя добраться. Результат работы алгоритма виден на последнем рисунке 18: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й – 9, до 4-й – 20, до 5-й – 20, до 6-й – 11.

1.2.4. Алгоритм Крускала – алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые описан Джозефом Крускалом в 1956 году. Алгоритм Крускала объединяет вершины графа в несколько связных компонент, каждая из которых является деревом (**дерево** – связный граф, не содержащий циклов). На каждом шаге из всех ребер, соединяющих вершины из различных компонент связности, выбирается ребро с наименьшим весом.

Пример 4:

Изображение

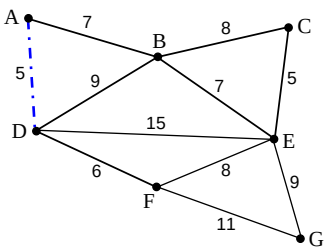


Рис. 19.

Словесное описание

Ребра AD и CE имеют минимальный вес, равный 5. Произвольно выбирается ребро AD (выделено пунктиром на рис. 19).

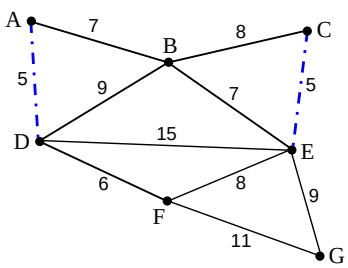


Рис. 20.

Теперь наименьший вес, равный 5, имеет ребро CE. Так как добавление CE не образует цикла, то выбираем его в качестве второго ребра (рис. 20).

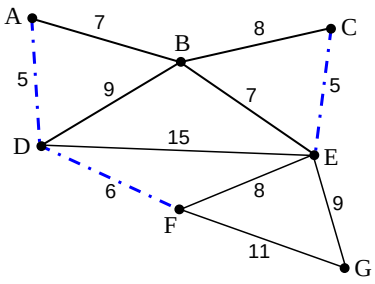


Рис. 21.

Аналогично выбираем ребро DF, вес которого равен 6 (рис. 21).

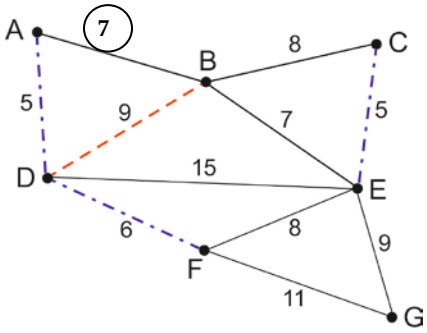


Рис. 22.

Следующие ребра – AB и BE с весом 7. Произвольно выбирается ребро AB, выделенное на рисунке 22. Ребро BD выделено (пунктиром), так уже существует путь (пунктир с точкой) между B и D, поэтому, если бы это ребро было выбрано, то образовался бы цикл ABD.

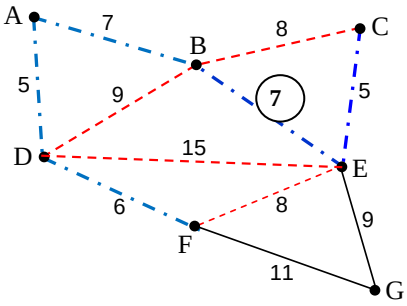


Рис. 23.

Аналогичным образом выбирается ребро BE, вес которого равен 7. На этом этапе пунктиром выделено гораздо больше ребер: BC, потому что оно создаст цикл BCE, DE и цикл DEBA, и FE, потому что оно сформирует цикл FEBAD (рис. 23).

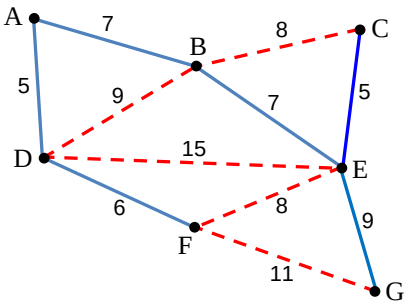


Рис. 24.

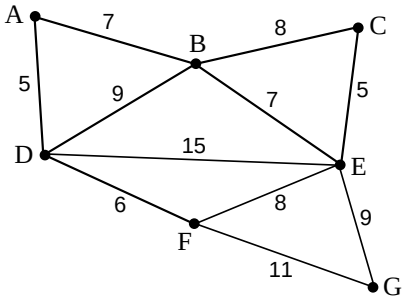
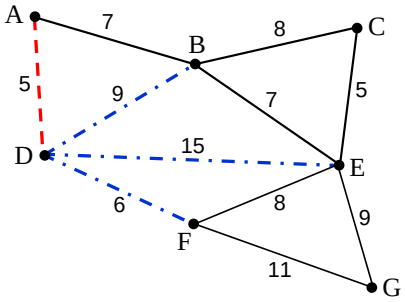
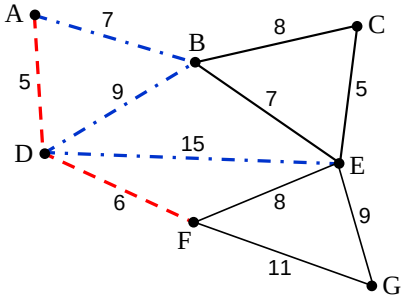
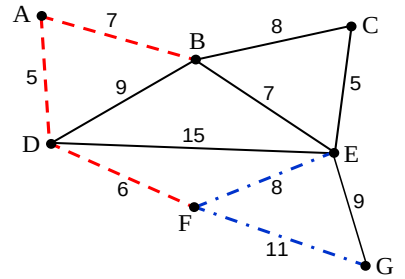
Алгоритм завершается добавлением ребра EG с весом 9. Минимальное остовное дерево построено (рис. 24).

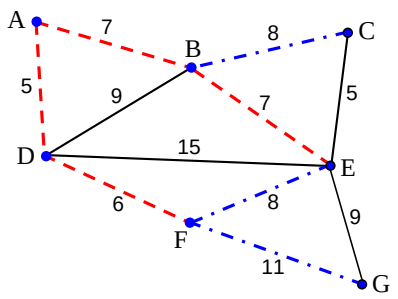
1.2.5. Алгоритм Прима – алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые был открыт в 1930 году чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 году и независимо от них Эдсгером Дейкстрой в 1959 году.

Как и алгоритм Крускала, алгоритм Прима следует общей схеме алгоритма построения минимального остовного дерева: на каждом шаге мы добавляем к строящемуся остову безопасное ребро. Построение начинается с дерева, включающего в себя одну (произвольную) вершину. В течение работы алгоритма дерево разрастается, пока не охватит все вершины ис-

ходного графа. На каждом шаге алгоритма к текущему дереву присоединяется самое легкое из ребер.

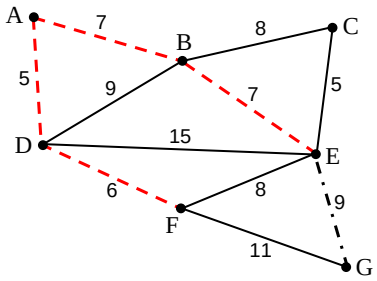
Пример 5:

Изображение	Словесное описание
	Пусть дан исходный взвешенный граф. Числа возле ребер показывают их веса, которые можно рассматривать как расстояния между вершинами (рис. 25).
Рис. 25.	
	В качестве начальной вершины произвольно выбирается вершина D. Каждая из вершин A, B, E и F соединена с D единственным ребром. Вершина A – ближайшая к D и выбирается как вторая вершина вместе с ребром AD (рис. 26).
Рис. 26.	
	Следующая вершина – ближайшая к любой из выбранных вершин D или A. B удалена от D на 9 и от A – на 7. Расстояние до E равно 15, а до F – 6. F является ближайшей вершиной, поэтому она включается в дерево F вместе с ребром DF (рис. 27).
Рис. 27.	
	Аналогичным образом выбирается вершина B, удаленная от A на 7 (рис. 28).
Рис. 28	



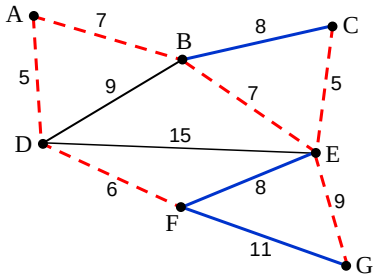
В этом случае есть возможность выбрать либо C, либо E, либо G. C удалена от B на 8, E удалена от B на 7, а G удалена от F на 11. E – ближайшая вершина, поэтому выбирается E и ребро BE (рис. 29).

Рис. 29.



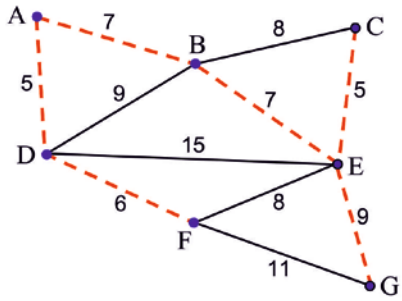
Здесь доступны только вершины C и G. Расстояние от E до C равно 5, а до G – 9. Выбирается вершина C и ребро EC (рис. 30).

Рис. 30.



Единственная оставшаяся вершина – G. Расстояние от F до нее равно 11, от E – 9. E ближе, поэтому выбирается вершина G и ребро EG (рис. 31).

Рис. 31.



Выбраны все вершины, минимальное остовное дерево построено (выделено пунктиром). В этом случае его вес равен 39 (рис. 32).

Рис. 32.

Алгоритм Прима начинает построение минимального остовного дерева с конкретной вершины графа и постепенно расширяет построенную часть дерева; в отличие от него алгоритм Крускала делает упор на ребрах графа.

1.2.6. Метод ближайшего соседа (k-nearest neighbor algorithm, KNN). О свойствах нового объекта мы судим, полагаясь на ранее полученные знания, наблюдения. Человек, сталкиваясь с новой задачей, использует свой жизненный опыт, вспоминает аналогичные ситуации, которые когда-то с ним происходили. Например, встретив иностранца на улице, мы можем догадаться о его происхождении по речи, жестам и внешности. Сходство объектов ле-

жит в основе данного алгоритма, который способен выделить среди всех наблюдений k известных объектов (k-ближайших соседей), похожих на новый неизвестный ранее объект. На основе классов ближайших соседей выносится решение касательно нового объекта. Важной задачей данного алгоритма является подбор коэффициента k – количество записей, которые будут считаться похожими.

1.2.7. Алгоритмы нахождения максимального потока

Так же, как дорожную карту можно смоделировать ориентированным графом, чтобы найти кратчайший путь из одной точки в другую, ориентированный граф можно интерпретировать как некоторую транспортную сеть и использовать его для решения задач о потоках вещества в системе трубопроводов. Представим, что некоторый продукт передается по системе от источника, где данный продукт производится, к стоку, где он потребляется. Источник производит продукт с некоторой постоянной скоростью, а сток с той же скоростью потребляет продукт. Интуитивно потоком продукта в любой точке системы является скорость движения продукта. С помощью транспортных сетей можно моделировать течение жидкостей по трубопроводам, движение деталей на сборных линиях, передачу тока по электрическим сетям, информации – по информационным сетям и т. д.

Каждое ориентированное ребро сети можно рассматривать как канал, по которому движется продукт. Каждый канал имеет заданную пропускную способность, которая характеризует максимальную скорость перемещения продукта по каналу, например, 200 литров жидкости в минуту для трубопровода или 20 ампер для провода электрической цепи. Вершины являются точками пересечения каналов; через вершины, отличные от источника и стока, продукт проходит не накапливаясь. В задачах о максимальном потоке можно найти максимальную скорость пересылки продукта от источника к стоку, при которой не будет нарушаться ограничения пропускной способности.

1.2.8. Алгоритмы нахождения максимального паросочетания

Пусть дан граф G = (V,E), **паросочетание** M в графе G – это множество попарно несмежных ребер, то есть ребер, не имеющих общих вершин (рис. 33).

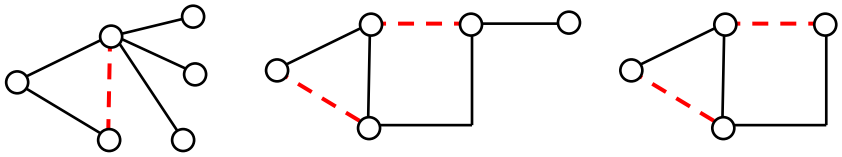


Рис. 33.

Максимальное паросочетание – это такое паросочетание, которое не содержится ни в каком другом паросочетании этого графа.

1.2.1.1. Алгоритм Куна (венгерский алгоритм) – алгоритм оптимизации, решающий задачу о назначениях за определенное время, был разработан и опубликован Харолдом Куном в 1955 году. Автор дал ему имя «венгерский метод» в связи с тем, что алгоритм в значительной степени основан на более ранних работах двух венгерских математиков (Дениша Кёнига и Йёне Эгервари).

Пример 6: предположим, есть три работника: Иван, Петр и Андрей. Нужно распределить между ними выполнение трех видов работ (которые мы назовем A, B, C), каждый работник должен выполнять только одну разновидность работ. Как это сделать так, чтобы потратить наименьшую сумму денег на оплату труда рабочих?

Сначала необходимо построить матрицу стоимостей работ (рис. 34):

	A	B	C
Иван	1000 руб.	2000 руб.	3000 руб.
Петр	3000 руб.	3000 руб.	3000 руб.
Андрей	3000 руб.	3000 руб.	2000 руб.

Рис. 34.

Венгерский алгоритм, примененный к приведенной выше таблице, дает следующее распределение: Иван выполняет работу A, Петр – работу B, Андрей – работу C.

1.3. Алгоритмы поиска

1.3.1. Метод штрафов

Основная задача **метода штрафных функций** состоит в преобразовании задачи минимизации функции $Z = F(x)$ с соответствующими ограничениями, наложенными на *x*, в задачу поиска минимума без ограничений функции $z = F(x) + P(x)$. Функция *P(x)* является штрафной. Необходимо, чтобы при нарушении ограничений она «штрафовала» функцию, то есть увеличивала ее значение. В этом случае минимум функции *Z* будет находиться внутри области ограничений. Функция *P(x)*, удовлетворяющая этому условию, может быть не единственной.

1.3.2. Поиск в глубину – проходит граф ветка за веткой. Это один из методов обхода графа: для каждой непройденной вершины необходимо найти все непройденные смежные вершины и повторить поиск для них (рис. 35).

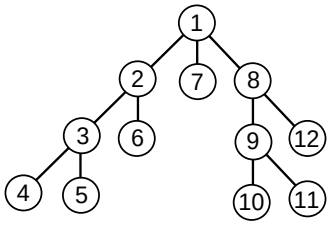


Рис. 35.

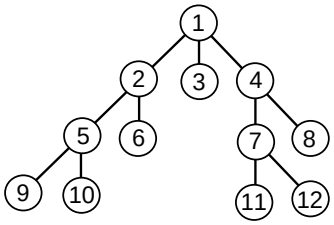


Рис. 36.

1.3.3. Поиск в ширину – проходит граф уровень за уровнем. Поиск в ширину реализуется с помощью структуры «очередь». Для этого занесем в очередь исходную вершину. Затем будем работать, пока очередь не опустеет, таким образом, выберем элемент из очереди и добавим все смежные ему элементы, которые еще не использованы (рис. 36).

1.3.4. Поиск по первому наилучшему совпадению – проходит граф в порядке важности, используя очередь приоритетов.

Существует целое семейство алгоритмов поиска по первому наилучшему совпадению (Best-First-Search) с различными функциями оценки. Ключевым компонентом этих алгоритмов является эвристическая функция, обозначаемая как *h(n):* оценка стоимости наименее дорогостоящего пути от узла *n* до целевого узла.

Пример 7: в задаче поиска маршрута в Румынии можно оценивать стоимость наименее дорогостоящего пути от Арада до Бухареста с помощью расстояний по прямой до Бухареста, измеряемых в узловых точках маршрута от Арада до Бухареста (рис. 37).

Начальное состояние:

Рис. 37.

После развертывания узла Arad (рис. 38):

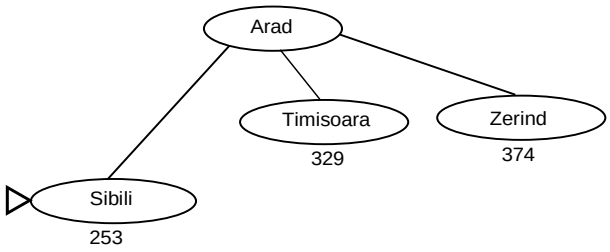


Рис. 38.

После развертывания узла Sibili (рис. 39):

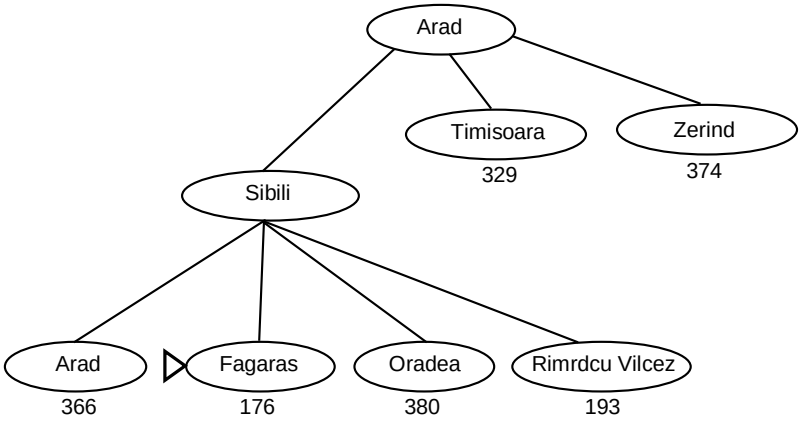


Рис. 39.

После развертывания узла Faragaras (рис. 40):

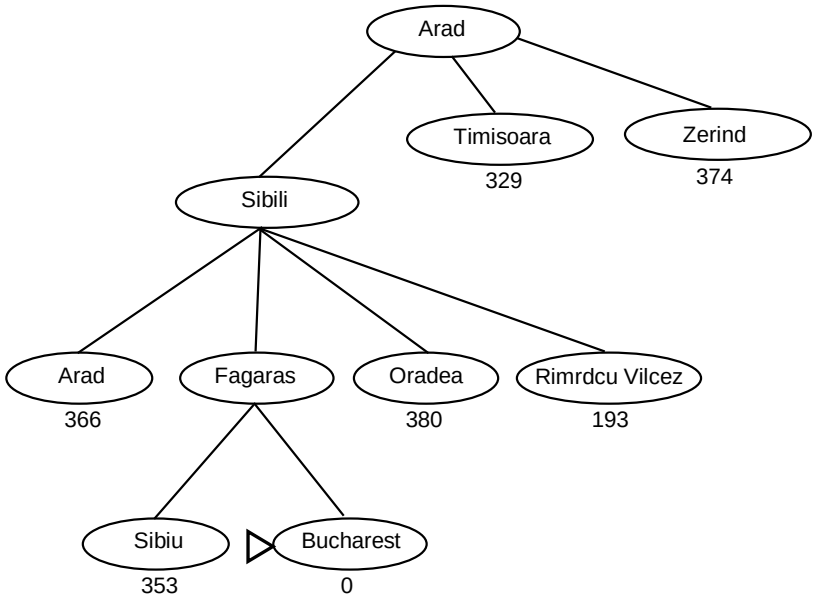


Рис. 40.

1.3.5. Алгоритм Ли – волновой алгоритм поиска пути на карте, алгоритм трассировки. С его помощью можно построить путь, или трассу, между двумя любыми элементами в лабиринте. Из начального элемента распространяется волна в четыре направления. Элемент, в который пришла волна, образует фронт волны. Элементы первого фронта волны являются

источниками вторичных волн. Элементы второго фронта генерируют волну третьего фронта и так далее. Процесс заканчивается тогда, когда достигается конечный элемент. На втором этапе алгоритма строится трасса. Используя волновой алгоритм, можно найти трассу в лабиринте с любым количеством стен. В этом заключается преимущество его использования, недостаток заключается в том, что при построении трассы требуется большой объем памяти.

1.4. Алгоритмы сортировки

Сортировка – это общая задача многих компьютерных приложений. Практически любой список данных ценнее, когда он отсортирован по некоторому определенному принципу. У каждого алгоритма сортировки есть свои преимущества и недостатки. Важно выбрать тот алгоритм, который лучше подходит для решения конкретной задачи.

1.4.1. Bogosort (случайная сортировка, сортировка ружья, или обезьянья сортировка) является одним из самых неэффективных алгоритмов сортировки. Чаще всего используют в образовательных целях, противопоставляя другим, более реалистичным алгоритмам. Если bogosort использовать для сортировки колоды карт, то сначала в алгоритме нужно проверить, лежат ли все карты по порядку, и если не лежат, то случайным образом перемешать ее, проверить, лежат ли теперь все карты по порядку, и повторять процесс, пока не отсортируется колода (рис. 41).

Рис. 41.

1.4.2. Наивная сортировка – генерация всех *n!* возможных перестановок и проверка на отсортированность.

1.4.3. Блинная сортировка – единственная операция, допустимая в алгоритме, – это переворот элементов последовательности до какого-либо индекса. В отличие от традиционных алгоритмов, в которых минимизируют количество сравнений, в блинной сортировке требуется сделать как можно меньше переворотов. Процесс можно визуально представить как стопку блинов, которую тасуют путем взятия нескольких блинов сверху и их переворачивания (рис. 42).

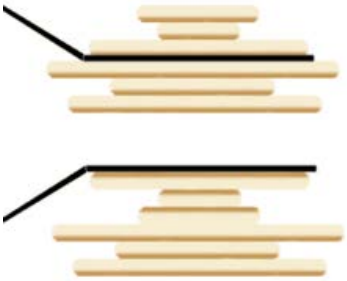


Рис. 42.

Усложненный вариант представляет собой блинную сортировку последовательности, элементы которой содержат дополнительный бинарный параметр. Эту задачу предложили Билл Гейтс и Христос Пападимитриу в 1979 году. Она стала известна как «задача о подго-

ревших блинах» (англ. *«burnt pancake problem»*): «Каждый блин в стопке подгорел с одной стороны. Требуется отсортировать блины по возрастанию (убыванию) диаметра так, чтобы они все лежали на тарелке подгоревшей стороной вниз».

В 2007 году был создан биологический компьютер на основе генетически модифицированной кишечной палочки, который решал задачу о подгорелых блинах. Роль блинов играли фрагменты дезоксирибонуклеиновой кислоты (3’ и 5’ концы которых были разными сторонами блина). Бактерия, выстроив фрагменты в нужном порядке, приобретала устойчивость к антибиотику и не погибала. Время, затраченное на поиск правильной комбинации, показывало минимально необходимое число переворотов фрагмента.

1.4.4. Блочная сортировка (корзинная сортировка) – алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем (рис. 43). Каждый блок затем сортируется отдельно либо рекурсивно тем же методом либо другим. Затем элементы помещаются обратно в массив (рис. 44).

Пример 8:

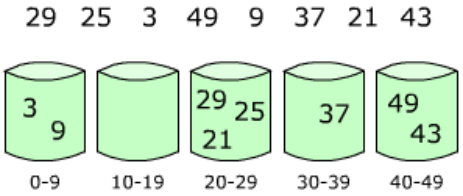


Рис. 43.

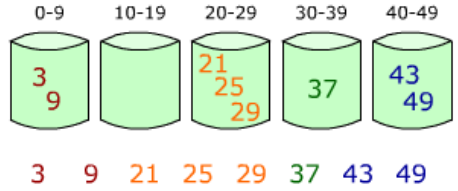


Рис. 44.

1.4.5. Быстрая сортировка – сортировка с разбиением исходного набора данных на две половины так, что любой элемент первой половины был упорядочен относительно любого элемента второй половины; алгоритм применяется рекурсивно к каждой половине (использует подход «разделяй и властвуй»).

1.4.6. Гномья сортировка основана на технике, используемой обычным голландским садовым гномом при сортировке линии цветочных горшков. Гном смотрит на следующий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперед, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперед; если нет следующего горшка, он закончил.

1.4.7. Сортировка пузырьком – алгоритм, состоящий из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются n-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает – массив отсортирован. При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на свое место в конце массива рядом с предыдущим наибольшим элементом, а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырек в воде, отсюда и название алгоритма, рис. 45).

Пример 9:

1	**1**	**1**	**1**
2	**2**	**2**	**2**
4	**4**	**4**	**3**
5	**5**	**3**	**4**
6	**3**	**5**	**5**
3	**6**	**6**	**6**
7	**7**	**7**	**7**
8	**8**	**8**	**8**

Рис. 45.

1.4.8. Сортировка расческой – это довольно упрощенный алгоритм сортировки, изначально спроектированный Влодзимежом Добосиевичем в 1980 г. Позднее он был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в апреле 1991 г. Сортировка расческой улучшает сортировку пузырьком, ее основная идея: устранить «черепа», или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком («кролики», большие значения в начале списка, не представляют проблемы для сортировки пузырьком). В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен одному. Основная идея сортировки расческой в том, что этот промежуток может быть гораздо больше, чем единица.

2. Алгоритмы сжатия данных

2.1. Алгоритмы сжатия без потерь – методы сжатия данных: видео, аудио, графики, документов, представленных в цифровом виде, при использовании которых закодированные данные могут быть восстановлены с точностью до бита. При этом оригинальные данные полностью восстанавливаются из сжатого состояния.

2.2. Алгоритмы сжатия с потерями – метод сжатия (компрессии) данных, при использовании которого распакованные данные отличаются от исходных данных, но степень отличия не является существенной с точки зрения их дальнейшего использования. Этот тип компрессии часто применяется для сжатия аудио- и видеоданных, статических изображений, в Интернете, особенно в потоковой передаче данных, и цифровой телефонии.

3. Вычислительная геометрия

3.1. Построение выпуклой оболочки набора точек

Если представить себе доску, в которую вбито – но не по самую шляпку – много гвоздей, взять веревку, связать на ней скользящую петлю (лассо) и набросить ее на доску, а потом затянуть, то веревка окружит все гвозди, но касаться она будет только некоторых, самых внешних. Те гвозди, которых она касается, составят выпуклую оболочку для всей группы гвоздей.

3.1.1. Алгоритм Грэхема – алгоритм построения выпуклой оболочки в двумерном пространстве. В этом алгоритме задача о выпуклой оболочке решается с помощью стека, сформированного из точек-кандидатов. Все точки входного множества заносятся в стек, а потом точки, не являющиеся вершинами выпуклой оболочки, со временем удаляются из не-

го. По завершении работы алгоритма в стеке остаются только вершины оболочки в порядке их обхода против часовой стрелки.

Стек (англ. *«stack»* – стопка) – структура данных, представляющая из себя список элементов, организованных по принципу LIFO (англ. *«last in – first out»:* последним пришел – первым вышел). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

3.2. Триангуляция

В геометрии триангуляция (в наиболее общем значении) – это разбиение геометрического объекта на симплексы. Например, на плоскости это разбиение на треугольники.

3.3. Локализация точки

3.3.1. Алгоритм точки в многоугольнике – проверка принадлежности данной точки простому многоугольнику.

3.3.2. Метод луча – принадлежность точки простому многоугольнику.

3.4. Пересечения

3.4.1. Алгоритм Бентли-Оттмана – поиск всех точек пересечения отрезков на плоскости. Позволяет найти все точки пересечений прямолинейных отрезков на плоскости. В нем применяется метод выметающей прямой (рис. 46). В методе используется вертикальная выметающая прямая движущаяся слева направо, при этом отрезки, которые она пересекает при данной координате *x*, можно упорядочить по координате *y*, тем самым их можно сравнивать между собой (какой выше, какой ниже). Это сравнение можно осуществить, например, используя уравнение прямой, проходящей через две точки (отрезки заданы двумя своими конечными точками): $\frac{(x-x_1)}{(x_2-x_1)} = \frac{(y-y_1)}{(y_2-y_1)}$, где $(x_1, y_1), (x_2, y_2)$ – координаты, соответственно, первой и второй точек отрезка. Выметающая прямая перемещается по так называемым точкам-событиям (левым и правым концам отрезков, а также точкам – пересечениям отрезков).

3.4.2. Алгоритм Коэна-Сазерленда – алгоритм отсечения отрезков, то есть алгоритм, позволяющий определить часть отрезка, которая пересекает прямоугольник. Был разработан Дэном Коэном и Айвеном Сазерлендом в Гарварде в 1966–1968 гг. (рис. 47).

Алгоритм разделяет плоскость на девять частей прямыми, которые образуют стороны прямоугольника. Каждой из девяти частей присваивается четырехбитный код. Биты (от младшего до старшего) значат «левее», «правее», «ниже», «выше». Иными словами, у тех трех частей плоскости, которые слева от прямоугольника, младший бит равен 1 и так далее. Алгоритм определяет код конечных точек отрезка.

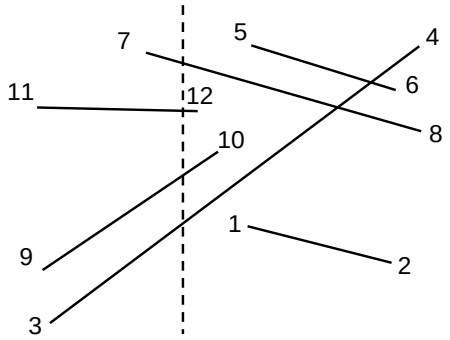


Рис. 46.

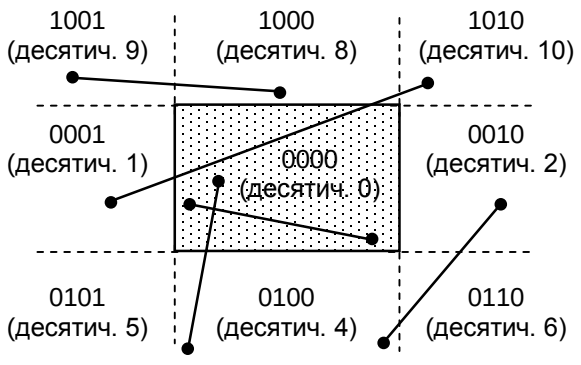


Рис. 47.

4. Компьютерная графика

4.1. Алгоритм художника – простейший программный вариант решения «проблемы видимости» в трехмерной компьютерной графике. При этом грани, которые выводятся позже, закрывают собою невидимые части более дальних граней.

4.2. Затенение по Фонгу – модель освещения и метод интерполяции в трехмерной компьютерной графике.

Интерполяция – способ нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений.

4.3. Затенение по Гуро – алгоритм моделирования различных эффектов света и цвета на поверхности объекта в трехмерной компьютерной графике.

5. Компьютерное зрение – теория и технология создания машин, которые могут производить обнаружение, слежение и классификацию объектов.

6. Криптографические алгоритмы

Криптография – наука о методах обеспечения конфиденциальности (невозможности прочтения информации посторонним) и аутентичности (целостности и подлинности авторства, а также невозможности отказа от авторства) информации.

6.1. Шифрование с симметричным (скрытым) ключом.

6.2. Асимметричное шифрование (с публичным ключом).

6.3. Алгоритмы цифровой подписи.

6.4. Алгоритмы разделения секрета – под разделением секрета понимают любой метод распределения секрета среди группы участников, каждому из которых достается доля секрета. Секрет потом может воссоздать только коалиция участников.

7. Разработка программного обеспечения

7.1. Алгоритмы распределенных систем – способ решения трудоемких вычислительных задач с использованием нескольких компьютеров, чаще всего объединенных в параллельную вычислительную систему.

7.2. Алгоритмы выделения и освобождения памяти.

7.3. Алгоритмы синхронизации процессов.

8. Медицинские алгоритмы

8.1. Генетический алгоритм – это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путем случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию. Является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких, как наследование, мутации и отбор. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

8.2. Нейронные сети, искусственные нейронные сети – математические модели, а также их программные или аппаратные реализации, построенные по принципу организации и функционирования биологических нейронных сетей – сетей нервных клеток живого организма.

9. Вычислительная алгебра

9.1. Алгоритм Штрассена – быстрое умножение матриц. Он был разработан Фолькером Штрассеном в 1969 году. Несмотря на то, что алгоритм Штрассена является не самым быстрым из существующих алгоритмов быстрого умножения матриц, он проще программируется, поэтому именно он чаще используется на практике.

9.2. Преобразования Хаусхолдера – вычисление обратной матрицы, собственных векторов и собственных значений матрицы, был разработан в 1958 году.

9.3. Метод Гаусса – стандартный метод решения систем линейных уравнений. Это метод последовательного исключения переменных, когда с помощью элементарных преобразований система уравнений приводится к равносильной системе треугольного вида, из которой последовательно, начиная с последних (по номеру) переменных, находятся все остальные переменные.

10. Теоретико-числовые алгоритмы

10.1. Умножение Карацубы – алгоритм быстрого умножения чисел.

10.2. Алгоритм Фюрера – на данный момент считается самым быстрым алгоритм умножения больших чисел, был построен в 2007 году швейцарским математиком Мартином Фюрером из университета штата Пенсильвания. В основе алгоритма лежит так называемая «сверстка». Допустим, что мы перемножаем числа 123 и 456 «в столбик», однако без выполнения переноса. Результат будет выглядеть так (рис. 48):

		1	2	3
	×	4	5	6
		6	12	18
	5	10	15	
4	8	12		
4	13	28	27	18

Рис. 48.

Последовательность (4, 13, 28, 27, 18) называется ациклической, или линейной, сверткой от последовательностей (1, 2, 3) и (4, 5, 6). Зная ациклическую свертку двух последовательностей, рассчитать произведение несложно: достаточно выполнить перенос (например, в самом правом столбце, мы оставляем 8 и добавляем 1 к столбцу, содержащему 27). В примере 13 это приводит к результату 56088.

10.3. Алгоритм быстрого возведения в степень – алгоритм, предназначенный для возведения числа x в натуральную степень n за меньшее число умножений, чем это требуется в определении степени.

10.4. Факторизация – разложение числа на простые множители.

10.5. Решето Эратосфена – алгоритм нахождения всех простых чисел до некоторого целого числа *n*.

11. Численные алгоритмы

11.1. Метод бисекции – метод деления отрезка пополам.

11.2. Метод Ньютона (метод касательных) – нахождение нулей функций с помощью производной.

12. Алгоритмы оптимизации

12.1. Линейное программирование

Имеется некий однородный груз, который нужно перевезти с n складов на m заводов. Для каждого склада i известно, сколько в нем находится груза a_i, а для каждого завода известна его потребность b_j в грузе. Стоимость перевозки пропорциональна расстоянию от склада до завода (все расстояния c_{ij} от i-го склада до j-го завода известны). Требуется составить наиболее дешевый план перевозки. Решающими переменными в данном случае являются x_{ij} – количества груза, перевезенного из i-го склада на j-й завод. Они удовлетворяют ограничениям:

$$x_{i1} + x_{i2} + \cdots + x_{im} \le a_i,$$
$$x_{1j} + x_{2j} + \cdots + x_{nj} \le b_j.$$

Целевая функция имеет вид: $f(x) = x_{11}c_{11} + x_{12}c_{12} + \ldots + x_{nm}c_{nm}$, которую надо минимизировать.

Пример 10: предприятие производит изделия трех видов, поставляет их заказчикам и реализует на рынке. Заказчикам требуется 1000 изделий первого вида, 2000 изделий второго вида и 2500 изделий третьего вида. Условия спроса на рынке ограничивают число изделий первого вида 2000 единицами, второго – 3000 и третьего – 5000 единицами.

Для изготовления изделий используется четыре типа ресурсов. Количество ресурсов, потребляемых для производства одного изделия, общее количество ресурсов и прибыль от реализации каждого вида изделия заданы на рисунке 49.

Тип ресурсов	**Вид изделий**			**Всего ресурсов**
	1	**2**	**3**	
1	500	300	1000	25000000
2	1000	200	100	30000000
3	150	300	200	20000000
4	100	200	400	40000000
Прибыль	20	40	50	

Рис. 49.

Как организовать производство, чтобы: обеспечить заказчиков, не допустить затоваривания, получить максимальную прибыль?

12.2. Симплекс-метод – алгоритм решения оптимизационной задачи линейного программирования путем перебора вершин выпуклого многогранника в многомерном пространстве. Метод был разработан советским математиком Л. В. Канторовичем в 1937 году. После-

довательность вычислений симплекс-методом можно разделить на две основные фазы: нахождение исходной вершины множества допустимых решений; последовательный переход от одной вершины к другой, ведущий к оптимизации значения целевой функции.

12.3. Муравьиные алгоритмы (алгоритм оптимизации подражания муравьиной колонии) – первая версия алгоритма, предложенная доктором наук Марко Дориго в 1992 году, была направлена на поиск оптимального пути в графе. В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождению продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют ее при возвращении, если в конечном итоге находят источник питания. Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа. На коротком пути, для сравнения, прохождение будет более быстрым и, как следствие, плотность феромонов остается высокой. Испарение феромонов также имеет свойство избежания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым, был бы самым привлекательным. В этом случае исследования пространственных решений были бы ограниченными. Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, другие муравьи, скорее всего, пойдут по этому пути, и положительные отзывы в конечном итоге приводят всех муравьев к одному кратчайшему пути.

Задачи для самостоятельной работы

1. Запишите алгоритм вычисления абсолютной величины числа, начертите блок-схему.
2. Запишите алгоритм решения неравенства $ax < b$, начертите блок-схему.
3. Запишите алгоритм вычисления значения функции:

$$y = \begin{cases} x-1, \text{ если } x < 1. \\ x + 1, \text{ если } 1 \le x < 5. \\ x^2-5, \text{ если } x \ge 5. \end{cases}$$ Постройте график этой функции.

4. Запишите алгоритмы решения с помощью циркуля и линейки следующих задач:
   (1) разделить данный отрезок пополам;
   (2) построить биссектрису данного угла;
   (3) описать окружность около данного треугольника;
   (4) вписать окружность в данный треугольник;
   (5) построить прямую, перпендикулярную к данной;
   (6) построить угол, равный данному;
   (7) построить треугольник с данными сторонами.
5. Запишите алгоритм и начертите блок-схему проверки существования треугольника с заданными сторонами.
6. Разработайте N–S-диаграммы для алгоритмов решения следующей задачи: определить, является ли число a делителем числа b.
7. Ответьте на вопрос: существует ли перечислимое, но неразрешимое множество натуральных чисел?
8. Докажите, что множество $M = \{1, 4, 9, \dots n^2, \dots\}$ является перечислимым и разрешимым.

9. Постройте дерево перебора всех путей в глубину для графа с рисунка 50.

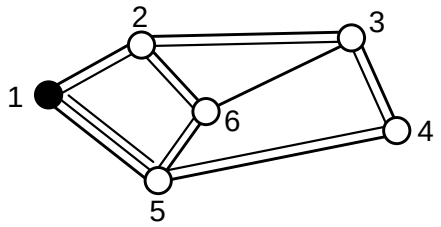


Рис. 50.

10. Используя алгоритм Крускала, построить минимальное остовное дерево (рис. 51).

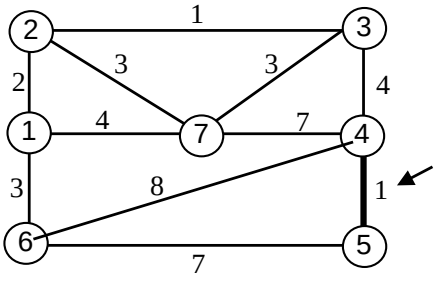


Рис. 51.

11. Запишите в виде алгоритма стихотворение:

Приметы

Собираясь на экзамен,
Валя говорила:
Если только палец мамин
Окунуть в чернила,
Если я перед доскою
Как-нибудь украдкой
Ухитрюсь одной рукою
Взять себя за пятку,
Если, сняв ботинок в школе,
Повторю заклятье,
А потом мешочек соли
Приколю на платье,
Если я в троллейбус новый
Сяду на Садовой,
А в троллейбусе вожатый
Будет бородатый,
Если я в пути не встречу
Ни единой кошки
Или вовремя замечу
И сверну с дорожки,
Не покажется священник
В нашем переулке,
И дадут мне дома денег
На кино и булки,
Если я зашью монеты –
В фартук под оборки,
То, по всем моим приметам,
Получу по всем предметам
Круглые пятерки!

С. Маршак

ЧАСТЬ II. ОСНОВНЫЕ ПОНЯТИЯ: МАШИНЫ ТЬЮРИНГА

Лекционные занятия № 6–7 «Машина Тьюринга, функции, вычислимые по Тьюрингу»

План занятия:

1. Описание машины Тьюринга (МТ).
2. Принцип работы МТ.
3. Конструирование МТ.
4. Вычислимые по Тьюрингу функции.
5. Операции над МТ.
6. Тезис Тьюринга.
7. Конечные автоматы, МТ и современные ЭВМ.

1. Описание машины Тьюринга (МТ)

Определение 1.1: машина Тьюринга (МТ) – это математическое уточнение понятия алгоритма с помощью описания абстрактного вычислительного устройства, состоящего из: ленты, считывающей (и печатающей) головки, и управляющего устройства, названного по имени английского математика Алана Тьюринга, сформулировавшего его в 1937 году, за девять лет до появления первой ЭВМ.

ЭНИАК (англ. «*Eniac*» **–** электронный числовой интегратор и вычислитель) – первый электронный цифровой компьютер общего назначения, который можно было перепрограммировать для решения широкого спектра задач.

МТ есть математическая (воображаемая) машина, а не машина физическая. Она есть такой же математический объект, как функция, производная, интеграл, группа и т. д. И так же, как и другие математические понятия, понятие «МТ» отражает объективную реальность, моделирует некие реальные процессы. Именно Алан Тьюринг предпринял попытку смоделировать действия математика (или другого человека), осуществляющего некую умственную созидательную деятельность. Такой человек, находясь в определенном «умонастроении» («состоянии»), просматривает некоторый текст. Затем он вносит в этот текст какие-то изменения, проникается новым «умонастроением» и переходит к просмотру следующих записей.

Бесконечная в обе стороны **лента** (рис. 52) разбита на ячейки (клетки). Во всякой ячейке в каждый дискретный (фиксированный) момент времени находится в точности один символ **из внешнего алфавита $A = \{a_0, a_1, \dots, a_n\}, n \geq 1$.**

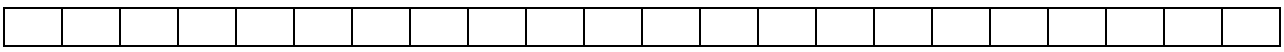


Рис. 52.

Некоторый символ a_0 алфавита A называется **пустым**, а любая ячейка, содержащая в данный момент пустой символ, называется пустой ячейкой (в данный момент). В качестве пустого символа обычно используют **0** или **λ.**

Словом в алфавите *A* (или в любом другом алфавите) называется любая последовательность букв соответствующего алфавита. Лента предполагается потенциально не ограниченной в обе стороны. Это следует понимать следующим образом: в каждый момент времени лента конечна (то есть содержит конечное число ячеек), но размеры ленты (число ячеек на ней) при необходимости можно увеличивать.

Управляющая головка передвигается вдоль ленты (рис. 53) и может останавливаться напротив какой-либо клетки, то есть воспринимать символ. В одном такте работы машины управляющая головка может сдвигаться на одну клетку (вправо – R, влево – L) или останавливаться на месте – E.

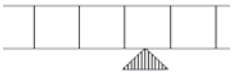


Рис. 53.

Головка считывает содержимое обозреваемой ячейки и записывает в нее (печатает в ней) вместо обозреваемого символа некоторый символ из внешнего алфавита. Записываемый в ячейку символ может, в частности, совпадать с тем, который обозревается в данный момент.

Управляющее устройство (головка) в каждый момент времени находится в некотором состоянии q_i, принадлежащем множеству $Q = \{q_0, q_1, \dots, q_m\}$. Множество Q называется **внутренним алфавитом** (или множеством внутренних состояний). Так называемая память МТ (множество Q – конечно).

Замечание: в дальнейшем, если не оговаривается противное, будем считать, что $|Q| \geq 2$, q_1 – начальное состояние, q_0 – заключительное состояние. Находясь в состоянии q_1, машина начинает работать, а попав в состояние q_0 – машина останавливается.

Работа МТ определяется программой (функциональной схемой). Программа состоит из команд.

Определение 1.2: команда – это действие устройства управления, которое записывается следующим образом:

$$q_i a_j \rightarrow q_k a_l S,$$

где q_i – внутреннее состояние МТ в данный момент;

a_j – считываемый в этот момент символ;

q_k – внутреннее состояние МТ в следующий момент;

a_l – символ, на который изменяется символ a_j (в частности оставляет его без изменения);

S – символ сдвига головки МТ: вправо – R, влево – L, на месте – E.

Выражения $q_i a_j$ и $q_k a_l S$ называются левой и правой частями команды. В левой части ни одной команды q_0 не встречается.

Определение 1.3: программа МТ – это система команд вида $q_i a_j \rightarrow q_k a_l S$, где все левые части должны быть различны, q_i (a_j) – символы внутреннего (внешнего) алфавитов, S – символ сдвига головки МТ: вправо – R, влево – L, на месте – E; $q_i, q_k \in Q, (1 \leq i \leq m, 0 \leq k \leq m), a_j, a_l \in A$.

Пример 1: является ли данный набор команд программой МТ?

$$q_1 1 \to q_1 0\ R$$
$$q_1 0 \to q_2 1\ L$$
$$q_1 0 \to q_3 0\ E$$
$$q_2 1 \to q_0 0\ E$$
$$q_0 1 \to q_1 0\ R$$

Программу можно задать в виде таблицы, которую называют **тьюринговой функциональной схемой** (рис. 54).

	a_0	a_1	…	a_j	…	a_n
q_1				…		
q_2				…		
…				…		
q_i	…	…	…	$q_k a_l S$	…	…
…				…		
q_m				…		

Рис. 54.

В клетке таблицы на пересечении i-й строки и j-го столбца вписана правая часть команды $q_k a_l S$, соответствующая левой части той же команды $q_i a_j$. Если в программе МТ команда с левой частью $q_i a_j$ отсутствует, то в таблице на пересечении строки q_i и столбца a_j ставится прочерк.

Работа МТ полностью определяется ее программой, то есть две МТ с общей функциональной схемой неразличимы, и различные МТ имеют разные программы.

Определение 1.4: МТ – это система, задаваемая тройкой $T = (A, Q, P)$, которая удовлетворяет следующим условиям:

1. Множества $A = \{a_0, a_1, \dots, a_n\}$ – внешний алфавит МТ и $Q = \{q_0, q_1, \dots, q_m\}$ – внутренний алфавит состояний управляющего устройства конечны, не пересекаются и не содержат букв R, L, E.

2. Пустой символ $a_0 \in A$; начальное и заключительное состояние $q_{1,}, q_0 \in Q$.

3. P – такая программа с внешним алфавитом A и внутренним алфавитом Q, что:

3.1 не существует в P двух различных команд вида $q_i a_j \to q_k a_l S$ с одинаковыми левыми частями;

3.2 q_0 не входит в левую часть ни одной команды из P.

4. МТ – воображаемое устройство, состоящее из управляющего устройства, реализующего функционирование МТ в дискретном времени по программе P; ленты, разделенной на ячейки; читающей-пишущей головки.

2. Принцип работы МТ

МТ функционирует в дискретном времени $t = 0,1,2, \dots$. Выполнение одной команды называется шагом. Вычисление (или работа) МТ есть последовательность шагов одного за

другим без пропусков, начиная с первого. Работа МТ начинается с задания в первый (начальный) момент:

1) слова на ленте, то есть последовательности символов из алфавита А, записанных в ячейках ленты слева направо;

2) положения считывающей головки;

3) внутреннего состояния машины, находящегося в той ячейке, на которую указывает головка.

Совокупность этих трех условий (в данный момент времени) называется конфигурацией (в данный момент времени).

Определение 2.5: конфигурацией МТ (машинным словом) называется слово К в алфавите $A \cup Q$ вида Uq_iV, где U – слово на ленте левее головки, V – слово на ленте правее слова U, начиная с символа, находящегося в той ячейке, на которую указывает головка (рис. 55).

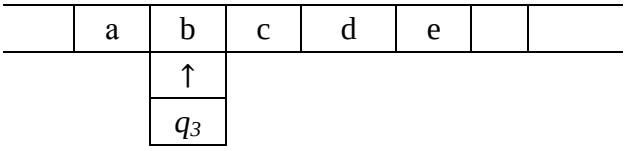


Рис. 55.

Конфигурацию в данный момент времени можно задать следующим словом: $K = aq_3bcde$, где $U = a, V = bcde$.

Стандартная начальная конфигурация – q_1V, а **стандартная заключительная конфигурация** – q_0V (рис. 56).

Замечание: условимся считать, что **стандартная МТ** считывает информацию в начальный момент времени с первого символа ленты. При этом будем считать, что в каждый момент времени на ленте записано конечное число непустых символов.

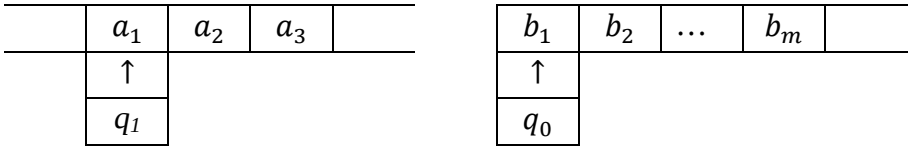


Рис. 56.

Если работа МТ завершается, то заключительное состояние возвращается к началу слова. Считывать слова всегда будем слева направо.

Если при выполнении некоторой команды МТ из конфигурации K_i получается конфигурация K_{i+1}, то K_{i+1} называется непосредственно выводимой из K_i.

Обозначение: T: $K_i \to K_{i+1}$.

Определение 2.6: протокол работы МТ – детерминированная (определенная на каждом шаге, конечная или бесконечная) последовательность конфигураций $K_{i_1}, K_{i_2}, \ldots$, такая, что $T: K_{i_r} \to K_{i_{r+1}}$, то есть при выполнении некоторой команды МТ из конфигурации K_{i_r} получается конфигурация $K_{i_{r+1}}$.

Работа МТ считается законченной в двух случаях:

1) в некоторый момент времени выполняется команда, правая часть которой содержит q_0;

2) в программе нет ни одной команды, соответствующей конфигурации, полученной на некотором шаге.

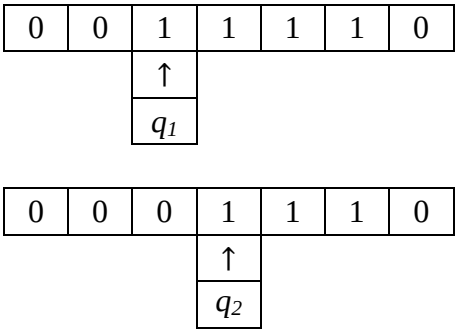
В остальных случаях будем считать работу МТ бесконечной.

Определение 2.7: говорят, что **машина Т применима к слову R**, если, начав работу на слове R, МТ (Т) остановится через конечное число шагов; если при этом получается слово Q, то пишут $T(R) = Q$. Если Т не останавливается, то Т **не применима к слову Q.**

МТ считается **применимой к множеству слов** $\{R_i\}$, если она применима к каждому слову R_i.

Зоной работы машины Т (на слове R) называется множество всех ячеек, которые за время работы машины хотя бы один раз обозреваются головкой.

Часто используют обозначение $[P]^m$ для слов вида PP…P (m раз), где $m \geq 0$, при $m = 0$ считаем, что $[P]^m$ – пустое слово (рис. 57).



$$K_1 = q_1 1111 = q_1 1^4 \rightarrow K_2 = q_2 1^3$$

Рис. 57.

Если в какой-то момент времени цепочка $K_1 \rightarrow K_2 \rightarrow \cdots \rightarrow K_m$ обрывается, то говорят, что МТ применима к начальной конфигурации (начальным данным); если последовательность бесконечна, то говорят, что МТ не применима.

Пример 2:

Дана МТ с внешним алфавитом $A = \{0, 1\}$ (0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2\}$ и со следующей функциональной схемой (программой):

$$q_1 0 \rightarrow q_2 0 L$$
$$q_2 0 \rightarrow q_0 1 E$$
$$q_1 1 \rightarrow q_1 1 L$$
$$q_2 1 \rightarrow q_2 1 L$$

Выпишем последовательно конфигурации машины при переработке слова $R = 101$, исходя из стандартного начального положения (рис. 58).

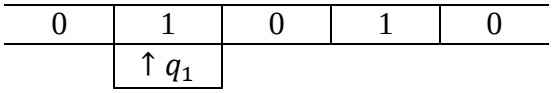


Рис. 58.

1. На первом шаге действует команда $q_1 1 \to q_1 1L$. В результате получается конфигурация (рис. 59):

	0	1	0	1	
	$\uparrow q_1$				

Рис. 59.

2. На втором шаге действует команда $q_1 0 \to q_2 0L$ (рис. 60):

	0	0	1	0	1	
	$\uparrow q_2$					

Рис. 60.

3. На третьем шаге действует команда $q_2 0 \to q_0 1E$ (рис. 61):

	1	0	1	0	1	
	$\uparrow q_0$					

Рис. 61.

Эта конфигурация является заключительной, потому что машина оказалась в заключительном состоянии остановки q_0. Таким образом, исходное слово $R = 101$ переработано в слово $Q = 10101$.

Полученную последовательность конфигураций можно записать более коротким способом: $K_1 = q_1 101 \to K_2 = q_1 0101 \to K_3 = q_2 00101 \to K_4 = q_0 10101$.

Замечание: для простоты изображения различных конфигураций МТ будем в дальнейшем записывать информацию в виде слова, не изображая ленты и ее разбивки на ячейки, а вместо изображения управляющей головки и состояния машины записывать только состояние:

$$K_2 = q_1 0^3 101 \text{ или } 0\ 0\ 0\ 1\ 0\ 1$$
$$q_1$$

Пример 3:

МТ задается внешним алфавитом $A = \{0, 1, *\}$ (0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2, q_3\}$ и программой, записанной в виде следующей таблицы (рис. 62), исходная конфигурация крайняя правая, заключительная тоже крайняя правая:

	q_1	q_2	q_3
0	$q_1 0L$	$q_3 1R$	$q_1 0L$
1	$q_2 0L$	$q_2 1L$	$q_3 1R$
*	$q_0 0$	$q_2 * L$	$q_3 * R$

Рис. 62.

Посмотрим, как эта машина перерабатывает некоторые слова, и постараемся обнаружить закономерности в ее работе. Применим машину к слову 11*11 (рис. 63).

В результате работы на ленте записано подряд столько единиц, сколько их было всего записано по обе стороны от звездочки перед началом работы машины.

		1	1	*	1	1
						q_1
		1	1	*	1	0
					q_2	
		1	1	*	1	0
				q_2		
		1	1	*	1	0
			q_2			
		1	1	*	1	0
		q_2				
	0	1	1	*	1	0
	q_2					
	1	1	1	*	1	0
		q_3				
	1	1	1	*	1	0
			q_3			
	1	1	1	*	1	0
				q_3		
	1	1	1	*	1	0
					q_3	
	1	1	1	*	1	0
						q_1
	1	1	1	*	1	0
					q_1	
	1	1	1	*	0	0
				q_2		
	1	1	1	*	0	0
			q_2			
	1	1	1	*	0	
		q_2				
	1	1	1	*	0	
	q_2					
0	1	1	1	*	0	
q_2						
1	1	1	1	*	0	
	q_3					
1	1	1	1	*	0	

		q_3				
1	1	1	1	*	0	
			q_3			
1	1	1	1	*	0	
				q_3		
1	1	1	1	*	0	
					q_3	
1	1	1	1	*	0	
				q_1		
1	1	1	1	0	0	
			q_0			

Рис. 63.

3. Конструирование МТ

Создание (синтез) МТ – это процесс написания соответствующей программы, являющийся значительно более сложной задачей, нежели процесс применения данной машины к словам.

Пример 4: попытаемся построить такую МТ, которая из n записанных подряд единиц оставила бы на ленте n-2 единицы, так же записанные подряд, если $n \geq 2$, и работала бы вечно, если n = 0,1, исходная конфигурация крайняя правая, заключительная тоже крайняя правая.

В качестве внешнего алфавита можно взять двухэлементное множество $A = \{0, 1\}$. Количество необходимых внутренних состояний будет определено в процессе составления программы. Предположим, что машина начинает работать, когда в состоянии q_1 крайняя правая единица из n, записанных на ленте.

Начнем с того, что сотрем первую единицу, если она имеется, перейдем к обозреванию следующей левой ячейки и сотрем там единицу, если она в этой ячейке записана. На каждом таком переходе машина должна переходить в новое внутреннее состояние, ибо в противном случае будут стерты вообще все единицы, записанные подряд:

$$q_1 1 \to q_2 0L,$$
$$q_2 1 \to q_3 0L.$$

Машина находится в состоянии q_3 и обозревает третью справа ячейку из тех, в которых записано данное слово (n единиц). Не меняя содержимое обозреваемой ячейки, машина должна остановиться, то есть перейти в заключительное состояние q_0, независимо от содержимого ячейки:

$$q_3 0 \to q_0 0\text{E},$$
$$\text{q}_3 1 \to \text{q}_0 1\text{E}.$$

Теперь остается рассмотреть ситуации, когда на ленте записана всего одна единица или не записано ни одной.

Если на ленте записана одна единица, то после первого шага (выполнив команду $q_1 1 \to q_2 0L$) машина будет находиться в состоянии q_2 и будет обозревать вторую справа ячейку, в которой записан 0. По условию в таком случае машина должна работать вечно. Это можно обеспечить такой командой $q_2 0 \to q_2 0R$, выполняя которую шаг за шагом, машина будет двигаться по ленте неограниченно вправо.

Если на ленте не записано ни одной единицы, то машина по условию также должна работать вечно. В этом случае в начальном состоянии q_1 обозревается ячейка с содержимым 0 и вечная работа машины обеспечивается следующей командой: $q_1 0 \to q_1 0R$.

Запишем функциональную схему (рис. 64):

	q_1	q_2	q_3
0	$q_1 0R$	$q_2 0R$	$q_0 0E$
1	$q_0 0L$	$q_2 0L$	$q_0 1E$

Рис. 64.

4. Вычислимые по Тьюрингу функции

Определение 4.8: функцию $f: X \to Y$ будем называть **частичной**, если она определена не для каждого $x \in X$. Множество тех $x \in X$, для которых однозначно указано соответствующее значение функции f, называется областью определения функции. Если область определения D_f функции f совпадает со всем множеством X (то есть D_f = X), то функция называется **всюду определенной**.

Будем рассматривать словарные частичные функции $f: A^* \to A^*$, где A^* – множество слов конечной длины в алфавите A.

Определение 4.9: функция называется вычислимой по Тьюрингу, если существует МТ, вычисляющая ее, то есть такая МТ, которая вычисляет ее значения для тех наборов значений аргумента, для которых функция определена, и работающая вечно, если функция для данного набора значений аргументов не определена.

Остается договориться о некоторых условностях для того, чтобы это определение стало до конца точным.

1. Речь идет о функциях (или, возможно, о частичных функциях), заданных на множестве натуральных чисел и принимающих так же натуральные значения.

2. Значения $x_1, x_2, \dots, x_n$ аргументов будем располагать на ленте в виде следующего слова: $0\underbrace{1 \dots 1}_{x_1}0\underbrace{1 \dots 1}_{x_2}0 \dots 0\underbrace{1 \dots 1}_{x_n}0$ (первая интерпретация).

Определение 4.10: говорят, что МТ **правильно вычисляет частичную словарную функцию** f, если для любого слова $P \in A^*$ выполнено:

1) если $f(P)$ определено и $f(P) = Q$, то МТ применима к начальной конфигурации $q_1 P$ и заключительной конфигурацией является $q_0 Q$;

2) если $f(P)$ не определено, то МТ не применима к начальной конфигурации $q_1 P$.

Аналогично можно определить вычисление числовых функций на МТ. Под числовыми функциями будем понимать функции, значениями которых и значениями их аргументов являются неотрицательные целые числа. В дальнейшем рассматриваются как всюду определенные функции, у которых $D_f = N_0$, так и частичные числовые функции, у которых $D_f \subseteq N_0$.

Пример 5: функция $\frac{y}{2}$ определена только для четных целых положительных чисел, а для остальных не определена.

При вычислении числовых функций на МТ пользуются специальным кодированием чисел: натуральному числу n ставят в соответствие набор n + 1 единиц, который обозначают через 1^n. Слово 1^{n+1} называется **основным машинным кодом,** или просто кодом числа n (рис. 65).

Таблица кодирования

Число	Код числа	Сокращенная запись
0	1	1
1	11	1^2
2	111	1^3
…	…	…
n	11..1	1^{n+1}
…	…	…

Рис. 65.

Набор чисел $x_1, x_2, \dots, x_n$ будем записывать в виде слова (вторая интерпретация) $1^{x_1+1} * 1^{x_2+1} * \dots * 1^{x_n+1}$ (код набора чисел в алфавите $\{\lambda, *, 1\}$) или $1^{x_1+1}01^{x_2+1}0 \dots 01^{x_n+1}$ (код набора чисел в алфавите $\{0, 1\}$).

Пример 6: построим МТ с внешним алфавитом $A = \{\lambda, 1\}$, правильно вычисляющую функцию O(x) = 0 (нулевая функция или оператор аннулирования).

Словесное предписание: «сотри у данного слова (последовательности единиц) все единицы, кроме одной, которая соответствует нулю, и остановись, полученное слово и есть результат».

Действие стирания единиц: $q_1 1 \to q_1 \lambda R$. Алгоритм, заданный одной этой командой, будет стирать по одной единице у исходного слова, пока управляющая головка не будет указывать на пустую ячейку, тогда действие этой команды закончится. Остается применить заключительную команду: $q_1 \lambda \to q_0 1E$, в пустую клетку ставится единица, соответствующая числу 0, и завершает работу.

Таким образом, программа МТ состоит из двух команд $P: \begin{cases} q_1 1 \to q_1 \lambda R \\ q_1 \lambda \to q_0 1E \end{cases}$ и применима к каждому слову в алфавите. Работу МТ можно представить в виде следующей последовательности конфигураций: $q_1 1^{x+1} \to q_1 1^x \to q_1 1^{x-1} \to \dots \to q_1 1^2 \to q_1 1^1 \to q_1 \lambda \to q_0 1$.

Пример 7: построим МТ с внешним алфавитом $A = \{\lambda, 1\}$, правильно вычисляющую функцию S(x) = x+1.

Словесное предписание: «припиши к данному слову (последовательности единиц) еще одну единицу и остановись, полученное слово и есть результат».

Рассмотрим два способа реализации:

1. Приписываем дополнительную единицу в начало слова:

$$P_1: \begin{cases} q_1 1 \to q_1 1L, \text{ сдвиг влево, оставляя 1 без изменений.} \\ q_1 \lambda \to q_0 1E, \text{ записать 1 в пустую ячейку и остановиться.} \end{cases}$$

Последовательность конфигураций: $q_1 1^{x+1} \to q_1 \lambda\, 1^{x+1} \to q_0 1^{x+2}$.

Данная МТ P_1 применима ко всем $x \in N_0$.

В частном случае $S(2)$:

$K_1 = q_1\, 111$

$K_2 = q_1\, \lambda 111$

$K_3 = q_0\, 1111$

2. Приписываем дополнительную единицу в конец слова

$$P_2: \begin{cases} q_1 1 \to q_1 1R, \text{ цикл движется вправо до конца слова.} \\ q_1 \lambda \to q_2 1L, \text{ записать 1 в пустую ячейку и сдвиг влево.} \\ q_2 1 \to q_2 1L, \text{ цикл движется влево к началу слова.} \\ q_1 \lambda \to q_0 \lambda R, \text{ конец цикла, остановка.} \end{cases}$$

Последовательность конфигураций $q_1 1^{x+1} \to 1 q_1 1^x \to \ldots \to \lambda 1^x q_1 1 \to 1^{x+1} q_1 \lambda \to 1^{x+1} q_2 1 \to 1^x q_2 1^2 \to \ldots \to q_2 1^{x+2} \to q_2 \lambda\, 1^{x+2} \to q_0 1^{x+2}$.

В частном случае $S(2)$:

$K_1 = q_1\, 111$

$K_2 = 1q_1\, 11$

$K_3 = 11q_1\, 1$

$K_4 = 111q_1\, \lambda$

$K_5 = 11q_2\, 11$

$K_6 = 1q_2\, 111$

$K_7 = q_2\, 1111$

$K_8 = q_2\, \lambda 1111$

$K_9 = q_0\, 1111$

Определение 4.11: машины Тьюринга T_1 и T_2 называются эквивалентными (в алфавите A), если для всякого входного слова R (в алфавите A) выполняется соотношение $T_1(R) \simeq T_2(R)$ (условно равны), означающее следующее: результаты $T_1(R)$ и $T_2(R)$ определены или не определены одновременно (то есть машины T_1 и T_2 либо обе применимы, либо обе не применимы к слову R) и, если эти результаты определены, то $T_1(R) = T_2(R)$. Символ $\simeq$ называется **знаком условного равенства**.

5. Операции над МТ

Прямое построение МТ для решения даже простых задач может оказаться затруднительным. Однако существуют способы, позволяющие конструировать сложные МТ из более простых. Рассмотрим основные операции над МТ: композицию (синонимы: произведение, суперпозиция), возведение в степень, ветвление и итерацию.

Будем рассматривать МТ, которые правильно вычисляют частичные словарные функции $f: A^* \to A^*$, где A^* – множество слов конечной длины в алфавите A.

5.1. Композиция

Композицией двух функций $f_1(R)$ и $f_2(R)$ называется функция $g(R) = f_2\big(f_1(R)\big)$, которая получается при применении f_2 к результату вычисления f_1. Для того чтобы $g(R)$ была

определена на данном слове R, необходимо и достаточно, чтобы f_1 была определена на R и f_2 была определена на $f_1(R)$.

Пусть машины T_1 и T_2 вычисляют соответственно $f_1(R)$ и $f_2(R)$, они имеют какие-то внешние алфавиты $A = \{a_0, a_1, \dots, a_m\}$, $A = \{a_0', a_1', \dots, a_k'\}$, внутренние алфавиты $Q = \{q_0, q_1, \dots, q_n\}$, $Q = \{q_0', q_1', \dots, q_t'\}$ и соответственно программы P_1 и P_2.

Композицией машин $T_1 = (A_1, Q_1, P_1)$ и $T_2 = (A_2, Q_2, P_2)$ будем называть машину $T = T_1^\circ T_2$ с внешним алфавитом $A_1 \cup A_2$, внутренним алфавитом $Q = \{q_1, \dots, q_n, q_0 = q_1', q_2', \dots, q_t'\}$ и программой $P = P_1' \cup P_2'$, где P_1' получается из P_1 заменой во всех командах программы P_1 заключительного символа q_0 на символ q_1' – какое-либо начальное состояние T_2, а программа P_2' получается из P_2 следующим образом: заключительное состояние не меняется, но зато каждый из остальных символов q_j' (j = 1, 2, .., t) заменяется символом q_{n+j}'. Новая программа P определяет машину T, называемую композицией машин T_1 и T_2 (по паре состояний (q_0, q_1')) и обозначаемую через $T_1^\circ T_2$ или $T_1 T_2$, более подробно $T = T(T_1, T_2, (q_0, q_1'))$.

Схематически композиция обозначается следующим образом:

$$R \underset{T_1}{\rightarrow} f_1(R) \underset{T_2}{\rightarrow} f_2\big(f_1(R)\big).$$

Замечание: в том случае, когда одна из перемножаемых машин имеет два или несколько заключительных состояний, умножение определяется совершенно аналогично, но только обязательно должно присутствовать указание, какое из заключительных состояний предыдущего сомножителя отождествляется с начальным состоянием последующего.

Операция композиции машин ассоциативна, то есть $(T_1^\circ T_2)^\circ T_3 = T_1^\circ (T_2^\circ T_3)$, но не коммутативна $T_1^\circ T_2 \neq T_2^\circ T_1$.

Пример 8: пусть машина T_1 задана тьюринговой функциональной схемой (рис. 66):

	0	1
q_1	$q_2$0R	$q_1$1R
q_2	$q_2$0R	$q_0$1E

Рис. 66.

Эта машина отыскивает ближайшую справа группу единиц после данной группы единиц и нулей.

Машина T_2 задана таблицей (рис. 67):

	0	1
${q_1}^2$	${q_0}^2$0E	${q_1}^2$0R

Рис. 67.

Эта машина стирает все единицы, если они есть, до ближайшего справа нуля.

В исходном состоянии считывающие головки машин T_1 и T_2 находятся у крайней левой заполненной ячейки.

Тогда машина $T = T(T_1, T_2, (q_0, {q_1}^2))$, являющаяся их произведением, будет иметь таблицу с тремя состояниями, ${q_0}^2$ – заключительное состояние машины T_2 (рис. 68):

	0	1
q_1	$q_2$0R	$q_1$1R
q_2	$q_2$0R	$q_3$1E
q_3	$q_0$0E	$q_3$0R

Рис. 68.

Машина T отыскивает ближайшую справа группу единиц и стирает их.

5.2. Операция возведения в степень

n-й степенью машины T называется произведение $TT \ldots T = T^n$ с n сомножителями.

Так, например, зная МТ, вычисляющую функцию S(x) = x+1, можно построить МТ, вычисляющую функцию F(x) = x+5: $T°T°T°T°T = T^5$.

5.3. Операция итерации (или организация цикла)

Операция итерации применима к одной машине. Суть операции состоит в следующем: пусть машина T_1 имеет несколько заключительных состояний и q' – одно из них, а q'' – какое-либо состояние машины T_1, не являющееся заключительным. Заменим всюду в программе P символ q' на символ q''. Получим программу $P_{\text{нов}}$, определяющую машину $T\,(q', q'')$. Машина T называется **итерацией** машины T_1 по паре состояний (q', q'').

Замечание: если T_1 имеет только одно заключительное состояние, то после выполнения итерации получается машина, не имеющая заключительного состояния вовсе.

$q_i = q_j$ – этим можно организовать цикл.

Пример 9: пусть МТ, отправляясь от числа, воспринятого в стандартном положении, стирает число правее данного (если такое имеется), задана функциональной схемой (рис. 69):

	q_1	q_2	q_3	q_4	q_5	q_6
0	$q_2$0R	$q_3$0L	$q_4$0 L	$q_4$0 L	$q_0$0R	$q_0$0E
1	$q_1$1R	$q_6$0R	–	$q_5$1 L	$q_5$1 L	$q_6$0R

Рис. 69.

Эта команда содержит две команды с заключительным состоянием q_0. Применив операцию зацикливания (заменив в команде $q_6 0 \rightarrow q_0$0E состояние q_0 состоянием q_1), получим МТ, которая, отправляясь от числа, воспринятого в стандартном положении, стирает все числа правее данного (если таковые имеются) до первого встречного промежутка из двух и более пустых ячеек и возвращается к стандартному положению первоначально воспринятого числа (рис. 70).

5.4. Ветвление

Пусть машины Тьюринга T_1, T_2 и T_3 имеют общий алфавит A и задаются программами P_1, P_2, P_3 соответственно. Считаем, что внутренние алфавиты этих машин попарно не пересекаются (иначе их внутренние состояния можно переобозначить). Пусть q_0', q_0'' – какие-либо заключительные состояния машины T_1. Заменим всюду в программе P_1 состояние q_0' некоторым начальным состоянием $q_1{}^2$ машины T_2, а состояние q_0'' начальным состоянием $q_1{}^3$ машины T_3. Затем новую программу объединим с P_2 и P_3. Получим программу P, задающую МТ $(T) = T\left(T_1, (q_0', q_1{}^2), T_2\left(q_0'', q_1{}^3\right), T_3\right)$.

Эта машина называется **разветвлением** машин T_2 и T_3, управляемым машиной T_1.

	1	0	1	
q_1				
1	1	0	1	
	q_1			
1	1	0	1	
		q_1		
1	1	0	1	
			q_2	
1	1	0	0	0
				q_6
1	1	0	0	0
				q_0

	1	1	0	0	0	
					q_1	
	1	1	0	0	0	0
						q_2
	1	1	0	0	0	0
					q_3	
	1	1	0	0	0	0
				q_4		
	1	1	0	0	0	0
			q_4			
	1	1	0	0	0	0
		q_4				
	1	1	0	0	0	0
	q_5					
0	1	1	0	0	0	0
q_5						
0	1	1	0	0	0	0
	q_0					

Рис. 70

Пример 10: sg(x) $= \begin{cases} 0, & x = 0 \\ 1, & x \geq 1 \end{cases}$ (рис. 71).

Sgn (сигнум, от лат. *«signum»* – знак) – кусочно-постоянная функция. Обозначается sgn(x) или sg(x) (данное обозначение используется в теории алгоритмов). Функцию sgn(x) ввел Леопольд Кронекер в 1878 году, определяя ее следующим образом:

$$sgn\,(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}.$$

	0	1	Комментарии
q_1	–	$q_2 1R$	T_1, если $f(0) = 0$
q_2	$q_0 0L$	$q_3 1R$	
q_3	$q_4 0L$	$q_3 0R$	T_2 стирает слово
q_4	$q_4 0L$	$q_5 1L$	T_3 возвращает в начало
q_5	$q_0 0R$	$q_5 1L$	

Рис. 71.

Сделаем проверку (рис. 72):

$x = 0$

0	0	1	0	0
		q_1		
0	0	1	0	0
			q_2	
0	0	1	0	0
		q_0		

$x = 3$

	1	1	1	1	0
	q_1				
	1	1	1	1	0
		q_2			
	1	1	1	1	0
			q_3		
	1	1	0	1	0
				q_3	
	1	1	0	0	0
					q_3
	1	1	0	0	0
				q_4	
	1	1	0	0	0
			q_4		
	1	1	0	0	0
		q_4			
	1	1	0	0	0
	q_5				
0	1	1	0	0	0
q_5					
0	1	1	0	0	0
	q_0				

Рис. 72.

6. Тезис Тьюринга

МТ дает один из путей уточнения понятия алгоритма. В связи с этим возникают вопросы: можно ли считать, что способ задания алгоритмов с помощью МТ является универсальным? Может ли всякий алгоритм задаваться таким образом? На эти вопросы современная теория алгоритмов предлагает ответ в виде следующей гипотезы, которая называется тезисом Тьюринга.

Тезис Тьюринга: всякий алгоритм может быть реализован МТ.

Это утверждение нельзя доказать, так как оно связывает нестрогое интуитивное понятие алгоритма со строгим определением МТ. Эта гипотеза может быть опровергнута, если удастся привести пример алгоритма, который не может быть реализован с помощью тьюринговой функциональной системы. Однако все известные до сих пор алгоритмы могут быть реализованы в виде МТ.

7. Конечные автоматы, МТ и современные ЭВМ

Наряду с рассматриваемым основным вариантом МТ изучаются его различные модификации, например, машины, имеющие несколько лент. В каждом такте работы такой машины устройство управления по информации, считанной со всех лент, и по своему внутреннему состоянию вырабатывает команду для каждой ленты. Наличие нескольких лент облегчает решение задач на машине.

Рассмотрим машину Т с двумя лентами, на каждой из которых используется свой алфавит. Пусть машина использует первую ленту только для чтения, вторую – только для записи и в каждом такте работает следующим образом: читается буква с первой ленты, пишется буква на вторую ленту, и обе головки сдвигаются на одну ячейку вправо. Перед началом работы на первую ленту помещается исходное слово, а вторая лента пуста. После считывания всего слова (при первом считывании пустой буквы) машина останавливается, и слово на второй ленте является результатом работы машины. Такая машина называется конечным автоматом. Конечные автоматы и автоматы других типов детально изучаются в теории автоматов.

Изучение машин Тьюринга и практика составления программ для них закладывают фундамент алгоритмического мышления. Сущность которого состоит в том, что нужно уметь разделять тот или иной процесс вычисления или какой-либо другой деятельности на простые составляющие шаги. В МТ расчленение (анализ) вычислительного процесса на простейшие операции доведено до предельной возможности: распознавание единичного рассмотренного вхождения символа, перемещение точки наблюдения данного ряда символов в соседнюю точку и изменение имеющейся в памяти информации. Конечно, такое мелкое дробление вычислительного процесса, реализуемого в МТ, значительно его удлиняет. Но вместе с тем логическая структура процесса, расчлененного до атомарного состояния, значительно упрощается и предстает в некотором стандартном виде, весьма удобном для теоретических исследований.

Таким образом, понятие МТ есть теоретический инструмент анализа алгоритмического процесса, а значит, анализа существования алгоритмического мышления.

В современных ЭВМ алгоритмический процесс расчленен не на столь мелкие составляющие, как в МТ. Наоборот, создатели ЭВМ стремятся к укрупнению выполняемых машиной процедур. Так, для выполнения операции сложения на МТ составляется целая программа, а в современной ЭВМ такая операция считается простейшей. Далее, МТ обладает бесконечной внешней памятью (не ограниченная в обе стороны лента), но ни в одной реально существующей машине бесконечной памяти быть не может. Это говорит о том, что МТ отображают потенциальную возможность неограниченного увеличения объема памяти современных ЭВМ.

Подводя итоги, можно сделать вывод, что современные ЭВМ есть некие реальные физические модели машин Тьюринга. Огрубленные с точки зрения теории, но созданные в це-

лях реализации конкретных вычислительных процессов. В свою очередь, понятие МТ и теория таких машин есть теоретический фундамент и обоснование современных ЭВМ.

Практическое занятие № 8 «Машины Тьюринга»

1. Машина T_1 работает в алфавите $M = \{\lambda, 0, 1\}$. Множество внутренних состояний машины $Q = \{q_1, q_2\}$.

Функциональная таблица (рис. 73):

	λ	0	1
q_1	$q_1 1R$	$q_0 0E$	$q_2 1R$
q_2	$q_2 1R$	$q_2 1R$	$q_2 1R$

Рис. 73.

Записать список команд и проверить применимость машины T_1 к словам $\propto_0 = 0, \propto_1 = 1$.

2. Машина T_2 работает в алфавите $M = \{\lambda, 0, 1\}$. Множество внутренних состояний машины $Q = \{q_1\}$.

Список команд:

$$q_1\lambda \to q_0 1E$$
$$q_1 0 \to q_1 \lambda R$$
$$q_1 1 \to q_1 \lambda R$$

Проверить применимость машины T_2 к исходному слову $\propto_0 = 10$.

Машина T_3 работает в алфавите $M = \{\lambda, 0, 1\}$. Множество внутренних состояний машины $Q = \{q_1, q_2\}$.

Функциональная таблица (рис. 74):

	λ	0	1
q_1	$q_2 1R$	$q_1 \lambda R$	$q_1 \lambda R$
q_2	$q_2 1R$	$q_0 0E$	$q_0 1E$

Рис. 74.

Проверить применимость машины T_3 к слову $\propto_0 = 10$.

3. Определить, что делает машина T_4 в алфавите $M = \{\lambda, *\}$.

Список команд:

$$q_1\lambda \to q_0 * E$$
$$q_1 * \to q_1 * L.$$

Исходное слово: $\propto_0 = ***$.

4. Записать протокол работы машины T_5 в алфавите $M = \{\lambda, 1, +\}$.

Функциональная таблица (рис. 75):

	λ	1	+
q_1		$q_2\lambda R$	$q_0\lambda R$
q_2	$q_2\lambda R$	$q_2 1R$	$q_3 1L$
q_3	$q_0\lambda R$	$q_3 1L$	

Рис. 75.

Исходное слово: $\propto_0 = 1111 + 11$.

5. Машина T_6 вычисляет характеристическую функцию предиката $P(n)$ – «число n является нечетным».

$$f_p(n) = \begin{cases} 1, \text{ n – нечетное,} \\ 0, \text{ n – четное.} \end{cases}$$

Число n записывается в унарной системе буквой $*$. Составить программу для машины T_6 для слов $\propto_0 = **, \propto_1 = ***$.

6. Выяснить, применима ли машина Тьюринга, задаваемая программой P, к слову W (исходя из стандартного положения). Если машина Тьюринга применима, то выписать результат применения к слову W.

P: $q_1 0 \to q_2 1R$
$q_1 1 \to q_1 0L$
$q_2 0 \to q_3 1R$, a) $W_1 = 10^3 1$, b) $W_2 = (10)^2 1$
$q_2 1 \to q_3 0L$
$q_3 0 \to q_1 0R$

7. По заданной машине Тьюринга T и начальной конфигурации K_1 найти заключительную конфигурацию (q_0 – заключительное состояние).

(1) T: $q_1 0q_2 1R$, $q_1 1q_2 0L$, $q_2 0q_0 1E$, $q_2 1q_1 1L$; a) $K_1 = 1^2 01q_1 1^2$, b) $K_1 = 101q_1 01^2$.

(2) T: $q_1 0q_1 1L$, $q_1 1q_2 1R$, $q_2 1q_1 0R$, $q_2 0q_0 0L$; a) $K_1 = 1q_1 01^3$, b) $K_1 = 1q_1 1^4$.

8. Вычислить, применима ли машина Тьюринга T, задаваемая программой P, к слову $\propto$. Предполагается, что q_1 – начальное состояние, q_0 – заключительное состояние, и в начальный момент машина обозревает самую левую единицу на ленте.

(1) P: $q_1 0q_2 1R$, $q_1 1q_2 1L$, $q_2 0q_3 1R$, $q_2 1q_3 0R$, $q_3 1q_1 1R$; a) $\propto_1 = 1^3 01^2$, b) $\propto_2 = 1^2 0^2 1$, c) $\propto_3 = 1^5$, d) $\propto_4 = 1^2 (01)^2$.

$q_1 0 q_1 1 R$
$q_1 1 q_2 0 R$

(2) P: $q_2 0 q_1 1 R$, a) $\propto_1 = (10)^3 1$, b) $\propto_2 = 10^2 1^2$, c) $\propto_3 = 10^3 1$.
$q_2 1 q_3 1 L$
$q_3 0 q_1 1 L$

Практическое занятие № 9 «Операции над машинами Тьюринга»

1. Построить композицию $T_1 \cdot T_2$ машин Тьюринга T_1 и T_2 по паре состояний (g_{10}, g_{21}) и найти результат применения композиции $T_1 \cdot T_2$ $(T_2 \cdot T_1)$ к слову $R = 1^4 0^2 1^3 0 1^2$ (g_{20} – заключительное состояние T_2) (рис. 76):

T_1:

	0	1				
g_{11}	$g_{10}0L$	$g_{12}1R$		g_{21}	$g_{22}1L$	$g_{22}1L$
g_{12}	$g_{13}0R$	$g_{13}1R$		g_{22}	$g_{20}0R$	$g_{21}0L$
g_{13}	$g_{11}0R$	$g_{11}0R$				

T_2:

Рис. 76.

2. Найти результат применения итерации машины Тьюринга T по паре состояний (g_0, g_1) к слову $R = 1^6$ (заключительными состояниями являются g_0 и $g_0{}'$) (рис. 77).

	g_1	g_2	g_3	g_4	g_5	g_6
0	$g_0 0R$	$g_0 0R$	$g_4 0R$	$g_5 1L$	$g_6 0L$	$g_0 0R$
1	$g_2 0R$	$g_3 0L$	$g_3 1R$	$g_4 1R$	$g_5 1L$	$g_6 1L$

Рис. 77.

3. Найти результат применения машины $T = T(T_1, (g_{10}{}', g_{21}), T_2, (g_{10}{}'', g_{31}), T_3)$ к слову $R = 1^x 0^2 1$, $x = 3$ (g_{20} – заключительное состояние T_2, g_{30} – заключительное состояние T_3) (рис. 78):

T_1				T_2			T_3		
	g_{11}	g_{12}	g_{13}		g_{21}	g_{22}		g_{31}	g_{32}
	$g_{12}0R$	$g_{10}{}'0L$	$g_{10}{}''0R$		$g_{22}0L$	$g_{20}1R$		$g_{32}0R$	$g_{30}1E$
	$g_{11}1R$	$g_{13}1R$	$g_{13}1R$		$g_{21}1L$	$g_{22}0L$		$g_{31}1R$	$g_{31}1R$

Рис. 78.

4. **Перенос нуля *A*.** Данная машина осуществляет перевод начальной конфигурации $g_1 001^x 0$ в слово $g_0 01^x 0$ 0.

5. **Левый сдвиг** Б$^-$**.** Данная машина перерабатывает слово $01^x g_1 0$ в слово $g_0 01^x 0$.

6. **Правый сдвиг** Б$^+$**.** Данная машина перерабатывает слово $g_1 01^x 0$ в слово $01^x g_0 0$.

7. **Транспозиция В.** Данная машина перерабатывает слово $01^x g_1 01^y 0$ в слово $01^y g_0 01^x 0$.

Практическое занятие № 10 «Конструирование машин Тьюринга»

1. Построить машину Тьюринга, которая применима ко всем словам в алфавите $\{a, b\}$ и делает следующее: любое слово $x_1 x_2 \dots x_n$, где $x_i \in \{a, b\}$, для всех $1 \leq i \leq n$, преобразует в слово $x_2 \dots x_n x_1$. Проверить для слов: R = ba, R = abb.

2. Построить машину Тьюринга, применимую ко всем словам $x_1 x_2 \dots x_n$, в алфавите $\{a, b\}$ и переводящую их в слово $\propto$.

$$\propto = \begin{cases} x_n, \text{ если } x_{n-1} = a \\ b^{n-1} x_n, \text{ если } x_{n-1} = b,\ n > 1. \end{cases}$$

3. Построить машину Тьюринга с внешним алфавитом $A = \{\lambda, 0, 1, 2, \dots 9\}$, правильно вычисляющую функцию $s(x) = x + 1$ в десятичной системе счисления (два или три варианта).

4. Построить машину Тьюринга с внешним алфавитом $A = \{0,1,*\}$, правильно вычисляющую функцию $sum(x, y) = x + y$.

5. Написать формулу числовой функции $f(x_1, x_2)$, вычисляемой машиной множеством внутренних состояний Q, если машина задана своей программой (рис. 79). Проверьте работу машины Тьюринга с некоторым набором значений аргумента.

	g_1	g_2	g_3	g_4	g_5	g_6
λ		$g_5 \lambda R$	$g_4 1L$	$g_0 1E$	$g_6 \lambda E$	$g_0 1E$
1	$g_2 \lambda R$	$g_3 1R$	$g_3 1R$	$g_4 1L$	$g_5 \lambda R$	

Рис. 79.

6. Построить в алфавите $A = \{0,1\}$ машину Тьюринга, которая применима к словам вида $1^{2m+1}01^{2n+1}$ $(m \geq 0, n \geq 0)$ и $1^{2m}01^{2n}$ $(m \geq 1, n \geq 1)$, но не применима к словам вида $1^{2m}01^{2n-1}$ и $1^{2m-1}01^{2n}$ $(m \geq 1, n \geq 1)$. К словам иного вида машина может быть как применима, так и не применима.

7. Построить машину Тьюринга, вычисляющую алгоритм Евклида.

Практические занятия № 11–12 «Функции, вычислимые по Тьюрингу»

1. Построить машину Тьюринга, правильно вычисляющую функцию f:
   (1) $f(x) = 0$
   (2) $f(x) = x + 2$
   (3) $f(x) = x - 1$
   (4) $f(x) = x \dot{-} 1$
   (5) $f(x) = sg(x)$
   (6) $f(x) = x - 5$
   (7) $f(x) = x \dot{-} 5$
   (8) $f(x, y) = y$
   (9) $f(x, y) = x$
   (10) $f(x, y) = x \dot{-} y$

(11) $f(x) = \overline{sg}(x-1)$
(12) $f(x) = sg(x-3)$
(13) $f(x) = x+2$, в десятичной системе счисления
(14) $f(x) = \begin{cases} 1, \text{ если x делится на 2} \\ 0, \text{ если x не делится на 2} \end{cases}$
(15) $f(x) = \frac{x}{2}$
(16) $f(x) = \begin{cases} 1, \text{ если x делится на 3} \\ 0, \text{ если x не делится на 3} \end{cases}$
(17) $f(x) = \max(x, 3)$
(18) $f(x) = |x-5|$
(19) $f(x) = \left[\frac{1}{x}\right]$

(20) $f(x) = \left[\frac{x}{2}\right]$
(21) $f(x) = \frac{2}{4-x}$
(22) $f(x,y) = \frac{x}{2} + y$
(23) $f(x) = 2^{1-x}$
(24) $f(x) = 3x$

2. По программе машины Тьюринга напишите формульное выражение функции $f(x,y)$, вычисляемой этой машиной (рис. 80):

	g_1	g_2	g_3	g_4	g_5	g_6
0	$g_2 0R$	$g_1 0L$	$g_4 0R$	$g_4 0L$	$g_6 0R$	$g_0 0E$
1	$g_1 1R$	$g_3 0R$	$g_3 0R$	$g_5 1L$	$g_5 1L$	$g_0 1E$

Рис. 80.

Задачи для самостоятельной работы

1. По заданной машине Тьюринга T и начальной конфигурации K_1 найти заключительную конфигурацию (q_0 – заключительное состояние).

(1) T: $\begin{matrix} q_1 0 q_2 0 L \\ q_1 1 q_1 0 R \\ q_2 0 q_2 1 L \\ q_2 1 q_0 0 R \end{matrix}$, a) $K_1 = 10^3 q_1 01$, b) $K_1 = 1^2 q_1 1^3 01$.

(2) T: $\begin{matrix} q_1 0 q_0 1 E \\ q_1 1 q_2 0 R \\ q_2 0 q_1 0 R \\ q_2 1 q_2 1 L \end{matrix}$, a) $K_1 = 1^2 q_1 1^3 01$, b) $K_1 = 1 q_1 1^4$.

2. Вычислить, применима ли машина Тьюринга T, задаваемая программой P, к слову $\propto$. Предполагается, что q_1 – начальное состояние, q_0 – заключительное состояние и в начальный момент машина обозревает самую левую единицу на ленте.

(1) P: $\begin{matrix} q_1 0 q_1 0 R \\ q_1 1 q_2 0 R \\ q_2 1 q_1 0 R \\ q_2 0 q_0 1 E \end{matrix}$, a) $\propto_1 = 1^3 01$, b) $\propto_2 = 1^2 0^2 1$.

(2) P: $\begin{matrix} q_1 0 q_2 1 L \\ q_1 1 q_2 1 R \\ q_2 1 q_1 1 R \end{matrix}$, a) $\propto_1 = 1^2 0^2 1$, b) $\propto_2 = 1^6$, c) $\propto_3 = 1^2 01^3$.

(3) P: $\begin{matrix} q_1 0 q_2 1 R \\ q_1 1 q_2 1 R \\ q_2 0 q_3 0 R \\ q_2 1 q_1 0 L \\ q_3 0 q_2 1 E \\ q_3 1 q_0 0 L \end{matrix}$, a) $\propto_1 = 1^2$, b) $\propto_2 = 1^2 0^2 1$, c) $\propto_3 = 10^4 1$.

3. Построить композицию $T_1 \cdot T_2$ машин Тьюринга T_1 и T_2 по паре состояний (g_{10}, g_{21}) и найти результат применения композиции $T_1 \cdot T_2$ ($T_2 \cdot T_1$) к слову R (g_{20} – заключительное состояние T_2) (рис. 81–83):

(1) T_1: T_2:

	0	1			0	1
g_{11}	$g_{12}0R$	$g_{12}1R$		g_{21}	$g_{22}1R$	$g_{21}0L$
g_{12}	$g_{10}1L$	$g_{11}0R$		g_{22}	$g_{21}1R$	$g_{20}1E$

Рис. 81.

а) $R = 1^3 0^2 1^2$, б) $R = 1^4 01(110)^3$

(2) T_1: T_2:

	0	1			0	1
g_{11}	$g_{10}0L$	$g_{12}1R$		g_{21}	$g_{22}0L$	$g_{21}1L$
g_{12}	$g_{13}0R$	$g_{13}1R$		g_{22}	$g_{23}0L$	$g_{22}1L$
g_{13}	$g_{11}0R$	$g_{11}0R$		g_{23}	$g_{20}0R$	$g_{23}1L$

Рис. 82.

а) $R = 1^3 0^2 1^2$, б) $R = 1^4 01(110)^3$

(3) T_1: T_2:

	0	1			0	1
g_{11}	$g_{12}0R$	$g_{11}1R$		g_{21}	$g_{22}0L$	$g_{21}1L$
g_{12}	$g_{13}0R$	$g_{11}1R$		g_{22}	$g_{23}0L$	$g_{22}1L$
g_{13}	$g_{10}1L$			g_{23}	$g_{20}0R$	$g_{23}1L$

Рис. 83.

а) $R = 1^3 0^2 1^2$, б) $R = 1^4 01(110)^3$

4. Найти результат применения итерации машины Тьюринга T по паре состояний (g_0, g_2) к слову R (заключительными состояниями являются g_0 и $g_0{}'$) (рис. 84–85).

	g_1	g_2	g_3	g_4	g_5
0	$g_0 0E$	$g_4 0E$	$g_5 0E$	$g_4 1R$	$g_0{}' 1L$
1	$g_2 0R$	$g_3 0R$	$g_1 0R$		

Рис. 84.

а) $R = 1^4 01$, б) $R = 1^3 01^3$.

5. Найти результат применения машины $T = T(T_1, (g_{10}', g_{21}), T_2, (g_{10}'', g_{31}), T_3)$ к слову R (g_{20} – заключительное состояние T_2, g_{30} – заключительное состояние T_3) (рис. 83).

T_1				T_2			T_3		
	g_{11}	g_{12}			g_{21}			g_{31}	g_{32}
0	$g_{12}0R$	$g_{10}'0R$		0	$g_{20}1E$		0	$g_{32}1L$	$g_{30}1E$
1	$g_{11}1R$	$g_{10}''1L$		1	$g_{21}0R$		1	$g_{31}1R$	$g_{32}0L$

Рис. 85.

а) $R = 101^3$, б) $R = 1^3 01$.

6. **Удвоение Г.** Данная машина перерабатывает слово $g_1 01^x 0$ в слово $g_0 01^x 01^x 0$.

7. **Циклический сдвиг Ц.** Данная машина перерабатывает слово $01^x 01^y g_1 01^z 0$ в слово $01^z g_0 01^x 01^y 0$.

8. **Копирование К.** Данная машина перерабатывает слово $g_1 01^x 01^y$ в слово $01^x 01^y g_0 01^x 01^y$.

9. Дана конечная совокупность единиц, вписанных в ячейки, взятые подряд без пропусков. Постройте функциональную схему такой машины Тьюринга, которая записывала бы в десятичной системе число этих единиц, то есть пересчитывала бы набор единиц (дешифратор).

10. Построить машину Тьюринга, которая подсчитывает количество букв «a» в любом слове, записанном в алфавите $\{\lambda, a, b\}$.

11. Построить машину Тьюринга, которая подсчитывает количество букв «b» в любом слове, записанном в алфавите $\{\lambda, a, b\}$ и содержащем более одной буквы «b», иначе машина должна заменить исходное слово на «bbb».

12. Построить машину Тьюринга, преобразующую любое слово в алфавите $\{\lambda, a, b\}$, содержащее хотя бы две буквы «a», в слово «$baabab$», иначе слово стереть.

13. Построить машину Тьюринга, преобразующую любое слово в алфавите $\{\lambda, a, b\}$, содержащее хотя бы две буквы «a», в слово «$baabab$», иначе должна оставить слово без изменения.

14. Построить машину Тьюринга, преобразующую любое слово в алфавите $\{\lambda, a, b\}$, содержащее хотя бы две буквы «b», в слово «abb», иначе в слово, полученное из исходного заменой «a» на «b» и наоборот.

15. Построить машину Тьюринга, которая любое слово в алфавите $\{\lambda, a, b\}$ удваивает.

16. Построить машину Тьюринга, которая каждое слово $x_1 x_2 \ldots x_n$ в алфавите $\{\lambda, a, b\}$ преобразует в слово $x_n x_{n-1} \ldots x_2 x_1$.

17. По программе машины Тьюринга написать аналитическое выражение для функций $f(x)$ и $f(x, y)$, вычисляемых машиной (рис. 86):

а)				б)		
	g_1	g_2			g_1	g_2
0	$g_2 1L$	$g_0 0R$		0	$g_2 0R$	$g_1 0L$
1	$g_1 1R$	$g_2 1L$		1	$g_1 1R$	$g_0 0R$

Рис. 86.

18. Построить машину Тьюринга, правильно вычисляющую функцию f:

(1) $f(x) = 2x$

(2) $f(x) = \frac{x}{3}$

(3) $f(x) = \left[\frac{x}{3}\right]$

(4) $f(x) = \left[\frac{1}{x-3}\right]$

(5) $f(x) = 2x + 1$

(6) $f(x) = x^2$

(7) $f(x) = x \div 3$

(8) $f(x) = x - 3$

(9) $f(x) = 2x \div 3$

(10) $f(x) = 2x - 3$

(11) $f(x) = \overline{sg}(x) = \begin{cases} 1, \text{ если } x=0 \\ 0, \text{ если } x \geq 1 \end{cases}$

(12) $f(x) = sg\ (x \div 3)$

(13) $f(x) = \overline{sg}(x - 3)$

(14) $f(x) = sg(2x - 3)$

(15) $f(x) = \overline{sg}(2x - 3)$

(16) $f(x) = \overline{sg}(2x \div 3)$

(17) $f(x, y) = 2x + y$

(18) $f(x, y) = x - y$

(19) $f(x, y) = x \cdot y$

(20) $f(x, y) = \max(x, y)$

(21) $f(x, y) = \frac{2-x}{2|3-y^2|}$

(22) $f(x, y) = \frac{2+x}{2-y}$

(23) $f(x, y) = \frac{4-2x}{y^2}$

(24) $f(x) = x \div 1$ (в десятичной системе)

(25) $f(x) = x \div 2$ (в десятичной системе)

ЧАСТЬ III. ОСНОВНЫЕ ПОНЯТИЯ: НОРМАЛЬНЫЕ АЛГОРИФМЫ МАРКОВА

Лекционные занятия № 13–14 «Нормальные алгорифмы Маркова»

План занятия:

1. ***Марковские подстановки.***
2. ***Нормальные алгорифмы и их применение к словам.***
3. ***Нормально вычислимые функции.***
4. ***Принцип нормализации Маркова.***
5. ***Основные способы композиции нормальных алгоритмов.***

1. Марковские подстановки

Теория нормальных алгорифмов (в авторской транскрипции) была разработана советским математиком А. А. Марковым (1903–1979 гг.) в конце 40-х годов XX века. Его подход к вопросу алгоритмизации связывает неформальное понятие алгоритмической вычислимости с переработкой слов в некотором конечном алфавите в соответствии с определенными правилами. В качестве элементарного преобразования используется **подстановка** одного слова вместо другого. Множество таких подстановок определяет схему алгоритма. Правила, определяющие порядок применения подстановок, а также правила остановки являются общими для всех нормальных алгоритмов.

Определение 1.1. Алфавитом назовем всякое непустое конечное множество символов, не содержащее знаки $\longrightarrow$ и $\cdot$. Его элементы – **буквы**, а любые последовательности букв – **слова** в алфавите A. Пустую последовательность букв будем называть **пустым словом** и обозначать λ. Если A и B два алфавита, причем $A \subseteq B$, то алфавит B будем называть **расширением** алфавита A.

Слова будем обозначать латинскими буквами P, Q, R. Одно слово может быть составной частью другого слова. Тогда первое слово называется **подсловом** второго или **вхождением** во второе.

Определение 1.2. Будем говорить, что имеется **вхождение** слова P в слово R, если существуют слова V_1 и V_2 (возможно, пустые), такие, что $R = V_1 P V_2$ (1). Если слово V_1 имеет наименьшую дину из всех слов вида (1), то говорят о первом вхождении P в слово R.

Пример 1: в алфавите A = {a, b} рассмотрим следующие слова:

1. P_1 = aa входит в слово R = abaabab, так как $R = V_1P_1V_2$, где V_1 = ab, V_2 = bab. В этом случае V_1 и V_2 определены однозначно.
2. P_2 = aba входит в слово R = abaabab, так как $R = V_1 P V_2$, где $V_1 = \lambda$, V_2 = abab; $R = W_1 P_2 W_2$, где W_1 = aba, W_2 = b. В этом случае V_1 и V_2 определены неоднозначно.

Замечание: в дальнейшем будем разыскивать первые вхождения данных слов в некоторые слова и заменять их на другие слова, возможно, пустые.

Пример 2: в результате замены первого вхождения слова P = ba в слове R = aababab на слово: 1) Q = a; 2) Q = b; 3) Q = baa; 4) на пустое слово, получим следующие слова (рис. 87):

	P = ba	
Q = a	aa**ba**bab	aa**a**bab
Q = b	aa**ba**bab	aa**b**bab
Q = baa	aa**ba**bab	aa**baa**bab
λ	aa**ba**bab	aabab

Рис. 87.

Если же вхождение пустого слова в любое слово R заменить на некоторое слово Q, получится слово QR.

Определение 1.3. Пусть A – алфавит, не содержащий в качестве букв знаки ⟶ и ·. **Обычной формулой подстановки** в алфавите A называется выражение P⟶Q, где P и Q – слова в алфавите A. **Заключительной формулой подстановки** в алфавите A называется выражение P⟶ Q, где P и Q – слова в алфавите A. P и Q называются левой и правой частями формулы подстановки.

Определение 1.4. Марковской подстановкой (P⟶Q или P⟶ · Q) называют операцию над словами в данном алфавите A. Эта операция задается с помощью упорядоченной пары слов (P, Q) и состоит в том, что в заданном слове R находят **первое (!)** вхождение слова P (если такое имеется) и, не изменяя остальных частей слова R, заменяют в нем это вхождение словом Q. Полученное слово называется результатом применения марковской подстановки к слову R. Если же первого вхождения слова P в слово R нет (и, следовательно, нет вообще ни одного вхождения P в R), то считается, что марковская подстановка не применима к слову R.

Пример 3: пусть для слов в алфавите A = {a, b, c} заданы следующие марковские подстановки:

1) ca ⟶ ab;
2) bca ⟶ λ;
3) abca ⟶ a;
4) b⟶a.

Примените каждую из них к слову abcabcabcabcab.

Пример 4 (рис. 88):

Преобразуемое слово	**Марковская подстановка**	**Результат**
13**8578**926	85789, 00	13**00**26
т**ара**рам	ара, λ	трам
ш**ра**м	ра, ар	ш**ар**м
функция	λ, μ-	μ-функция
лог**ика**	ика, λ	лог
книга	λ, λ	книга
поляна	пор, т	(неприменима)

Рис. 88.

2. Нормальные алгорифмы и их применение к словам

Определение 2.1. Произвольная конечная непустая упорядоченная последовательность марковских подстановок называется **нормальной схемой в алфавите A** и обозначается S_a. Таким образом, схема нормального алгоритма имеет вид:

$S_a: \begin{cases} P_1 \to Q_1 \\ P_2 \to Q_2 \\ \quad \dots \\ P_m \to (\cdot)Q_m \end{cases}$, здесь $P_i, Q_i \in A$ или $P_i, Q_i \in B \supset A$.

Если схема нормального алгоритма задана в некотором расширении В алфавита А ($A \subseteq B$), то говорят, что задана **нормальная схема над А,** или **нормальная схема в алфавите В.**

Нормальный алгоритм над алфавитом А задается конечным алфавитом В, не содержащим $\longrightarrow$ и $\cdot$ и включающим в себя А или совпадающим с А (то есть $B \supseteq A$) и нормальной схемой в алфавите В.

Определение 2.2. Нормальным алгорифмом Маркова (НАМ) называется непустой упорядоченный набор формул подстановки:

$S_a: \begin{cases} P_1 \to Q_1 \\ P_2 \to Q_2 \\ \quad \dots \\ P_m \to (\cdot)Q_m \end{cases}$, здесь $P_i, Q_i \in A$ или $P_i, Q_i \in B \supset A$.

Правила выполнения НАМ:

1. Задается некоторое входное слово R. Где именно оно записано – не важно, в НАМ этот вопрос не оговаривается.

2. Работа НАМ сводится к последовательности шагов. На каждом шаге входящие в НАМ формулы подстановки просматриваются сверху вниз и выбирается первая из формул, применимых к водному слову R, то есть самая верхняя из тех, левая часть которых входит в R. Далее выполняется подстановка согласно найденной формуле. Получается новое слово R_1. На следующем шаге это слово R_1 берется за исходное и к нему применяется та же самая процедура. И так далее $R \longrightarrow R_1 \longrightarrow \dots \longrightarrow R_i \longrightarrow R_{i+1} \longrightarrow \dots$.

3. Если на очередном шаге была применена обычная формула подстановки, то работа НАМ продолжается.

4. Если же на очередном шаге была применена заключительная формула подстановки, то после нее работа НАМ прекращается. То слово, которое получилось в этот момент, и есть выходное слово.

5. Если на очередном шаге к текущему слову не применима ни одна формула, то и в этом случае работа НАМ прекращается, а выходным словом считается текущее слово.

Таким образом, НАМ останавливается по двум причинам: либо применена заключительная формула, либо ни одна из формул не подошла. То и другое считается «хорошим» окончанием работы НАМ. В обоих случаях говорят, что НАМ применим к входному слову.

6. Может случиться и так, что НАМ никогда не остановится, это происходит, когда на каждом шаге есть применимая формула и эта формула обычная. Тогда говорят, что НАМ **неприменим** к входному слову R. В этом случае ни о каком результате не может быть и речи.

Необходимые уточнения:

1. Если левая часть P входит в R, то говорят, что эта формула **применима к R**. Если не входит, то формула считается **неприменимой к R** и подстановка не выполняется.

2. Если левая часть P входит в R несколько раз, то на правую часть Q, по определению, заменяется только первое вхождение P в R.

3. Если правая часть формулы подстановки – пустое слово, то подстановка сводится к вычеркиванию части P из R (отметим попутно, что в формулах подстановки не принято как-либо обозначать пустое слово, $P \longrightarrow$).

4. Если в левой части формулы подстановки указано пустое слово, то подстановка $\longrightarrow$ Q сводится, по определению, к приписыванию Q слева к слову R.

Замечание: формула с пустой левой частью применима к любому слову, формула с пустыми левой и правой частями не меняет слово.

Определение 2.3. НАМ над алфавитом A называется следующее правило построения последовательности $R \longrightarrow R_1 \longrightarrow \ldots \longrightarrow R_i \longrightarrow R_{i+1} \longrightarrow \ldots$ слов в алфавите $B \supseteq A$, исходя из данного слова R в алфавите A. Если процесс построения последовательности слов $R \longrightarrow R_1 \longrightarrow \ldots \longrightarrow R_{m-1} \longrightarrow R_m$ обрывается, то говорят, что рассматриваемый нормальный алгоритм **применим к слову R.** Последний член последовательности R_m называется результатом применения нормального алгоритма к слову R. Говорят, что нормальный алгоритм **перерабатывает R в R_m.** Если последовательность слов бесконечна, то говорят, что нормальный алгоритм **не применим к слову R.**

Пример 5: тождественный НАМ над алфавитом A применим к каждому слову в алфавите A, и результатом его работы является это же слово.

Такой НАМ может быть задан алфавитом B = A (не содержащим $\longrightarrow$ и .) и нормальной схемой $\{ \longrightarrow$, то есть в этой подстановке имеется только одна формула подстановки, являющаяся заключительной, с пустыми левыми и правыми частями.

Замечание: рассмотренный НАМ вычисляет тождественную всюду определенную словарную функцию F(R) = R или числовую функцию f(x) = x.

Пример 6: НАМ над алфавитом A = {a, b}, заданный алфавитом B = A и нормальной схемой S_a: $\begin{cases} a \longrightarrow b \\ b \longrightarrow a \end{cases}$, не применим ни к одному слову в алфавите A.

3. Нормально вычислимые функции

Как и МТ, НАМ не производят собственно вычислений: они лишь производят преобразования слов, заменяя в них одни буквы другими по предписанным им правилам. В свою очередь, мы предписываем им такие правила, результаты применения которых мы можем интерпретировать как вычисления.

Пример 7: в алфавите A = {1} схема $\lambda \longrightarrow \cdot\, 1$ определяет НАМ, который к каждому слову в алфавите A = {1} (все такие слова следующие: λ, 1, 11, 111,….) приписывает слева единицу. Следовательно, алгоритм реализует функцию *f(x) = x+1.*

Определение 3.1. Функция f, заданная на некотором множестве слов алфавита A, называется **нормально вычислимой** (или **вычислимой по Маркову**), если найдется такое расширение B данного алфавита A ($B \supseteq A$) и такая схема н.а. S_a в алфавите B, что каждое слово R (в алфавите A) из области определения функции f этот алгоритм перерабатывает в слово f(R).

В качестве функции f можно брать как словарные, так и числовые функции. Будем рассматривать числовые функции *f,* определенные на множестве N_0, принимающие значения из N_0.

Как и раньше, целое неотрицательное число x будем изображать словом из x+1 единиц. Набор чисел $x_1, x_2, \ldots, x_n$ условимся изображать словом $1^{x_1+1} * 1^{x_2+1} * \ldots * 1^{x_n+1}$.

Пример 8: НАМ, вычисляющий функцию s(x) = x+1, задается алфавитом B = {1} и нормальной схемой $\{1 \longrightarrow \cdot\, 11$.

Словесное предписание: «Припиши к данному слову еще одну единицу и остановись. Полученное слово и есть результат».

Действие приписывания единицы можно задать с помощью подстановки $\{1 \longrightarrow 11$. Но алгоритм, заданный этой подстановкой, будет бесконечно прибавлять по одной единице к исходному слову, никогда не останавливаясь, то есть процесс никогда не закончится. Для указания остановки пользуются заключительной подстановкой $\{1 \longrightarrow \cdot\, 11$.

Рассматриваемый нормальный алгоритм применим к каждому слову в алфавите {1}, и его работа состоит из двух слов $1^{x+1} \longrightarrow 1^{x+2}$.

4. Принцип нормализации Маркова

Принцип Маркова: всякая эффективно вычислимая функция является нормально вычислимой или всякий алгоритм представим в виде нормального алгоритма.

Математически доказать принцип нормализации невозможно, поскольку понятие произвольного алгоритма или эффективно вычислимой функции не являются строго определенными математическими понятиями. Справедливость этого принципа основана на том, что все известные в настоящее время алгоритмы являются нормализуемыми, а способы композиции алгоритмов, позволяющие строить новые алгоритмы из уже известных, не выводят за пределы класса нормализуемых алгоритмов.

Теорема 4.1. Всякая функция, вычислимая по Тьюрингу, будет так же и нормально вычислимой.

Верно и обратное утверждение: всякая нормально вычислимая функция вычислима по Тьюрингу.

5. Основные способы композиции нормальных алгоритмов

5.1. Суперпозиция. При суперпозиции двух алгоритмов C и D выходное слово первого алгоритма C рассматривается как входное слово второго алгоритма D. Результат суперпозиции алгоритмов C и D можно представить в виде H(R) = D · C = D(C(R)). Суперпозиция может выполняться для любого конечного числа алгоритмов.

Пример 9: рассмотрим алгоритм, позволяющий по любому слову в алфавите A = {a, b} определить, имеет ли оно четную длину.

Условимся считать, что этот алгоритм должен перерабатывать слова в алфавите A = {a, b}, имеющие четную длину (и только их), в пустое слово.

НАМ задается алфавитом B = A и нормальной схемой:

S_a: $\begin{cases} a \to | \\ b \to | . \\ \| \to \lambda \end{cases}$

Этот алгоритм может быть задан в виде следующего предписания:

1. Вместо каждой буквы из алфавита A = {a, b} запиши |, переходи к п. 2.
2. Выясни, содержит ли полученное слово вхождение ‖. Если да, переходи к п. 3, если нет, – к п. 4.
3. Выброси каждое вхождение слова ‖, переходи к п. 4.
4. Сравни полученное слово с λ. Если оно совпадает, то переходи к п. 5, иначе – к п. 6.
5. Слово имеет четную длину, переходи к п.7.
6. Слово имеет нечетную длину, переходи к п.7.
7. Процесс вычисления остановить.

Можно выделить два этапа: нахождение длины исходного слова и распознавание четности длины.

C: S_c: $\begin{cases} a \to | \\ b \to | \end{cases}$

D: S_d: $\{|| \to \lambda$

D · C

abbbb → |bbbb → ||bbb → |||bb →... → |||| →λ ||| → λ|| →|.

5.2. Разветвление. Разветвление алгоритмов представляет собой композицию трех алгоритмов C, D, K. Обозначая результат этой композиции буквой H, будем считать, что область определения алгоритма H совпадает с пересечением областей определения всех трех алгоритмов C, D, K, а для любого слова R из этого пересечения:

H(R) = C (R), если K(R) = λ,

H(R) = D(R), если K(R) ≠ λ, где λ – пустая строка.

Пример 10: пусть даны алфавит A = {a, b}, нормальные алгоритмы в том же алфавите, имеющие следующие нормальные схемы:

C = {ab → ba

D = {ba →ab

K = $\begin{cases} ab \to ba \\ ba \to \lambda \end{cases}$

Рассмотрим работу над словами R = aba и Q = bab

1. C(aba) = baa
   D(aba) = aab
   K (aba) = aa
   H(aba) = aab, так как K(R) ≠ λ
2. C(bab) = bba
   D(bab) = abb
   K (bab) = λ
   H(bab) = bba, так как K(R) = λ

5.3. Повторение (итерация). Итерация представляет собой композицию двух алгоритмов C и D. Обозначая результат этой композиции через H, определим, что для любого входного слова R **соответствующее ему выходное слово** H(R) получается в результате последовательного многократного алгоритма C до тех пор, пока не получится слово, перерабатываемое алгоритмом D в некоторое фиксированное слово.

Замечание: все композиции нормальных алгоритмов приводят к нормальным алгоритмам.

Пример 11: A = {a, b}, C = {ab → ba, D = {bbbaa → ab, R = ababb

ababb →ba**ab**b →b**ab**ab →bb**ab**a →**bbbaa** → ab.

Практические занятия № 15–16 «Нормальные алгорифмы Маркова»

1. Нормальный алгоритм задан алфавитом A = {a, b} и нормальной схемой:

$$S_a = \begin{cases} \alpha a a \to \alpha \\ \alpha a \to \alpha \\ a \to \cdot \\ ab \to b \alpha a \\ \to \alpha \end{cases}$$ примените этот алгоритм к словам :

(1) $R_1 = aaaa$

(2) $R_2 = bbb$

(3) $R_3 = bab$

(4) $R_4 = aba$

2. Нормальный алгоритм задан алфавитом A = {a, b, 1} и нормальной схемой:

$S_a = \begin{cases} a \longrightarrow 1 \\ b \longrightarrow 1 \\ 11 \longrightarrow \lambda \end{cases}$ примените этот алгоритм к слову abaabbb.

3. Нормальный алгоритм в алфавите A = {a, b} задан нормальной схемой:

$S_a = \begin{cases} ba \longrightarrow ab \\ a \longrightarrow \lambda \\ b \longrightarrow \cdot\, b \end{cases}$, примените его к следующим словам и проанализируйте его работу:

(1) $R_1 = bbaab$

(2) $R_2 = aaa$

(3) $R_3 = bbbbb$

(4) $R_4 = aabaabb$

4. Нормальный алгоритм задан в алфавите A = {a, b, c, d} и нормальной схемой:

$S_a = \begin{cases} ad \longrightarrow \cdot\, dc \\ ba \longrightarrow \lambda \\ a \longrightarrow bc \\ bc \longrightarrow bba \\ \lambda \longrightarrow a \end{cases}$ примените этот алгоритм к слову bdc и проанализируйте его работу.

5. Примеры на составления НАМ:

(1) **Вставка и удаление символов**. $A = \{a, b, c, d\}$. В слове R требуется заменить первое вхождение подслова bb на ddd и удалить все вхождения символа c. Например, abbcabbca⟶adddabba.

(2) **Перестановка символов.** $A = \{a, b\}$. Преобразовать слово R так, чтобы в его начале оказались все символы a, а в конце b. Например, babba⟶aabbb.

(3) **Использование спецзнака.** $A = \{a, b\}$. Удалить из непустого слова R его первый символ, пустое слово не менять.

(4) **Перемещение спецзнака.** $A = \{a, b\}$. Требуется приписать символ a к концу слова R. Например, bbab⟶bbaba.

(5) **Смена спецзнака.** $A = \{a, b\}$. В слове R заменить на aa последнее вхождение символа a, если такое есть. Например, bababb⟶babaabb.

(6) **Перенос символа через слово.** $A = \{a, b\}$. Перенести в конец непустого слова R его первый символ. Пустое слово не менять. Например, bbaba⟶babab.

(7) **Использование нескольких спецзнаков.** $A = \{a, b\}$. Удвоить слово R, то есть приписать слева или справа его копию. Например, abb⟶abbabb.

Задачи для самостоятельной работы

1. Написать нормальную схему алгоритма для функций:

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

(2) $f(x) = 2x$

(3) $f(x, y) = x + y$

(4) $f(x, y, z) = x + y + z$

(5) $f(x, y, z) = x$

(6) $f(x, y, z) = y$

(7) $f(x, y, z) = z$

(8) $f(x, y, z) = x + 2y + 3z$

(9) $f(x, y) = |x - y|$

(10) $f(x) = sg(x) = \begin{cases} 0, x = 0 \\ 1, x > 0 \end{cases}$

(11) $f(x) = \overline{sg}(x) = \begin{cases} 1, x = 0 \\ 0, x > 0 \end{cases}$

(12) $f(x) = x \dot{-} 1 = \begin{cases} 0, x \leq 1 \\ x - 1, x > 1 \end{cases}$

(13) $f(x) = x - 1$

(14) $f(x) = x - 5$

(15) $f(x) = \begin{cases} 1, \text{ если x делится на 2} \\ 0, \text{ если x не делится на 2} \end{cases}$

(16) $f(x, y) = \frac{x}{2} + y$

(17) $f(x) = 2^{1-x}$

(18) $f(x) = \frac{2}{4-x}$

(19) $f(x) = \left[\frac{1}{x}\right]$

(20) $f(x) = \left[\frac{x}{3}\right]$

(21) $f(x) = \begin{cases} 1, \text{ если x делится на 3} \\ 0, \text{ если x не делится на 3} \end{cases}$

(22) $f(x) = x + 1$ в десятичной системе счисления

(23) $f(x) = 2x$ в десятичной системе счисления

(24) $f(x) = x - 1$

2. Построить НАМ, подсчитывающий количество букв «b» в слове, записанном в алфавите A = {a,b} и содержащем более одной буквы «b», иначе заменить исходное слово на «bbb».

3. Построить НАМ, применимый ко всем словам $x_1 x_2 \ldots x_n$ в алфавите $\{a, b\}$ и переводящим их в слово $\propto$.

$$\propto = \begin{cases} x_n, \text{ если } x_{n-1} = a \\ b^{n-1} x_n, \text{ если } x_{n-1} = b,\ n > 1. \end{cases}$$

Проверить работу построенного алфавита над некоторыми словами: R_1 = abba, R_2 = bbaaa.

4. Построить НАМ, вычисляющий данную числовую функцию $f(x, y) = x + 2y$. Проверить построенный НАМ над некоторым набором значений переменных: f(0, 1) = 2.

5. Написать формулу для функции $y = f(x, y, z)$, вычисляемой НАМ. Проверить работу алгоритма над некоторым набором значений аргумента:

$$\begin{cases} *1 \rightarrow \alpha \\ \alpha\alpha 1 \rightarrow 11\alpha\alpha \\ \alpha\alpha \rightarrow \cdot \\ \alpha 1 \rightarrow \alpha \end{cases}$$

ЧАСТЬ IV. ОСНОВНЫЕ ПОНЯТИЯ: РЕКУРСИВНЫЕ ФУНКЦИИ

Лекционные занятия № 17–18 «Рекурсивные функции»

План занятия:

1. Происхождение рекурсивных функций.
2. Основные вычислимые операторы.
3. Тезис А. Чёрча.

1. Происхождение рекурсивных функций

Термин «рекурсивная функция» в теории алгоритмов используется для обозначения трех классов функций:

- примитивно рекурсивные функции;
- общерекурсивные функции;
- частично-рекурсивные функции (совпадают с классом вычислимых по Тьюрингу функций).

Термины «частично рекурсивная функция» и «общерекурсивная функция» образовались в силу исторических причин и, по сути, являются результатом неточного перевода английских терминов: «*partial recursive function*» и «*total recursive function*», которые по смыслу более правильно переводить как «рекурсивные функции, определенные на части множества возможных аргументов» и «рекурсивные функции, определенные на всем множестве возможных аргументов». Наречие «частично» относится не к прилагательному «рекурсивные», а к области определения функции. Возможно, более правильным названием было бы «частично определенные рекурсивные функции» и просто «везде определенные рекурсивные функции».

Определение 1.1. Функция называется **эффективно вычислимой**, если существует алгоритм, позволяющий вычислить ее значения.

Переход от алгоритма к эффективно вычислимой функции дает определенные преимущества. Дело в том, что все требования, которые предъявляются к алгоритмам, выполняются и для совокупности всех вычислимых функций, которая получила название совокупности рекурсивных функций.

На первый взгляд может показаться, что все числовые функции эффективно вычислимы, а если вычислимость какой-либо функции и вызывает сомнение, то лишь потому, что для ее вычисления пока еще не придумали подходящий алгоритм, однако математическими методами удалось доказать, что некоторые функции не являются вычислимыми. Это доказательство стало возможным только после того, как были получены математические модели алгоритма. Рекурсивная функция – одна из таких моделей. Она была разработана в 1930-х годах благодаря трудам А. Чёрча, К. Гёделя, Дж. Эрбрана, С. Клини.

Курт Гёдель впервые описал класс всех рекурсивных функций как класс всех числовых функций, определяемых в некоторой нормальной системе. Алонзо Чёрч пришел к тому же классу функций в 1936 году.

Сначала были выбраны простейшие функции, эффективная вычислимость которых была очевидна (своего рода «аксиомы»). Затем сформулированы некоторые правила (названные операторами), на основе которых можно строить новые функции из уже имеющихся (своего рода «правила вывода»). Требуемым классом функций стала совокупность всех функций, получающихся из простейших с помощью выбранных операторов.

Напомним, что рассматриваются числовые функции, заданные на множестве целых неотрицательных чисел $N_0 = \{0,1,2, \dots\}$ и принимающие значения из этого же множества.

Выбираются **простейшие функции:**

1) $O(x) = 0$ (оператор аннулирования);
2) $s(x) = x + 1$ (оператор сдвига, функция следования);
3) $I_n^m(x_1, x_2,\dots, x_n) = x_m$, $1 \le m \le n$ (оператор проектирования).

Ясно, что все три простейшие функции всюду определены и интуитивно вычислимы (их правильную вычислимость можно установить с помощью МТ).

Из исходных базисных функций будем образовывать другие функции с помощью трех операций: суперпозиции, примитивной рекурсии и минимизации.

2. Основные вычислимые операторы

2.1. Суперпозиция функций. Рассмотрим функции $f_1(x_1, x_2,\dots, x_n)$, $f_2(x_1, x_2,\dots, x_n)$, …, $f_m(x_1, x_2,\dots, x_n)$ и функцию $\varphi(x_1, x_2,\dots, x_m)$. Говорят, что функция $\psi(x_1, x_2,\dots, x_m)$ получена суперпозицией функций $f_i(x_1, x_2,\dots, x_n)$, если справедливо равенство:

$$\psi(x_1, x_2,\dots, x_m) = \varphi\,(\, f_1(x_1, x_2,\dots, x_n), f_2(x_1, x_2,\dots, x_n), \dots, f_m(x_1, x_2,\dots, x_n)).$$

Операция суперпозиции обозначается через $\psi = S(\varphi, f_1, .., f_m)$.

Пример 1: постоянные функции $f(x) = 1$ и $f(x) = 3$ получаются суперпозицией из базисных функций $O(x)$ и $s(x)$ следующим образом:

- $f(x) = 1 = s(o(x))$;
- $f(x) = 3 = s(s\left(s\left(o(x)\right)\right))$.

Чтобы из любого числа x получить фиксированное число, нужно сначала числу x сопоставить число 0: $O(x) = 0$, а затем к полученному результату, то есть к нулю, с помощью функции следования $s(x)$ прибавлять по единице до тех пор, пока не получится нужное число.

Пример 2: функция $o\,(x_1, x_2, \dots, x_n) = 0$ получается с помощью операции суперпозиции из функций $O(x)$ и $I_n^1(x_1, x_2,\dots, x_n)$ следующим образом: $o\,(x_1, x_2, \dots, x_n) = o\,(I_n^1(x_1, x_2, \dots, x_n)) = 0$ (функция обнуления n переменных не является базисной).

Пример 3: функция $g(x) = x + 3$ получается с помощью операции суперпозиции из функций $s(x)$ и $I_n^m(x_1, x_2,\dots, x_n)$ следующим образом: $g(x) = x + 3 = s\,(s(s(x)))$.

1.1. Схема примитивной рекурсии. Пусть заданы какие-нибудь числовые частичные функции: n-местная g и (n+2)-местная h. Говорят, что (n+1)-местная частичная функция f возникает из функций g и h примитивной рекурсией, если для всех натуральных значений x_1, $x_2,\dots,x_n$, y имеем:

$$\begin{cases} f(x_1,x_2, \dots, x_n, 0) = g(x_1, \dots, x_n) & (1) \\ f(x_1,x_2, \dots, x_n, y+1) = h\big(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)\big) & (2) \end{cases}$$

Это определение мы будем применять и для n = 0, говоря, что одноместная частичная функция f возникает примитивной рекурсией из постоянной одноместной функции, равной числу a, и двуместной частичной функции h, если:

$$f(0) = a; \tag{3}$$
$$f(x+1) = h(x, f(x)) . \tag{4}$$

Такая система напоминает способ задания функций с помощью известного метода математической индукции, когда указывают значение функции в некоторой начальной точке и закон, согласно которому значение функции в каждой последующей точке вычисляют через ее значение в предыдущей точке. Например, значение f(3) вычисляют:

$$f(0) = a,$$
$$f(1) = h(0, f(0)),$$
$$f(2) = h(1, f(1)),$$
$$f(3) = h(2, f(2)).$$

Встает вопрос, для каждых ли частичных функций g, h от n и n+2 переменных существует частичная функция f от n+1 переменной, удовлетворяющая условиям (1, 2) или соответственно (3, 4)? И будет ли такая частичная функция единственной?

Так как область определения функций есть множество всех натуральных чисел, то ответ на оба вопроса, очевидно, положительный. Если функция f существует, то из (1) последовательно находим:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, 1) = h(x_1, \ldots, x_n, 0, g(x_1, \ldots, x_n))$$
$$\ldots$$
$$f(x_1, \ldots, x_n, m+1) = h(x_1, \ldots, x_n, m, f(x_1, \ldots, x_n, m)) \tag{5}$$

и поэтому f определена однозначно.

Чтобы при заданных частичных функциях g, h найти функцию f, возникающую из них примитивной рекурсией, достаточно равенства (5) принять в качестве равенств, определяющих значения функции f.

Таким образом, для любых n-местной функции g и (n+2)-местной функции h существует одна и только одна частичная (n+1)-местная функция f, возникающая из g и h примитивной рекурсией. Символически пишут: f = R(g, h).

Из соотношения (5) вытекает, что если функции g и h всюду определены, то и функция f всюду определенная.

Замечание: схему примитивной рекурсии можно записывать по любой переменной, указывая отдельно, по какой переменной ведется рекурсия.

Пример 4: покажем, что следующие функции могут быть получены из простейших с помощью оператора примитивной рекурсии:

$$sum(x, y) = x + y$$
$$p(x, y) = x \cdot y \quad \cdot$$

Для функции $sum\,(x,y) = x + y$ верны следующие равенства:

$$\begin{cases} sum\,(x,0) = x+0 = x = I_1^1(x) \\ sum\,(x,y+1) = x+(y+1) = (x+y)+1 = sum\,(x+y)+1 = s\left(sum(x,y,)\right) = \left(I_3^3(x,y,z)\right). \end{cases}$$

Это и есть схема примитивной рекурсии для функции $\mathrm{sum}\,(x,y) = x + y$, основывающаяся на базисных функциях $I_1^1(x)$ и $s(x)$.

Начальные примитивно рекурсивные функции $g(x) = I_1^1(x)$, $h(x, y, z) = z + 1 = s(I_3^3(x, y, z))$.

Отождествляя аргументы x и y, получим новую функцию f(x) = 2x. Она тоже примитивно рекурсивна, поскольку отождествление аргументов – это вариант оператора суперпозиции. Далее из функций f(x,y) = x+y, g(x) = 2x и тождественных констант, используя суперпозицию, можно получить 3x, 4x, …, а также все линейные функции вида ax + b, a, b ϵ N.

Аналогично для операции умножения:

$$\begin{cases} p\,(x,0) = x \cdot 0 = 0 = o(x) \\ p\,(x,y+1) = x \cdot (y+1) = (x \cdot y)+x = p\,(x,y)+ x = sum(x, p(x, y)). \end{cases}$$

Записанные выражения показывают, что функция $p(x,y) = x \cdot y$ получается из базисной функции $o(x)$ и функции $sum\,(x,y) = x + y$ с помощью схемы примитивной рекурсии и суперпозиции. Начальные примитивно рекурсивные функции $g(x) = 0(x)$, h(x,y,z) = z + x.

Отождествляя аргументы x и y, получим новую функцию $f(x) = x^2$. Она тоже примитивно рекурсивна, а далее с помощью подстановки получаем степенные функции x^3, x^4,…

Определение 2.2: функция называется **примитивно рекурсивной**, если она может быть получена из базисных функций с помощью конечного числа применений операций суперпозиции и примитивной рекурсии.

Замечание: все основные функции арифметики, алгебры и анализа (с поправкой на целочисленность) являются примитивно рекурсивными. И так как исходные базисные функции являются всюду определенными и операции подстановки и примитивной рекурсии сохраняют это свойство, то из определения примитивно рекурсивной функции следует, что каждая примитивно рекурсивная функция является всюду определенной.

Существуют функции всюду определенные, но не являющиеся примитивно рекурсивными (об этом далее). Примеры таких функций показывают, что существует более широкий класс функций, чем примитивно рекурсивные функции. Его можно построить, если к примитивно рекурсивным функциям применить новый оператор, действие которого не сводится к многократному использованию суперпозиции и примитивной рекурсии. Это так называемый оператор минимизации.

2.3. Операция минимизации (μ-оператор). Говорят, что n-местная частичная функция g получается с помощью операции минимизации из (n+1)-местной частичной функции f, если для любых $x_1, x_2,\ldots, x_n, y \in N$ значение $g(x_1, x_2,\ldots, x_n)$ определено и равно y тогда и только тогда, когда для любого z < y значение $f(x_1, x_2,\ldots, x_n, z)$ ($f(x_1, x_2,\ldots, x_n, 0)$, $f(x_1, x_2,\ldots, x_n, 1)$, …, $f(x_1, x_2,\ldots, x_n, y-1)$) определено и не равно 0, а $f(x_1, x_2,\ldots, x_n, y) = 0$. В этом случае пишут: $g(x_1, x_2,\ldots, x_n) = \mu y[f(x_1, x_2,\ldots, x_n, y) = 0]$.

Замечание: операцию минимизации можно применять по любой переменной, входящей в функцию, указывая отдельно, по какой переменной эта операция проводится.

Оператор минимизации является удобным средством для построения обратных функций. Так, функция $f^{-1}(x) = \mu_t(f(t) = x)$ (наименьший t такой, что $f(t) = x$) является обратной для функции $f(x)$. В применении к одноместным функциям оператор минимизации иногда называют оператором обращения.

Рассмотрим функцию одного аргумента f(x) = x+1. Сделаем замену x = t: f(t) = t+1. Рассмотрим уравнение f(t) = x: t+1 = x. Фиксируем конкретное значение x = a – const, a ϵ N_0 и решаем уравнение относительно t. Решив уравнение, ищем наименьшее из всех возможных t, выбираем t_0, если оно существует.

Затем выполняем проверку: для всех чисел меньше найденного (t'< t_0) левая часть нашего уравнения должна быть определена.

Меняя значения x, получаем другие уравнения и, следовательно, другие наименьшие решения. Полученные соотношения между значениями x и t_0 обозначаем через функцию g(x) = $\mu_t(f(t) = x)$. В нашем случае f(x) = x+1, g(x) = $\mu_t(t + 1 = x)$(рис. 89)

x	0	1	2	3	4	…
g(x)	–	0	1	2	3	…

Рис. 89.

1. t+1 = 0, не существует решений t_0 ϵ N_0
2. t+1 = 1, t_0 = 0
3. t+1 = 2, t_0 = 1, t' < 1, t'= 0
4. t+1 = 3, t_0 = 2, t' < 2, t'= 0

Ответ: g(x) = x-1

Определение 2.3. Функция $f(x_1, x_2,\ldots, x_n)$ называется **частично рекурсивной**, если она может быть получена за конечное число шагов из базисных функций при помощи операций суперпозиции, схем примитивной рекурсии и μ-оператора.

Определение 2.4. Функция $f(x_1, x_2,\ldots, x_n)$ называется **общерекурсивной**, если она частично рекурсивна и всюду определена.

Примерами общерекурсивных функций являются: $f(x, y) = x + y$, $f(x, y) = x \cdot y$, $f(x, y) = x + n$.

Множество частично рекурсивных функций включает в себя множество общерекурсивных функций, а общерекурсивные функции включают в себя примитивно рекурсивные функции. Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Рекурсивными функциями мы стремимся исчерпать все мыслимые функции, поддающиеся вычислению с помощью какой-нибудь определенной процедуры механического характера. Подобно тезису Тьюринга в теории рекурсивных функций выдвигается соответствующая естественнонаучная гипотеза.

3. Тезис А. Чёрча

Тезис А. Чёрча: всякая эффективно вычислимая функция является частично рекурсивной.

Или: числовая функция тогда и только тогда алгоритмически вычислима, когда она частично рекурсивна.

Этот тезис нельзя доказать, так как он связывает нестрогое понятие интуитивно вычислимой функции со строгим математическим понятием частично рекурсивной функции. Но

этот тезис можно опровергнуть, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.

Легко понять, что любая примитивно рекурсивная функция является частично рекурсивной, так как по определению операторы для построения частично рекурсивных функций включают в себя операторы для построения примитивно рекурсивных функций.

Также понятно, что примитивно рекурсивная функция определена везде и поэтому является общерекурсивной функцией (у примитивно рекурсивной функции нет повода «зависать», так как при ее построении используются операторы, определяющие везде определенные функции).

Теорема 2.1: функция, возникающая примитивной рекурсией из правильно вычислимых на машине Тьюринга функций, сама правильно вычислима на машине Тьюринга.

Следствие: всякая примитивно рекурсивная функция вычислима по Тьюрингу.

Напомним, что в начале изучения теории алгоритмов мы поставили задачу охарактеризовать вычислимые (с помощью какого-либо алгоритма) функции. Учитывая тезис Тьюринга, под алгоритмом достаточно понимать машину Тьюринга. Учитывая предыдущее следствие, возникает вопрос: исчерпывается ли класс вычислимых функций примитивно рекурсивными, то есть всякая ли вычислимая (по Тьюрингу) функция будет непременно примитивно рекурсивной?

Теорема 2.2: существуют функции, не являющиеся примитивно рекурсивными.

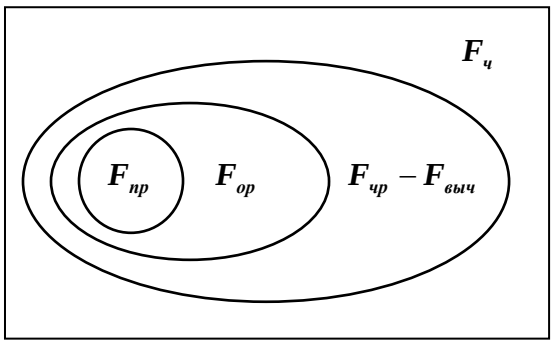
Не все вычислимые функции можно описать как примитивно рекурсивные. Это устанавливается, если привести пример вычислимой функции, не являющейся примитивно рекурсивной. Идея примера состоит в том, чтобы построить такую вычислимую функцию, которая обладала бы свойством, каким не обладает ни одна примитивно рекурсивная функция. Таким свойством может служить скорость роста функций. Примером такой функции является функция Аккермана, она растет быстрее любой примитивно рекурсивной функции. Данная функция вычислима, но ее нельзя получить из простейших функций с помощью оператора примитивной рекурсии.

Функция Аккермана принимает два неотрицательных целых числа в качестве параметров и возвращает натуральное число, обозначаемое $A = (m, n)$. Данная функция растет очень быстро, например, число $A = (4,4)$ настолько велико, что количество цифр в записи количества цифр этого числа многократно превосходит количество атомов в наблюдаемой части Вселенной.

Теорема 2.3: всякая частично рекурсивная функция вычислима по Тьюрингу.

Теорема 2.4: если функция вычислима по Тьюрингу, то она частично рекурсивна.

Класс функций, вычислимых по Тьюрингу, совпадает с классом частично рекурсивных функций. Совпадение этих двух классов вычислимых функций, в основе построения которых лежали совершенно разные подходы к формализации понятия вычислимости функций, говорит о том, что эти походы оказались весьма состоятельными, и служит косвенным подтверждением того, что как тезис Тьюринга, так и тезис Чёрча не только не безосновательны, но и имеют все права на признание (рис. 90).



$F_ч$ – класс числовых функций;
$F_{выч}$ – класс вычислимых функций;
$F_{чр}$ – класс частично-рекурсивных функций;
$F_{ор}$ – класс общерекурсивных функций;
$F_{пр}$ – класс примитивно рекурсивных функций.

Рис. 90.

Практические занятия № 19–20 «Рекурсивные функции»

1. Доказать, что если функция $f(x_1, x_2, x_3, x_4)$ примитивно рекурсивна, то следующие функции примитивно рекурсивны:

(1) $g(x_1, x_2, x_3, x_4) = f(x_2, x_1, x_3, x_4)$ – перестановка аргументов;

(2) $\varphi(x_1, x_2, x_3, x_4) = f(x_2, x_3, x_4, x_1)$ – циклическая перестановка аргументов;

(3) $h(x_1, x_2, x_3, x_{4,}, x_5) = f(x_1, x_2, x_3, x_4)$ – введение фиктивного аргумента;

(4) $\psi(x_1, x_2, x_3) = f(x_1, x_2, x_3)$ – отождествление аргументов.

2. Доказать, что любая примитивно рекурсивная функция всюду определена.

3. Доказать, что следующие функции примитивно рекурсивны:

(1) $f(x) = x + 5$

(2) $f(x) = 3$

(3) $f(x) = x + n$

(4) $f(x, y) = x + y$

(5) $f(x) = 3x$

(6) $f(x) = sg(x)$

(7) $f(x) = \overline{sg}(x)$

(8) $f(x, y) = x^y$

(9) $f(x, y) = x \dot{-} y = \begin{cases} 0, x = 0 \\ x - y, x > 0 \end{cases}$

(10) $f(x, y) = |x - y|$

(11) $f(x) = x^2 + 3$

4. Доказать, что функции s(x), o(x), $I_n^m(x_1, x_2 \dots x_n)$, $C_k(x) = k, F_k(x) = x + k$ – общерекурсивны.

5. Применяя операцию примитивной рекурсии к функциям g(x) и h(x,y,z) по переменной y, построить функцию $f(x, y) = R(g, h)$, записав ее в «аналитической» форме:

(1) g(x) = x, h(x,y,z) = x+y

(2) g(x) = x, h(x,y,z) = x+z

(3) g(x) = 2x, h(x,y,z) = x+2z

(4) g(x) = x, h(x,y,z) = z^x

(5) g(x) = x, h(x,y,z) = z^y

(6) g(x) = x, h(x,y,z) = x^z

6. Применить операцию минимизации к следующим функциям:

(1) $f(x) = x + 1$

(2) $f(x) = x \dot{-} 2$

(3) $f(x) = x - 2$

(4) $f(x) = \frac{x}{2}$

(5) $f(x) = \left[\frac{x}{2}\right]$

Задачи для самостоятельной работы

1. Доказать, что следующие функции примитивно рекурсивны:

(1) $f(x) = x^2 + 3$

(2) $f(x) = x^2 + 2y^2$

(3) $f(x) = (x + 2)^2$

(4) $f(x, y, z) = 3x + 4y + 5z$

(5) $f(x) = 3^{x+2}$

(6) $f(x) = sg(2^x)$

(7) $f(x, y) = 2^x \dot{-} y$

(8) $f(x) = x^4 + 5$

(9) $f(x) = (x + 2)!$

(10) $f(x, y) = x \dot{-} y$

(11) $f(x, y) = x^2 + 3^y$

(12) $f(x, y) = sg(x \dot{-} 3) + y!$

(13) $f(x, y) = x + (2 \dot{-} y)$

2. Применяя операцию примитивной рекурсии к функциям g(x) и h(x,y,z) по переменной y, построить функцию $f(x,y) = R(g,h)$, записав ее в «аналитической» форме:

(1) $g(x) = x^2$, $h(x,y,z) = x^2+2y$

(2) $g(x) = x^2$, $h(x,y,z) = (x+2)xz$

(3) $g(x) = 2^x$, $h(x,y,z) = 2^x y$

(4) $g(x) = 3^x$, $h(x,y,z) = 3^x z$

(5) $g(x) = 3^x$, $h(x,y,z) = 3^y z$

(6) $g(x) = 3^x$, $h(x,y,z) = 3^x y$

3. Применить операцию минимизации к следующим функциям:

(1) $f(x) = sg(x-3)$

(2) $f(x) = sg(x \dot{-} 3)$

(3) $f(x) = \frac{x}{5}$

(4) $f(x) = \left[\frac{x}{5}\right]$

(5) $f(x) = 5x \dot{-} 2$

(6) $f(x) = 5x + 2$

(7) $f(x) = 5x - 2$

ЧАСТЬ V. НЕКОТОРЫЕ ВОПРОСЫ ТЕОРИИ АЛГОРИТМОВ

Лекционное занятие № 21 «Машины Поста»

План занятия:

1. ***Принцип работы МП.***
2. ***Примеры работы МП.***

1. Принцип работы МП

В 1936 году американский математик и логик Эмиль Леон Пост (1897–1954) предложил абстрактную вычислительную конструкцию, позволяющую формально определить алгоритм и названную впоследствии машиной Поста (МП). При разработке вычислительной конструкции Пост руководствовался принципом создания максимально простой абстракции: минимум операций при обработке информации, входная информация должна быть закодирована с использованием минимального набора символов.

Несмотря на «примитивность» МП, любой существующий алгоритм может быть записан в виде программы для МП. **Тезис Поста:** «Всякий алгоритм представим в форме МП». Этот тезис одновременно является формальным определением алгоритма.

Алгоритм (по Посту) – программа для машины Поста, приводящая к решению поставленной задачи.

Тезис Поста является гипотезой. Его невозможно строго доказать (так же, как и тезис Тьюринга), потому что в нем фигурирует, с одной стороны, интуитивное понятие «всякий алгоритм», а с другой стороны – точное понятие «машина Поста». Для того чтобы опровергнуть гипотезу Поста, необходимо придумать алгоритм, который невозможно записать в виде программы для машины Поста. На сегодняшний день такого алгоритма не существует.

МП – это абстрактная (то есть не существующая в арсенале действующей техники), но очень простая вычислительная машина. Она способна выполнять лишь самые элементарные действия, и потому ее описание и составление простейших программ может быть доступно студентам в ходе самостоятельной работы. Тем не менее на МП можно запрограммировать – в известном смысле – любые алгоритмы. Разработка программ для машин Поста – достаточно эффективный этап в обучении алгоритмизации, так как в процессе написания этих программ студенты учатся разбивать интуитивно понятные вычислительные процедуры на элементарные действия. Изучение МП полезно не только студентам математических направлений, но и школьникам, интересующимся информатикой и математикой.

МП состоит из каретки (или считывающей и записывающей головки) и разбитой на секции бесконечной в обе стороны ленты. Каждая секция ленты может быть либо пустой – 0

(или X), либо помеченной меткой 1 (или V). За один шаг каретка может сдвинуться на одну позицию влево или вправо, считать, поставить или стереть символ в том месте, где она стоит.

Работа МП определяется программой, состоящей из конечного числа строк. Для работы машины нужно задать программу и ее начальное состояние (то есть состояние ленты и позицию каретки). Кареткой управляет программа, состоящая из строк команд. Каждая команда имеет следующий синтаксис:

- i – номер команды;
- K – действие каретки;
- j – номер следующей команды (отсылка).

Всего для МП существует шесть типов команд:

- V j – поставить метку, перейти к j-й строке программы;
- X j – стереть метку, перейти к j-й строке программы;
- ← j – сдвинуться влево, перейти к j-й строке программы;
- ⟶ j – сдвинуться вправо, перейти к j-й строке программы;
- ? j_1; j_2 – если в ячейке нет метки, то перейти к j_1-й строке программы, иначе перейти к j_2-й строке программы;
- ! – конец программы (стоп, останов).

Если номер строки перехода в командах не указан, то происходит переход к следующей строке.

У команды «стоп» отсылки нет. После запуска возможны варианты:

- работа может закончиться невыполнимой командой (стирание несуществующей метки или запись в помеченное поле);
- работа может закончиться командой «стоп»;
- работа никогда не закончится.

Приведем список недопустимых действий, ведущих к аварийной остановке машины:

- попытка записать 1 (V) в заполненную ячейку;
- попытка стереть метку в пустой ячейке;
- бесконечное выполнение (вообще говоря, это трудно назвать аварийным остановом, но бессмысленное повторение одних и тех же действий – зацикливание – ничуть не лучше вышеперечисленного).

Будем говорить, что мы можем применить программу к текущему состоянию МП, если выполнение программы не приведет к зацикливанию, то есть рано или поздно мы выполним команду «останов».

2. Примеры работы МП

Пример 1: программа, которая не применима ни к одному состоянию МП.

1. ⟶ 1;
2. !.

Пример 2: программа, которая применима к любому состоянию МП.

1. !.

Пример 3: программа, не применимая ни к какому состоянию МП, и зона работы для любого начального состояния бесконечна.

1. ⟶ 1.

Пример 4: машина, не применимая ни к какому состоянию МП, и зона работы для любого начального состояния ограничена одним и тем же числом ячеек, не зависящим от выбранного начального состояния ленты.

1. ⟶ 1;
2. ← 1.

Пример 5: на ленте проставлена метка в единственной ячейке. Каретка стоит на некотором расстоянии левее этой ячейки. Необходимо подвести каретку к ячейке, стереть метку и остановить каретку слева от этой ячейки.

Поскольку нам известно, что каретка стоит напротив пустой ячейки, но неизвестно, сколько шагов нужно совершить до пустой ячейки, мы можем сразу сделать шаг вправо; проверить, заполнена ли ячейка; если она пустая, то повторять эти действия до тех пор, пока не наткнемся на заполненную ячейку. Как только мы ее найдем, мы выполним операцию стирания, после чего нужно будет лишь сместить каретку влево и остановить выполнение программы.

Программа для МП:

1) ⟶ 2;
2) ? 1; 3;
3) X4;
4) ← 5;
5) !.

Пример 6: прибавление единицы к числу (рис. 91):

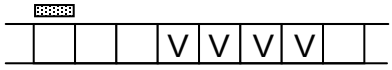


Рис. 91.

1. ⟶ 2 – сдвинуть каретку вправо, перейти к команде 2 (рис. 92):

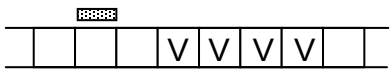


Рис. 92.

2. ? 1; 3 – если метки нет, то перейти к команде 1, иначе к команде 3 (рис. 93):

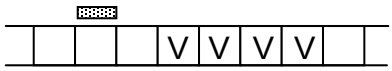


Рис. 93.

3. ⟶ 2 – сдвинуть каретку вправо, перейти к команде 2 (рис. 94):

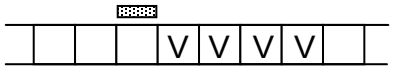


Рис. 94.

4. ? 1; 3 – если метки нет, то перейти к команде 1, иначе – к команде 3 (рис. 95):

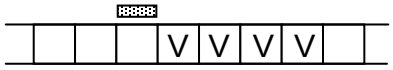


Рис. 95.

5. ⟶ 2 – сдвинуть каретку вправо, перейти к команде 2 (рис. 96):

Рис. 96.

6. ? 1; 3 – если метки нет, то перейти к команде 1, иначе – к команде 3 (рис. 97):

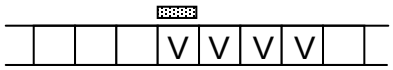


Рис. 97.

7. ← 4 – сдвинуть каретку влево, перейти к команде 4 (рис. 98):

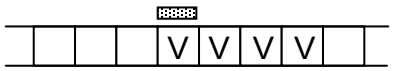


Рис. 98.

8. V 5 – поставить метку, перейти к команде 5.

9. ! (рис. 99):

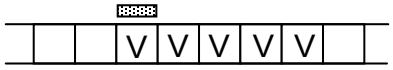


Рис. 99.

Пример 7: на ленте задан массив меток, увеличить длину массива на две метки (каретка находится либо слева от массива, либо над одной из ячеек самого массива).

1. ? 2; 3.
2. ⟶ 1 – передвинули каретку к массиву.
3. ⟶ 4.
4. ? 5; 3 – передвинули каретку к концу массива.
5. V6.
6. ⟶ 7 – ставим две метки в конце массива.
7. V8.
8. !.

Пример 8: даны два массива меток, которые находятся на некотором расстоянии друг от друга, требуется соединить их в один массив (каретка находится над крайней левой меткой первого массива).

1. X 2.
2. ⟶ 3.
3. ? 4; 2.
4. V5.
5. ⟶ 6.
6. ? 8; 7.
7. !.
8. ← 9.
9. ? 10; 8.
10. ⟶ 1.

Пример 9: на ленте задана последовательность массивов, включающая в себя один и более массивов, при этом два соседних массива отделены друг от друга одной пустой ячейкой. Необходимо на ленте оставить один массив длиной, равной сумме длин массивов, присутствовавших изначально (каретка находится над крайней левой меткой первого (левого) массива).

1. X 2.
2. ⟶ 3.
3. ? 4; 2.
4. V5.
5. ⟶ 6.
6. ? 10; 7.
7. ← 8.
8. ? 9; 7.
9. ⟶ 1.
10. !.

Лекционное занятие № 22 «Машины с неограниченными регистрами»

План занятия:

1. ***Описание машины с неограниченными регистрами (МНР).***
2. ***Принцип работы МНР.***
3. ***Вычисление функций на МНР.***

1. Описание машины с неограниченными регистрами (МНР)

Машина с неограниченными регистрами (МНР) – это абстрактная машина, более сходная с реальными компьютерами по сравнению с машиной Тьюринга. Она имеет следующее описание (рис. 100):

- регистры R_1 , $R_2, ...,$ в которых содержатся соответственно натуральные числа $r_1,\ r_2, ...;$
- число регистров бесконечно, но только конечное множество регистров $R_1,\ R_2, ... R_k$ содержит числа, отличные от нуля. Все остальные регистры заполнены нулями.

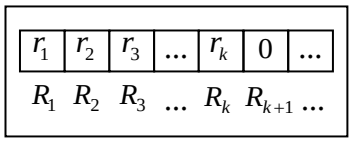


Рис. 100.

Программа МНР – это конечная последовательность, состоящая из следующих типов команд: $Z(n), S\ (n), T\ (m,n), J(m,n,q)$, где m,n,q ϵ \{1,2,3,…}. Эти команды выполняют следующие действия:

– команда обнуления $Z(n)$ делает содержимое регистра R_n равным нулю;

– команда прибавления единицы $S\ (n)$ к содержимому регистра R_n прибавляет число, равное 1;

– команда переадресации $T\ (m,n)$ заменяет содержимое регистра R_n на содержимое регистра R_m;

– команда условного перехода $J(m,n,q)$ сравнивает содержимое регистров R_m и R_n. При $r_m = r_n$ в качестве следующей команды выполняется команда с номером q, в противном случае выполняется следующая по порядку команда программы.

Команды обнуления, прибавления единицы и переадресации называются арифметическими командами.

Пример 1: регистры МНР имеют вид (рис. 101):

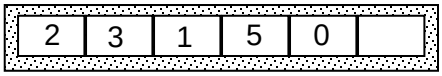


Рис. 101.

Выполним команду $S\ (2)$. Эта команда прибавит число 1 к числу 3 в регистре R_2 и не затронет остальные регистры. В результате получим следующее содержимое регистров (рис. 102):

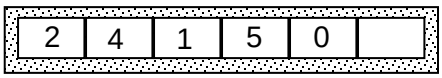


Рис. 102.

Выполним затем команду $T\ (2,1)$. Содержимое регистра R_2 (число 4) заменит старое содержимое в регистре R_1 (число 2) (рис. 103):

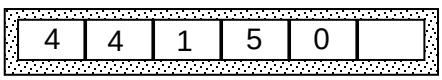


Рис. 103.

2. Принцип работы МНР

МНР работает по тактам: такт 1, такт 2, … . Первый такт работы МНР с программой $I_1, I_2, \dots, I_n$ – выполнение первой команды I_1. Затем выполняются команды $I_2, I_3, \dots$.

МНР останавливается тогда и только тогда, когда невозможно выполнить предписанную команду. Это означает, что МНР только что совершила i-й такт работы и следующим $i+1$ тактом должна выполнить несуществующую команду. Эта ситуация при выполнении программы $I_1, I_2, \dots, I_n$ возникает ровно в одном из трех случаев:

- если в i-м такте выполнена I_n – последняя команда программы и эта команда не является командой условного перехода, тогда следующим $i+1$ тактом должна выполняться несуществующая команда I_{n+1};
- если в i-м такте выполнена команда условного перехода $J(m, n, q)$, где $r_m = r_k$ и $q > n$, тогда следующим $i+1$ тактом должна выполняться несуществующая команда I_q;
- если в i-м такте выполнена I_n – последняя команда программы и эта команда является командой условного перехода $J(m, n, q)$, при $r_m \neq r_k$, тогда следующим $i+1$ тактом должна выполняться несуществующая команда I_{n+1}.

Если выполнение программы завершилось, то число r_1 из регистра R_1 считается результатом применения алгоритма к исходному набору чисел $r_1, r_2, \dots$.

Если выполнение программы никогда не заканчивается, то нет результата вычислений. В этом случае алгоритм не применим к исходным данным. Тем самым при работе МНР возможно лишь два случая завершения работы: выдача результата и бесконечная работа. Третий случай (безрезультативная остановка) невозможен.

3. Вычисление функций на МНР

Как и в случае с МТ, мы можем указать, как МНР вычисляет частичную функцию $f(x_1, x_2, \dots, x_n)$ от n аргументов. Рассмотрим набор аргументов $x_1, x_2, \dots, x_n$ и разместим число x_1 в регистре R_1, число x_2 в регистре R_2, …, число x_n в регистре R_n. Все остальные регистры заполнены нулями. Получаем начальную конфигурацию МНР. После окончания работы в регистре R_1 должно быть значение функции $f(x_1, x_2, \dots, x_n)$. Если значение $f(x_1, x_2, \dots, x_n)$ не определено, то МНР должна работать бесконечно.

Пример 2: составить программу для МНР, которая вычисляет функцию $f(x, y) = x + y$.

Рассмотрим работу МНР со следующей программой:

$$I_1: J(2, 3, 5)$$
$$I_2: S(1)$$
$$I_3: S(3)$$
$$I_4: J(1, 1, 1)$$

В регистр R_1 перед первым тактом работы машины занесено число x, в R_2 – число y, остальные регистры заполнены нулями (рис. 104).

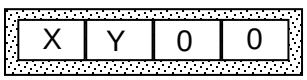


Рис. 104.

Первым тактом работы МНР исполняется команда $I_1: J\ (2, 3, 5)$, где R_2 сравнивается с R_3, то есть y сравнивается с 0. Допустим, например, что $y = 2$. Поскольку $y \neq 0$, то вторым тактом работы исполнится команда $I_2: S\ (1)$, а третьим – $I_3: S\ (3)$. В результете к числам в регистрах R_1 и R_3 добавится число 1, а каждый из них увеличится на 1 (рис. 105).

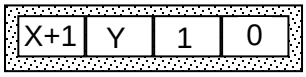


Рис. 105.

Далее команда $I_4: J\ (1, 1, 1)$ сравнивает регистр R_1 с самим собой, получаем равенство и переход к первой команде $I_1: J\ (2, 3, 5)$. Цикл повторится второй раз, и к числам в регистрах R_1 и R_2 снова добавится число 1. Получим регистры (рис. 106):

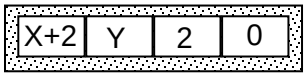


Рис. 106.

Далее $I_4: J\ (1, 1\ ,1)$ задает переход к первой команде $I_1: J\ (2, 3, 5)$.

МНР готова выполнить команду $I_1: J\ (2, 3, 5)$ и совершить третий проход цикла, однако в регистрах R_2 и R_3 содержатся равные числа 2 и y. Тогда, $I_1: J\ (2, 3, 5)$ вызывает команду с номером 5, но такой команды нет. Поэтому МНР останавливается, в регистре R_1 содержится результат $x + 2$.

В общем случае произойдет y проходов цикла, добавляющих к числам в регистре R_1 и R_3 число 1. Вначале $y + 1$ прохода цикла в момент исполнения команды $I_1: J\ (2, 3, 5)$ в регистре R_1 имеем $x + y$, в регистре R_3 – число y. По команде $I_1: J\ (2, 3, 5)$ МНР останавливается, имея в регистре R_1 результат вычислений $x + y$.

Пример 3: нигде не определенная функция может вычисляться, например, следующей программой $-\ J\ (1, 1, 1)$.

Теорема 3.1. Простейшие функции
s(x) = x + 1, $O(x_1, x_2, \ldots, x_n)$ = 0, $I_n^m(x_1, x_2, \ldots, x_n) = x_m$, $1 \leq m \leq n$ вычислимы в МНР.

Укажем программы для данных функций:

- $S\ (1)$, если в регистре R_1 занесено число x, то МНР выполнит один такт работы, сменит число x в регистре R_1 на $x+1$ и остановится;
- $Z(n)$ вычисляет нулевую функцию;
- $T\ (m, 1)$, МНР выполнит один такт работы, перешлет в регистр R_1 число x_m и остановится.

Лекционное занятие № 23 «Алгоритмически неразрешимые задачи»

План занятия:

1. ***Алгоритмическая неразрешимость.***
2. ***Отсутствие общего метода решения задачи.***
3. ***Информационная неопределенность задачи.***
4. ***Логическая неразрешимость (теорема Гёделя «о неполноте»).***

1. Алгоритмическая неразрешимость

За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и научных проблем. Мы уже задавались вопросом: а существуют **ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения?**

Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным – мы констатируем, что **мы не только сейчас не знаем соответствующего алгоритма, но мы не можем принципиально никогда его найти.**

Успехи математики к концу XIX века привели к формированию мнения, которое выразил Д. Гильберт, – **«в математике не может быть неразрешимых проблем»**, в связи с этим формулировка проблем Гильбертом на конгрессе 1900 года в Париже была руководством к действию, констатацией отсутствия решений.

Первой фундаментальной теоретической работой, связанной с доказательством алгоритмической неразрешимости, была работа Курта Гёделя – его известная теорема о неполноте символических логик. Это была строго формулированная математическая проблема, для которой не существует решающего ее алгоритма. Усилиями различных исследователей список алгоритмически не разрешимых проблем был значительно расширен. Сегодня принято при доказательстве алгоритмической неразрешимости некоторой задачи сводить ее к ставшей классической задаче – «проблема останова».

В теории вычислимости **алгоритмически неразрешимой задачей** называется задача, имеющая ответ да или нет для каждого объекта из некоторого множества входных данных, для которой (принципиально) не существует алгоритма, который бы, получив любой возможный в качестве входных данных объект, останавливался и давал правильный ответ после конечного числа шагов.

Теорема 1.1: не существует алгоритма (МТ), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм, и данные заданы символами на ленте МТ) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, то есть невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

Тем не менее можно попытаться сформулировать причины, ведущие к алгоритмической неразрешимости, эти причины достаточно условны, так как все они сводимы к проблеме останова, однако такой подход позволяет более глубоко понять природу алгоритмической неразрешимости.

2. Отсутствие общего метода решения задачи

Пример 1: распределение девяток в записи числа π.

Определим функцию $f(n) = i$, где n – количество девяток подряд в десятичной записи числа π = 3,141592, а i – номер самой левой девятки из n девяток подряд: f(1) = 5.

Задача состоит в вычислении функции f(n) для произвольно заданного n.

Поскольку число π является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток (равно как и любых других цифр) в десятичной записи числа π. Вычисление f(n) связано с вычислением последующих цифр в разложении π,

до тех пор, пока мы не обнаружим n девяток подряд, однако у нас нет общего метода вычисления f(n), поэтому для некоторых n вычисления могут продолжаться бесконечно – мы даже не знаем в принципе (по природе числа π), существует ли решение для всех n.

Пример 2: вычисление совершенных чисел. Совершенные числа – это числа, которые равны сумме своих делителей, например: 28 = 1 + 2 + 4 + 7 + 14.

Поставим задачу вычисления S(n) по произвольно заданному n. Нет общего метода вычисления совершенных чисел, мы даже не знаем, множество совершенных чисел конечно или счетно, поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос об останове алгоритма. Если мы проверили M чисел при поиске n-го совершенного числа – означает ли это, что его вообще не существует?

Пример 3: десятая проблема Гильберта.

Пусть задан многочлен n-й степени с целыми коэффициентами – P, существует ли алгоритм, который определяет, имеет ли уравнение P = 0 решение в целых числах?

Ю. В. Матиясевич показал, что такого алгоритма не существует, то есть отсутствует общий метод определения целых корней уравнения P = 0 по его целочисленным коэффициентам.

3. Информационная неопределенность задачи

Пример 4: позиционирование машины Поста на последний помеченный ящик.

Пусть на ленте машины Поста заданы наборы помеченных ящиков (кортежи) произвольной длины с произвольными расстояниями между кортежами и головка находится у самого левого помеченного ящика. Задача состоит в установке головки на самый правый помеченный ящик последнего кортежа. Попытка построения алгоритма, решающего эту задачу, приводит к необходимости ответа на вопрос – больше на ленте кортежей нет или они есть где-то правее? Информационная неопределенность задачи состоит в отсутствии информации либо о количестве кортежей на ленте, либо о максимальном расстоянии между кортежами – при наличии такой информации (при разрешении информационной неопределенности) задача становится алгоритмически разрешимой.

4. Логическая неразрешимость (теорема Гёделя о неполноте)

Пример 5: проблема «останова» (см. теорема 1.1).

Пример 6: проблема эквивалентности алгоритмов.

По двум произвольным заданным алгоритмам (например, по двум МТ) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Пример 7: проблема тотальности.

По произвольному заданному алгоритму определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи – является ли частичный алгоритм P всюду определенным?

Пример 8: проблема самоприменимости. Самоприменимость – это свойство алгоритма успешно завершаться на данных, представляющих собой формальную запись этого же алгоритма.

Задача распознавания самоприменимости сводится к тому, чтобы найти алгоритм, позволяющий за конечное число шагов по формальной записи некоего алгоритма узнать, является ли он самоприменимым или нет. Хотя эта задача несколько искусственна и не представляет самостоятельного интереса, но часто используется для того, чтобы доказать неразрешимость других, более сложных задач. Общий метод для подобных выводов состоит в том, что из предположения о существовании алгоритма, решающего некую задачу, выводится существование алгоритма, решающего задачу распознавания самоприменимости.

В теории алгоритмов проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются **частично разрешимыми проблемами.** В частности, проблема останова так же является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

Лекционное занятие № 24 «Сложность алгоритмов»

План занятия:

1. ***Вычислительная сложность алгоритма.***
2. ***Измерения сложности алгоритмов.***
3. ***О-сложность алгоритмов.***
4. ***Классы сложности.***

1. Вычислительная сложность алгоритма

В теории алгоритмов зачастую изучается лишь принципиальная возможность алгоритмического решения задач. При рассмотрении конкретной задачи не обращается внимание на ресурсы времени и памяти для соответствующих им разрешающих алгоритмов. Однако нетрудно понять, что принципиальная возможность алгоритмического решения задачи еще не означает, что оно может быть практически получено.

В информатике и теории алгоритмов **вычислительная сложность алгоритма** – это функция, определяющая зависимость объема работы, выполняемой некоторым алгоритмом, от размера входных данных. Раздел, изучающий вычислительную сложность, называется **теорией сложности вычислений**. Объем работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами. Время определяется количеством элементарных шагов, необходимых для решения задачи, тогда как пространство определяется объемом памяти или места на носителе данных.

Теория сложности вычислений возникла из потребности сравнивать быстродействие алгоритмов, четко описывать их поведение (время исполнения, объем необходимой памяти и т. д.) в зависимости от размера входных и выходных данных.

Количество элементарных операций, затраченных алгоритмом для решения конкретной задачи, зависит не только от размера входных данных, но и от самих данных. Например, количество операций алгоритма сортировки вставками значительно меньше в случае, если входные данные уже отсортированы.

2. Измерения сложности алгоритмов

Под сложностью алгоритма можно понимать количество элементарных действий в вычислительном процессе этого алгоритма. Существует несколько способов измерения сложности алгоритма. Программисты обычно сосредотачивают внимание на скорости алгоритма, но не менее важны и другие показатели – требования к объему памяти, свободному месте на диске. Использование быстрого алгоритма не приведет к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у компьютера.

Многие алгоритмы предлагают выбор между объемом памяти и скоростью. Задачу можно решить быстро, используя большой объем памяти, или медленнее, занимая меньший объем.

Пример 1: алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы. Результат будет получен мгновенно, но это потребует огромного объема памяти. Карта большого города может содержать десятки тысяч точек. Тогда описанная выше таблица должна содержать более 10 млрд. ячеек, то есть для того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гб памяти. Из этой зависимости проистекает идея объемно-временной сложности. При таком подходе алгоритм оценивается как с точки зрения скорости выполнения, так и с точки зрения потребленной памяти.

При сравнении различных алгоритмов важно знать, как их сложность зависит от объема входных данных. Допустим, при сортировке одним методом обработка тысячи чисел занимает 1 с., а обработка миллиона чисел – 10 с., при использовании другого алгоритма может потребоваться 2 с. и 5 с. соответственно. В таких условиях нельзя однозначно сказать, какой алгоритм лучше.

В общем случае сложность алгоритма можно оценить по порядку величины. Алгоритм имеет сложность O(f(n)), если при увеличении размерности входных данных N время выполнения алгоритма возрастает с той же скоростью, что и функция f(N).

Сложность вычислений определяется:

- числом элементарных операций (временная сложность);
- объемом используемых ресурсов памяти (пространственная сложность).

Различают: сложность в наихудшем случае; сложность в наилучшем случае; сложность в среднем.

Временная сложность – это время, необходимое для его выполнения в зависмости от исходных данных. Оно равно произведению числа элементарных действий на среднее время выполнения одного действия.

Временная сложность алгоритма (в худшем случае) – это функция размера входных и выходных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера. В задачах, где размер выхода не превосходит или пропорционален размеру входа, можно рассматривать временную сложность как функцию размера только входных данных.

Аналогично понятию временной сложности в худшем случае определяется понятие **«временная сложность алгоритма в наилучшем случае»**. Также рассматривают понятие

«среднее время работы алгоритма», то есть математическое ожидание времени работы алгоритма. Иногда говорят просто: «временная сложность алгоритма» или «время работы алгоритма», имея в виду временную сложность алгоритма в худшем, наилучшем или среднем случае (в зависимости от контекста).

По аналогии с временной сложностью определяют **пространственную сложность** алгоритма, только здесь говорят не о количестве элементарных операций, а об объеме используемой памяти.

Несмотря на то, что функция временной сложности алгоритма в некоторых случаях может быть определена точно, в большинстве случаев искать точное ее значение бессмысленно. Дело в том, что, во-первых, точное значение временной сложности зависит от определения элементарных операций (например, сложность можно измерять в количестве арифметических операций или операций на машине Тьюринга), а во-вторых, при увеличении размера входных данных вклад постоянных множителей и слагаемых низших порядков, фигурирующих в выражении для точного времени работы, становится крайне незначительным.

Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию **асимптотической сложности алгоритма**. При этом алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера.

Сложность алгоритма определяется исходя из вычислительной модели, в которой проводятся вычисления.

Наряду со сложностью важной характеристикой алгоритма является эффективность. Под эффективностью понимается выполнение следующего требования: не только весь алгоритм, но и каждый шаг его должны быть такими, чтобы исполнитель был способен выполнить их за разумное время. Например, если алгоритм, выдающий прогноз погоды на ближайшие сутки, выполняется неделю, то такой алгоритм просто-напросто никому не нужен. Если мы рассматриваем алгоритмы, реализующиеся на компьютере, то к требованию выполнения за разумное время прибавляется требование выполнения в ограниченном объеме оперативной памяти.

3. О-сложность алгоритмов

Принято говорить **об О-сложности алгоритмов:**

- O (1) – большинство операций в программе выполняется только раз или только несколько раз алгоритмами константной сложности, любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность;
- O (N) – время работы программы линейно;
- O (N^2), O (N^3), O (N^a) – полиномиальная сложность;
- O ($Log\ (N)$) – когда время работы логарифмическое, программа начинает работать намного медленнее с увеличением N, такое время работы встречается обычно в программах, которые делят большую проблему в маленькую и решают их по отдельности;
- O (2^n) – экспотенциальная сложность и др.

Правила для определения сложности:

- постоянные множители не имеют значения для определения порядка сложности;
- порядок сложности произведения двух функций равен произведению их сложностей;
- порядок сложности суммы функций определяется как порядок доминанты первого и второго слагаемых, то есть выбирается наибольший порядок.

4. Классы сложности

Классы сложности – это множество вычислительных задач, примерно одинаковых по сложности вычисления.

Более узко, классы сложности – это множество предикатов (функций, получающих на вход слово и возвращающих ответ 0 или 1), использующих для вычисления примерно одинаковые количества ресурсов.

Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих свойством: для каждого класса существует категория задач, которые являются «самыми сложными». Это означает, что любая задача из класса сводится к такой задаче, и притом сама задача лежит в классе.

Такие задачи называются **полными задачами** для данного класса. Наиболее известными являются NP – полные задачи.

Полные задачи – хороший инструмент для доказательства равенства классов. Достаточно для одной такой задачи предоставить алгоритм, решающий ее и принадлежащий более маленькому классу, и равенство будет доказано.

Классом сложности X называется множество предикатов P(x), вычислимых на МТ и использующих для вычисления O(f(n)) ресурса, где n – длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов МТ) или рабочая зона (количество использованных ячеек на ленте во время работы).

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие. Однако про большинстве включений не известно, являются ли они строгими. Одна из наиболее известных открытых проблем в этой области – равенство P и NP (рис. 107).

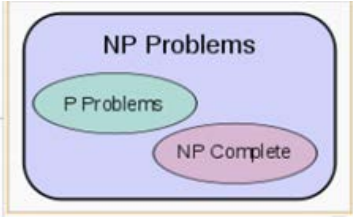


Рис. 107.

Класс P вмещает все те проблемы, решение которых считается «быстрым», которые могут быть решены за время, полиномиально зависящее от объема исходных данных. Сюда относятся сортировка, поиск во множестве и многие другие.

Класс NP содержит задачи, которые недетерминированная МТ в состоянии решить за полиномиальное количество времени, тогда как детерминированной МТ полиномиальный алгоритм неизвестен. Следует отметить, что недетерминированная МТ является лишь абстрактной моделью, в то время как современные компьютеры соответствуют детерминированной МТ с ограниченной памятью. В класс NP входят многие знаменитые проблемы, такие, как задача коммивояжера, задача выполнимости булевых функций, факторизация и другие.

МТ называется **детерминированной**, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила. Если существует пара (ленточный символ – состояние), для которой существует две и более команд, такая МТ называется **недетерминированной.**

Класс NP включает в себя класс P, а также некоторые проблемы (NP – полные задачи), для решения которых известны лишь алгоритмы, эспотенциально зависящие от размера входа (то есть не эффективные для больших входов).

Из определения классов P и NP сразу вытекает следствие: $P \subseteq NP$. Однако до сих пор ничего не известно о строгости этого включения, то есть существует ли задача, лежащая в *NP*, но не лежащая в *P*. Если такой задачи не существует, то все задачи, принадлежащие классу *NP*, можно будет решать за полиномиальное время, что сулит огромную выгоду в скорости вычислений. Сейчас самые сложные задачи из класса *NP* можно решить за экспоненциальное время, что считается неприемлемым с практической точки зрения.

Пример 2: верно ли, что среди чисел $\{-2, -3, 15, 14, 7, -10, \ldots\}$ есть такие, что их сумма равна *0* (задача о суммах подмножеств)? Ответ: *да*, потому что $-2 -3 + 15 -10 = 0$ легко проверяется несколькими сложениями (информация, необходимая для проверки положительного ответа, называется сертификатом).

Следует ли отсюда, что так же легко подобрать эти числа? Проверить сертификат так же легко, как найти его? Кажется, что подобрать числа сложнее, но это не доказано.

Впервые вопрос о равенстве классов P и NP был поставлен Стивеном Куком в 1971 году и независимо Леонидом Левиным в 1973 году. На сегодняшний день вопрос о равенстве двух классов P и NP считается одной из самых сложных открытых проблем в области теоретической информатики. Математический институт Клэя включил эту проблему в список проблем тысячелетия, предложив награду размером в один миллион долларов США за ее решение.

Попытки доказательства:

- 6 августа 2010 года сотрудник исследовательской лаборатории Hewlett-Packard в Пало-Альто Винэй Деолаликар разослал некоторым ученым на проверку свое доказательство неравенства P и NP. Стивен Кук назвал его «относительно серьезной попыткой решения проблемы P vs NP». Однако уже в том же месяце были найдены недостатки в доказательстве. Деолаликар заявил, что в следующей версии доказательства он постарается учесть все замечания. На викистранице «Deolalikar P vs NP paper», связанной с проектом Polymath, приводится критический анализ, собраны предполагаемые ошибки и некоторые опечатки в работе Деолаликара. Там же можно проследить за онлайн-реакцией на предложенное доказательство.
- В 2012 году профессор Восточно-украинского национального университета им. В. Даля, кандидат технических наук Анатолий Плотников опубликовал свой вариант доказательства неравенства $P \neq NP$. По его словам, ранее ему удалось решить эту задачу для частного случая. Текущее решение претендует на общее и в настоящее время проходит проверку.

В настоящее время большинство математиков считают, что эти классы не равны. Согласно опросу, проведенному в 2002 году среди 100 ученых, 61 человек считает, что ответ – «не равны», 9 – «равны», 22 затруднились ответить и 8 считают, что гипотеза не выводима из текущей системы аксиом и, таким образом, не может быть доказана или опровергнута.

Лекционное занятие № 25 «Нумерация алгоритмов»

План занятия:

1. ***Нумерация МТ.***
2. ***Нумерация МНР.***

1. Нумерация МТ

Зафиксируем счетные множества символов $\{a_0, a_1, \dots, a_i, \dots\}$ и $\{q_0, q_1, \dots, q_j, \dots\}$. Будем считать, что внешние и внутренние алфавиты всех МТ берутся из этих множеств. При этом будем считать, что a_0 принадлежит всем внешним алфавитам машин и интерпретируется как пустой символ, а q_0, q_1 принадлежат всем внутренним алфавитам машин и всегда означают заключительное и начальное состояния соответственно.

Опишем теперь единый способ представления информации о машинах с помощью кодирования. Каждому символу из множества $\{L, R, E, a_0, a_1, \dots, a_i, \dots, q_0, q_1, \dots, q_j, \dots\}$ поставим в соответствие двоичный набор согласно таблице (рис. 108).

Далее команде I машины Т, имеющей вид $qa \to \dot{q}\dot{a}d$, ставится в соответствие двоичный набор вида: $Код\ (I) = Код\ (q) Код\ (a)\ Код\ (\dot{q}) Код\ (\dot{a})\ Код\ (d)$, в котором коды букв приписаны друг к другу. Пусть машина имеет алфавиты $A = \{a_0, a_1, \dots, a_m\}$, $Q = \{q_0, q_1, \dots, q_n\}$. Упорядочим команды машины в соответствии с лексикографическим порядком левых частей команд: $q_1a_0, q_1\ a_1, q_1a_m, q_2a_0, \dots, q_2a_m, \dots, q_na_0, \dots, q_na_m$.

Пусть $I_1, \dots, I_{n(m+1)}$ – последовательность команд. Тогда машине Т поставим в соответствие двоичный набор вида: $Код\ (T) = Код\ (I_1) Код\ (I_2)$ ... $Код\ (I_{n(m+1)})$, полученный приписыванием к друг другу кодов команд.

	Символ	Код	Число нулей в коде
Символы сдвига	R	10	1
	L	100	2
	E	1000	3
Символы алфавита ленты	a_0	10000	4
	a_1	100000	6
	.	…	…
		100…00	2i+4
	a_i	…	…
	.		
Символы алфавита состояния	q_0	100000	5
	q_1	10000000	7
	…	…	…
	q_1	1000…000	2i+5

Рис. 108.

Пример 1: пусть дана машина Т, заданная $A = \{a_0, a_1\}$, $Q = \{q_0, q_1, \}$.
Т: $q_1a_0 \to q_0a_0\,E,\ q_1a_1 \to q_0a_1E$.

Имеем код: *Код* $(T) = 10^7 10^4 10^5 10^4 10^3 10^7 10^6 10^5 10^6 10^3$.

Легко видеть, что машина Т переводит конфигурации $q_1 a_1{}^x$ в конфигурации $q_0 a_1{}^x$, и поэтому, представляя натуральное число n как $a_1{}^{n+1}$, получаем, что машина Т вычисляет функцию $f(x) = x$.

Ясно, что указанное кодирование является алгоритмической процедурой. Имея код машины, можно однозначно восстановить множество ее команд – для этого надо выделить подслова, начинающиеся с единицы с нулями до следующей единицы. Пятерка таких подслов образует команду. Далее, вероятнее всего, имеется алгоритмическая процедура, позволяющая по произвольному слову из нулей и единиц выяснять – будет ли это слово служить некоторой МТ. Будем теперь рассматривать код МТ как двоичную запись натурального числа. Данное число назовем номером МТ. Поскольку все коды начинаются с единицы, то разным кодам соответствуют разные числа. Упорядочим МТ по возрастанию чисел, представляемых их кодами, и занумеруем их $T_0, T_1, \dots, T_n, \dots$.

Указанное упорядочение является эффективным в том смысле, что существует алгоритм, который по n выдает *Код* (T_n), и существует алгоритм, который, наоборот, по *Код* (T_n) выдает n. Если обозначить через $f_n(x)$ – одноместную функцию, которая вычисляет машину Тьюринга, то в результате получим нумерацию всех одноместных частично рекурсивных функций: $f_0(x), f_1(x), \dots, f_n(x)$. Каждая одноместная частично рекурсивная функция представлена в этой последовательности. Можно показать, что каждая такая функция представлена в этой последовательности бесконечное число раз. Аналогично можно определить нумерацию n-местных функций.

2. Нумерация МНР

Приведем аналогичную конструкцию по нумерации МНР-программ, которая позволит получить нумерацию МНР-вычислимых функций. Для начала определим нумерацию команд. Предположим:

- $\alpha\left(Z(n)\right) = 4(n-1)$;
- $\alpha\left(S(n)\right) = 4(n-1) - 1$;
- $\alpha\left(T(m,n)\right) = 4\,p(m-1, n-1) + 2$;
- $\alpha\left(J(m,n,q)\right) = 4\,\pi(m,n,q) + 3$.

Здесь $p(x,y) = 2^x(2y+1) - 1$, $\pi(x,y,z) = p\left(p(x-1, y-1), z-1\right)$.

Функция α эффективно вычислима. Для вычисления α^{-1} находят числа u, v: $x = 4u + v, 0 \le v < 4$, имеем:

- $\alpha^{-1}(x) = Z(u+1), v = 0$;
- $\alpha^{-1}(x) = S(u+1), v = 1$;
- $\alpha^{-1}(x) = T(l(u)+1, r(u)+1), v = 2$;
- $\alpha^{-1}(x) = J(m,n,q), v = 3, (m,n,q) = \pi^{-1}(u)$.

Следовательно, функция α^{-1} эффективно вычислима.

Пусть теперь дана МНР- программа $P = I_1, \dots, I_s$. Определим ее номер:

$$\gamma(P) = \tau\left(\alpha(I_1), \dots, \alpha(I_s)\right),$$

где $\tau(x_1, \dots, x_s) = 2^{x_1} + 2^{x_1+x_2+1} + 2^{x_1+x_2+x_3+2} + \dots + 2^{x_1+\dots+x_s+s-1} - 1$.

Ясно, что тем самым определена биекция γ множества программ в множестве натуральных чисел, причем функции γ и γ^{-1} эффективно вычислимы, по программе P

эффективно находится ее номер $n = \gamma(P)$, а по номеру n можно эффективно найти программу P, такую, что $n = \gamma(P)$. Число $\gamma(P)$ называется номером программы P.

Пример 2: если $P = I_1\, I_2\, I_3$, $I_1 = T(3;1)$, $I_2 = S(4)$, $I_3 = Z(6)$, то $\alpha\left(T(1,3)\right) = 18$, $\alpha\left(S(4)\right) = 13$, $\alpha\left(Z(6)\right) = 20$.

Следовательно, $\gamma\,(P) = 2^{18} + 2^{32} + 2^{53} - 1$.

Пусть теперь дано n = 4127. Поскольку $4127 = 2^5 + 2^{12} - 1$, то $P = I_1\, I_2$, где $\alpha\,(I_2) = 12 - 5 - 1 = 6$. Следовательно, $I_1 = S(2)$, $I_2 = T(2;1)$.

Разумеется, существуют и другие способы нумерации программ, для нас в данном случае важна была эффективная вычислимость функций γ и γ^{-1}.

Таким образом, получаем эффективную нумерацию МНР-программ: $P_0, P_1, \dots, P_n, \dots$. Теперь, имея нумерацию МНР-программ, можно занумеровать вычислимые функции. Для любого $\alpha \in N, n \geq 1$ определим ${f_a}^{(n)}$ – n-местная функция, вычислимая программой с номером α. Это дает нумерацию-местных МНР – вычислимых функций ${f_0}^{(n)}, {f_1}^{(n)}, {f_2}^{(n)}, \dots$.

Например, если α = 4127, то имеем:

$$ {f_{4127}}^{(1)}(x) = 1, n = 1, {f_{4127}}^{(1)}(x_1, \dots x_{1n}) = x_2 + 1, n > 1. $$

Поскольку для всякой частично рекурсивной функции $f^{(n)}$ существует вычисляющая ее МНР-программа P, то $f^{(n)} = {f_a}^{(n)}$, где $\alpha = \gamma$. Число α называют номером (индексом) функции $f^{(n)}$.

Задачи для самостоятельной работы

1. На ленте заданы два массива – m и n, $m > n$. Вычислить разность этих массивов (каретка располагается над левой ячейкой правого массива).

2. На ленте заданы два массива. Найти модуль разности длин массивов (каретка располагается над первой ячейкой левого массива).

3. На ленте задан массив. Удвоить массив в два раза (каретка располагается над первой ячейкой массива).

4. На ленте задан массив. Вычислить остаток от деления длины заданного массива на 3 (каретка располагается над первой ячейкой массива).

5. На ленте ПМ расположен массив из n меток. Составить программу, действуя по которой машина выяснит, делится ли число n на 3. Если да, то после массива через одну пустую ячейку поставить метку.

6. На ленте имеется некоторое множество меток (общее количество меток не менее 1). Между метками множества могут быть пропуски, длина которых составляет одну ячейку. Заполнить все пропуски метками.

7. На ленте имеется массив из n отмеченных ячеек (каретка обозревает крайнюю левую метку). Справа от данного массива на расстоянии в m ячеек находится еще одна метка. Составьте для МП программу, придвигающую данный массив к данной ячейке.

8. Известно, что на ленте МП находится метка. Напишите программу, которая находит ее.

9. Дан массив меток (каретка располагается где-то над массивом, но не над крайними метками). Стереть все метки, кроме крайних, и поставить каретку в исходное положение.

10. На ленте МП расположен массив из n меток (метки расположены через пробел). Нужно сжать массив так, чтобы все n меток занимали n расположенных подряд ячеек.

11. Дано несколько массивов меток. Удалить четные массивы (каретка находится над первым массивом).

12. На ленте МП расположено n массивов меток, отделенных друг от друга свободной ячейкой (каретка находится над крайней левой меткой первого массива). Определить количество массивов.

13. На ленте МП расположен массив из $2n - 1$ меток. Составить программу удаления средней метки массива.

14. На ленте МП расположен массив из $2n$ ячеек. Составить программу, по которой МП раздвинет на расстояние в одну ячейку две половины данного массива.

15. На ленте расположены два массива разной длины (каретка обозревает крайний элемент одного из них). Составьте программу для МП, сравнивающую длины массивов и стирающую больший из них. Отдельно продумайте случай, когда длины массивов равны.

16. На ленте МП находятся два массива в m и n меток. Составить программу выяснения, одинаковы ли массивы по длине.

17. Дано n массивов меток. Массивы разделены тремя пустыми ячейками. Количество меток в массиве не меньше двух. Если количество меток в массиве кратно трем, то стереть метки в этом массиве через одну, в противном случае стереть весь массив (каретка находится над крайней левой меткой первого массива).

18. Подготовить сообщение по одной из предложенных тем:

(1) десятая проблема Гильберта;

(2) аналог десятой проблемы Гильберта для уравнений степени 3 (уравнения в рациональных числах);

(3) теорема Гёделя о неполноте символических логик;

(4) теорема Райса;

(5) проблема умирающей матрицы;

(6) проблема единичной матрицы;

(7) проблема свободности матричной полугруппы;

(8) проблема разрешения (Entscheidungsproblem);
(9) выводимость формулы в арифметике Пеано;
(10) проблема соответствий Поста;
(11) идеальный архиватор, создающий для любого входного файла программу наименьшего возможного размера, печатающую этот файл;
(12) идеальный оптимизирующий по размеру компилятор;
(13) функция Радо;
(14) задача поиска усердных бобров или «Busy Beaver»;
(15) NP – полные задачи;
(16) задача Кука;
(17) задачи тысячелетия.

Практические занятия № 26–27 «Контрольная работа»

1. Представить в алфавите $\{0,1\}$ машину Тьюринга, обладающую следующими свойствами (предполагается, что в начальный момент обозревается самый левый символ слова и в качестве пустого слова берется 0):

(1) Машина имеет одно состояние, одну команду и применима к любому слову в алфавите $\{0,1\}$.

(2) Машина имеет две команды, не применима ни к какому слову в алфавите $\{0,1\}$, и зона работы на каждом слове бесконечна.

(3) Машина имеет две команды, не применима ни к какому слову в алфавите $\{0,1\}$, и зона работы на любом слове ограничена одним и тем же числом ячеек, не зависящим от выбора слова.

(4) Машина имеет три команды, применима к словам $10^{2n}1$ $(n \geq 1)$ и не применима к словам $10^{2n+1}1$ $(n \geq 0)$.

(5) Машина имеет пять команд, применима к словам 1^{3n} $(n \geq 1)$ и не применима к словам $1^{3n+\propto}$ $(\propto = 1{,}2 \text{ и } n \geq 0)$.

2. Известна машина Тьюринга с внешним алфавитом $A = \{0,1\}$, алфавитом внутренних состояний $Q = \{g_0, g_1\}$ и программой:

$$g_1 0 \rightarrow g_0 1R$$
$$g_1 1 \rightarrow g_1 1R$$

Определить, в какое слово перерабатывает машина каждое из следующих слов, если в начальном состоянии g_1 обозревается указанная ячейка:

(1) 10110011 (обозревается 4 ячейка);
(2) 11011101 (обозревается 2 ячейка);
(3) 100111 (обозревается 3 ячейка);
(4) 1111011 (обозревается 4 ячейка);
(5) 1101111 (обозревается 3 ячейка);
(6) 1111111 (обозревается 4 ячейка);
(7) 11111 (обозревается 5 ячейка);
(8) 111…1 (k единиц, обозревается k ячейка).

3. Машина Тьюринга задается следующей функциональной схемой (рис. 109):

	g_1	g_2	g_3
0		$g_3 1R$	$g_1 0L$
1	$g_2 0L$	$g_2 1L$	$g_3 1R$
$*$	$g_0 0L$	$g_2 * L$	$g_3 * R$

Рис. 109.

Определите, в какое слово перерабатывает машина каждое из следующих слов, исходя из положения, когда g_1 – крайняя правая единица. После этого постарайтесь усмотреть общую закономерность в работе машины. Придумайте свой алгоритм, решающий эту задачу.

(1) 111∗111, 1111∗11;
(2) 111∗1, 1∗11;
(3) 11∗111, 111∗11;
(4) 11111∗, ∗11111.

4. Машина Тьюринга задается следующей функциональной схемой (рис.110):

	g_1	g_2	g_3	g_4
0	$g_4 0R$	$g_3 0L$	$g_1 0R$	$g_0 0L$
1	$g_2 \propto E$	$g_1 \beta E$	$g_1 1R$	$g_1 1L$
$\propto$	$g_1 \propto L$	$g_2 \propto R$	$g_3 1L$	$g_4 0R$
β	$g_1 \beta L$	$g_2 \beta R$	$g_3 0L$	$g_4 1R$

Рис. 110.

Определите, в какое слово перерабатывает машина каждое из следующих слов, исходя из начального положения g_1:

(1) 11111 (2 ячейка, считая слева);
(2) 111 (1 ячейка, считая слева);
(3) 1111111111 (4 ячейка, считая слева);
(4) 111111 (2 ячейка, считая слева);
(5) 1^{15} (6 ячейка, считая слева).

5. Остановится ли когда-нибудь машина Тьюринга, заданная следующей программой (рис. 111):

	g_1	g_2	g_3
0	$g_1 0R$	$g_3 0L$	$g_0 0E$
1	$g_2 1R$	$g_1 0R$	$g_2 1L$

Рис. 111.

Если она начнет перерабатывать следующие слова:

(1) 111101;
(2) 11111;
(3) 10101.

6. Остановится ли когда-нибудь машина Тьюринга, заданная следующей программой (рис. 112):

	g_1	g_2	g_3	g_4
0	$g_2 0R$	$g_3 0L$	$g_1 1L$	$g_0 0E$
1	$g_2 1R$	$g_4 0R$	$g_0 1E$	$g_2 0R$

Рис. 112.

Если она начнет перерабатывать следующие слова:

(1) 1110101;
(2) 1111;
(3) 1010101.

7. Известно, что на ленте записано слово из n единиц 11…1, $n \geq 1$. Постройте машину Тьюринга с внешним алфавитом $A = \{0,1\}$, которая отыскивала бы левую единицу этого слова, если в начальный момент головка машины обозревает одну из ячеек с буквой данного слова.

8. Сконструируйте машину Тьюринга с внешним алфавитом $A = \{0,1\}$, которая каждое слово в алфавите $A_1 = \{1\}$ перерабатывает в пустое слово.

9. На ленте машины Тьюринга записаны два набора единиц. Они разделены *. Составьте функциональную схему машины так, чтобы она, исходя из начального положения g_1 – крайняя правая единица, выбрала больший из этих наборов, а меньший стерла. Звездочка должна быть сохранена, чтобы было видно, какой из наборов выбран. Рассмотрите примеры работы этой машины применительно к словам:

(1) 1*11;
(2) 11*1;
(3) 11*111;
(4) 111*11;
(5) 11*1111;
(6) 1111*11.

10. Написать формулу числовой функции $f(x_1, x_2, \ldots x_n)$, вычислимой машиной Тьюринга с множеством внутренних состояний $\{g_0, g_1, \ldots g_6\}$. Проверить работу с некоторым набором значений аргумента (рис. 113).

№	n		g_1	g_2	g_3	g_4	g_5	g_6
1	2	λ		$g_5 1R$	$g_4 \lambda R$	$g_0 \lambda E$		$g_0 \lambda E$
		1	$g_2 1R$	$g_3 \lambda R$	$g_3 \lambda R$	$g_4 \lambda R$	$g_6 1R$	$g_6 \lambda R$

№	n		g_1	g_2	g_3	g_4	g_5	g_6
2	2	λ	$g_4 1R$	$g_3 \lambda L$	$g_0 \lambda E$	$g_5 1L$	$g_6 1R$	$g_0 1E$
		1	$g_2 1R$	$g_1 1R$	$g_3 \lambda L$	$g_4 1R$	$g_5 1L$	$g_6 1R$

Рис. 113.

11. НАМ в алфавите A = {a, b}, задается схемой $S:\begin{cases} ba \to ab \\ ab \to \lambda \end{cases}$. Примените его к слову bbabab.

12. НАМ в алфавите A = {a, b}, задается схемой $S:\begin{cases} ab \to a \\ b \to \lambda \\ a \longrightarrow b \end{cases}$. Примените его к слову bbaab.

13. НАМ в алфавите A = {a, b}, задается схемой $S:\begin{cases} ab \to a \\ b \to \lambda \\ a \longrightarrow bb \end{cases}$. Примените его к слову baab.

14. Построить НАМ, подсчитывающий количество букв «b» в слове, записанном в алфавите A = {a,b} и содержащем более одной буквы «b», иначе заменить исходное слово на «bbb».

15. Построить НАМ: перенести первый символ непустого слова R в конец слова в алфавите A = {a, b}.

16. Построить НАМ: перенести последний символ непустого слова R в начало слова в алфавите A = {a, b}.

17. Написать нормальную схему алгоритма для функций:

(1) f(x)=$\begin{cases} 1, \text{ если x делится на 2} \\ 0, \text{ если x не делится на 2} \end{cases}$

(2) $f(x,y) = \frac{x}{2} + y$

(3) $f(x,y,z) = z$

(4) $f(x) = 2^{1-x}$

18. Доказать, что следующая функция примитивно рекурсивна:

(1) $f(x) = x^2 + 3$

(2) $f(x,y) = x^2 + 2y^2$

(3) $f(x) = (x+2)^2$

19. Применяя операцию примитивной рекурсии к функциям g(x) и h(x,y,z) по переменной y, построить функцию $f(x,y) = R(g,h)$, записав ее в «аналитической» форме:

(1) $g(x) = x^2$, h(x,y,z) = xz

(2) g(x) = x, h(x,y,z) = x+y-z

(3) g(x) = x, h(x,y,z) = x+z.

20. Применить операцию минимизации к функции:

(1) $f(x) = 3$

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

(3) $f(x)$ = x−3

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Интуитивное понятие алгоритма.

2. Характерные черты алгоритма (дискретность, детерминированность, элементарность).

3. Характерные черты алгоритма (массовость, реализуемость, результативность).

4. Конструктивный объект.

5. Виды алгоритмов.

6. Типы частных алгоритмов.

7. Формы записи алгоритма.

8. Формализация понятия алгоритма.

9. Современное состояние теории алгоритмов.

10. Понятие вычислимой функции.

11. Разрешимые множества.

12. Перечислимые множества.

13. Алгоритм Дейкстры.

14. Алгоритм Крускала.

15. Алгоритм Прима.

16. Поиск в глубину.

17. Поиск в ширину.

18. Алгоритмы сортировки.

19. Алгоритмы слияния.

20. Сжатие без потерь.

21. Сжатие с потерями.

22. Алгоритм разделения секрета.

23. Описание машины Тьюринга.

24. Принцип работы машины Тьюринга.

25. Конструирование машины Тьюринга.

26. Вычислимые по Тьюрингу функции.

27. Операции над машинами Тьюринга.

28. Тезис Тьюринга.

29. Конечные автоматы.

30. Машина с неограниченными регистрами.

31. Машина Поста.

32. Происхождение рекурсивных функций.

33. Операция суперпозиции.

34. Операция примитивной рекурсии.

35. Операция минимизации.

36. Виды рекурсивных функций.

37. Тезис Чёрча.

38. Универсальная функция.

39. Марковские подстановки.

40. Нормальные алгорифмы и их применение к словам.

41. Нормально вычислимые функции.

42. Принцип нормализации Маркова.

43. Основные способы композиции нормальных алгоритмов (суперпозиция, объединение).

44. Основные способы композиции нормальных алгоритмов (разветвление, итерация).

45. Эквивалентность различных теорий алгоритмов.

46. Алгоритмически неразрешимые проблемы.

47. Нумерация алгоритмов.

48. Элементы теории сложности вычислений.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов, С. А. Лекции о сложности алгоритмов / С. А. Абрамов. – М. : изд-во МЦНМО, 2009. – 256 с.

2. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений / В. Е. Алексеев, В. А. Таланов. – М. : Бином, 2006. – 320 с.

3. Верещагин, Н. К. Лекции по математической логике и теории алгоритмов. – Ч. 3 : Вычислимые функции / Н. К. Верещагин, А. Шень. – 3-е изд., стереотип. – М. : МЦНМО, 2008. – 192 с.

4. Гамова, А. Н. Математическая логика и теория алгоритмов / А. Н. Гамова. – Саратов : изд-во СГУ, 1999. – 76 с.

5. Ершов, С. С. Элементы теории алгоритмов : учебное пособие / С. С. Ершов. – Челябинск : изд. центр ЮУрГУ, 2009. – 64 с.

6. Ершов, Ю. Л. Математическая логика / Ю. Л. Ершов, Е. А. Палютин. – М. : Наука, 1987. – 336 с.

7. Игошин, В. И. Задачи и упражнения по математической логике и теории алгоритмов : учеб. пособие для студ. высш. учеб. заведений / В. И. Игошин. – М. : изд. центр «Академия», 2006. – 304 с.

8. Игошин, В. И. Математическая логика и теория алгоритмов : учеб. пособие для студ. высш. учеб. заведений / В. И. Игошин. – М. : изд. центр «Академия», 2004. – 448 с.

9. Кормен, Т. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест [и др.]. – М. : Вильямс, 2012. – 1296 с.

10. Крупский, В. Н. Теория алгоритмов : учеб. пособие для студ. вузов / В. Н. Крупский, В. Е. Плиско. – М. : изд. центр «Академия», 2009. – 208 с.

11. Кузюрин, Н. Н. Эффективные алгоритмы и сложность вычислений / Н. Н. Кузюрин, С. А. Фомин. – М., 2011. – 363 с.

12. Лавров, И. А. Задачи по теории множеств, математической логике и теории алгоритмов / И. А. Лавров, Л. Л. Максимова. – М. : ФИЗМАТЛИТ, 2009. – 256 с.

13. Лихтарников, Л. М. Математическая логика. Курс лекций. Задачник-практикум и решения / Л. М. Лихтарников, Т. Г. Сукачева. – СПб. : Лань, 1999. – 288 с.

14. Мальцев, А. И. Алгоритмы и рекурсивные функции / А. И. Мальцев. – М. : Наука, 1965. – 392 с.

15. Пильщиков, В. Н. Машина Тьюринга и алгоритмы Маркова. Решение задач : учебно-методическое пособие / В. Н. Пильщиков, В. Г. Абрамов, А. А. Вылиток [и др.]. – М. : МГУ, 2006. – 47 с.

16. Прокопенко, Н. Ю. Элементы теории алгоритмов / Н. Ю. Прокопенко. – Южно-Сахалинск : СахГУ, 2006. – 108 с.

17. Редькин, Н. П. Дискретная математика / Н. П. Редькин. – М. : ФИЗМАТЛИТ, 2009. – 264 с.

18. Стивенс, Р. Delfi. Готовые алгоритмы / Р. Стивенс. – М. : Пресс, 2001. – 384 с.

19. Тишин, В. В. Дискретная математика в примерах и задачах / В. В. Тишин. – СПб. : БХВ-Петербург, 2009. – 352 с.

20. Фалевич, Б. Я. Теория алгоритмов / Б. Я. Фалевич. – М. : Машиностроение, 2004. – 160 с.

21. Шауцукова, Л. З. Информатика 10–11 / Л. З. Шауцукова. – М. : Просвещение, 2000. – Режим доступа : http://book.kbsu.ru/theory/chapter7/1_7_0.html.

22. Шиханович, Ю. А. Минимум по теории алгоритмов для нематематиков / Ю. А. Шиханович. – М. : Научный мир, 2009. – 160 с.

Электронные ресурсы:

1. Алгоритм ближайшего соседа. – Режим доступа : http://www.basegroup.ru/library/ analysis/ regression/knn/

2. Алгоритм поиска минимального остовного дерева. – Режим доступа : http:// www.algolib.narod.ru/Graph/Ostov.html

3. Библиотека алгоритмов на графах. Теория алгоритмов – примеры – задачи. – Режим доступа : http://urban-sanjoo.narod.ru/flows.html

4. Викентьева, О. Л. Математическая логика и теория алгоритмов, конспект лекций для студентов специальностей АСУ, ЭВТ, КЗИ. – Режим доступа : http://rudocs.exdat.com/ docs/index-68239.html?page=5

5. Вычислительная сложность. – Режим доступа : https://ru.wikipedia.org/ wiki/ %D0 %92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D 1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE %D1%81%D1%82%D1%8C

6. Бекман, Игорь. Компьютерные науки. Курс лекций. Лекция № 7. Алгоритмы. – Режим доступа : http://profbeckman.narod.ru/Komp.files/Lec1.pdf

7. Маркин, П. М. Дискретная математика. Теория алгоритмов. – Режим доступа : http:// share.auditory.ru/2013/Anton.Kuzmishchev/miem/4%20%D0%A1%D0%B5%D0%BC%D0%B5% D1%81%D1%82%D1%80/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F%20%D0% 90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2/teory a_algoritmov.pdf

8. Носов, В. А. Основы теории алгоритмов и анализа их сложности, курс лекций. – Режим доступа : http://www.ict.edu.ru/ft/003139/theoralg.pdf

9. Плиско, В. Е. Теория алгоритмов. – Режим доступа : http:// denis. kraynov.2009. homepage.auditory.ru/2011/Roman.Komkov/Study/4sem/Mat.logika/ta.pdf

10. Проект «Теория алгоритмов» по курсу «Практикум на ЭВМ» выполнен Ф. Еремеевым, рук. проекта С. В. Колосков. – М. : МГТУ им. Н. Э. Баума, 2004 г. – Режим доступа : http://th-algoritmov.narod.ru/3.htm

11. Список алгоритмов. – Режим доступа : https://ru.wikipedia.org/wiki/%D0% A1%D0 %BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80 %D0%B8%D1%82%D0%BC%D0%BE%D0%B2

12. Функция Радо, или Busy Beaver. – Режим доступа : http://falcao. livejournal. com/ 26513.html

Учебное издание

ИЗБРАННЫЕ ВОПРОСЫ ТЕОРИИ АЛГОРИТМОВ

Учебно-методическое пособие

Составители:
МЕРКУЛОВА Ольга Олеговна,
НИКИТИНА Алла Борисовна,
ФЁДОРОВ Олег Анатольевич

Корректор В. А. Яковлева.
Верстка Г. С. Лосева.

Подписано в печать 25.12.2018. Бумага «Mondi».
Гарнитура «Times New Roman». Формат 60x84$^{1}/_{8}$.
Тираж 500 экз. (1-й завод 1–100 экз.).
Объем 13,02 усл. п. л. Заказ № 877-18.

Сахалинский государственный университет.
693008, Южно-Сахалинск, ул. Ленина, 290, каб. 32.
Тел. (4242) 45-23-16, факс (4242) 45-23-17.
E-mail: polygraph@sakhgu.ru,
izdatelstvo@sakhgu.ru