МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»
ИНСТИТУТ ПРИБОРОСТРОЕНИЯ, АВТОМАТИЗАЦИИ
И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

В.А. Лобанова, О.А. Воронина, Н.Г. Лобанова

ТЕОРИЯ АЛГОРИТМОВ

Орёл
ОГУ имени И.С. Тургенева
2017

УДК 519.712(075)
ББК 22.12я7
Л68

Печатается по решению
редакционно-издательского совета
ОГУ имени И.С. Тургенева.
Протокол № 6 от 22.02.2017 г.

Рецензенты:

доктор технических наук,
профессор кафедры информационных систем
федерального государственного бюджетного
образовательного учреждения высшего образования
«Орловский государственный университет имени И.С. Тургенева»
В.И. Раков,

кандидат технических наук, доцент,
заведующий кафедрой систем информационной безопасности
федерального государственного бюджетного
образовательного учреждения высшего образования
«Брянский государственный технический университет»
М.Ю. Рытов

Лобанова, В.А.

Л68 Теория алгоритмов: учебное пособие / В.А. Лобанова, О.А. Воронина, Н.Г. Лобанова. – Орёл: ОГУ имени И.С. Тургенева, 2017. – 95 с.

ISBN 978-5-9929-0496-3

В учебном пособии рассмотрены основы таких разделов теории алгоритмов, как классическая теория алгоритмов (машина Поста, машина Тьюринга, алгоритмически неразрешимые задачи), асимптотический анализ сложности алгоритмов, сложностные классы и практический сравнительный анализ вычислительных алгоритмов.

Предназначено студентам очной формы обучения, обучающимся по направлениям «Информационная безопасность» и «Конструирование и технология ЭС», а также может быть полезно для подготовки студентов различных специальностей, изучающих математическую логику и теорию алгоритмов.

УДК 519.712(075)
ББК 22.12я7

ISBN 978-5-9929-0496-3

СОДЕРЖАНИЕ

Введение 4
1. Исторический обзор алгоритмов 5
1.1. Определение алгоритма 5
1.2. Машина Поста 8
Вопросы для самоконтроля 11
1.3. Машина Тьюринга и алгоритмически неразрешимые проблемы 12
Вопросы для самоконтроля 17
2. Введение в анализ алгоритмов 19
2.1. Сравнительные оценки алгоритмов 19
2.2. Граф-машина 30
2.3. Модель данных 31
2.4. Сложность алгоритма 34
3. Алгоритмы сортировки 41
3.1. Сортировка и поиск 41
3.2. Сортировка всплытия Флойда 44
3.3. Задачи поиска 51
3.4. Сортировка с вычисляемыми адресами 54
3.5. Эффективность методов оптимизации 56
4. Алгоритмы машинной математики 64
4.1. Сортировка включением. 64
4.2. Обменная сортировка. 66
4.3. Сортировка выбором 69
4.4. Сортировка разделением (Quicksort) 69
4.5. Сортировка с помощью дерева (Heapsort) 70
4.6. Сортировка со слиянием 78
4.7. Сравнение методов внутренней сортировки. 80
4.8. Задания для выполнения. 80
5. Алгоритмы машинной графики 82
5.1. Краткие теоретические сведения 82
5.2. Задания для выполнения 90
Литература 93

ВВЕДЕНИЕ

Для решения типовых задач в математике используются определенные правила, описывающие последовательности действий. Например, правила сложения дробных чисел, решения квадратных уравнений и т. д. Обычно любые инструкции и правила представляют собой последовательность действий, которые необходимо выполнить в определенном порядке. Для решения задачи надо знать, **что дано, что следует получить, какие действия и в каком порядке следует для этого выполнить.** Предписание, определяющее порядок выполнения действий над данными с целью получения искомых результатов, и есть алгоритм.

Алгоритм – заранее заданное понятное и точное предписание возможному исполнителю совершить определенную последовательность действий для получения решения задачи за конечное число шагов.

Теория алгоритмов – это наука, изучающая общие свойства и закономерности алгоритмов, разнообразные формальные модели их представления. На основе формализации понятия алгоритма возможно сравнение алгоритмов по их эффективности, проверка их эквивалентности, определение областей применимости.

Разработанные в 1930-х годах разнообразные формальные модели алгоритмов (Пост, Тьюринг, Черч), равно как и предложенные в 1950-х годах модели Колмогорова и Маркова, оказались эквивалентными в том смысле, что любой класс проблем, разрешимых в одной модели, разрешим и в другой.

В настоящее время полученные на основе теории алгоритмов практические рекомендации находят всё большее распространение в области проектирования и разработки программных систем. В связи с этим в государственный стандарт введена специальная дисциплина – «Математическая логика и теория алгоритмов». Описание государственного стандарта регламентирует включение в состав дисциплины ряда понятий и методов теории алгоритмов, которые и отражены в настоящем учебном пособии.

1. ИСТОРИЧЕСКИЙ ОБЗОР АЛГОРИТМОВ

1.1. Определение алгоритма

Первым дошедшим до нас алгоритмом в его интуитивном понимании – конечной последовательности элементарных действий, решающих поставленную задачу, считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел (алгоритм Евклида). Отметим, что в течение длительного времени, вплоть до начала XX века, само слово «алгоритм» употреблялось в устойчивом сочетании «алгоритм Евклида». Для описания пошагового решения других математических задач использовалось слово «метод».

Начальной точкой отсчета современной теории алгоритмов можно считать работу немецкого математика Курта Гёделя (1931 г. – теорема о неполноте символических логик), в которой было показано, что некоторые математические проблемы не могут быть решены алгоритмами из некоторого класса. Общность результата Геделя связана с тем, совпадает ли использованный им класс алгоритмов с классом всех (в интуитивном смысле) алгоритмов. Эта работа дала толчок к поиску и анализу различных формализаций алгоритма.

Первые фундаментальные работы по теории алгоритмов были независимо опубликованы в 1936 году Аланом Тьюрингом, Алоизом Черчем и Эмилем Постом. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление Черча были эквивалентными формализмами алгоритма. Сформулированные ими тезисы (Поста и Черча – Тьюринга) постулировали эквивалентность предложенных ими формальных систем и интуитивного понятия алгоритма. Важным развитием этих работ стала формулировка и доказательство алгоритмически неразрешимых проблем.

В 1950-е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова.

К 1960-70-м годам оформились следующие направления в теории алгоритмов:

- Классическая теория алгоритмов (формулировка задач в терминах формальных языков, понятие задачи разрешения, введение сложностных классов, формулировка в 1965 году Эдмондсом проблемы PNP, открытие класса NP-полных задач и его исследование);

- Теория асимптотического анализа алгоритмов (понятие сложности и трудоёмкости алгоритма, критерии оценки алгоритмов, методы получения асимптотических оценок, в частности, для рекурсивных алгоритмов, асимптотический анализ трудоемкости или времени выполнения), в развитие которой внесли существенный вклад Кнут, Ахо, Хопкрофт, Ульман, Карп;
- Теория практического анализа вычислительных алгоритмов (получение явных функций трудоёмкости, интервальный анализ функций, практические критерии качества алгоритмов, методика выбора рациональных алгоритмов), основополагающей работой в этом направлении, очевидно, следует считать фундаментальный труд Д. Кнута «Искусство программирования для ЭВМ».

Цели и задачи теории алгоритмов

Обобщая результаты различных разделов теории алгоритмов, можно выделить следующие цели и соотнесенные с ними задачи, решаемые в теории алгоритмов:

- формализация понятия «алгоритм» и исследование формальных алгоритмических систем;
- формальное доказательство алгоритмической неразрешимости ряда задач;
- классификация задач, определение и исследование сложностных классов;
- асимптотический анализ сложности алгоритмов;
- исследование и анализ рекурсивных алгоритмов;
- получение явных функций трудоемкости в целях сравнительного анализа алгоритмов;
- разработка критериев сравнительной оценки качества алгоритмов.

Практическое применение результатов теории алгоритмов

Полученные в теории алгоритмов теоретические результаты находят достаточно широкое практическое применение, при этом можно выделить следующие два аспекта:

Теоретический аспект: при исследовании некоторой задачи результаты теории алгоритмов позволяют ответить на вопрос: является ли эта задача в принципе алгоритмически разрешимой – для алгоритмически неразрешимых задач возможно их сведение к задаче останова машины Тьюринга. В случае алгоритмической разрешимости задачи следующий важный теоретический вопрос – это вопрос о принадлежности этой задачи к классу NP–полных задач, при утвердитель-

ном ответе на который можно говорить о существенных временных затратах для получения точного решения для больших размерностей исходных данных.

Практический аспект: методы и методики теории алгоритмов (в основном разделов асимптотического и практического анализа) позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения данной задачи с учетом особенностей их применения (например, при ограничениях на размерность исходных данных или объем дополнительной памяти);
- получение временных оценок решения сложных задач;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время, что важно для криптографических методов;
- разработку и совершенствование эффективных алгоритмов решения задач в области обработки информации на основе практического анализа.

Формализация понятия алгоритма

Во всех сферах своей деятельности, в частности, в сфере обработки информации, человек сталкивается с различными способами или методиками решения задач. Они определяют порядок выполнения действий для получения желаемого результата – мы можем трактовать это как первоначальное, или интуитивное, определение алгоритма. Некоторые дополнительные требования приводят к неформальному определению алгоритма:

Определение 1.1. Алгоритм – это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых элементарных операций для решения задачи, общее для класса возможных исходных данных.

Пусть D – область (множество) исходных данных задачи, а R – множество возможных результатов, тогда можно говорить, что алгоритм осуществляет отображение D --> R. Поскольку такое отображение может быть неполным, то вводятся следующие понятия:

Алгоритм называется частичным алгоритмом, если мы получаем результат только для некоторых d є D, и полным алгоритмом, если алгоритм получает правильный результат для всех d є D.

Несмотря на усилия исследователей, отсутствует одно исчерпывающе строгое определение понятия «алгоритм», в теории алгоритмов были введены различные формальные определения алгоритма

и удивительным научным результатом является доказательство эквивалентности этих формальных определений в смысле их равномощности.

Варианты словесного определения алгоритма принадлежат российским ученым А.Н. Колмогорову и А.А. Маркову

Определение 1.2. (Колмогоров): Алгоритм – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 1.3. (Марков): Алгоритм – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Отметим, что различные определения алгоритма в явной или неявной форме постулируют следующий ряд требований:

- алгоритм должен содержать конечное количество элементарно выполнимых предписаний, т.е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т.е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т.е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т.е. удовлетворять требованию правильности.

Другие формальные определения понятия алгоритма связаны с введением специальных математических конструкций (машина Поста, машина Тьюринга, рекурсивно-вычислимые функции Черча) и постулированием тезиса об эквивалентности такого формализма и понятия «алгоритм».

1.2. Машина Поста

Основные понятия и операции

Одной из фундаментальных статей, результаты которой лежат в основе современной теории алгоритмов, является статья Эмиля Поста (Emil Post) «Финитные комбинаторные процессы, формулировка 1», опубликованная в 1936 году в сентябрьском номере «Журнала символической логики».

Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решение общей проблемы – это такое решение, которое доставляет ответ для каждой конкретной проблемы.

Например, решение уравнения 3*x + 9 = 0 – это одна из конкретных проблем, а решение уравнения a*x + b = 0 – это общая проблема, тем самым алгоритм (сам термин «алгоритм» не используется Постом) должен быть универсальным, т.е. должен быть соотнесен с общей проблемой.

Основные понятия алгоритмического формализма Поста – это пространство символов (язык L), в котором задаётся конкретная проблема и получается ответ, и набор инструкций, т.е. операций в пространстве символов, задающих как сами операции, так и порядок выполнения инструкций.

Постовское пространство символов –
это бесконечная лента ячеек (ящиков):

_	V	_	_	V	V	V	_	V

Каждый ящик или ячейка могут быть помечены или не помечены.

Конкретная проблема задается «внешней силой» (термин Поста) пометкой конечного количества ячеек, при этом очевидно, что любая конфигурация начинается и заканчивается помеченным ящиком. После применения к конкретной проблеме некоторого набора инструкций решение представляется так же в виде набора помеченных и непомеченных ящиков, распознаваемое той же внешней силой.

Пост предложил набор инструкций (элементарных операций), которые выполняет «работник». Отметим, что в 1936 году не было еще ни одной электронной вычислительной машины. Этот набор инструкций является, очевидно, минимальным набором битовых операций:

1) пометить ящик, если он пуст;

2) стереть метку, если она есть;

3) переместиться влево на 1 ящик;

4) переместиться вправо на 1 ящик;

5) определить помечен ящик или нет, и по результату перейти на одну из двух указанных инструкций;

6) остановиться.

Отметим, что формулировка инструкций 1 и 2 включает защиту от неправильных ситуаций.

Программа представляет собой нумерованную последовательность инструкций, причем переходы в инструкции 5 производятся на указанные в ней номера других инструкций.

Финитный 1-й – процесс

Программа (набор инструкций в терминах Поста) является одной и той же для всех конкретных проблем, поэтому соотнесена с общей проблемой – таким образом Пост формулирует требование универсальности.

Далее Пост вводит следующие понятия:

- набор инструкций *применим* к общей проблеме, если для каждой конкретной проблемы не возникает коллизий в инструкциях 1 и 2, т.е. никогда программа не стирает метку в пустом ящике и не устанавливает метку в помеченном ящике;
- набор инструкций *заканчивается* (за конечное количество инструкций), если выполняется инструкция (6);
- набор инструкций задаёт *финитный 1 – процесс*, если набор применим, и заканчивается для каждой конкретной проблемы;
- финитный 1-й – процесс для общей проблемы есть 1 – решение, если ответ для каждой конкретной проблемы правильный (это определяется внешней силой).

Способ задания проблемы и формулировка 1

По Посту, проблема задаётся внешней силой путем пометки конечного количества ящиков ленты. В более поздних работах по машине Поста принято считать, что машина работает в единичной системе счисления (0 = V; 1 = VV; 2 = VVV; 3 = VVVV), т.е. ноль представляется одним помеченным ящиком, а целое положительное число – помеченными ящиками в количестве, на единицу больше его значения.

Поскольку множество конкретных проблем, составляющих общую проблему, счетное, то можно установить взаимно однозначное соответствие (биективное отображение) между множеством положительных целых чисел N и множеством конкретных проблем.

Общая проблема называется, по Посту, *1-заданой*, если существует такой финитный 1 – процесс, что будучи применим к n є N в качестве исходной конфигурации ящиков, он задает *n-ю* конкретную проблему в виде набора помеченных ящиков.

Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы и ее решению, получаем ответ по номеру проблемы – это и есть в терминах статьи Поста *формулировка 1*.

Эмиль Пост завершает свою статью следующей фразой: «Автор ожидает, что его формулировка окажется логически эквивалентной рекурсивности в смысле Геделя – Черча. Цель формулировки, однако, в том, чтобы предложить систему не только определенной логической силы, но и психологической достоверности. В этом последнем смысле подлежат рассмотрению всё более и более широкие формулировки. С другой стороны, нашей целью будет показать, что все они логически сводимы к формулировке 1. В настоящий момент мы выдвигаем это умозаключение в качестве *рабочей гипотезы*. … Успех вышеизложенной программы заключался бы для нас в превращении этой гипотезы не столько в определение или аксиому, сколько в закон природы».

Таким образом, гипотеза Поста состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представления и интерпретации конкретных проблем сводимы к формулировке 1.

АЛГОРИТМ	очевидно <------- гипотеза ------->	ПРОГРАММА ПОСТА ФОРМУЛИРОВКА 1

Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1 Эмиля Поста.

Обоснование этой гипотезы происходит сегодня не путем строго математического доказательства, а на пути эксперимента – действительно, всякий раз, когда нам указывают алгоритм, его можно перевести в форму программы машины Поста, приводящей к тому же результату.

Вопросы для самоконтроля

1. Понятие общей и конкретной проблем по Посту.
2. Пространство символов и примитивные операции в машине Поста.
3. Понятие финитного 1-го-процесса в машине Поста.
4. Способы задания проблем и формулировка 1.
5. Гипотеза Поста.

1.3. Машина Тьюринга и алгоритмически неразрешимые проблемы

Машина Тьюринга. Алан Тьюринг (Turing) в 1936 году в трудах Лондонского математического общества опубликовал статью «О вычислимых числах в приложении к проблеме разрешения», которая наравне с работами Поста и Черча лежит в основе современной теории алгоритмов.

Предыстория создания этой работы связана с формулировкой Давидом Гильбертом на Международном математическом конгрессе в Париже в 1900 году неразрешенных математических проблем. Одной из них была задача доказательства непротиворечивости системы аксиом обычной арифметики, которую Гильберт в дальнейшем уточнил как «проблему разрешимости» – нахождение общего метода для определения выполнимости данного высказывания на языке формальной логики.

Статья Тьюринга как раз и давала ответ на эту проблему – вторая проблема Гильберта оказалась неразрешимой. Но значение статьи Тьюринга выходило далеко за рамки той задачи, по поводу которой она была написана.

Приведем характеристику этой работы, принадлежащую Джону Хопкрофту: «Работая над проблемой Гильберта, Тьюрингу пришлось дать четкое определение самого понятия метода. Отталкиваясь от интуитивного представления о методе как о некоем алгоритме, т.е. процедуре, которая может быть выполнена механически, без творческого вмешательства, он показал, как эту идею можно воплотить в виде подробной модели вычислительного процесса. Полученная модель вычислений, в которой каждый алгоритм разбивался на последовательность простых, элементарных шагов, и была логической конструкцией, названной впоследствии машиной Тьюринга».

Машина Тьюринга является расширением модели конечного автомата, расширением, включающим потенциально бесконечную память с возможностью перехода (движения) от обозреваемой в данный момент ячейки к ее левому или правому соседу.

Формально машина Тьюринга может быть описана следующим образом. Пусть заданы:

- конечное множество состояний – Q, в которых может находиться машина Тьюринга;
- конечное множество символов ленты – Г;

- функция δ (функция переходов или программа), которая задается отображением пары из декартова произведения Q x Г (машина находится в состоянии q_i и обозревает символ γ_i) в тройку декартова произведения Q x Г x {L,R} (машина переходит в состояние q_i, заменяет символ γ_i на символ γ_j и передвигается влево или вправо на один символ ленты) – Q x Г-->Q x Г x {L,R};
- один символ из Г-->e (пустой);
- подмножество Σ є Г --> определяется как подмножество входных символов ленты, причем e є (Г-Σ);
- одно из состояний – q_0 є Q является начальным состоянием машины.

Решаемая проблема задается путем записи конечного количества символов из множества Σ є Г – Si є Σ на ленту:							
e	S1	S2	S3	S4	.....	Sn	e

после чего машина переводится в начальное состояние и головка устанавливается у самого левого непустого символа – $(q_0 \uparrow \omega)$, после чего в соответствии с указанной функцией переходов (q_i, S_i)-->(q_j, S_k, L или R) машина начинает заменять обозреваемые символы, передвигать головку вправо или влево и переходить в другие состояния, предписанные функций переходов.

Остановка машины происходит в том случае, если для пары (q_i, S_i) функция перехода не определена.

Алан Тьюринг высказал предположение, что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной Тьюринга. Это предположение известно как тезис Черча – Тьюринга. Каждый компьютер может моделировать машину Тьюринга (операции перезаписи ячеек, сравнения и перехода к другой соседней ячейке с учетом изменения состояния машины). Следовательно, он может моделировать алгоритмы в любом формализме, и из этого тезиса следует, что все компьютеры (независимо от мощности, архитектуры и т.д.) эквивалентны с точки зрения принципиальной возможности решения алгоритмических задач.

Алгоритмически неразрешимые проблемы

За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и науч-

ных проблем. Зададимся вопросом: а существуют ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения?

Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным – констатируем, что не только сейчас не знаем соответствующего алгоритма, но и 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 в данный момент.

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 неразрешимых проблем был значительно расширен. Сегодня принято при доказательстве алгоритмической неразрешимости некоторой задачи сводить ее к ставшей классической задаче – «задаче останова».

Имеет место быть следующая теорема:

Теорема 1. Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм, и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, т.е. 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:

а) отсутствие общего метода решения задачи:

Проблема 1. Распределение девяток в записи числа π.

Определим функцию f(n) = i, где n – количество девяток подряд в десятичной записи числа π, а i – номер самой левой девятки из n девяток подряд: π=3,141592… f(1) = 5.

Задача состоит в вычислении функции f(n) для произвольно заданного n.

Поскольку число π является иррациональным и трансцендентным, то нет 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 не найдены n девяток подряд, однако общего метода вычисления f(n) нет, поэтому для некоторых n вычисления могут продолжаться бесконечно – неизвестно в принципе (по природе числа π), существует ли решение для всех n.

Проблема 2. Вычисление совершенных чисел.

Совершенные числа – это числа, которые равны сумме своих делителей, например: 28 = 1+2+4+7+14.

Определим функцию S(n) = n-е по счёту совершенное число и поставим задачу вычисления S(n) по произвольно заданному n. Нет общего метода вычисления совершенных чисел, неизвестно, 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 проверено M чисел при поиске n-го совершенного числа – означает ли это, что его вообще не существует?

Проблема 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, т.е. отсутствует общий метод определения целых корней уравнения P = 0 по его целочисленным коэффициентам;

б) информационная неопределенность задачи:

Проблема 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: когда после обнаружения конца кортежа сделан сдвиг вправо по пустым ящикам на M позиций и не обнаружено начало следующего кортежа – 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;

в) логическая неразрешимость (в смысле теоремы Гёделя о неполноте)

Проблема 5. Проблема «останова»;

Проблема 6. Проблема эквивалентности алгоритмов.

По двум произвольным заданным алгоритмам (например, по двум машинам Тьюринга) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Проблема 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 определённым?

Проблема соответствий Поста над алфавитом Σ.

В качестве более подробного примера алгоритмически неразрешимой задачи рассмотрим проблему соответствий Поста (Э. Пост, 1943 г.). Выделим эту задачу, поскольку на первый взгляд она выглядит достаточно «алгоритмизуемой», однако она сводима к проблеме останова и является алгоритмически неразрешимой.

Постановка задачи:

Пусть дан алфавит Σ: $|\Sigma| >= 2$ (для односимвольного алфавита задача имеет решение) и дано конечное множество пар из $\Sigma^+ x \Sigma^+$, т.е. пары непустых цепочек произвольного языка над алфавитом Σ: (x_1, y_1), ……, (x_m, y_m).

Проблема: выяснить, существует ли конечная последовательность этих пар, не обязательно различных, такая, что цепочка, составленная из левых подцепочек, совпадает с последовательностью правых подцепочек – такая последовательность называется решающей.

В качестве примера рассмотрим Σ = {a,b}:

1. Входные цепочки: (abbb, b), (a, aab), (ba, b)

Решающая последовательность для этой задачи имеет вид:

(a,aab) (a,aab) (ba,b) (abbb,b), так как : a a ba abbb ≡ aab aab b b

2. Входные цепочки: (ab,aba), (aba,baa), (baa,aa)

Данная задача вообще не имеет решения, так как нельзя начинать с пары *(aba,baa)* или *(baa,aa)*, поскольку не совпадают начальные символы подцепочек, но если начинать с цепочки *(ab,aba)*, то в последующем не будет совпадать общее количество символов «а», так как в других двух парах количество символов «а» одинаково.

В общем случае можно построить частичный алгоритм, основанный на идее упорядоченной генерации возможных последовательностей цепочек (отметим, что имеется счетное множество таких последовательностей) с проверкой выполнения условий задачи. Если последовательность является решающей – то получаем результативный ответ за конечное количество шагов. Поскольку общий метод определения отсутствия решающей последовательности не может быть указан, так как задача сводима к проблеме «останова» и, следовательно, является алгоритмически неразрешимой, то при отсутствии решающей последовательности алгоритм порождает бесконечный цикл.

В теории алгоритмов такого рода 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 также является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

Вопросы для самоконтроля

1. Исторические аспекты создания и разработки теории алгоритмов.
2. Цели и задачи классической теории алгоритмов.
3. Цели и задачи теории асимптотического анализа алгоритмов.
4. Цели и задачи практического анализа алгоритмов.
5. Теоретический и практический аспекты применения результатов теории алгоритмов.

6. Формализация алгоритма, определения Колмогорова и Маркова.

7. Требования к алгоритму, связанные с формальными определениями.

Упражнения

Упражнение 1.1. Постройте машину Тьюринга, которая записывает входное двоичное слово в обратном порядке.

Упражнение 1.2. Постройте машину Тьюринга, которая удваивает исходное слово, т.е. из слова α формирует слово $\alpha * \alpha$.

Упражнение 1.3. Постройте машину Тьюринга, которая вычисляет предикат «α – четное число».

Упражнение 1.4. Постройте машину Тьюринга, которая вычисляет сложение n чисел.

Упражнение 1.5. Постройте машину Тьюринга для распознавания строки с одинаковым количеством 0 и 1.

2. ВВЕДЕНИЕ В АНАЛИЗ АЛГОРИТМОВ

2.1. Сравнительные оценки алгоритмов

При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение проблемы выбора связано с построением системы сравнительных оценок, которая, в свою очередь, существенно опирается на формальную модель алгоритма.

Будем рассматривать в дальнейшем, придерживаясь определений Поста, применимые к общей проблеме правильные и финитные алгоритмы, т.е. алгоритмы, дающие *1-решение* общей проблемы. В качестве формальной системы будем рассматривать абстрактную машину, включающую процессор с фон – Неймановской архитектурой, поддерживающей адресную память, и набор «элементарных» операций, соотнесенных с языком высокого уровня.

В целях дальнейшего анализа примем следующие допущения:

- каждая команда выполняется не более чем за фиксированное время;
- исходные данные алгоритма представляются машинными словами по β битов каждое.

Конкретная проблема задается N словами памяти, таким образом, на входе алгоритма – N_β = N*β бит информации. Отметим, что в ряде случаев, особенно при рассмотрении матричных задач, N является мерой длины входа алгоритма, отражающей линейную размерность.

Программа, реализующая алгоритм для решения общей проблемы, состоит из M машинных инструкций по $\beta_м$ битов – M_β = M*$\beta_м$ бит информации.

Кроме того, алгоритм может требовать следующих дополнительных ресурсов абстрактной машины:

- S_d – память для хранения промежуточных результатов;
- S_r – память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

При решении конкретной проблемы, заданной N словами памяти, алгоритм выполняет не более, чем конечное количество «элементарных» операций абстрактной машины в силу условия рассмотрения

только финитных алгоритмов. В связи с этим введем следующее определение:

Определение. Трудоёмкость алгоритма.

Под трудоёмкостью алгоритма для данного конкретного входа F_a(N) будем понимать количество «элементарных» операций, совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной системы, требуемых алгоритмом для решения конкретных проблем. Очевидно, что для различных областей применения веса ресурсов будут различны, что приводит к следующей комплексной оценке алгоритма:

$\psi_A = c_1 * F_a(N) + c_2 * M + c_3 * S_a + c_4 * S_r$, где c_i – веса ресурсов.

Система обозначений в анализе алгоритмов

При более детальном анализе трудоемкости алгоритма оказывается, что не всегда количество элементарных операций, выполняемых алгоритмом на одном входе длины N, совпадает с количеством операций на другом входе такой же длины. Это приводит к необходимости введения специальных обозначений, отражающих поведение функции трудоемкости данного алгоритма на входных данных фиксированной длины.

Пусть D_A – множество конкретных проблем данной задачи, заданное в формальной системе. Пусть D є D_A – задание конкретной проблемы и |D| = N.

В общем случае существует собственное подмножество множества D_A, включающее все конкретные проблемы, имеющие мощность N:

обозначим это подмножество через D_N: D_N = {D є D_N,: |D| = N};

обозначим мощность множества D_N через M_{DN} --> $M_{DN} = |D_N|$.

Тогда содержательно данный алгоритм, решая различные задачи размерности N, будет выполнять в каком-то случае наибольшее количество операций, а в каком-то случае наименьшее количество операций. Ведем следующие обозначения:

1. $F_a\hat{}$(N) – *худший случай* – наибольшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N:

$F_a\hat{}(N) = \max_{D \in D_N} \{F_a(D)\}$ - худший случай на D_N.

2. $F_a^{\vee}$(N) – *лучший случай* – наименьшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N:

$F_a^{\vee}(N) = \min_{D \in D_N} \{F_a(D)\}$ – лучший случай на D_N.

3. $\bar{F}_a$(N) – средний случай – среднее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N:

$\bar{F}_a(N) = (1 / M_{DN}) * \sum_{D \in D_N} \{F_a(D)\}$ – средний случай на D_N.

В зависимости от влияния исходных данных на функцию трудоемкости алгоритма может быть предложена следующая классификация, имеющая практическое значение для анализа алгоритмов:

1. Количественно-зависимые по трудоемкости алгоритмы. Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений:

$$F_a(\mathrm{D}) = F_a(|\mathrm{D}|) = F_a(\mathrm{N}).$$

Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами – умножение матриц, умножение матрицы на вектор и т.д.

2. Параметрически-зависимые по трудоемкости алгоритмы. Это алгоритмы, трудоемкость которых определяется не размерностью входа (как правило, для этой группы размерность входа обычно фиксирована), а конкретными значениями обрабатываемых слов памяти:

$$F_a(\mathrm{D}) = F_a(d_1,\ldots,d_n) = F_a(P_1,\ldots,P_m),\ \mathrm{m} =< \mathrm{n}.$$

Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов. Очевидно, что такие алгоритмы, имея на входе два числовых значения – аргумент функции и точность, – выполняют существенно зависящее от значений количество операций:

а) вычисление x^k последовательным умножением

$\Rightarrow F_a(\mathrm{x}, \mathrm{k}) = F_a(\mathrm{k})$;

б) вычисление $e^x = \Sigma (x^n/\mathrm{n}!)$, с точностью до $\xi \Rightarrow F_a = F_a(\mathrm{x}, \xi)$.

3. Количественно-параметрические по трудоемкости алгоритмы.

Однако в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае:

$$F_a(\mathrm{D}) = F_a(\|\mathrm{D}\|, P_1,\ldots,P_m) = F_a\ (\mathrm{N}, P_1,\ldots,P_m).$$

В качестве примера можно привести алгоритмы численных методов, в которых параметрически-зависимый внешний цикл по точности включает в себя количественно-зависимый фрагмент по размерности.

4. Порядково-зависимые по трудоемкости алгоритмы. Среди разнообразия параметрически-зависимых алгоритмов выделим еще оду группу, для которой количество операций зависит от порядка расположения исходных объектов.

Пусть множество D состоит из элементов ($d_1,\ldots,d_n$), и ||D||=N,

Определим ^{D_p}= {($d_1,\ldots,d_n$)} – множество всех упорядоченных N-ок из $d_1,\ldots,d_n$, отметим, что $|^{D_p}|$ = n!.

Если $F_a(\mathrm{i}^{D_p}) \neq F_a(\mathrm{j}^{D_p})$, где i^{D_p}, j^{D_p} є D_p, то алгоритм будем называть порядково-зависимым по трудоемкости.

Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве. Рассмотрим более подробно алгоритм поиска максимума в массиве S, содержащим n элементов:

```
MaxS (S,n; Max)
Max ← S1
   For i←2 to n
         if Max < Si
                then Max ← Si
```

(количество выполненных операций присваивания зависит от порядка следования элементов массива)

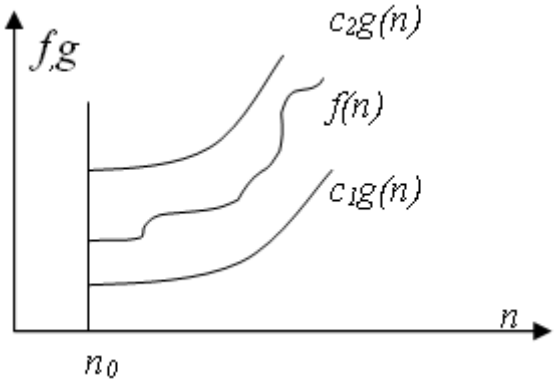
Асимптотический анализ функций

При анализе поведения функции трудоемкости алгоритма часто используют принятые в математике асимптотические обозначения, позволяющие показать скорость роста функции, маскируя при этом конкретные коэффициенты.

Такая оценка функции трудоемкости алгоритма называется *сложностью алгоритма* и позволяет определить предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных.

В асимптотическом анализе приняты следующие обозначения:

1. Оценка Θ (тетта). Пусть f(n) и g(n) – положительные функции положительного аргумента, n >= 1 (количество объектов на входе и количество операций – положительные числа), тогда:



f(n) = Θ(g(n)), если существуют положительные c1, c2, n0, такие, что: c1 * g(n) =< f(n) =< c2 * g(n), при n > n0.

Обычно говорят, что при этом функция g(n) является асимптотически точной оценкой функции f(n), так как по определению функция f(n) не отличается от функции g(n) с точностью до постоянного множителя.

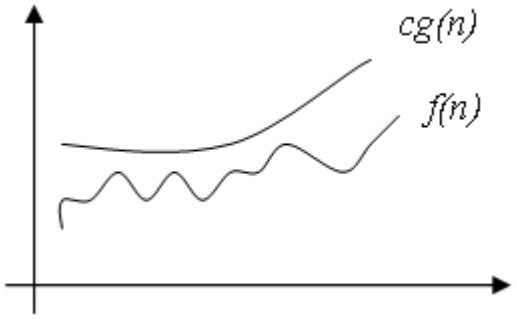
Отметим, что из f(n) = Θ(g(n)) следует, что g(n) = Θ(f(n)).

Примеры:

1) f(n)=4n^2+nlnN+174 – f(n)= Θ(n^2);

2) f(n)=Θ(1) – запись означает, что f(n) или равна константе, не равной нулю, или f(n) ограничена константой на ∞ : f(n) = 7+1/n = = Θ(1).

2. Оценка O (O большое). В отличие от оценки Θ, оценка O требует только, чтобы функция f(n) не превышала g(n), начиная с n > n0, с точностью до постоянного множителя:



$\exists$ c > 0, n0 > 0 : 0 =< f(n) =< c * g(n), $\forall$ n > n0.

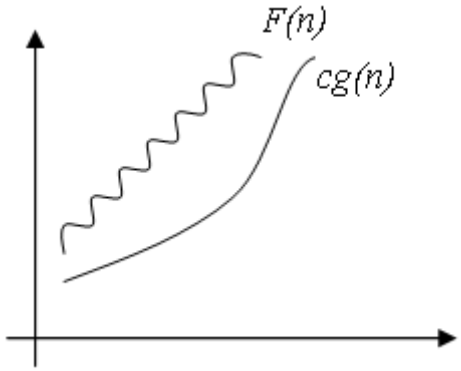
Вообще, запись O(g(n)) обозначает класс функций, таких, что все они растут не быстрее, чем функция g(n) с точностью до постоянного множителя, поэтому иногда говорят, что g(n) мажорирует функцию f(n).

Например, для всех функций:

f(n) = 1/n, f(n) = 12, f(n) = 3*n + 17, f(n) = n*Ln(n), f(n) = 6*n^2 + + 24*n + 77 будет справедлива оценка O(n^2).

Указывая оценку O, есть смысл указывать наиболее «близкую» мажорирующую функцию, поскольку, например, для f(n) = n^2 справедлива оценка O(n^2), однако она не имеет практического смысла.

3. Оценка Ω (Омега). В отличие от оценки O, оценка Ω является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем g(n) с точностью до постоянного множителя:



$\exists$ c > 0, n0 >0 : 0 =< c * g(n) =< f(n).

Например, запись Ω(n*Ln(n)) обозначает класс функций, которые растут не медленнее, чем g(n) = n*Ln(n). В этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием большим единицы.

Асимптотическое обозначение O восходит к учебнику Бахмана по теории простых чисел (Bachman, 1892), обозначения Θ, Ω введены Д. Кнутом (Donald Knuth).

Отметим, что не всегда для пары функций справедливо одно из асимптотических соотношений, например, для f(n) = $n^{1+sin(n)}$ и g(n) = n не выполняется ни одно из асимптотических соотношений.

В асимптотическом анализе алгоритмов разработаны специальные методы получения асимптотических оценок, особенно для класса рекурсивных алгоритмов. Очевидно, что Θ оценка является более прдпочтительной, чем оценка O. Знание асимптотики поведения функции трудоемкости алгоритма – его сложности – дает возможность делать прогнозы по выбору более рационального с точки зрения трудоемкости алгоритма для больших размерностей исходных данных.

Пример 1:

1. For i <-- 1 to n

2. For j <-- 1 to n
3. Sum <-- Sum + A[i,j]
4. end for
5. Return (Sum)
6. End

Алгоритм выполняет одинаковое количество операций при фиксированном значении n, и, следовательно, является количественно-зависимым. Применение методики анализа конструкции «Цикл» дает:

$$F_a(n) = 1+1+ n *(3 + 1+ n *(3 + 4))=7n^2+4*n + 2 = \Theta(n^2).$$

Под *n* понимается линейная размерность матрицы, в то время как на вход алгоритма подается n^2 значений.

Пример 2. Задача поиска максимума в массиве:

```
MaxS (S,n; Max)
 Max <-- S[1]
 For i <-- 2 to n
 if Max < S[i]
 then Max <-- S[i]
end for
return Max
End
```

Данный алгоритм является количественно-параметрическим, поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

А) Худший случай. Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. Трудоемкость алгоритма в этом случае равна:

$$F_a^{\wedge}(n)=1+1+1+ (n-1) (3+2+2)=7 n - 4 = \Theta(n).$$

Б) Лучший случай. Минимальное количество переприсваивания максимума (ни одного на каждом проходе цикла) будет в том случае, если максимальный элемент расположен на первом месте в массиве. Трудоемкость алгоритма в этом случае равна:

$$F_a^{\vee}(n)=1+1+1+ (n-1) (3+2)=5 n - 2 = \Theta(n).$$

В) Средний случай. Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением максимума. На очередном шаге, когда просматривается *к*-й элемент массива, переприсваивание максимума

произойдет, если в подмассиве из первых *к* элементов максимальным элементом является последний. Очевидно, что в случае равномерного распределения исходных данных вероятность того, что максимальный из *к* элементов расположен в определенной (последней) позиции, равна 1/*к*. Тогда в массиве из n элементов общее количество операций переприсваивания максимума определяется как:

$$\sum_{i=1}^{N} 1/i = Hn \approx Ln(N) + \gamma, \quad \gamma \approx 0,57.$$

Величина *Hn* называется *n*-м гармоническим числом. Таким образом, точное значение (математическое ожидание) среднего количества операций присваивания в алгоритме поиска максимума в массиве из n элементов определяется величиной *Hn* (на бесконечности количества испытаний), тогда:

$\overline{F}_a$(n) = 1 +(n-1)*(3+2) +2*(Ln(n) + γ)=5*n +2*Ln(n) - 4+2*γ=Θ(n).

Переход к временным оценкам. Сравнение двух алгоритмов по их функции трудоемкости вносит некоторую ошибку в получаемые результаты. Основной причиной этой ошибки является различная частотная встречаемость элементарных операций, порождаемая разными алгоритмами, и различие во временах выполнения элементарных операций на реальном процессоре. Таким образом, возникает задача перехода от функции трудоемкости к оценке времени работы алгоритма на конкретном процессоре:

Дано: $F_a(D_A)$ – трудоёмкость алгоритма; требуется определить время работы программной реализации алгоритма – $T_A(D_A)$.

На пути построения временных оценок мы сталкиваемся с целым набором различных проблем, пофакторный учет которых вызывает существенные трудности. Укажем основные из этих проблем:

- неадекватность формальной системы записи алгоритма и реальной системы команд процессора;
- наличие архитектурных особенностей существенно влияющих на наблюдаемое время выполнения программы, таких как конвейер, кеширование памяти, предвыборка команд и данных, и т.д.;
- различные времена выполнения реальных машинных команд;
- различие во времени выполнения одной команды в зависимости от значений операндов;
- различные времена реального выполнения однородных команд в зависимости от типов данных;

- неоднозначность компиляции исходного текста, обусловленная как самим компилятором, так и его настройками.

Попытки различного подхода к учету этих факторов привели к появлению разнообразных методик перехода к временным оценкам:

1) пооперационный анализ. Идея пооперационного анализа состоит в получении пооперационной функции трудоемкости для каждой из используемых алгоритмом элементарных операций с учетом типов данных. Следующим шагом является экспериментальное определение среднего времени выполнения данной элементарной операции на конкретной вычислительной машине. Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$T_A(N) = \sum F_{aoni}(N) * \bar{t}_{oni};$$

2) метод Гиббсона. Метод предполагает проведение совокупного анализа по трудоемкости и переход к временным оценкам на основе принадлежности решаемой задачи к одному из следующих типов:

- задачи научно-технического характера с преобладанием операций с операндами действительного типа;
- задачи дискретной математики с преобладанием операций с операндами целого типа;
- задачи баз данных с преобладанием операций с операндами строкового типа.

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций (рис. 1), создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач – $\bar{t}_{тип\ задачи}$.

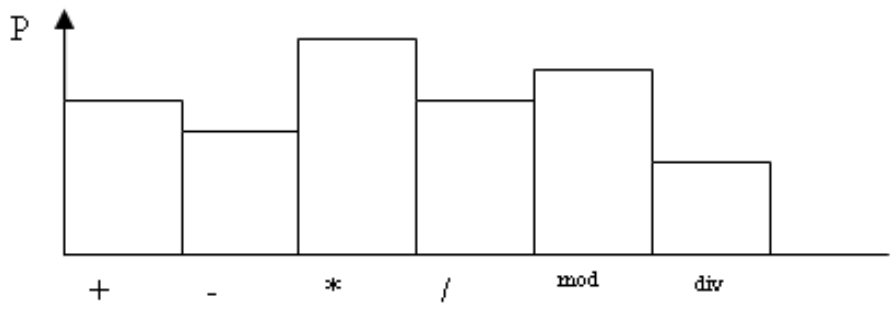


Рис 1. Возможный вид частотной встречаемости операций

На основе полученной информации оценивается общее время работы алгоритма в виде:

$$T_A(\mathrm{N}) = F_a(\mathrm{N}) *;$$

3) метод прямого определения среднего времени. В этом методе так же проводится совокупный анализ по трудоемкости – определяется $F_a(\mathrm{N})$, после чего на основе прямого эксперимента для различных значений $N_э$ определяется среднее время работы данной программы $T_э$ и на основе известной функции трудоемкости рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером – $\bar{t_a}$. Эти данные могут быть (в предположении об устойчивости среднего времени по N) интерполированы или экстраполированы на другие значения размерности задачи следующим образом:

$$\bar{t_a} = T_э(T_э) / F_a(N_э), \mathrm{T(N)} = \bar{t_a} * F_a(\mathrm{N}).$$

Пример пооперационного временного анализа. В ряде случаев именно пооперационный анализ позволяет выявить тонкие аспекты рационального применения того или иного алгоритма решения задачи. В качестве примера рассмотрим задачу умножения двух комплексных чисел:

$$(a + bi)*(c + di) = (ac - bd) + i(ad + bc) = e + if.$$

1. Алгоритм A1 (прямое вычисление e, f – четыре умножения)

```
MultComplex1 (a, b, c, d; e, f)
    e←a*c - b*d          f_A1 =8 операций
    f←a*d + b*c          f_*  =4 операций
    Return (e, f)        f_±  =2 операций
End.                     f_←  =2 операций
```

2. Алгоритм A2 (вычисление e, f за три умножения)

```
MultComplex2 (a, b, c, d; e, f)
    z1←c*(a + b)
    z2←b*(d + c)         f_A2 =13 операций
    z3←a*(d - c)         f_*  =3 операций
    e←z1 - z2            f_±  =5 операций
    f←z1 + z3            f_←  =5 операций
    Return (e, f)
End.
```

Пооперационный анализ этих двух алгоритмов не представляет труда, и его результаты приведены справа от записи соответствующих алгоритмов.

По совокупному количеству элементарных операций алгоритм A2 уступает алгоритму A1, однако в реальных компьютерах операция умножения требует большего времени, чем операция сложения, и можно путем пооперационного анализа ответить на вопрос: при каких условиях алгоритм A2 предпочтительнее алгоритма A1?

Введем параметры q и r, устанавливающие соотношения между временами выполнения операции умножения, сложения и присваивания для операндов действительного типа.

Тогда мы можем привести временные оценки двух алгоритмов к времени выполнения операции сложения/вычитания – t_+:

$t_* = q * t_+$, $q > 1$;

$t_{\leftarrow} = r * t_+$, $r < 1$, тогда приведенные к t_+ временные оценки имеют вид:

$T_{A1} = 4*q*t_+ + 2*t_+ + 2*r*t_+ = t_+*(4*q + 2 + 2*r)$;

$T_{A2} = 3*q*t_+ + 5*t_+ + 5*r*t_+ = t_+*(3*q + 5 + 5*r)$.

Равенство времен будет достигнуто при условии: $4*q + 2 + 2*r = 3*q + 5 + 5*r$,

откуда:

$q = 3 + 3r$ и, следовательно, при $q > 3 + 3r$ алгоритм A2 будет работать более эффективно.

Таким образом, если среда реализации алгоритмов A1 и A2 – язык программирования, обслуживающий его компилятор и компьютер, на котором реализуется задача, – такова, что время выполнения операции умножения двух действительных чисел более чем втрое превышает время сложения двух действительных чисел, в предположении, что $r << 1$, а это реальное соотношение, то для реализации более предпочтителен алгоритм A2.

Конечно, выигрыш во времени пренебрежимо мал, если мы перемножаем только два комплексных числа, однако если этот алгоритм является частью сложной вычислительной задачи с комплексными числами, требующей существенно значимого по времени количества умножений, то выигрыш во времени может быть ощутим. Оценка такого выигрыша на одно умножение комплексных чисел следует из только что проведенного анализа:

$$\Delta T = (q - 3 - 3*r)* \ .$$

2.2. Граф-машина

Трудно найти язык описания алгоритмов, одновременно пригодный для обучения, с одной стороны, и достаточно мощный для реализации реальных приложений (включая научные) – с другой. В качестве базового языка описания алгоритмов предлагается использовать язык визуального программирования GRAPH [2].

В системе GRAPH произвольная программа интерпретируется некоторой вычислимой функцией:

$$f : in(D) \rightarrow out(D),$$

где *in(D)* – множество входных данных программного модуля f;

out(D) – множество выходных (вычисляемых) данных программного модуля f.

Определим граф состояний G как ориентированный помеченный граф, вершины которого – суть состояния, а дугами отмечаются переходы системы из состояния в состояние.

Каждая вершина графа помечается соответствующей локальной вычислимой функцией f_k. Одна из вершин графа, соответствующая начальному состоянию, объявляется начальной вершиной и, таким образом, граф оказывается инициальным. Дуги графа проще всего интерпретировать как события. С позиций данной работы, событие – это изменение состояния объекта O, которое влияет на развитие вычислительного процесса.

На каждом конкретном шаге работы алгоритма в случае возникновения коллизии, когда из одной вершины исходят несколько дуг, соответствующее событие определяет дальнейшей ход развития вычислительного процесса алгоритма. Активизация того или иного события так или иначе зависит от состояния объекта, которое, в свою очередь, определяется достигнутой конкретизацией структур данных D объекта O.

Для реализации событийного управления на графе состояний G введем множество предикативных функций $P = \{P_1, P_2, \ldots, P_I\}$. Под предикатом будем понимать логическую функцию Pi(D), которая в зависимости от значений данных D принимает значение, равное 0 или 1. Дугам графа G поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \rightarrow S_j$ на графе состояний G, инициируется, если модель объекта O на текущем шаге работы алгоритма находится в состоянии *Si* и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

В общем случае, предложенная концепция (без принятия дополнительных соглашений) допускает одновременное наступление нескольких событий в том случае, когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности. Возникает вопрос: на какое из наступивших событий объект программирования должен отреагировать в первую очередь?

Традиционное решение этой проблемы связано с использованием механизма приоритетов. В связи с чем все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющими их приоритеты. Отметим, что принятое уточнение обусловлено ресурсными ограничениями, свойственными однопроцессорной ЭВМ.

Определение. Определим универсальную алгоритмическую модель языка GRAPH четверкой <D, $\mathfrak{F}$, P, G>, где

· D – множество данных (ансамбль структур данных) некоторой предметной области O;

· $\mathfrak{F}$ – множество вычислимых функций некоторой предметной области;

· P – множество предикатов, действующих над структурами данных предметной области D;

· G – граф состояний объекта O.

Граф в данном случае заменяет текстовую (вербальную) форму описания алгоритма программы, при этом:

1. Реализуется главная цель – представление алгоритма в визуальной (графосимволической) форме.

2. Происходит декомпозиционное расслоение основных компонент описания алгоритма программного продукта. Так, структура алгоритма представляется графом G, элементы управления собраны во множестве предикатов P и, как правило, значимы не только для объекта O, но и для всей предметной области. Спецификация структур данных, а также установка межмодульного информационного интерфейса по данным «пространственно» отделена от описания структуры алгоритма и элементов управления.

2.3. Модель данных

Предложенная алгоритмическая модель < D, $\mathfrak{F}$, P, G > в конечном счете описывает некоторую вычислимую функцию $f_G(D)$ и в этом

смысле может служить «исходным материалом» для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

В качестве конструктивного объекта данных в системе GRAPH используются любые возможные структуры данных, доступные базовому языку программирования C++, на котором компилируются программы с визуальных образов GRAPH (рис. 2).

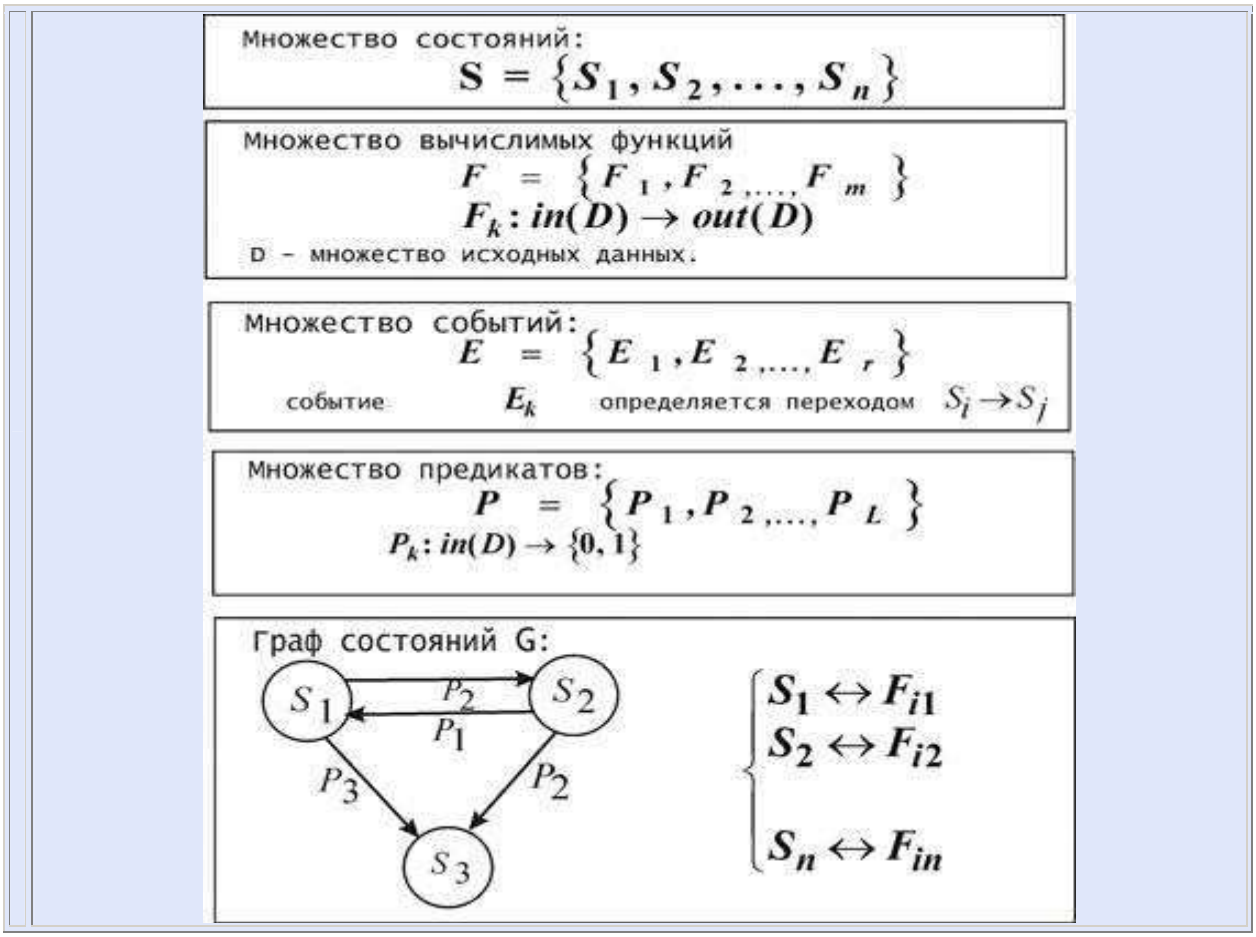


Рис. 2. Алгоритмическая модель языка GRAPH

В системе GRAPH вводится стандарт на организацию межмодульного информационного интерфейса. Стандарт обеспечивается выполнением пяти основных правил:

1) вводится единое для всей предметной области хранилище данных, актуальных для предметной области программирования (ПОП). Полное описание данных размещено в словаре данных ПОП. Любые переменные, не описанные в словаре данных, считаются локальными данными тех объектов ГСП, где они используются;

2) в пределах ГСП описание типов данных централизовано размещается в архиве типов данных;

3) в базовых модулях в качестве механизма доступа к данным допускается только передача параметров по адресам данных;

4) привязка данных объектов ПОП реализована в паспортах объектов ПОП;

5) в технологии ГСП не рекомендуется использовать иные способы организации межпрограммных связей по данным.

Предложенный стандарт позволяет полностью отделить задачу построения межмодульного информационного интерфейса от кодирования процедурной части программы, а также частично автоматизировать процессы построения информационного интерфейса.

Здесь под предметной областью программирования понимается следующее.

Определение. Под предметной областью программирования в дальнейшем понимается среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (словарь и библиотека программных модулей).

Словарь данных представляет собой таблицу, в которой каждому данному присвоено уникальное имя, задан тип, начальное значение данного и краткий комментарий его назначения в ПОП.

Технология ГСП поддерживает жесткие стандарты на описание и документирование программных модулей, представление и поддержку информационного обеспечения программных модулей предметной области. Таким образом, для каждой предметной области строится единая информационная среда, позволяющая унифицировать проектирование написания программных модулей разными разработчиками.

Кроме словаря данных и каталога типов данных информационную среду определяют объекты ГСП. Под объектом понимается специальным образом построенный в рамках технологии ГСП программный модуль, выполняющий определенные действия над данными ПОП.

С информационной точки зрения, каждый объект ГСП fi представляет собой функциональное отображение области определения объекта D_i^{in} на область значений D_i^{out}:

$$f_i : D_i^{in} \to D_i^{out}.$$

В общем случае $D_i^{in} \cap D_i^{out} \neq \varnothing$ (в объекте могут быть модифицируемые данные) и $D_i^{in} \cdot D_i^{out} \in D$, где D – полная область данных ПОП. Для двух произвольных объектов ПОП fi и fj в общем случае справедливо: $(D_i^{in} \cup D_i^{out}) \cap (D_j^{in} \cup D_j^{out}) \neq \varnothing$.

Формально сущность проблемы организации передачи данных между объектами в рамках некоторого модуля-агрегата f_Σ можно определить как задачу построения области данных агрегата f_Σ - $D_\Sigma = D_\Sigma^{in} \cup D_\Sigma^{out}$ и установления соответствий между данными $D_\Sigma = \{d_1, d_2, \ldots d_{n_\Sigma}\}$ и данными $D_i = \{d_1^i, d_2^i, \ldots, d_{n_i}^i\}$ объектов $f_1, f_2, \ldots, f_m$, из которых составлен агрегат f_Σ (рис. 3).

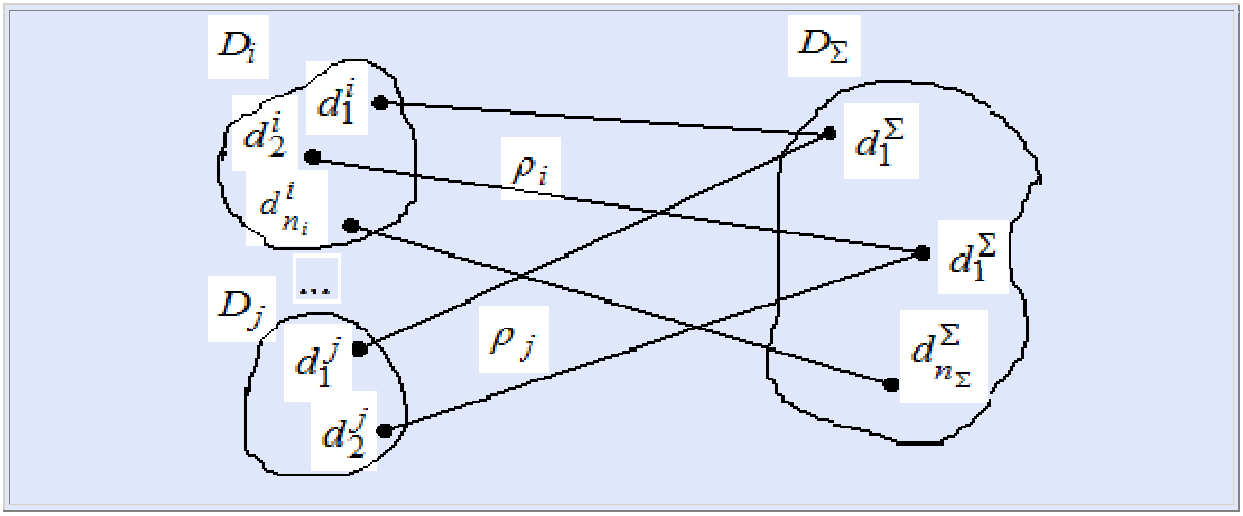


Рис. 3. Информационный межмодульный интерфейс

2.4. Сложность алгоритма

Классы сложности. Временная и пространственная сложность алгоритма. Классы DTIME и DSPACE.

После определения разрешимости хочется иметь меру сложности вычисления. Здесь и далее рассматриваются только разрешимые задачи и всюду определенные (не зацикливающиеся ни на одном входе) машины Тьюринга. Под временем вычисления понимаем число шагов машины Тьюринга до получения результата.

Определение. Пусть $t : N \to N$. Машина Тьюринга T имеет **временную сложность** t(n), если для каждого входного слова α

длины n T выполняет не больше t(n) шагов до остановки. Также будем обозначать временную сложность машины Тьюринга T как timeT (n).

Используемой памятью будем считать число ячеек на ленте, использованных для записи, не считая длины входа.

Ленточной сложностью машины Тьюринга называется функция $s_T(a)$, которая равна мощности просматриваемой активной зоны ленты (исключая мощность входного слова).

Следует помнить, что пространственная сложность может быть меньше длины входа.

Следующим шагом могло бы стать разумное определение «оптимального» алгоритма для данной алгоритмической задачи.

К сожалению, такой подход оказался бесперспективным, и соответствующий результат (теорема об ускорении), установленный на заре развития теории сложности вычислений М. Блюмом, послужил на самом деле мощным толчком для ее дальнейшего развития. Приведем этот результат (без доказательства).

Теорема. Существует разрешимая алгоритмическая задача, для которой выполнено следующее. Для произвольного алгоритма A, решающего эту задачу и имеющего сложность в наихудшем случае timeA(n), найдется другой алгоритм B (для этой же задачи) со сложностью timeB(n), такой, что

$$time_B(n) \le \log_2 time_A(n)$$

выполнено для почти всех n (т.е. для всех n, начиная с некоторого).

Эта теорема показывает, что любой вычислительный процесс машины Тьюринга T_1 можно улучшить с некоторого шага на МТ T_2, что, в свою очередь, с некоторого шага улучшается на МТ T_3 и т.д. Тем самым не существует вычисления наилучшего в абсолютном смысле.

Следует сразу отметить, что задача, о которой идет речь в этой теореме, выглядит довольно искусственно, и, по-видимому, ничего подобного не происходит для задач, реально возникающих на практике. Тем не менее теорема об ускорении не позволяет определить общее математическое понятие «оптимального»алгоритма, пригодное для всех задач, поэтому развитие теории эффективных алгоритмов пошло другим путем. Одним из центральных понятий этой теории стало понятие класса сложности. Так называется совокупность тех алгоритмических задач, для которых существует хотя бы один алгоритм с теми или иными сложностными характеристиками.

Для формальных определений классов сложности обычно рассматривают не произвольные алгоритмы, а алгоритмы для так называемых задач разрешения (переборных задач), когда требуется определить, принадлежит или нет некоторый элемент некоторому множеству.

Учитывая необходимость кодирования данных, подаваемых на вход машине Тьюринга, эти задачи абсолютно эквивалентны задачам распознавания языков, когда на некотором алфавите Σ рассматривается подмножество слов $L \subset \Sigma^*$, и для произвольного слова $l \in \Sigma^*$ нужно определить, принадлежит ли оно языку L.

Суть подхода к определению наиболее сложных задач, называемых универсальными, состоит в сведении к ним любой переборной задачи. Решение универсальной задачи в этом смысле дает решение любой переборной задачи и поэтому универсальная задача оказывается не проще любой из переборных задач (рис. 4).

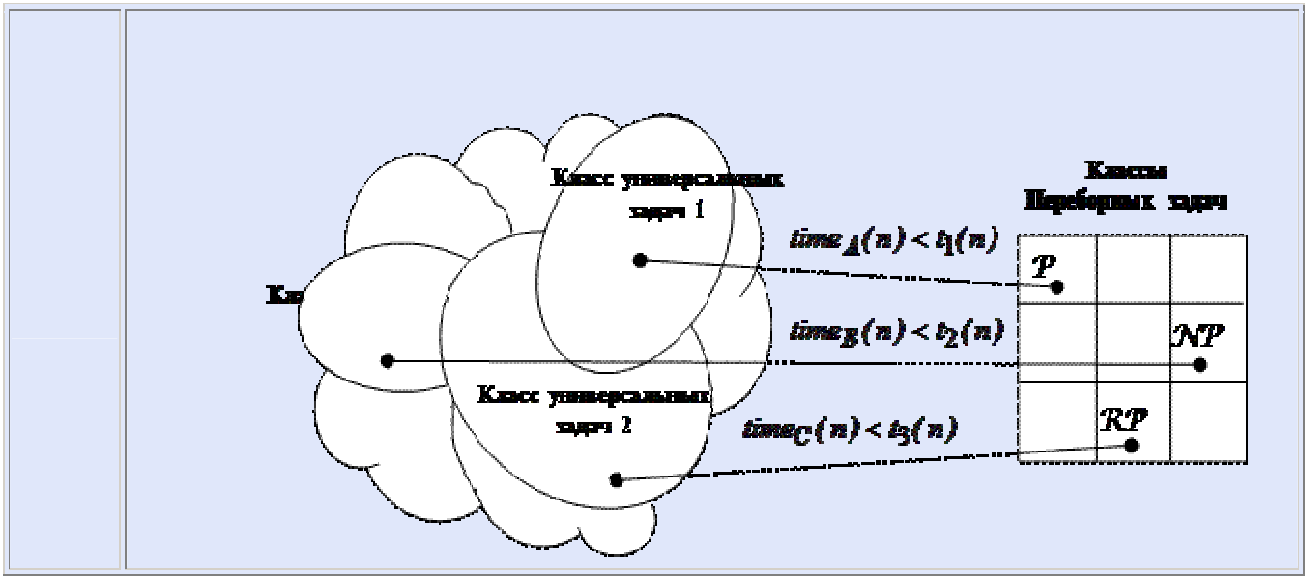


Рис. 4. Классификация сложности задач

Определение. Говорят, что неотрицательная функция $f(n)$ не превосходит по порядку функцию $g(n)$ и пишут $f(n) = O(g(n))$, если существует такая константа C, что $f(n) < Cg(n)$ для любого $n \in N$.

Выражениям «трудоемкость (сложность) алгоритма составляет $O(g(n))$», «решение задачи требует порядка $O(g(n))$ операций» или «алгоритм решает задачу за время $O(g(n))$» обычно при-

дается именно этот смысл. Например, трудоемкость $O(1)$ означает, что время работы соответствующего алгоритма не зависит от длины входного слова ($t(n) = const$).

Алгоритмы с трудоемкостью $O(n)$ называются линейными.

Алгоритм, сложность которого равна $O(n^c)$, где с – константа, называется полиномиальным.

Встречаются алгоритмы, сложность которых оценивается $O(n \log_2 n)$.

Для алгоритма со сложностью $O(c^n)$ (или $O(2^{n^k})$) говорят, что он имеет экспоненциальную сложность.

Определение. Алгоритм называется полиномиальным, если его сложность t(n) в наихудшем случае ограничена сверху некоторым полиномом (многочленом) от n.

В теории алгоритмов рассматриваются в основном переборные или «распознавательные» (задачи, решение которых сводится к получению ответа «да» или «нет») массовые задачи. Произвольные задачи обычно легко сводятся к переборным или «распознавательным» задачам.

Алгоритмы, решающие переборные задачи с полиномиальной сложностью, часто называю эффективными.

Может ли неполиномиальный алгоритм быть эффективным? Ответ утвердительный.

Во-первых, в реальных задачах, в которых время работы алгоритма велико – на практике событие редкое. Во-вторых, многие псевдополиномиальные алгоритмы являются эффективными, когда возникающие на практике числовые параметры не слишком велики.

Подчеркнем, что примеров задач, в которых нарушается основополагающее равенство «полиномиальность» = «эффективность», крайне мало по сравнению с числом примеров, на которых оно блестяще подтверждается.

Класс всех переборных задач с полиномиальной сложностью обозначается P.

Однако возможно, что класс всех переборных («распознавательных») задач шире класса P. Поэтому все переборные задачи обозначают через NP и называют NP-полными задачами.

С другой стороны, алгоритмическая задача называется труднорешаемой (NP-полной), если для нее не существует полиномиального алгоритма.

По этой причине задачи, решаемые с экспоненциальной сложностью, относя к классу NP-полных задач.

Историю современной теории сложности вычислений принято отсчитывать с работ С.А. Кука (1975 г.), в которых были заложены основы теории NP-полноты и доказано существование вначале одной, а затем достаточно большого числа (а именно, 21) естественных NP-полных задач. К 1979 году было известно уже более 300 наименований.

К настоящему времени количество известных NP-полных задач выражается четырехзначным числом и постоянно появляются новые, возникающие как в самой математике и теории сложности, так и в таких дисциплинах, как биология, социология, военное дело, теория расписаний, теория игр и т.д.

Более того, как для подавляющего большинства задач из класса NP в конечном итоге удается либо установить их принадлежность классу P (т.е. найти полиномиальный алгоритм), либо доказать NP-полноту.

Одним из наиболее важных исключений являются задачи типа дискретного логарифма и факторизации, на которых основаны многие современные криптопротоколы.

Тот факт, что большинство «естественных» массовых задач входят в класс NP, свидетельствует о чрезвычайной важности вопроса о совпадении классов P и NP. Безуспешным попыткам построения полиномиальных алгоритмов для NP-полных задач были посвящены усилия огромного числа выдающихся специалистов в данной области.

Ввиду этого можно считать, что NP-полные задачи являются труднорешаемыми со всех практических точек зрения, хотя, повторяем, строгое доказательство этого составляет одну из центральных открытых проблем современной математики.

Алгоритмическая сводимость задач

Пусть существует алгоритм A, который, будучи применимым ко всякому входному слову a задачи Z_1, строит некоторое входное слово $\beta = A(\alpha)$ задачи Z_2. Если при этом слово a дает ответ «да» тогда

и только тогда, когда ответ «да» дает слово β, то говорят, что задача Z_1 (полиномиально) сводится к задаче Z_2, и пишут $Z_1 \Rightarrow Z_2$.

Нетрудно показать, что если $Z_1 \Rightarrow Z_2$ и $Z_2 \in$ P, то $Z_1 \in$ P.

Понятие сводимости переборных задач помогает при определении класса сложности произвольной задачи. Доказав, что задача NP – полная, разработчик алгоритма получает достаточные основания для отказа от поиска эффективного и точного алгоритма. Дальнейшие усилия разработчика могут быть направлены, например, на получение приближенного решения либо решения важнейших частных случаев.

Подведем итоги. Мы определили два класса задач: P и NP. Причем имеет место включение $P \subseteq NP$.

В то же время имеется задача непустоты дополнения полурасширенного выражения, предполагающая создание башен неограниченной высоты при фиксированном n объеме входной информации. Эта задача алгоритмически разрешима, но для ее решения требуется больше чем $(...((2^2)^2)...)^2$ единиц памяти (ленточная сложность). Подобные задачи не принадлежат классу NP и образуют класс труднорешаемых задач.

Кроме того, имеется обширный класс нерешимых задач, которые нельзя решить алгоритмически.

Таким образом, имеются четыре основных класса массовых задач: P, NP, труднорешаемые и нерешаемые задачи.

При описании понятия алгоритма вводится важное понятие конструктивного объекта данных. Без данных не существует алгоритмов. Под конструктивным объектом данных в программировании понимается модель данных. В большинстве языков программирования понятие модели данных совпадает с понятием абстрактных типов данных.

Выбор представления данных (типа данных, модели данных) – один из важнейших этапов разработки алгоритма решения поставленной задачи. Точно так же, как не существует универсальных алгоритмов решения многих задач, обычно невозможно предложить универсальную модель данных, пригодную для их решения. Один и тот же конструктивный объект данных можно представить массивом, структурой, списком, классом, иерархией классов и т.д. Выбор модели данных зачастую зависит от решаемой задачи, используемого алгоритма, особенностей восприятия данных человеком.

Главные соображения, которыми нужно руководствоваться при таком выборе, состоят в следующем.

Во-первых, это естественность внешнего представления исходных данных и ответа, их привычность для человеческого восприятия. Это требование вытекает из специфики использования ЭВМ человеком как средства автоматизации его деятельности. Польза от разработанной программы может быть сведена к минимуму, если для понимания напечатанного ответа от человека требуется дополнительная сложная работа, связанная с переводом данных ответа в понятия исходной формулировки задачи.

Во-вторых, это возможность построения эффективного алгоритма решения задачи. Эта возможность реализуется за счет надлежащего выбора внутреннего представления исходных и промежуточных данных задачи (алгоритма). Как уже отмечалось, для построения более эффективного алгоритма наряду с внешним представлением исходных данных может потребоваться другое внутреннее представление, отличное от внешнего. Программа будет более эффективной, если предусмотреть перевод исходных данных в такое представление, которое обеспечит прямой доступ к нужным компонентам. Во многих задачах подобный перевод из внешнего представления во внутреннее является существенной частью процесса решения задачи, ему следует уделять должное внимание.

3. АЛГОРИТМЫ СОРТИРОВКИ

3.1. Сортировка и поиск

Сортировка вставками. Построение моделей алгоритмов в системе GRAPH

В качестве примера использования системы GRAPH для описания алгоритмов рассмотрим известный алгоритм сортировки «вставками».

Пусть нам задан массив натуральных чисел $A = \{a_1, a_2, ..., a_n\}$. Для простоты введем фиктивный наименьший элемент $a_0 = -\infty$ (для ЭВМ $a_0 = -32000$).

Создадим словарь данных ПОП. В первую очередь, в качестве конструктивного объекта для множества данных A создадим массив A (в языке C++ тип массива определяется описанием: typedef int MASSIV[200];). Переменные n, i, j, w описаны в табл. 1.

Таблица 1

Словарь данных ПОП

Словарь данных				
Имя данного	**Тип**	**Класс данного**	**Нач. значение**	**Комментарий**
A	MASSIV	I	{-32000,18,4,56,65,37,63,66}	Массив, который необходимо отсортировать
n	int	I	6	Размерность массива A
j	int	I	2	Цикл
i	int	V	0	Счетчик
w	int	V	0	Промежуточный элемент

Алгоритм сортировки вставками представлен на рис. 5.

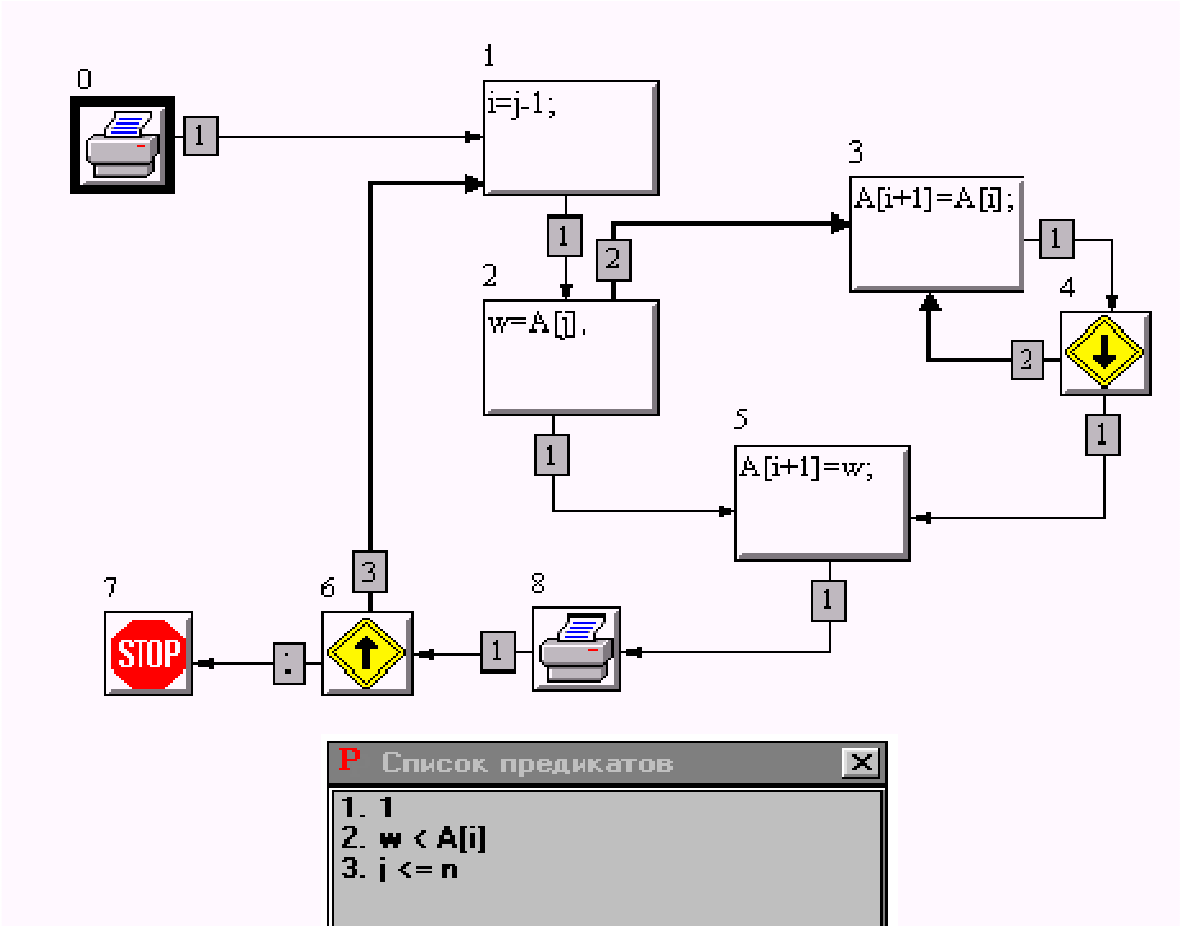


Рис. 5. Алгоритм сортировки

Здесь:

• образом  обозначена вычислимая функция (объект) вывода на печать содержимого массива A (`int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch();`);

• образом  обозначен объект «j++»;

• образом  обозначен объект «i--»;

• образом  обозначена пустая функция «// Конец».

Нулевому элементу массива (A[0]) присвоено значение -32000.

Работа алгоритма начинается с вызова корневой вершины (на рис. 5 обведена «жирно»). В данном случае – печать исходного массива данных. Далее, последовательно, начиная с элемента A[j] (пер-

воначально j=2) на участке массива A от j до 1 производится упорядочивание элементов в порядке возрастания их значений.

Для этого индексу «i» присваивается значение на 1 меньше j (вершина 1). В объекте 2 запоминается «старшее» (улучшаемое значение элемента) A[j]. При этом в цикле вершина 3 – вершина 4 производится перемещение элементов в направлении A[j], до тех пор, пока не выполнится логическая функция 2 (w<A[i]). В этом случае на «освободившееся» место вставляется элемент A[j] (объект 5). Очевидно, что на данный момент все элементы на участке от 1 до j оказываются упорядоченными.

В блоке 8 производится печать текущего состояния массива, в вершине 6 – увеличение индекса j на 1.

Алгоритм работает, пока не исчерпаются все числа массива A (предикат 3).

Оценим временную сложность данного алгоритма. Для решения этой задачи необходимо ввести понятие *инверсии* – разновидности понятия перестановки.

Инверсии

Чтобы подсчитывать эффективность различных алгоритмов сортировки, нужно уметь подсчитывать число перестановок, которые вынуждают повторять некоторый шаг алгоритма определенное число раз.

Пусть $A=(a_1,a_2,\ldots,a_n)$ – перестановка элементов множества {1,2,…,n}. Если $i<j$, а $a_i > a_j$, то пара (a_i,a_j) называется *инверсией* перестановки.

Таблицей инверсии перестановки A называют последовательность $D=(d_1,d_2,\ldots,d_n)$, где d_j – число элементов в перестановке A, таких, что для них выполняется условие $\forall(i=1,2,\ldots,k=find(A,j))\ a_i > a_k$, где функция $find(A,j)$ определяет место j-го элемента в перестановке A.

Например, таблица инверсий для перестановки $A=(5,9,1,8,2,6,4,7,3)$ имеет вид $D=(2,3,6,4,0,2,2,1,0)$.

По определению,

$$0 \le d_1 \le n-1, 0 \le d_2 \le n-2, \ldots, 0 \le d_{n-1} \le 1, d_n = 0.$$

М. Холл установил, что *таблица инверсий единственным образом определяет соответствующую перестановку*. Из любой таблицы ин-

версий можно однозначно восстановить перестановку, которая порождает данную таблицу.

Упражнение 1. Найдите перестановку для таблицы инверсии $D=(3,4,4,5,1,0,2,1,0)$.

Упражнение 2. Напишите программу, которая по заданной таблице инверсии восстанавливает перестановку.

Такое соответствие между перестановками и таблицами инверсий важно потому, что задачи, сформулированные в терминах перестановок, можно свести к эквивалентным им задачам, сформулированным в терминах инверсий.

Например, для инверсий легко подсчитать число всевозможных таблиц инверсий. Так как d_1 можно выбрать n различными способами, d_2 независимо от d_1 n-1 способами и т.д., d_n – единственным способом. Тогда различных таблиц инверсий $n(n-1)\cdot\ldots\cdot 1=n!$

Сложность алгоритмов сортировки определяется числом проверок условия w<A[i], выполняемых в цикле. Сравнение w<A[i] для конкретного j>=2 выполняется $1+d_j$ раз, где d_j – число элементов, больших a_j и стоящих слева от него. т.е. d_j – это число инверсий, у которых второй элемент a_j. Числа d_j составляют таблицу инверсий $0\le d_1\le n-1, 0\le d_2\le n-2,\ldots, 0\le d_{n-1}\le 1, d_n=0.$ В худшем случае, сортировка элементов $A=(a_1,a_2,\ldots,a_n)$ потребует

$$\sum_{j=2}^{n}(1+d_j)\le\sum_{j=2}^{n}(1+n-j)=\frac{n(n-1)}{2}=O(n^2)$$

сравнений.

Сложность сортировки вставками является квадратичной.

3.2. Сортировка всплытия Флойда

Большинство известных алгоритмов сортировки (сортировка вставками, пузырьковая сортировка, сортировка перечислением) требуют $O(n^2)$ сравнений при сортировке элементов $A=(a_1,a_2,\ldots,a_n)$. Рассмотрим один из наиболее элегантных и эффективных методов сортировки сложности $O(n\log_2 n)$, предложенный Флойдом.

Интересно отметить, что в методах сортировки сложности $0(n^2)$ при выборе наибольшего (наименьшего) элемента обычно «забывают» информацию о других забракованных элементах на эту роль, хотя эта проверка и выполнялась. Флойд предложил метод, в котором предшествующие проверки запоминаются и размещаются в специальной структуре данных – двоичном дереве.

Двоичные деревья на смежной памяти

Представление деревьев с помощью списочных структур данных, как правило, не представляет каких-либо трудностей. На рис. 6 схематично представлена некая древовидная структура.

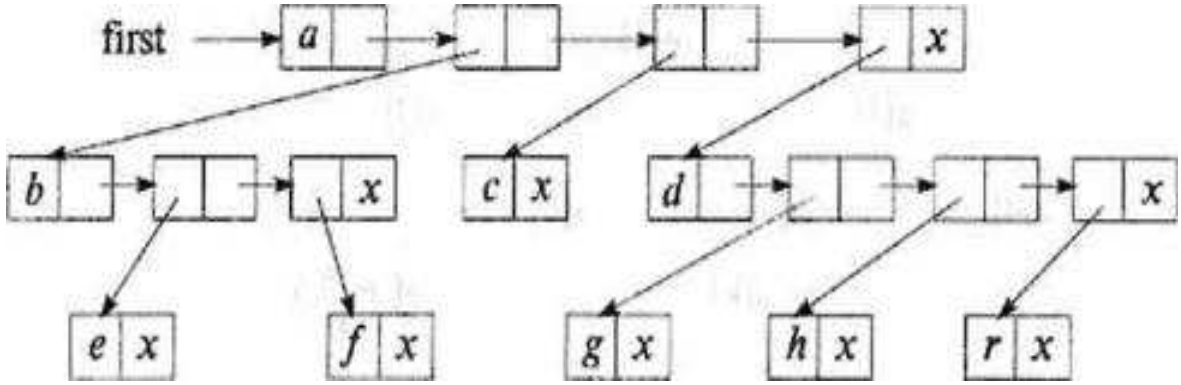


Рис. 6. Представление дерева на связанной памяти

Использование указателей для построения древовидных структур имеет свои неоспоримые преимущества (динамически изменяемое выделение памяти, наглядность, большая универсальность). Однако в отдельных случаях, таких как сортировка множества однородных элементов, выгоднее использовать смежное распределение элементов множества (массив).

Представление деревьев на смежной памяти (одномерный массив) предполагает неявное присутствие ребер, переход по которым выполняется посредством арифметических операций над индексами элементов массива — смежной памяти. Формирование таких деревьев с помощью адресной арифметики можно осуществлять двумя способами. Идея первого способа применима при любом постоянном количестве ребер, выходящих из вершин (регулярное дерево). Рассмотрим данный способ формирования на примере двоичного (бинарного) дерева (рис. 7).

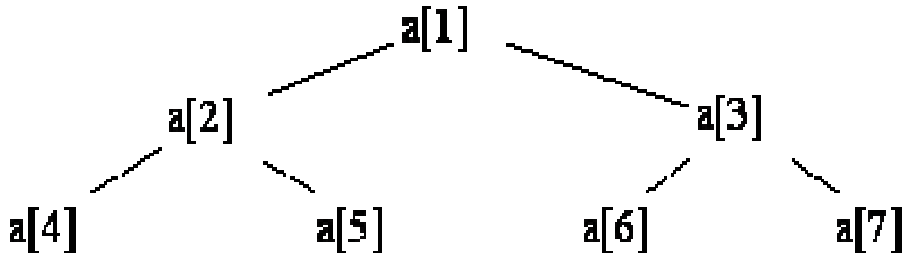


Рис. 7. Представление двоичного дерева на смежной памяти

Пусть имеется одномерный массив смежных элементов $A=(a_1,a_2,...,a_n)$ Неявная структура двоичного дерева определяется, как на рис. 7.

По дереву на рис. 7 легко перемещаться в обоих направлениях. Переход вниз на один уровень из вершины a[к] можно выполнить, удвоив индекс к (индекс левого поддерева) или удвоив и прибавив 1 (индекс правого поддерева). Переход вверх на один уровень из вершины a[m] можно выполнить, разделив m пополам и отбросив дробную часть. Рассмотренная структура применима к любому дереву с постоянным количеством ребер, выходящих из вершин.

Определение. ***Упорядоченным бинарным деревом называют*** *дерево, у которого значение в каждой из вершин не меньше, чем значения в его дочерних вершинах.*

Метод Флойда состоит из двух этапов:

1. На первом этапе первоначальное *неупорядоченное дерево* элементов $A=(a_1,a_2,...,a_n)$ за конечное число шагов *h* (число уровней дерева – высота дерева) превращается в упорядоченное дерево.

2. На втором этапе происходит перемена местами корня дерева с его последним листом. После чего дерево уменьшается на одну вершину (поледний элемент рассматриваемого массива), и все готово для определения нового наибольшего (наименьшего) элемента множества при помощи следующего применения процедуры всплытия Флойда (см. этап 1).

На рис. 8 для массива $A=(3,8,1,4,6,5,2)$ показана работа первого этапа алгоритма всплытия Флойда.

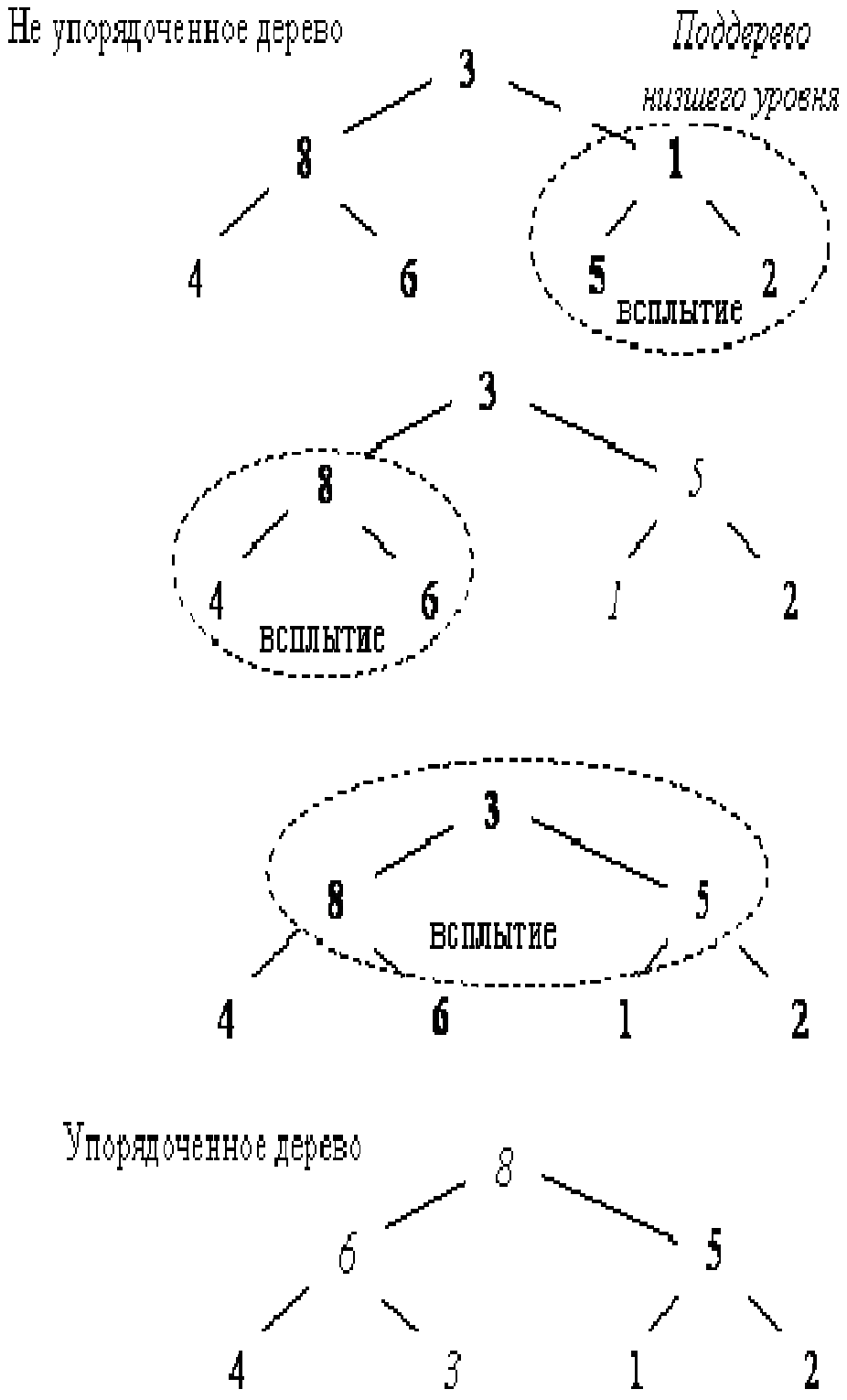


Рис. 8. Первый этап алгоритма Флойда

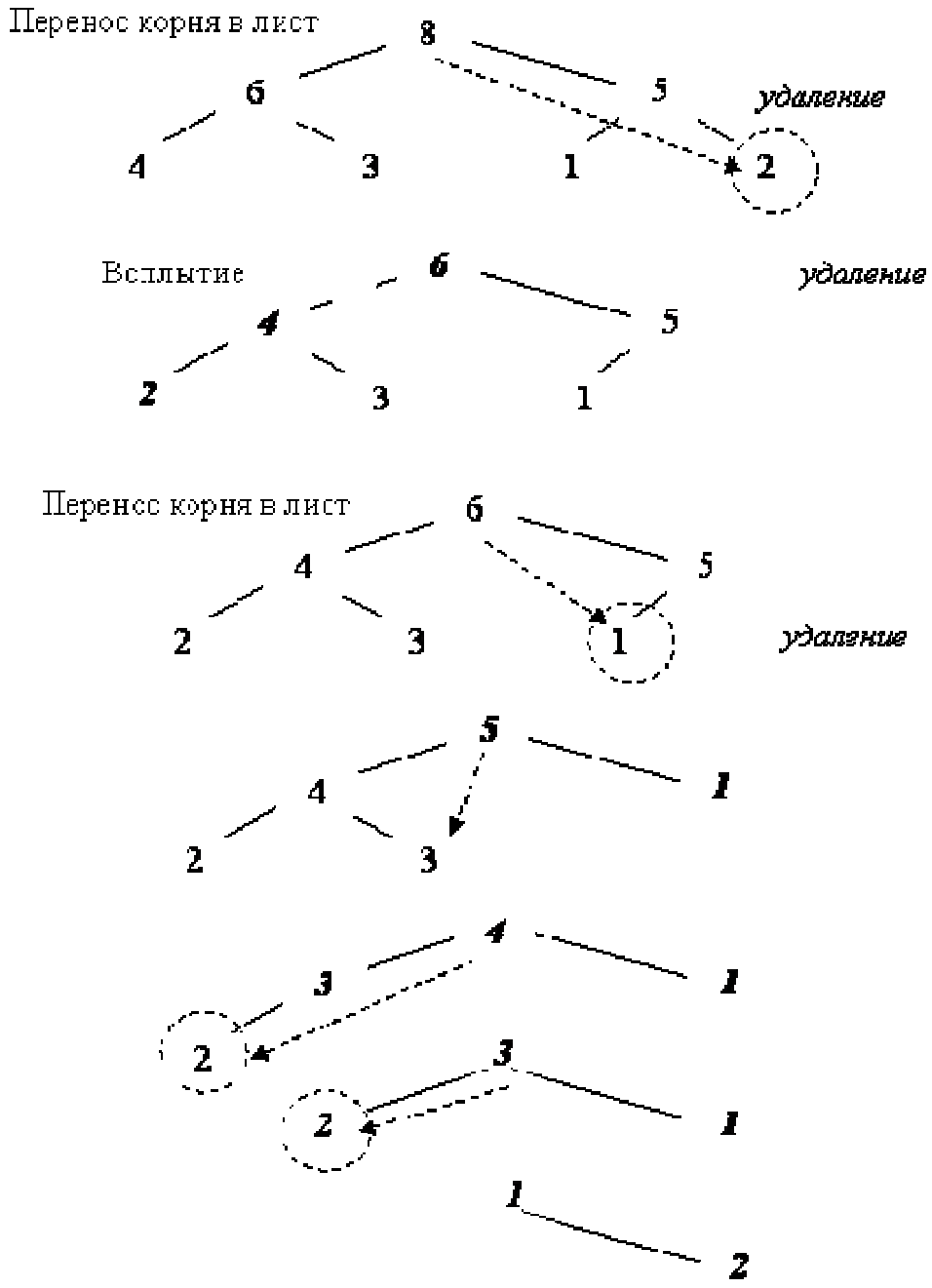


Рис. 9. Второй этап алгоритма Флойда

На втором этапе производится перенос значения максимальной вершины бинарного дерева (корень дерева) в последний элемент (лист дерева) массива, «укорачивание» дерева на один последний элемент и нахождение нового максимального элемента для новой конфигурации дерева (см. рис. 9).

На рис. 10 представлен алгоритм всплытия Флойда на бинарном дереве. Несколько комментариев к алгоритму. В вершине 0 запоми-

нается значение элемента, стоящего в корне дерева. Переменная m необходима для навигации по более низким уровням дерева. Ветка 2-3 необходима в случае, если рассматривается последний элемент списка при четном количестве элементов.

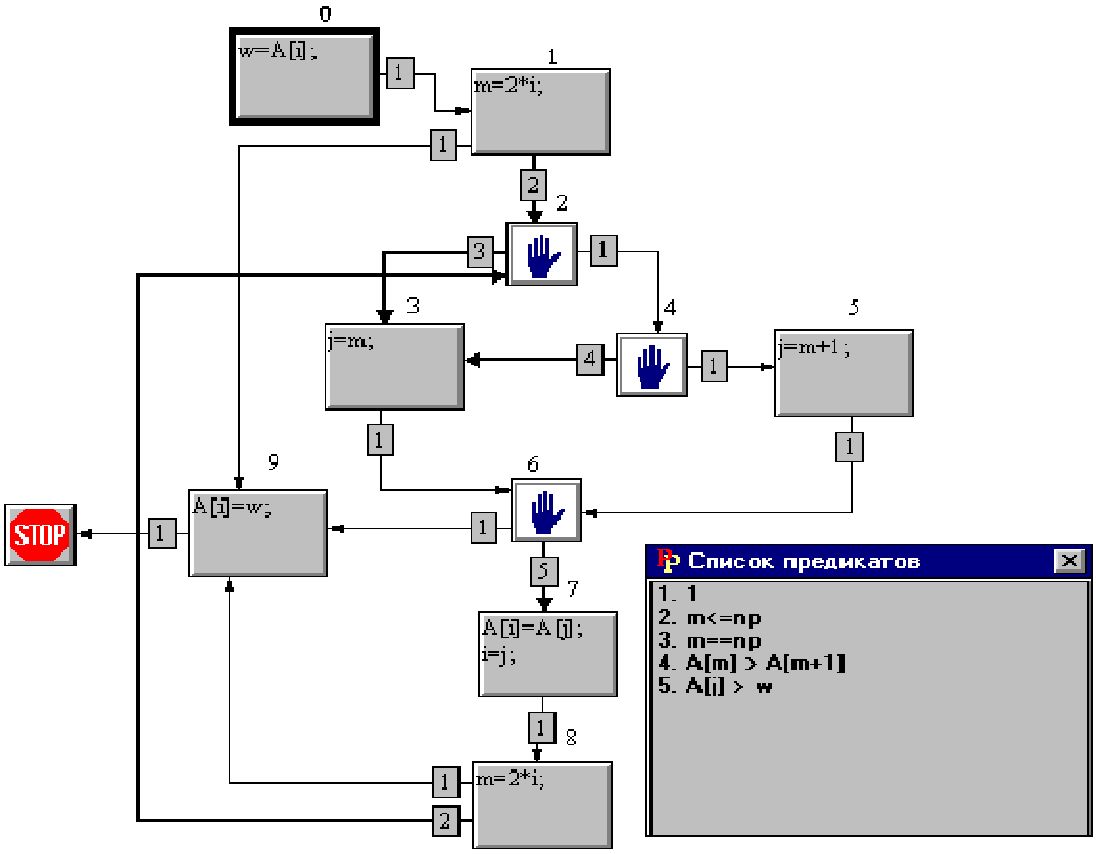


Рис. 10. Процедура всплытия Флойда на бинарном дереве

В вершине 7 производится текущая замена значений элементов при всплытии максимального элемента и установке корневого элемента на своем месте в последовательности вершин соответствующих элементов. Ветка алгоритма 8-2 отрабатывается, если необходима перестановка элементов на несколько уровней вниз в процессе всплытия максимального элемента.

На рис. 11 полностью представлен алгоритм сортировки всплытия Флойда. Схема алгоритма, представленная на рис. 11 выразительными возможностями языка GRAPH, самодостаточна, поэтому не нуждается в комментариях.

Оценим сложность алгоритма Флойда.

Рассмотрим первый этап. Пусть во время всплытия на каждом уровне бинарного дерева выполняется конечное число *C* операций

сравнения элементов массива *A*. Если положить, что высота дерева (число уровней в дереве) равна *h*, то сложность одного всплытия составит $C \cdot h = O(h)$. Высота регулярного бинарного дерева из *n* вершин легко находится из соотношения $n \le 2^0 + 2^1 + ... + 2^{h-1}$, где 2^{i-1} – количество вершин на i-м уровне дерева, i=1,2,..,n. Отсюда $h = [\log_2(n+1)]$.

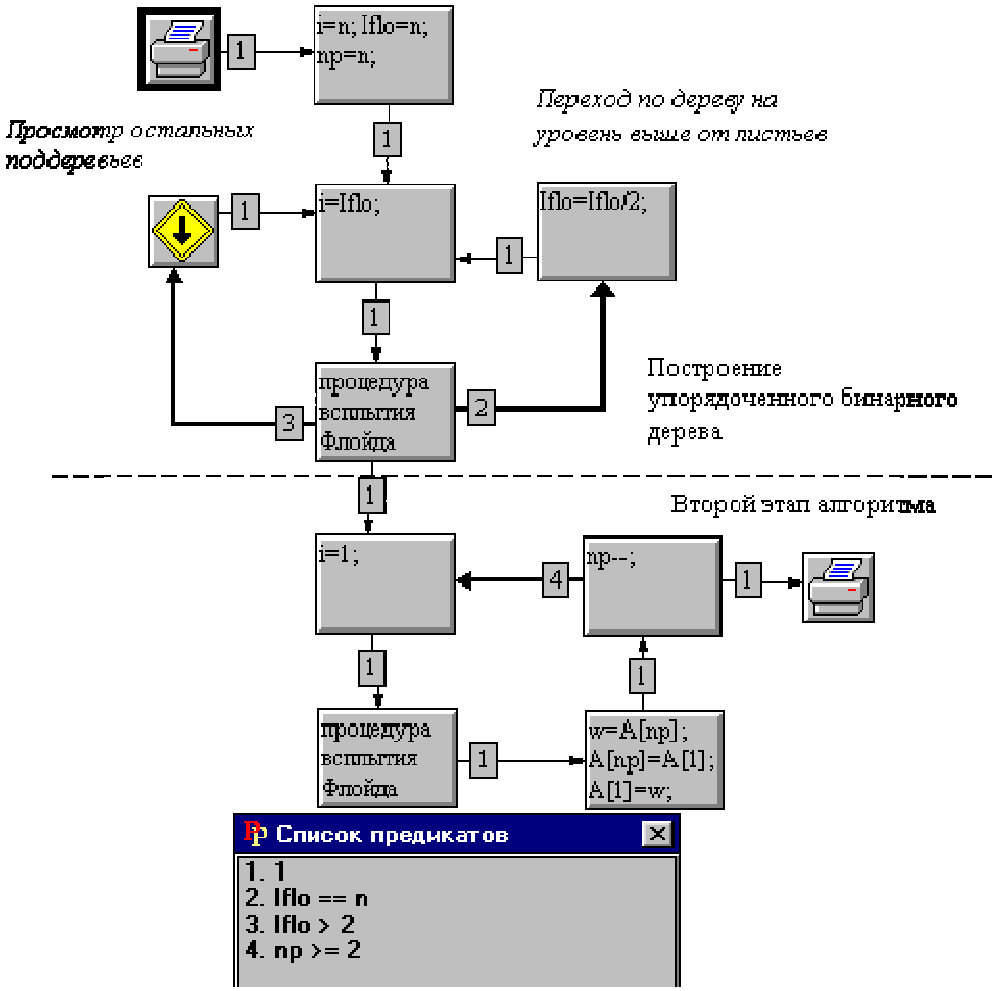


Рис. 11. Алгоритм сортировки всплытия Флойда

В полном варианте процедура всплытия Флойда на 1-м этапе выполняется n раз и затем n раз для каждого шага на 2-м этапе, тогда общая сложность алгоритма сортировки Флойда составит $O(n \log_2 n)$.

Оценка $O(n\log_2 n)$ вообще является наилучшей, на какую только можно надеяться при разработке алгоритмов сортировки, основанных на сравнениях элементов. Действительно, число возможных перестановок элементов множества $A=(a_1,a_2,\ldots,a_n)$ равно $n!$ и только одна перестановка из них удовлетворяет условию сортировки элементов. Двоичный поиск нужной перестановки среди множества $n!$ перестановок требует $\log_2 n!$ числа сравнений. Воспользуемся приближенной формулой Стирлинга для вычисления $n!$ при больших n $n!\approx\sqrt{2\pi n}n^n e^{-n}$. Тогда $\log_2 n!\approx\log_2\sqrt{2\pi n}e^{-n}+n\log_2 n$, отсюда сложность гипотетического «самого эффективного» алгоритма составляет $O(n\log_2 n)$.

3.3. Задачи поиска

Последовательный поиск

Задача поиска является фундаментальной в алгоритмах на дискретных структурах. Задачи сортировки и поиска на практике «идут» «рука об руку». Удивительно, что, накладывая незначительные ограничения на структуру исходных данных, можно получить множество разнообразных стратегий поиска с различной эффективностью.

Последовательный поиск элемента среди элементов множества $A=(a_1,a_2,\ldots,a_n)$ подразумевает исследование элементов множества *A* в том порядке, в каком они встречаются. Эта стратегия поиска является наиболее очевидной и самой простой.

Поиск начинается с первого элемента и продолжается до тех пор, пока нужный элемент не будет найден. После этого алгоритм останавливается.

Очевидно, что наихудшая оценка сложности алгоритма линейно зависит от мощности множества *A* ($O(n)$). Оценим *среднюю сложность* последовательного поиска.

Для нахождения элемента a_i требуется *i* сравнений. Для вычисления среднего времени поиска необходимо знать информацию о частоте обращений к каждому элементу множества. Предположим, что в некотором вычислительном эксперименте к каждому элементу обращаются с одинаковой частотой (частота обращений распределена

равномерно). Тогда средняя сложность алгоритма для *n* независимых испытаний (*n* процедур поиска), будет равна $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$, а *средняя сложность алгоритма* $O(n)$, т.е линейна.

Рассмотрим распределение частот обращения к элементам в общем случае (например, в некоторой поисковой системе). Пусть p_i – обозначает приведенную частоту обращения к элементу a_i (для заданного цикла испытаний). Если для каждого элемента известно количество поисков $m_i \;\; i = 1,2,...n; \;\; \sum_{i=1}^{n} m_i = n$, тогда $p_i = m_i / n, \;\; \sum_{i=1}^{n} p_i = 1$.

В этом случае среднюю сложность можно оценить по формуле

$$\frac{1}{n}\sum_{i=1}^{n} i p_i .$$

Средняя сложность алгоритма поиска в общем случае зависит от распределения частот p_i. Дж. Зипф заметил, что *k*-е наиболее употребительное в тексте на естественном языке слово встречается с частотой приблизительно обратно пропорциональной *k*, т.е $p_k = c / k$. Нормирующая константа выбирается из условия $\sum_{i=1}^{n} p_i = 1$.

Пусть элементы множества $A = (a_1, a_2, ..., a_n)$ упорядочены согласно указанным частотам, т.е $p_1 > p_2 > ... > p_n$. Тогда

$$c = \frac{1}{\sum_{i=1}^{n} 1/i} \approx \frac{1}{\ln n}$$

и среднее время успешного поиска составит $\frac{1}{n}\sum_{i=1}^{n} i p_i = \sum_{i=1}^{n} i \frac{c}{i} = \frac{n}{\ln n}$, что существенно эффективнее, чем при неотсортированном размещении данных.

Последний пример показывает, что даже простой последовательный поиск при удачном выборе структуры данных (отсортированный массив) может существенно повысить эффективность алгоритма поиска. На практике это общепринятая стратегия время от времени переупорядочивать данные.

Логарифмический поиск

Логарифмический поиск (бинарный, метод деления пополам) данных применим к сортированному множеству элементов $a_1 < a_2 < \ldots < a_n$, размещение которого может быть реализовано на смежной памяти.

Идея метода заключается в том, что для большей эффективности поиска элемента (например, a_{find}) необходимо построить алгоритм так, чтобы путь к нужному элементу был как можно короче.

Последнее можно реализовать, если начинать поиск с середины множества, т.е. с элемента $a_{[(1+n)/2]}$. Если $a_{find} < a_{[(1+n)/2]}$, то нужный элемент находится справа от $a_{[(1+n)/2]}$, иначе – наоборот. Поделив пополам правое или левое подмножество, еще раз уменьшим пространство поиска вдвое. Данную процедуру можно продолжать до тех пор, пока не найдем нужный элемент.

Естественной геометрической интерпретацией бинарного поиска является двоичное дерево сравнений.

Определение. Двоичное дерево называется ***деревом сравнений***, если для любой его вершины выполняется условие:

{вершины левого поддерева}<вершина корня<{вершины правого поддерева}.

На рис. 12 показан пример двоичного дерева сравнений для сортированного множества A={3, 5, 7, 9, 12, 19, 27, 44}.

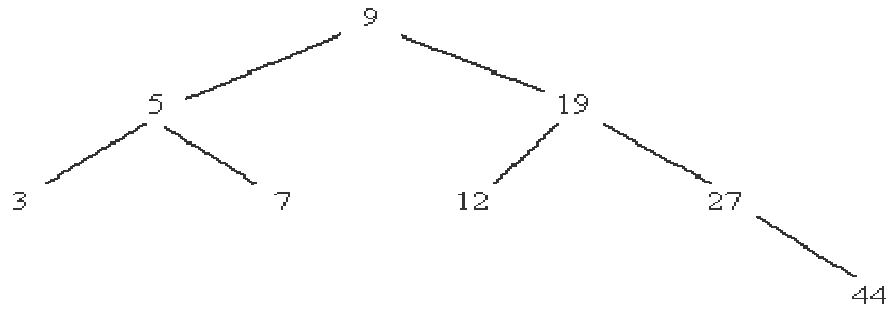


Рис. 12. Пример двоичного дерева сравнений

3.4. Сортировка с вычисляемыми адресами

В предыдущем разделе (на примере алгоритмов поиска) было показано, что учет особенностей исходного объекта может существенно повысить эффективность формируемого алгоритма. Для произвольного множества $A = (a_1, a_2, \ldots, a_n)$ эффективность алгоритма сортировки не может быть лучше $O(n \log_2 n)$. Однако если множество *A* имеет особенности, то можно построить и более эффективные алгоритмы.

Пусть $A = (a_1, a_2, \ldots, a_n)$ – исходная последовательность сортируемых ***целых чисел***. В этом случае с помощью вспомогательного множества индексов (*адресов*) $B = (b_r, b_{r+1}, \ldots, b_s)$ можно построить более эффективный алгоритм сортировки. Здесь $b_{a_i} = a_i$; $r = \min\{a_1, \ldots, a_n\}$; $s = \max\{a_1, \ldots, a_n\}$.

Пусть все элементы множества *A* принимают различные значения, т.е. $\forall i, j \quad a_i \neq a_j$. На рис. 13 представлен алгоритм сортировки с вычисляемыми адресами.

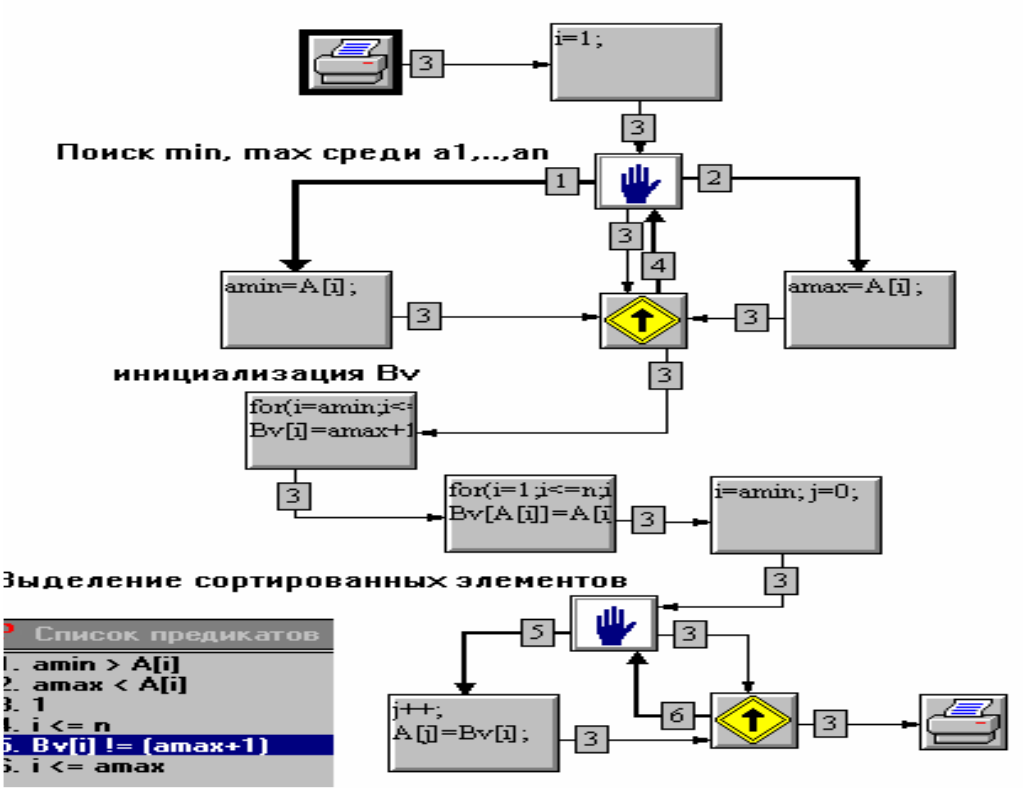


Рис. 13. Алгоритм сортировки с вычисляемыми адресами

Более подробно акторы (наборы практик) алгоритма представлены в табл. 2.

Таблица 2

Акторы алгоритмов

№	Имя актора
0.	amin=A[i];.
1.	amax=A[i];.
2.	i++;.
3.	//Ветвление.
4.	int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch();.
5.	i=1;.
6.	for(i=amin;i<=amax;i++) Bv[i]=amax+1;.
7.	for(i=1;i<=n;i++) Bv[A[i]]=A[i];.
8.	i=amin; j=0;.
9.	//Ветвление.
10.	j++; A[j]=Bv[i];.
11.	i++;.
12.	int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch();.

На фрагменте алгоритма 5, 0, 2, 1 осуществляется поиск максимального и минимального элементов множества *A*.

В модуле 6 производится инициализация элементов вспомогательного массива Bv значениями s + 1 (amax+1). В дальнейшем это значение служит признаком отсутствия в множестве *A* соответствующего элемента, помеченного значением s + 1. Например, для множества A = (3, 6, 2, 5) в массиве Bv значение 7 = 6 + 1 будут иметь следующие элементы: Bv[1], Bv[4].

В вершине 7 производится запись значения a_i в массив Bv по адресу (индексу) a_i. В примере массив Bv примет вид Bv = (7, 2, 3, 7, 5, 6).

Для построения отсортированного множества элементов *A* остается последовательно просмотреть, начиная с первого элемента, массив Bv и переписать его значения в массив A, пропуская элементы, помеченные значением s + 1.

Сортированное множество $A=(a_1<a_2<...<a_n)$ является результатом последовательного просмотра массива $B=(b_r,b_{r+1},...,b_s)$ при условии удаления из него незанятых элементов, равных значению s + 1. Алгоритм не содержит вложенных циклов, а значит, сложность его линейна $O(n)$.

Если элементы массива $A=(a_1,a_2,...,a_n)$ содержат одинаковые элементы, то описанный выше алгоритм необходимо модифицировать. Для этого достаточно ввести еще один массив $C=(c_r,c_{r+1},...,c_s)$ для подсчета кратности элементов массива *A*. В этом случае нет необходимости инициализации массива Bv значениями s + 1, поскольку нулевые значения элементов массива *C* являются признаками отсутствия соответствующих чисел в массиве *A*. Отсортированный массив можно расположить в массиве Bv.

Модифицированный алгоритм не содержит вложенных циклов, а значит, его сложность остается линейной $O(n)$.

Сортировка с вычисляемыми адресами является очень быстрым методом, но она может оказаться неэффективной при больших значениях *s*-*r* с точки зрения использования оперативной памяти (ленточная сложность).

3.5. Эффективность методов оптимизации

Численные методы решения экстремальных задач широко применяются – как при решении математических задач (аппроксимация функций, решение нелинейных систем уравнений, построение нейронных сетей), так и при построении сложных программных приложений (процедуры принятия решений, экспертные системы и т.д.). От эффективности методов оптимизации зачастую зависит эффективность численных методов там, где они используются. Не вдаваясь в теоретические аспекты методов оптимизации (они будут рассмат-

риваться в отдельном курсе лекций), остановимся на оценках сложности алгоритмов оптимизации применительно к некоторым классам задач.

Рассмотрим задачи численного решения задач *математического программирования*:

$$f_0(X) \to \min_x \ \big| \ X \in G, \quad f_j(X) \le 0, \quad j = 1,\dots, m, \tag{1}$$

где $X = (x_1, x_2, \dots, x_n)'$ – оптимизируемые переменные, G – замкнутое подмножество евклидова пространства $G \subset R_n$, $f_0(X)$ – оптимизируемая функция, $f_i(X), \quad i = 1,\dots, m$ – система ограничений.

На рис. 14 образно представлена задача математического (в данном случае, выпуклого программирования). Тонкими линиями представлены изолинии (линии равного уровня) оптимизируемой функции $f_0(X)$ в пространстве 2-х переменных $X = (x_1, x_2)'$. Четыре ограничения представлены жирными линиями. Задача оптимизации ставится как задача поиска наименьшего (минимального) значения оптимизируемой функции при условии выполнения всех заданных ограничений (*задача условной оптимизации*).

Эффективность алгоритмов оптимизации зависит от множества факторов. В первую очередь, от свойств оптимизируемой функции и ограничений по топологии пространства поиска, размерности вектора оптимизируемых параметров и т.д..

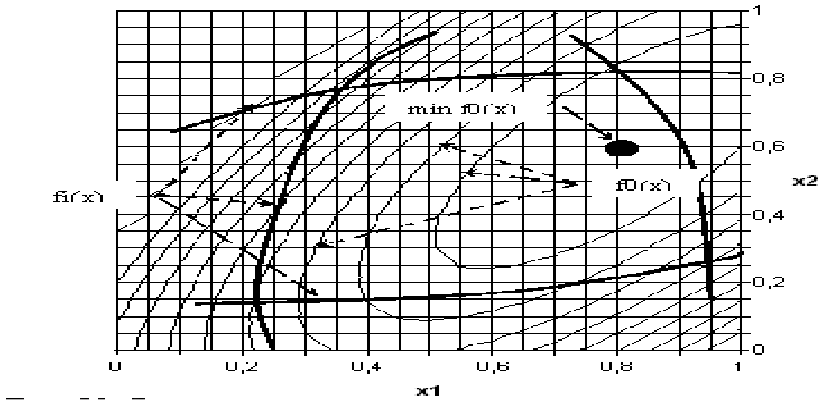


Рис. 14. Задача математического программирования

Для дискретной функции $y_k = f(X^k)$, заданной набором значений функции $y = (y_1, y_2, \dots, y_n)$ задача оптимизации решается доста-

точно просто. Для этого можно использовать любой из алгоритмов переборного поиска. При этом, если множество $y = (y_1, y_2, ..., y_N)$ упорядоченно, то решение находится за один шаг (эффективность алгоритма равна $O(1)$) и не зависит от размерности вектора независимых переменных. Иначе минимальное значение находится, в худшем случае, за *N* шагов (эффективность $O(N)$).

Значительно хуже обстоят дела для непрерывных функций. В первую очередь, в этом случае необходимо ввести понятие точности решения задачи оптимизации, поскольку *численными методами* (на ЭВМ), как правило, найти точное решение невозможно. Обычно вводят некоторую меру оценки погрешности решения поставленной задачи. Например, если X^* – точное решение задачи математического программирования, а $\tilde{X}$ – приближенное решение, полученное тем или иным алгоритмом, то оценку точности решения можно вычислить по формулам:

$$\nu = \sum_{i=1}^{n} \left| x_i^* - \tilde{x}_i \right|$$ – для оценки абсолютной погрешности;

$$\nu = \sum_{i=1}^{n} \left| x_i^* - \tilde{x}_i \right| / \left| x_i^* \right|$$ – для оценки относительной погрешности.

Численные методы (алгоритмы) оптимизации формируют некоторую последовательность (траекторию) $X_1, X_2, ..., X_N$ точек области *G*, такую, что последняя точка $\tilde{X} = X_N$ лежит в окрестностях решения задачи математического программирования (МП) x^*.

Определение. Траекторией алгоритма оптимизации F называется всякая последовательность $X^\infty = \{X_1, X_2, ...\}, \quad X_i \in G$.

Формально детерминированные алгоритмы оптимизации можно задать с помощью рекуррентного правила

$$X^{k+1} = F(X^k),$$

где $F(\bullet)$ – итерационная функция.

Определение. Трудоемкостью траектории (трудоемкостью алгоритма оптимизации) будем называть число $l(X^\infty)$ членов последова-

тельности, обеспечивающих попадание последней точки траектории в ν – окрестность решения задачи МП.

В методах оптимизации эффективность алгоритма традиционно оценивается по числу обращений к оптимизируемой функции. К особенности рассматриваемого класса задач можно отнести *приближенный характер формируемого решения*.

Часто невозможно найти точное решение поставленной задачи. Одним из общих подходов к решению NP-трудных задач, бурно развивающихся в настоящее время, является разработка приближенных алгоритмов с гарантированными оценками качества получаемого решения.

Точность приближенного алгоритма характеризует мультипликативная ошибка, которая показывает, в какое максимально возможное число раз может отличаться полученное решение от оптимального (по значению заданной целевой функции).

Определение. Алгоритм называется C-приближенным, если при любых исходных данных он находит допустимое решение со значением целевой функции, отличающимся от оптимума не более чем в C раз.

Заметим, что иногда также говорят об C-приближенных алгоритмах, причем смысл отклонения (больше или меньше единицы) обычно ясен из контекста и направления оптимизации (максимизации или минимизации). Мультипликативная ошибка может быть константой или зависеть от параметров входной задачи. Наиболее удачные приближенные алгоритмы позволяют задавать точность своей работы.

Эффективность алгоритмов оптимизации зависит:

- от свойств оптимизируемой функции;
- размерности и топологии пространства независимых переменных;
- точности решения задачи оптимизации.

Унимодальные, непрерывные одномерные функции

Пусть при $n = 1$ и $x \in [0;1]$ оптимизируемая функция является линейной функцией $y = ax + b$. В этом случае минимум или максимум функции находится на одном из концов отрезка [0; 1]. Для нахождения минимума функции достаточно вычислить два значения $f(0)$, $f(1)$ и выбрать минимальное. Сложность алгоритма минимальна $O(1)$, решение находится точно.

Пусть для одномерной задачи оптимизации ($x \in [0;1]$) задана унимодальная функция (например, функция, имеющая минимум (рис. 15).

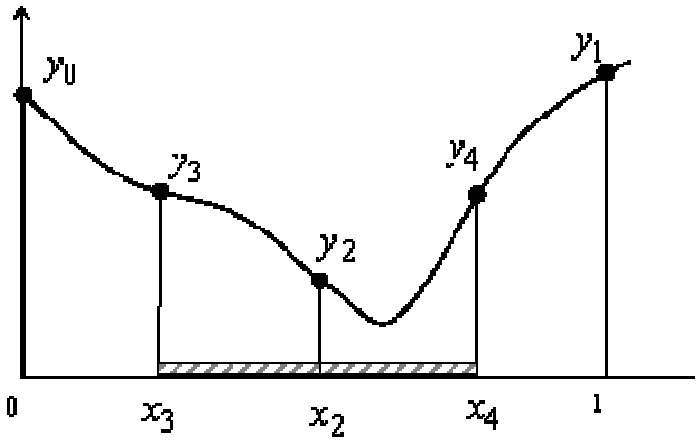


Рис. 15. Оптимизация унимодальной функции

Известно множество алгоритмов оптимизации унимодальных функций. Если задана непрерывная функция, то можно применить самый простой алгоритм двоичного деления *отрезка неопределенности*. Под ***отрезком неопределенности*** будем понимать отрезок области поиска, которому безусловно принадлежит минимум функции.

Первоначально отрезок неопределенности равен исходному отрезку $X_{min} = [0;1]$. Разделим исходный отрезок пополам и вычислим еще два промежуточных значения функции в точках $x_3 = 1/4$ и $x_4 = 3/4$. Новый отрезок неопределенности будет частью исходного отрезка, для которого выполняется условие $y_{i1} \le y_{i2} \le y_{i3}$ (i1, i2, i3 – смежные точки отрезка неопределенности). В нашем случае новый отрезок неопределенности равен $[x_3, x_4] = [0.25; 0.75]$ (заштрихованная линия). За 5 обращений к оптимизируемой функции мы в 2 раза уменьшили отрезок неопределенности. Далее, вычислив 2 раза исходную функцию (справа и слева от средней точки), мы уменьшим отрезок неопределенности еще в 2 раза.

Число итераций, необходимых для вычисления минимума функции с заданной точностью ν, можно определить из выражения

$$\nu \approx \left(\frac{1}{2}\right)^{\left[\frac{N-3}{2}\right]}$$ или

$$N \approx 3 - 2\log_2 \nu .$$

Из формулы следует, что предложенный алгоритм имеет сложность, равную $O(-\log_2 \nu)$, которая для непрерывных унимодальных функций зависит от точности поиска минимума функции. Для более эффективного алгоритма, например, метода золотого сечения, можно получить существенно меньшую сложность $O(\ln \nu / \ln 0.382)$. На рис. 16 показаны графики трудоемкости алгоритмов двоичного деления и золотого сечения в зависимости от точности поиска минимума функции.

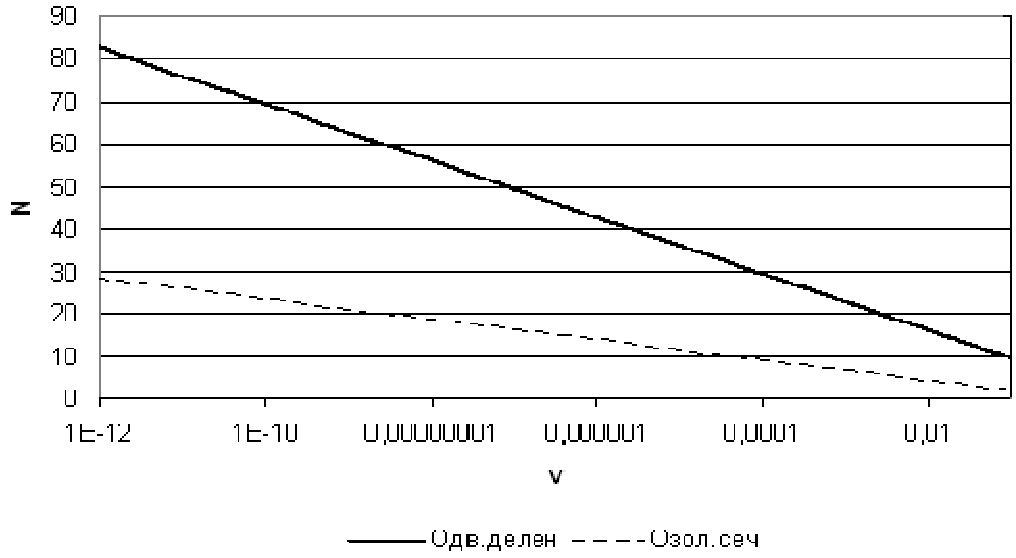


Рис. 16. Оптимизация многоэкстремальных функций

Для многоэкстремальных функций предложенная выше стратегия совершенно не годится. Функция содержит заранее неизвестное количество локальных минимумов и максимумов. В общем случае, без дополнительных предположений, накладываемых на функцию, данная задача, по-видимому, относится к классу *неразрешимых задач.* Не вдаваясь в детали проблемы оптимизации многоэкстремальных задач, рассмотрим аспекты формирования сложности алгоритма на простом примере.

Пусть оптимизируемой функцией является функция $y = a\sin(\omega x)$. Здесь a – амплитуда, ω – частота. Функция $y = a\sin(\omega x)$ является непрерывной, имеющей бесконечное число непрерывных производных высокого порядка, что, безусловно, хорошо само по себе. На отрезке $[0;2\pi]$ данная функция имеет приблизительно $[\omega]$ минимумов. Если на функцию наложить *условие Липшица*, ограничивающее скорость роста непрерывной функции, то для алгоритма оптимизации появляются некоторые «ориентиры», позволяющие строить достаточно эффективные алгоритмы.

Условие Липшица задается неравенством:

$$|f(x_1) - f(x_2)| \le L|x_1 - x_2|,$$

для любых $x_1, x_2 \in [0;1]$.

Упрощенно можно положить $|f'(x)| \le L$. Для примера $|y'| = |a\omega\cos(\omega x)| < |a\omega| < L$ или $|\omega| < \frac{L}{|a|}$, следовательно, исходный участок [0; 1] неопределенности будет содержать приблизительно $|\omega|$ минимумов функции (участков унимодальности функции), равномерно распределенных на исходном отрезке. Для исходной функции участки унимодальности приблизительно равны $d_1 = d_2 = ... = d_{[\omega]} = 1/[\omega]$. На каждом участке унимодальности функции можно применить алгоритм золотого сечения со сложностью для заданной длины отрезка неопределенности d $N \approx \frac{\ln(v/d)}{\ln 0.382}$, тогда интегральная трудоемкость алгоритма равна $N \approx [\omega]\frac{\ln(v\omega)}{\ln 0.382}$ и, следовательно, алгоритм многоэкстремальной оптимизации функции $y = a\sin(\omega x)$ можно оценить величиной $O([\omega]\frac{\ln(v\omega)}{\ln 0.382})$.

В общем случае, трудоемкость класса многоэкстремальной задачи математического программирования, порождаемой k-гладкими функциями (имеющих k непрерывных производных) $f_i(X), \quad i = 0,..., m$ оценивается снизу величиной

$$N(\nu) \geq C\left(\frac{1}{\nu}\right)^{n/k},$$

где C – константа, зависящая от свойств области G, n – размерность задачи оптимизации, k – гладкость оптимизируемых функций.

Катастрофический рост $N(\nu)$ *при* $\nu \to 0$ *и* $n \to \infty$ *показывает, что бессмысленно ставить вопрос о построении универсальных методов решения «всех вообще» гладких задач сколько-нибудь заметной размерности*.

Последнее высказывание относится как к детерминированным методам, так и стохастическим, в том числе и генетическим алгоритмам, которым в последнее время незаслуженно приписываются высокие оценки эффективности. Естественно, речь идет о методах, дающих гарантированные результаты. На отдельных задачах может повезти любому методу.

Сложность выпуклых экстремальных задач

Если на оптимизируемые функции задачи (3.1) наложить еще более жесткие ограничения, например, положить, что $f_i(X), \quad i = 0,\ldots,m$ являются выпуклыми непрерывными функциями, G – выпуклое множество, то сложность алгоритмов оптимизации можно оценить величиной $O(n\ln(1/\nu))$, что существенно лучше предыдущего случая.

Интересно, что для самого простого случая, когда все $f_i(X), \quad i = 0,\ldots,m$ функции являются линейными (*задача линейного программирования*) и можно было бы ожидать алгоритма точного решения, наилучший из известных методов – симплекс-метод – дает только ***полиномиальную оценку сложности***. Конечно, это верхняя оценка сложности, для реальных задач он работает значительно эффективнее.

4. АЛГОРИТМЫ МАШИННОЙ МАТЕМАТИКИ

В общей постановке задача ставится следующим образом. Имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (далее мы будем называть его ключом сортировки). Тип данных ключа должен включать операции сравнения ("=", ">", "<", ">=" и "<="). Задачей сортировки является преобразование исходной последовательности в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа. Метод сортировки называется устойчивым, если при его применении не изменяется относительное положение записей с равными значениями ключа.

Различают сортировку массивов записей, целиком расположенных в основной памяти (внутреннюю сортировку), и сортировку файлов, хранящихся во внешней памяти и не помещающихся полностью в основной памяти (внешнюю сортировку). Для внутренней и внешней сортировки требуются существенно разные методы. В этой части мы рассмотрим наиболее известные методы внутренней сортировки, начиная с простых и понятных, но не слишком быстрых, и заканчивая не столь просто понимаемыми усложненными методами.

Естественным условием, предъявляемым к любому методу внутренней сортировки, является то, что эти методы не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива. Мерой эффективности алгоритма внутренней сортировки являются число требуемых сравнений значений ключа (C) и число перестановок элементов (M).

Заметим, что поскольку сортировка основана только на значениях ключа и никак не затрагивает оставшиеся поля записей, можно говорить о сортировке массивов ключей.

4.1. Сортировка включением

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста (табл. 3). Пусть имеется массив ключей a[1], a[2], ..., a[n]. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...), до тех

пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i] или достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива).

Можно сократить число сравнений, применяемых в методе простых включений, если воспользоваться тем фактом, что при обработке элемента a[i] массива элементы a[1], a[2], ..., a[i-1] уже упорядочены, и воспользоваться для поиска элемента, с которым должна быть произведена перестановка, методом двоичного деления.

Таблица 3

Пример сортировки методом простого включения

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1-й	8 23 5 65 44 33 1 6
Шаг 2-й	8 5 23 65 44 33 1 6 5 8 23 65 44 33 1 6
Шаг 3-й	5 8 23 65 44 33 1 6
Шаг 4-й	5 8 23 44 65 33 1 6
Шаг 5-й	5 8 23 44 33 65 1 6 5 8 23 33 44 65 1 6
Шаг 6-й	5 8 23 33 44 1 65 6 5 8 23 33 1 44 65 6 5 8 23 1 33 44 65 6 5 8 1 23 33 44 65 6 5 1 8 23 33 44 65 6 1 5 8 23 33 44 65 6
Шаг 7-й	1 5 8 23 33 44 6 65 1 5 8 23 33 6 44 65 1 5 8 23 6 33 44 65 1 5 8 6 23 33 44 65 1 5 6 8 23 33 44 65

Дальнейшим развитием метода сортировки с включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Мы не будем описывать алгоритм в общем виде, а ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2n элементами алгоритм работает следующим образом. На первой фазе производится сортировка включением всех пар элементов массива, расстояние между которыми есть 2(n-1). На второй фазе производится сортировка включением элементов полученного масси-

ва, расстояние между которыми есть 2(n-2). И так далее, пока не дойдем до фазы с расстоянием между элементами, равным единице, и не выполним завершающую сортировку с включениями. Применение метода Шелла к массиву, используемому в наших примерах, показано в табл. 4.

Таблица 4

Пример сортировки методом Шелла

Начальное состояние массива	8 23 5 65 44 33 1 6
Фаза 1-я (сортируются элементы, расстояние между которыми четыре)	8 23 5 65 44 33 1 6 8 23 5 65 44 33 1 6 8 23 1 65 44 33 5 6 8 23 1 6 44 33 5 65
Фаза 2-я (сортируются элементы, расстояние между которыми два)	1 23 8 6 44 33 5 65 1 23 8 6 44 33 5 65 1 23 8 6 5 33 44 65 1 23 5 6 8 33 44 65 1 6 5 23 8 33 44 65 1 6 5 23 8 33 44 65 1 6 5 23 8 33 44 65
Фаза 3-я (сортируются элементы, расстояние между которыми один)	1 6 5 23 8 33 44 65 1 5 6 23 8 33 44 65 1 5 6 23 8 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65

В общем случае алгоритм Шелла естественно переформулируется для заданной последовательности из t расстояний между элементами h1, h2, ..., ht, для которых выполняются условия h1 = 1 и h(i+1) < hi.

4.2. Обменная сортировка

Простая обменная сортировка (в просторечии называемая «методом пузырька») для массива a[1], a[2], ..., a[n] работает следующим образом. Начиная с конца массива сравниваются два соседних элемента (a[n] и a[n-1]). Если выполняется условие a[n-1] > a[n], то значения элементов меняются местами. Процесс продолжается для a[n-1] и a[n-2] и т.д., пока не будет произведено сравнение a[2] и a[1]. Понятно, что после этого на месте a[1] окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но по-

следними сравниваются a[3] и a[2]. И так далее. На последнем шаге будут сравниваться только текущие значения a[n] и a[n-1]. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые «легкие») постепенно «всплывают» к верхней границе массива. Пример сортировки методом пузырька показан в табл. 5.

Таблица 5

Пример сортировки методом пузырька

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1-й	8 23 5 65 44 33 1 6 8 23 5 65 44 1 33 6 8 23 5 65 1 44 33 6 8 23 5 1 65 44 33 6 8 23 1 5 65 44 33 6 8 1 23 5 65 44 33 6 1 8 23 5 65 44 33 6
Шаг 2-й	1 8 23 5 65 44 6 33 1 8 23 5 65 6 44 33 1 8 23 5 6 65 44 33 1 8 23 5 6 65 44 33 1 8 5 23 6 65 44 33 1 5 8 23 6 65 44 33
Шаг 3-й	1 5 8 23 6 65 33 44 1 5 8 23 6 33 65 44 1 5 8 23 6 33 65 44 1 5 8 6 23 33 65 44 1 5 6 8 23 33 65 44
Шаг 4-й	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 5-й	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 6-й	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65
Шаг 7-й	1 5 6 8 23 33 44 65

Метод пузырька допускает три простых усовершенствования. Во-первых, как показывает табл. 5, на четырех последних шагах расположение значений элементов не менялось (массив оказался уже упорядоченным). Поэтому если на некотором шаге не было произведено ни одного обмена, то выполнение алгоритма можно прекращать. Во-вторых, можно запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Оче-

видно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса. В-третьих, метод пузырька работает неравноправно для «легких» и «тяжелых» значений. Легкое значение попадает на нужное место за один шаг, а тяжелое на каждом шаге опускается по направлению к нужному месту на одну позицию.

На этих наблюдениях основан метод шейкерной сортировки (ShakerSort). При его применении на каждом следующем шаге меняется направление последовательного просмотра. В результате, на одном шаге «всплывает» очередной наиболее легкий элемент, а на другом «тонет» очередной самый тяжелый. Пример шейкерной сортировки приведен в табл. 6.

Таблица 6

Пример шейкерной сортировки

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1-й	8 23 5 65 44 33 1 6 8 23 5 65 44 1 33 6 8 23 5 65 1 44 33 6 8 23 5 1 65 44 33 6 8 23 1 5 65 44 33 6 8 1 23 5 65 44 33 6 1 8 23 5 65 44 33 6
Шаг 2-й	1 8 23 5 65 44 33 6 1 8 5 23 65 44 33 6 1 8 5 23 65 44 33 6 1 8 5 23 44 65 33 6 1 8 5 23 44 33 65 6 1 8 5 23 44 33 6 65
Шаг 3-й	1 8 5 23 44 6 33 65 1 8 5 23 6 44 33 65 1 8 5 6 23 44 33 65 1 8 5 6 23 44 33 65 1 5 8 6 23 44 33 65
Шаг 4-й	1 5 6 8 23 44 33 65 1 5 6 8 23 44 33 65 1 5 6 8 23 44 33 65 1 5 6 8 23 33 44 65
Шаг 5-й	1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65 1 5 6 8 23 33 44 65

Шейкерная сортировка позволяет сократить число сравнений. Данную сортировку рекомендуется использовать в тех случаях, когда известно, что массив «почти упорядочен».

4.3. Сортировка выбором

При сортировке массива a[1], a[2], ..., a[n] методом простого выбора среди всех элементов находится элемент с наименьшим значением a[i], и a[1] и a[i] обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] до тех пор, пока мы не дойдем до подмассива a[n], содержащего к этому моменту наибольшее значение. Работа алгоритма иллюстрируется примером в табл. 7.

Таблица 7

Пример сортировки простым выбором

Начальное состояние массива	**8** 23 5 65 44 33 1 6
Шаг 1-й	1 **23** 5 65 44 33 8 6
Шаг 2-й	1 5 **23** 65 44 33 8 6
Шаг 3-й	1 5 6 **65** 44 33 8 23
Шаг 4-й	1 5 6 8 **44** 33 65 23
Шаг 5-й	1 5 6 8 23 **33** 65 44
Шаг 6-й	1 5 6 8 23 33 **65** 44
Шаг 7-й	1 5 6 8 23 33 44 **65**
Шаг 8-й	1 5 6 8 23 33 44 65

4.4. Сортировка разделением (Quicksort)

Метод сортировки разделением был предложен Чарльзом Хоаром в 1962 г. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть «методом быстрой сортировки – Quicksort».

Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент массива *x*, после чего массив просматривается слева, пока не встретится элемент a[i], такой что a[i] > *x*, а затем массив просматривается справа, пока не встретится элемент a[j], такой что a[j] < *x*. Эти два элемента меняются местами, и процесс просмотра, сравнения и обмена продолжается, пока мы не дойдем до элемента *x*. В результате массив окажется разбитым

на две части – левую, в которой значения ключей будут меньше *x*, и правую, со значениями ключей, большими *x*. Далее процесс рекурсивно продолжается для левой и правой частей массива до тех пор, пока каждая часть не будет содержать в точности один элемент. Понятно, что, как обычно, рекурсию можно заменить итерациями, если запоминать соответствующие индексы массива. Проследим этот процесс на примере нашего стандартного массива (табл. 8).

Таблица 8

Пример быстрой сортировки

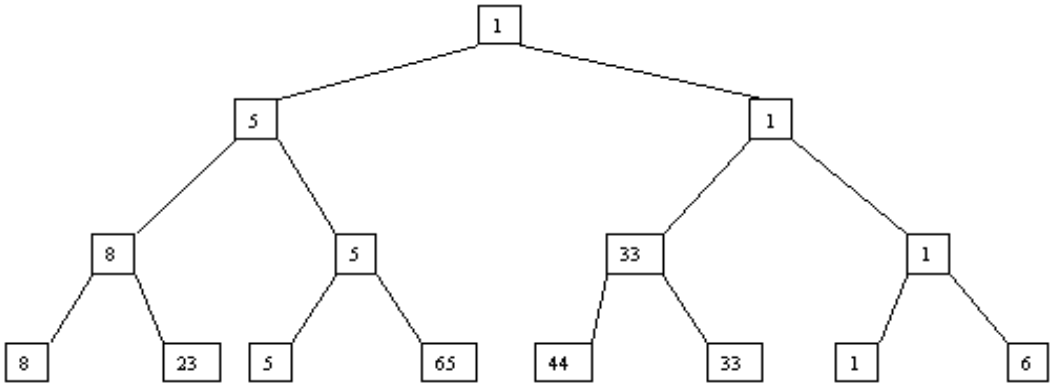
Начальное состояние массива	8 23 5 65 |44| 33 1 6
Шаг 1-й (в качестве x выбирается a[5])	|-------| 8 23 5 6 44 33 1 65 |-----| 8 23 5 6 1 33 44 65
Шаг 2-й (в подмассиве a[1], a[5] в качестве x выбирается a[3])	8 23 |5| 6 1 33 44 65 |----------| 1 23 5 6 8 33 44 65 |--| 1 5 23 6 8 33 44 65
Шаг 3-й (в подмассиве a[3], a[5] в качестве x выбирается a[4])	1 5 23 |6| 8 33 44 65 |------| 1 5 8 6 23 33 44 65
Шаг 4-й (в подмассиве a[3], a[4] выбирается a[4])	1 5 8 |6| 23 33 44 65 |--| 1 5 6 8 23 33 44 65

На самом деле, в большинстве утилит, выполняющих сортировку массивов, используется именно этот алгоритм.

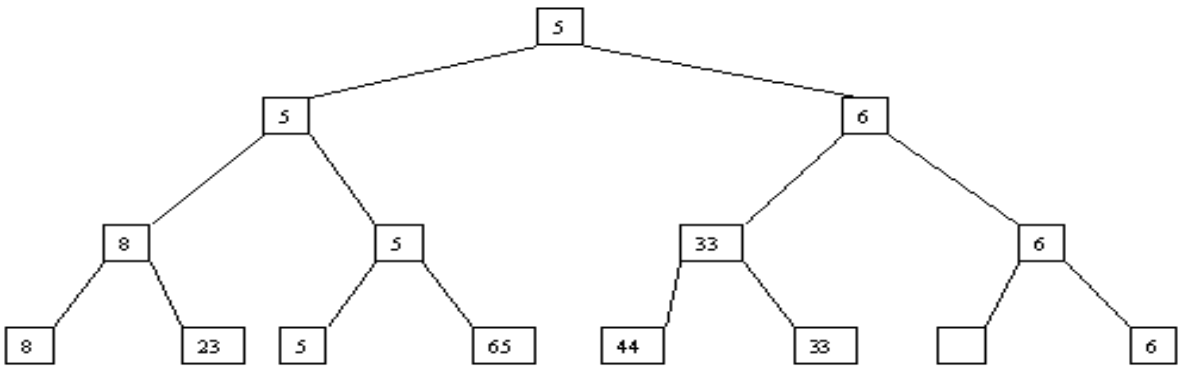
4.5. Сортировка с помощью дерева (Heapsort)

Начнем с простого метода сортировки с помощью дерева, при использовании которого явно строится двоичное дерево сравнения ключей. Построение дерева начинается с листьев, которые содержат все элементы массива. Из каждой соседней пары выбирается наименьший элемент, и эти элементы образуют следующий (ближе к корню уровень дерева). Из каждой соседней пары выбирается наименьший элемент и т.д., пока не будет построен корень, содержащий наименьший элемент массива. Двоичное дерево сравнения для масси-

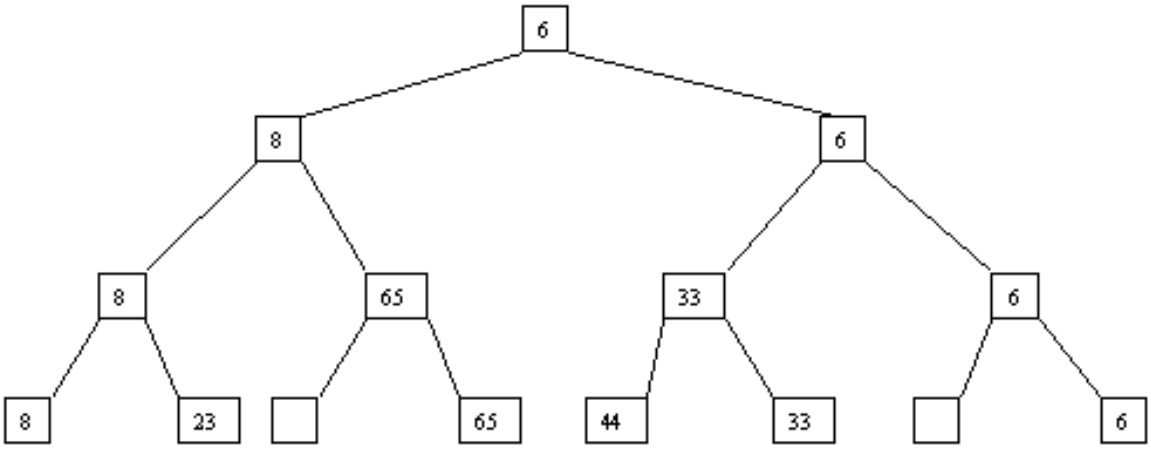
ва, используемого в примерах, показано на рис. 17. Итак, уже имеется наименьшее значение элементов массива. Для того чтобы получить следующий по величине элемент, спустимся от корня по пути, ведущему к листу с наименьшим значением. В этой листовой вершине проставляется фиктивный ключ с «бесконечно большим» значением, а во все промежуточные узлы, занимавшиеся наименьшим элементом, заносится наименьшее значение из узлов – непосредственных потомков (рис. 17, *а*, *б*, *в*, *г*, *д*, *е*, *ж*, *з*). Процесс продолжается до тех пор, пока все узлы дерева не будут заполнены фиктивными ключами.



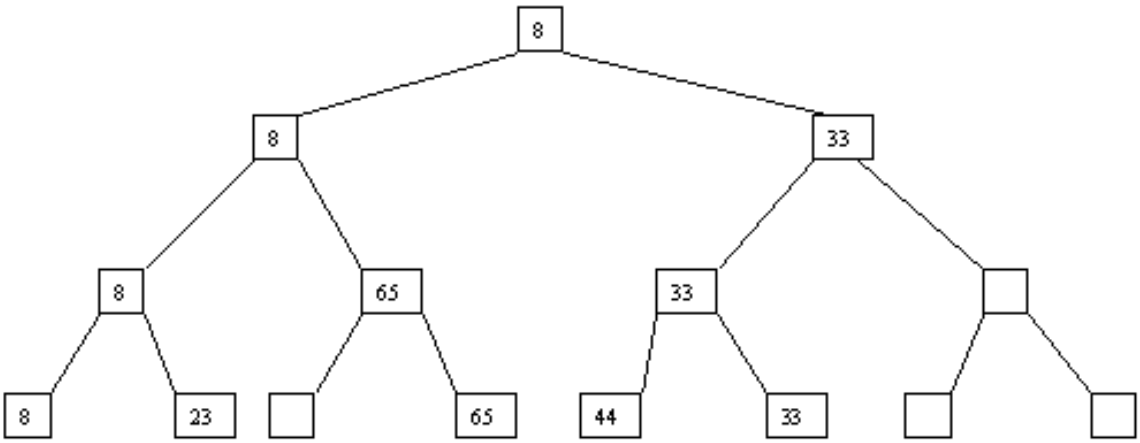
а



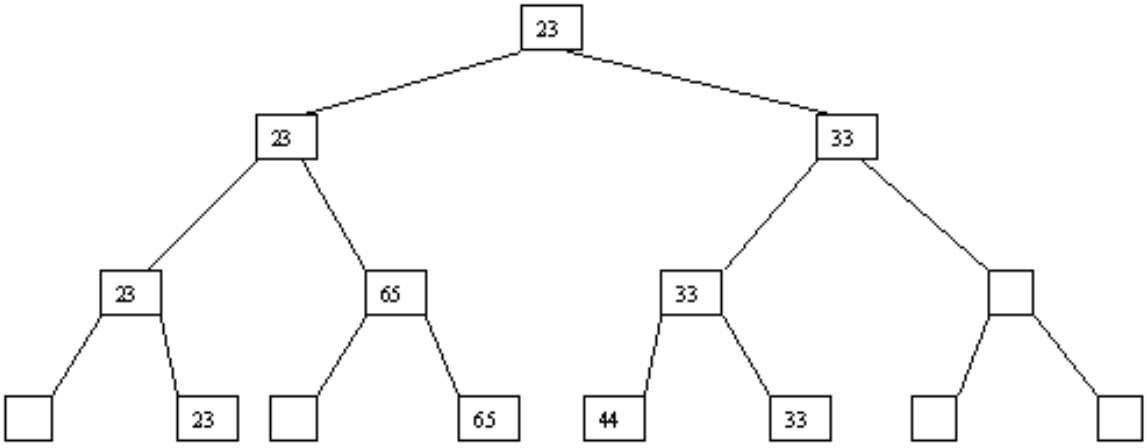
б


6
8
6
8
65
33
6
8
23
65
44
33
6

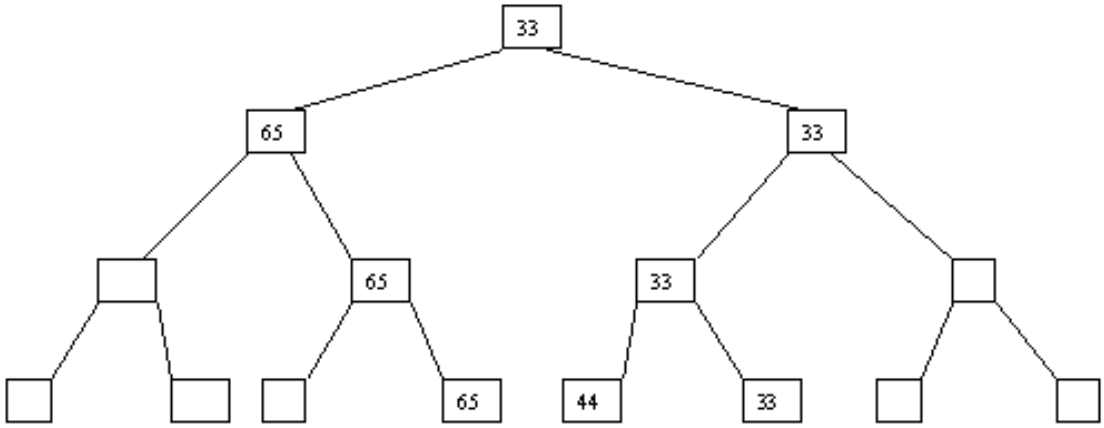
в


8
8
33
8
65
33
8
23
65
44
33

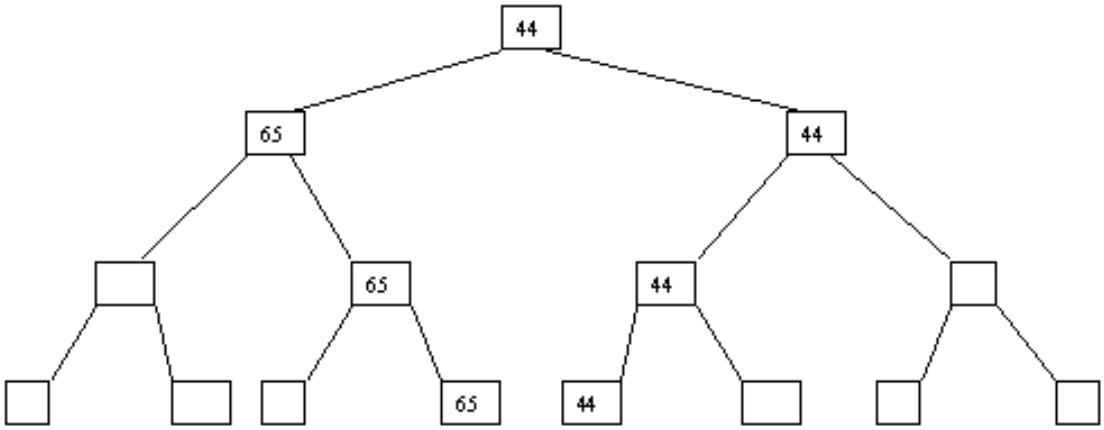
г


23
23
33
23
65
33
23
65
44
33

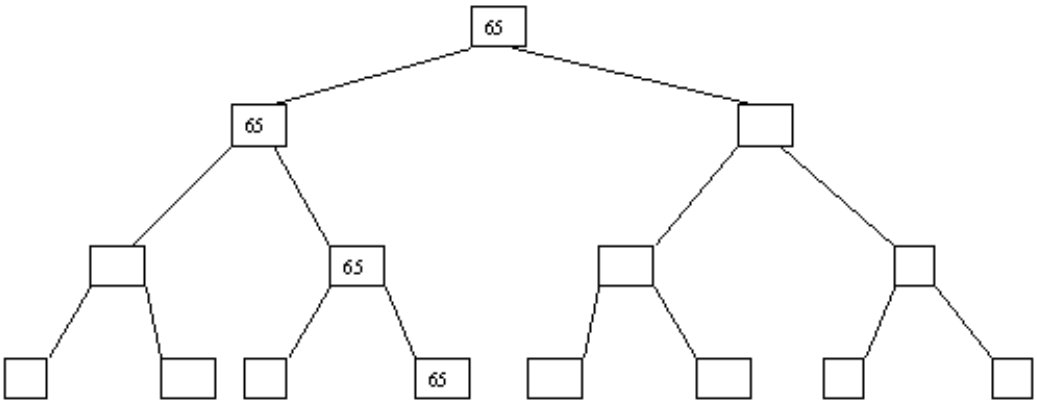
д



е



ж



з

Рис. 17. Сортировка с помощью дерева (Heapsort) пошагово

Имеется более совершенный алгоритм, который принято называть пирамидальной сортировкой (Heapsort). Его идея состоит в том,

что вместо полного дерева сравнения исходный массив a[1], a[2], ..., a[n] преобразуется в пирамиду, обладающую тем свойством, что для каждого a[i] выполняются условия a[i] <= a[2i] и a[i] <= a[2i+1]. Затем пирамида используется для сортировки.

Наиболее наглядно метод построения пирамиды выглядит при древовидном представлении массива, показанном на рис. 18. Массив представляется в виде двоичного дерева, корень которого соответствует элементу массива a[1]. На втором ярусе находятся элементы a[2] и a[3]. На третьем – a[4], a[5], a[6], a[7] и т.д. Как видно, для массива с нечетным количеством элементов соответствующее дерево будет сбалансированным, а для массива с четным количеством элементов n элемент a[n] будет единственным (самым левым) листом «почти» сбалансированного дерева.

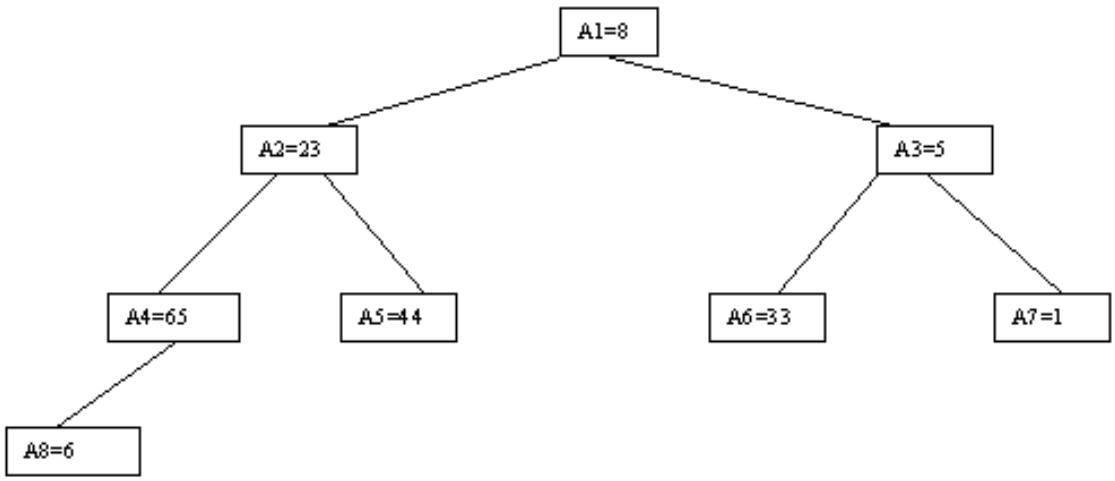
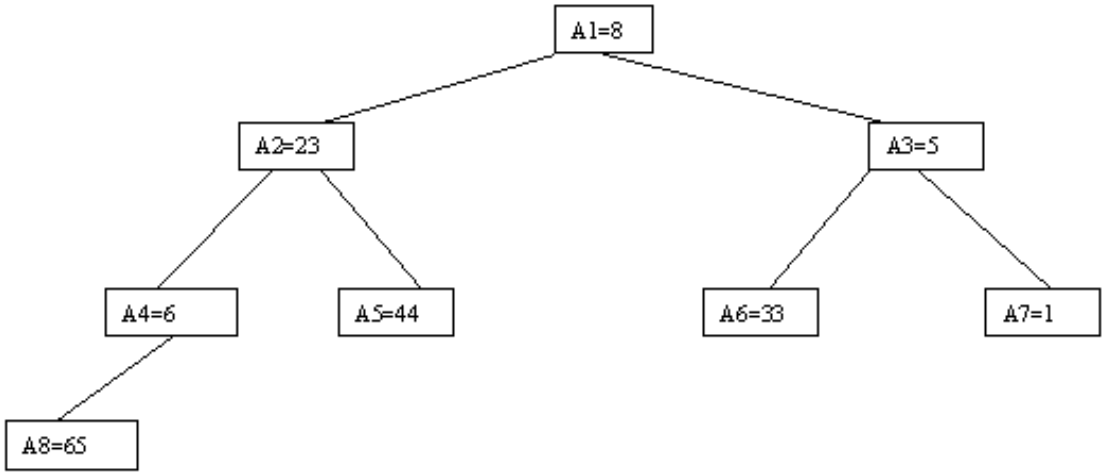


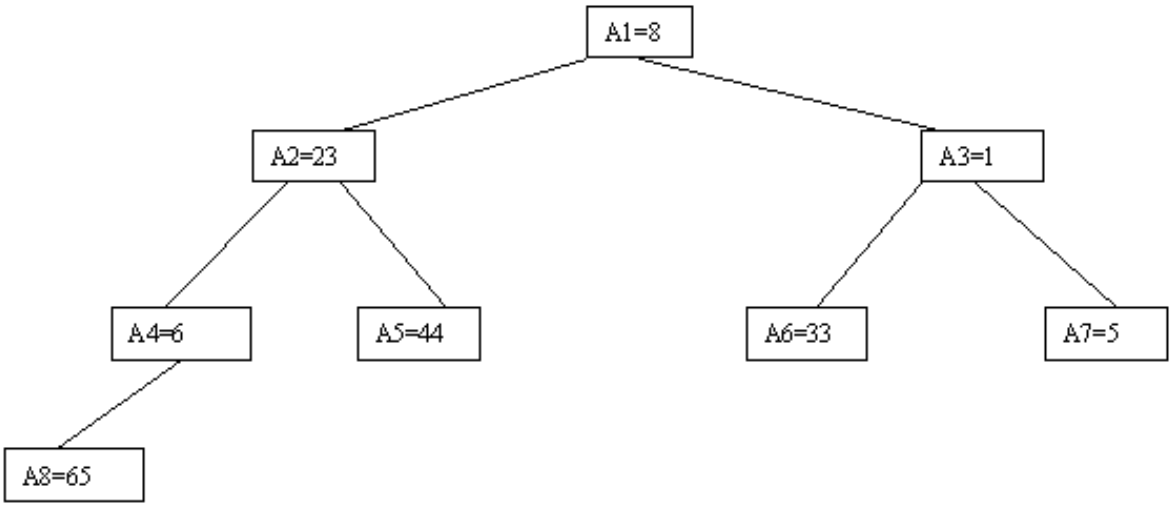
Рис. 18. Метод построения пирамиды

Очевидно, что при построении пирамиды нас будут интересовать элементы a[n/2], a[n/2-1], ..., a[1] – для массивов с четным числом элементов и элементы a[(n-1)/2], a[(n-1)/2-1], ..., a[1] – для массивов с нечетным числом элементов (поскольку только для таких элементов существенны ограничения пирамиды). Пусть i – наибольший индекс из числа индексов элементов, для которых существенны ограничения пирамиды. Тогда берется элемент a[i] в построенном дереве и для него выполняется процедура просеивания, состоящая в том, что выбирается ветвь дерева, соответствующая min(a[2i], a[2i+1]), и значение a[i] меняется местами со значением соответствующего элемента.

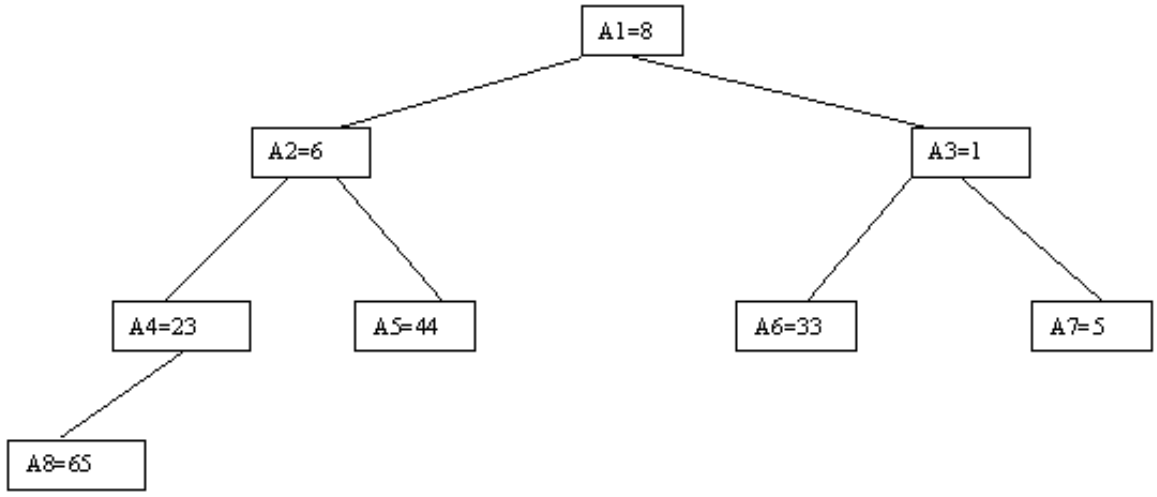
Если этот элемент не является листом дерева, для него выполняется аналогичная процедура и т.д. Такие действия выполняются последовательно для a[i], a[i-1], ..., a[1]. Легко увидеть, что в результате получается древовидное представление пирамиды для исходного массива (последовательность шагов для используемого в наших примерах массива показана на рис. 19, *а*, *б*, *в*, *г*).



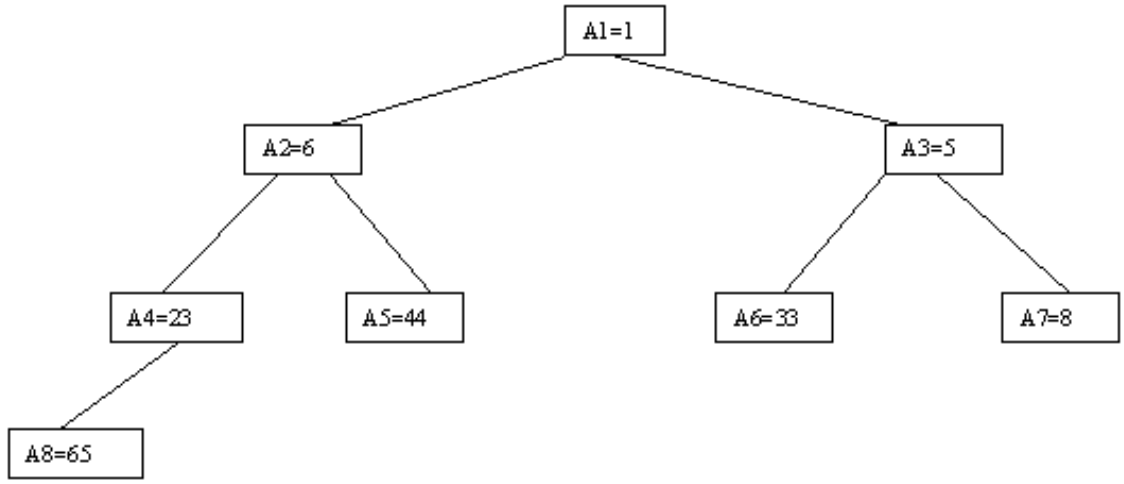
а



б



в



г

Рис. 19. Сортировка через древовидное представление пирамиды

В 1964 г. Флойд предложил метод построения пирамиды без явного построения дерева (хотя метод основан на тех же идеях). Построение пирамиды методом Флойда для нашего стандартного массива показано в табл. 9.

Таблица 9

Пример построения пирамиды

Начальное состояние массива	8 23 5 |65| 44 33 1 6
Шаг 1-й	8 23 |5| 6 44 33 1 65
Шаг 2-й	8 |23| 1 6 44 33 5 65
Шаг 3-й	|8| 6 1 23 44 33 5 65
Шаг 4-й	1 6 8 23 44 33 5 65 1 6 5 23 44 33 8 65

В табл. 10 показано, как производится сортировка с использованием построенной пирамиды. Суть алгоритма заключается в следующем. Пусть i – наибольший индекс массива, для которого существенны условия пирамиды. Тогда, начиная с a[1] до a[i], выполняются следующие действия. На каждом шаге выбирается последний элемент пирамиды (в нашем случае первым будет выбран элемент a[8]). Его значение меняется со значением a[1], после чего для a[1] выполняется просеивание. При этом на каждом шаге число элементов в пирамиде уменьшается на 1 (после первого шага в качестве элементов пирамиды рассматриваются a[1], a[2], ..., a[n-1]; после второго – a[1], a[2], ..., a[n-2] и т.д., пока в пирамиде не останется один элемент). В результате мы получим массив, упорядоченный в порядке убывания. Можно модифицировать метод построения пирамиды и сортировки, чтобы получить упорядочение в порядке возрастания, если изменить условие пирамиды на a[i] >= a[2i] и a[1] >= a[2i+1] для всех осмысленных значений индекса i.

Таблица 10

Сортировка с помощью пирамиды

Исходная пирамида	1 6 5 23 44 33 8 65
Шаг 1-й	65 6 5 23 44 33 8 1 5 6 65 23 44 33 8 1 5 6 8 23 44 33 65 1
Шаг 2-й	65 6 8 23 44 33 5 1 6 65 8 23 44 33 5 1 6 23 8 65 44 33 5 1
Шаг 3-й	33 23 8 65 44 6 5 1 8 23 33 65 44 6 5 1
Шаг 4-й	44 23 33 65 8 6 5 1 23 44 33 65 8 6 5 1
Шаг 5-й	65 44 33 23 8 6 5 1 33 44 65 23 8 6 5 1
Шаг 6-й	65 44 33 23 8 6 5 1 44 65 33 23 8 6 5 1
Шаг 7-й	65 44 33 23 8 6 5 1

Процедура сортировки с использованием пирамиды особо привлекательна для сортировки больших массивов.

4.6. Сортировка со слиянием

Сортировки со слиянием, как правило, применяются в тех случаях, когда требуется отсортировать последовательный файл, не помещающийся целиком в основной памяти. Однако существуют и эффективные методы внутренней сортировки, основанные на разбиениях и слияниях.

Один из популярных алгоритмов внутренней сортировки со слияниями основан на следующих идеях (для простоты будем считать, что число элементов в массиве, как и в нашем примере, является степенью числа 2). Сначала поясним, что такое слияние. Пусть имеются два отсортированных в порядке возрастания массива p[1], p[2], ..., p[n] и q[1], q[2], ..., q[n] и имеется пустой массив r[1], r[2], ..., r[2n], который мы хотим заполнить значениями массивов p и q в порядке возрастания. Для слияния выполняются следующие действия: сравниваются p[1] и q[1], и меньшее из значений записывается в r[1]. Предположим, что это значение p[1]. Тогда p[2] сравнивается с q[1] и меньшее из значений заносится в r[2]. Предположим, что это значение q[1]. Тогда на следующем шаге сравниваются значения p[2] и q[2] и т.д., пока не достигнутся границы одного из массивов. Тогда остаток другого массива просто дописывается в «хвост» массива r.

Пример слияния двух массивов показан на рис. 20.

Для сортировки со слиянием массива a[1], a[2], ..., a[n] заводится парный массив b[1], b[2], ..., b[n]. На первом шаге производится слияние a[1] и a[n] с размещением результата в b[1], b[2], слияние a[2] и a[n-1] с размещением результата в b[3], b[4], ..., слияние a[n/2] и a[n/2+1] с помещением результата в b[n-1], b[n]. На втором шаге производится слияние пар b[1], b[2] и b[n-1], b[n] с помещением результата в a[1], a[2], a[3], a[4], слияние пар b[3], b[4] и b[n-3], b[n-2] с помещением результата в a[5], a[6], a[7], a[8], ..., слияние пар b[n/2-1], b[n/2] и b[n/2+1], b[n/2+2] с помещением результата в a[n-3], a[n-2], a[n-1], a[n]. И т.д. На последнем шаге, например (в зависимости от значения n), производится слияние последовательностей элементов массива длиной n/2 a[1], a[2], ..., a[n/2] и a[n/2+1], a[n/2+2], ..., a[n] с помещением результата в b[1], b[2], ..., b[n].

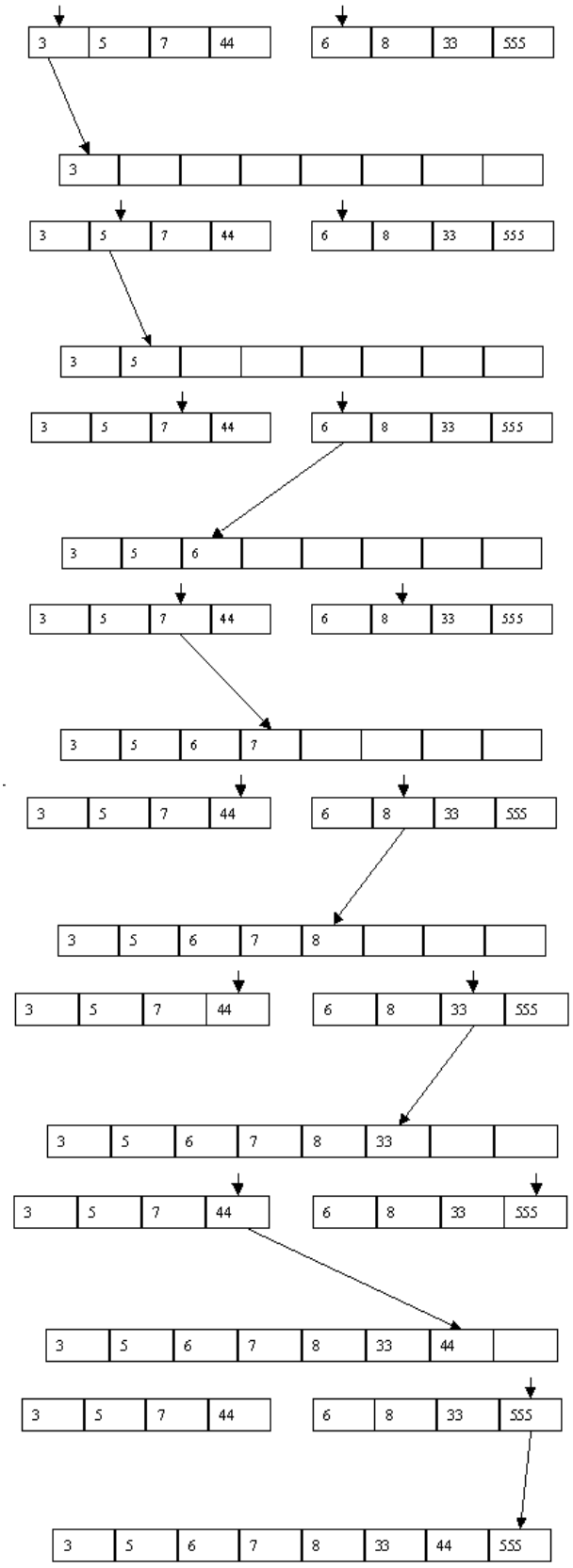


Рис. 20. Пример слияния двух массивов

Для случая массива, используемого в наших примерах, последовательность шагов показана в табл. 11.

Таблица 11

Пример сортировки со слиянием

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1-й	6 8 1 23 5 33 44 65
Шаг 2-й	6 8 44 65 1 5 23 33
Шаг 3-й	1 5 6 8 23 33 44 65

4.7. Сравнение методов внутренней сортировки

Для рассмотренных в начале этой главы простых методов сортировки существуют точные формулы, вычисление которых дает минимальное, максимальное и среднее число сравнений ключей (C) и пересылок элементов массива (M) (табл. 12)

Таблица 12

Характеристики простых методов сортировки

	Min	**Avg**	**Max**
Прямое включение	$C=n-1$ $M=2(n-1)$	$(n^2+n-2)/4$ $(n^2-9n-10)/4$	$(n^2-n)/2-1$ $(n^2-3n-4)/2$
Прямой выбор	$C=(n^2-n)/2$ $M=3(n-1)$	$(n^2-n)/2$ $n(\ln n+0.57)$	$(n^2-n)/2$ $n^2/4+3(n-1)$
Прямой обмен	$C=(n^2-n)/2$ $M=0$	$(n^2-n)/2$ $(n^2-n)0.75$	$(n^2-n)/2$ $(n^2-n)1.5$

Для оценок сложности усовершенствованных методов сортировки точных формул нет.

4.8. Задания для выполнения

1. Отсортируйте следующий список, используя рассмотренные методы внутренней сортировки, как в порядке возрастания, так и в порядке убывания:

1.1. 503, 087, 512, 061, 908, 170, 897, 275, 653, 426, 154, 509, 612, 677, 765, 703;

1.2. Последовательность игральных карт: 8, 7, К, 10, Д, 3, 6, 5, 9, 2, Т, В, 4;

1.3. 17, 23, 114, 28, 35, 72, 16, 84, 90, 65, 33, 43, 67, 100, 56, 77;

1.4. Последовательность букв: A, R, E. W, O, G, P, B, M, C, D, T, J, H, S, L;

1.5. 38, 8, 16, 6, 79, 76, 57, 24, 56, 2, 58, 48, 4, 70, 46, 47;

1.6. Последовательность игральных карт: 2, В, Т, К, 5, 9, 3, 8, Д, 10, 4, 6, 7;

1.7. 27, 9, 14, 8, 5, 11, 7, 2, 3, 13, 35, 1, 45, 4, 25, 30;

1.8. Последовательность букв: В, У, А, Н, Г, Ы, Щ, Ж, К, М, Ю, Т, З, Ч, Ф, О.

Для каждого алгоритма постройте блок-схему.

На основе анализа и экспериментов установите, какой из методов наиболее эффективный (дайте критерий) при различных диапазонах значений N.

2. В алгоритмах сортировки предполагалось, что все сортируемые ключи неотрицательны. Какие изменения необходимо внести в эти алгоритмы в том случае, когда ключами могут быть и отрицательные числа ?

3. Дан массив чисел a(1), a(2),..., a (n). Расположить числа в массиве так, чтобы в начале массива В были все положительные числа, а в конце – все отрицательные.

4. Даны 2 упорядоченных в порядке возрастания (убывания) массива чисел. В первом – 40 чисел, во втором – 50. Составить алгоритм и программу слияния этих массивов в один упорядоченный в порядке возрастания (убывания) массив.

5. Написать программу формирования ведомости об успеваемости студентов. Каждая запись этой ведомости должна содержать номер группы, фамилию студента, средний балл за последнюю сессию В каждой группе фамилии студентов разместить в порядке убывания среднего балла.

6. Составить программу, которая из произвольной строки, содержащей некоторый текст, выделяет все слова и печатает их в алфавитном порядке (по первой букве).

7. Из двух упорядоченных одномерных массивов (длины K и N) сформируйте массив размером K+N, упорядоченный:

а) так же, как исходные массивы;

б) в обратную сторону.

Используйте упорядоченность исходных массивов.

5. АЛГОРИТМЫ МАШИННОЙ ГРАФИКИ

5.1. Краткие теоретические сведения

Алгоритмы машинной графики можно разделить на два уровня: **нижний** и **верхний**. Группа алгоритмов нижнего уровня предназначена для реализации графических примитивов (линий, окружностей, заполнений и т.п.). Эти алгоритмы или подобные им воспроизведены в графических библиотеках языков высокого уровня или реализованы аппаратно в графических процессорах рабочих станций.

Среди алгоритмов **нижнего** уровня можно выделить следующие группы:

Простейшие – в смысле используемых математических методов и отличающиеся простотой реализации. Как правило, такие алгоритмы не являются наилучшими по объему выполняемых вычислений или требуемым ресурсам памяти.

Поэтому можно выделить ***вторую*** группу алгоритмов, использующих более сложные математические предпосылки (но часто и эвристические) и отличающихся большей эффективностью.

К ***третьей*** группе следует отнести алгоритмы, которые могут быть без больших затруднений реализованы аппаратно (допускающие распараллеливание, рекурсивные, реализуемые в простейших командах). В эту группу могут попасть и алгоритмы, представленные в первых двух группах.

Наконец, к ***четвертой*** группе можно отнести алгоритмы со специальным назначением (например, для устранения лестничного эффекта).

К алгоритмам ***верхнего*** уровня относятся в первую очередь алгоритмы удаления невидимых линий и поверхностей. Задача удаления невидимых линий и поверхностей продолжает оставаться центральной в машинной графике. От эффективности алгоритмов, позволяющих решить эту задачу, зависят качество и скорость построения трехмерного изображения.

К задаче удаления невидимых линий и поверхностей примыкает задача построения (закрашивания) полутоновых (реалистических) изображений, т.е. учета явлений, связанных с количеством и характером источников света, а также свойств поверхности тела (прозрачность, преломление, отражение света).

Однако при этом не следует забывать, что вывод объектов в алгоритмах верхнего уровня обеспечивается примитивами, реализующими алгоритмы нижнего уровня, поэтому нельзя игнорировать проблему выбора и разработки эффективных алгоритмов нижнего уровня.

Для разных областей применения машинной графики на первый план могут выдвигаться разные свойства алгоритмов. Для научной графики большое значение имеет универсальность алгоритма, быстродействие может отходить на второй план. Для систем моделирования, воспроизводящих движущиеся объекты, быстродействие становится главным критерием, поскольку требуется генерировать изображение практически в реальном масштабе времени.

Особенности растровой графики связаны с тем, что обычные изображения, с которыми сталкивается человек в своей деятельности (чертежи, графики, карты, художественные картины и т.п.), реализованы на плоскости, состоящей из бесконечного набора точек. Экран же растрового дисплея представляется матрицей дискретных элементов, имеющих конкретные физические размеры. При этом их число существенно ограничено. Поэтому нельзя провести точную линию из одной точки в другую, а можно выполнить только аппроксимацию этой линии с отображением ее на дискретной матрице (плоскости). Такую плоскость называют также целочисленной решеткой, растровой плоскостью или растром. Эта решетка представляется квадратной сеткой с шагом 1. Отображение любого объекта на целочисленную решетку называется разложением его в растр или просто растровым представлением.

Построение линий, окружностей, эллипсов

Проще всего начертить линию можно с помощью уравнения y = ax + b. При этом результаты надо округлять до целых, поэтому прямая будет неровная.

Если наоборот, нужно соединить 2 точки с заданными координатами (x1, y1), (x2, y2), то:

$$a = (y2-y1) / (x2-x1);\ \ b = y1 - a*x1.$$

Уравнение окружности выглядит следующим образом:

$$x = xc + r*\cos(a);\ \ y = yc + r*\sin(a),$$

где (xc, yc) – координаты центра окружности; r – радиус; a – угол для текущей точки (x, y).

Можно строить окружность прямо по этому уравнению, задав определенный шаг по углу (a = 0..360 с шагом DA). Однако если шаг

будет слишком мал, окружность за счет округления координат будет неровная и некоторые точки будут высвечиваться по несколько раз. Обычно шаг по углу выбирается равным 1/r радиан (чаще всего шаг изменения угла должен быть переменным для того, чтобы избежать разрывов или отсутствия изменения координат).

Можно осуществить простой алгоритм аппроксимации отрезками.

Например, координаты шести отрезков получаются с шагом 60^0 , затем они соединяются прямыми линиями.

Для быстрого построения используется симметрия окружности (вычисляются координаты точек только 1/8 части окружности – для сегмента от 0 до 45 градусов). Кроме того, можно уйти от операций синуса/косинуса, если выразить координаты следующей точки окружности из предыдущей (рекуррентная формула):

$$x2 = xc + (x1-xc) * CA + (y1-yc) * SA;$$
$$y2 = yc + (y1-yc) * CA - (x1-xc) * SA,$$

где SA = sin(DA), CA = cos(DA).

Начальная точка для рекуррентной формулы: x1 = xc, y1= yc + r.

При построении окружностей следует учитывать, что размеры пикселов по вертикали и горизонтали не совпадают (кроме VGA 640x480) и окружности будут вытягиваться в эллипсы. Для того чтобы избежать этого, нужно вводить по выравнивающие коэффициенты (которые можно определить по GetAspectRatio). Сами же эллипсы описываются уравнениями:

$$x = xc + rx * \cos(a); \quad y = yc + ry * \sin(a),$$

где rx, ry – полурадиусы.

Общие принципы построения для них те же, что и для окружностей.

Алгоритм Брезенхема

Алгоритм Брезенхема (Bresenham) был разработан в 1965 году для цифровых графопостроителей, а затем стал использоваться для растровых дисплеев.

Идея алгоритма заключается в том, что одна координата изменяется на единицу, а другая либо не изменяется, либо изменяется на единицу в зависимости от расположения соответствующей точки от ближайшего узла координатной сетки. Расстояние от точки отрезка до ближайшего узла по соответствующей ортогональной координате называется ошибкой. Алгоритм организован таким образом, что для вычисления второй координаты требуется только определять знак

этой ошибки. Величину ошибки Delta можно определить в соответствии со следующим выражением: Delta = Yуз - Yот,
где Yуз – координата Y ближайшего узла при X = 1; Yот – координата Y отрезка при X = 1.

Если при X = 1 координата Y точки отрезка равна 1/2, то узлы координатной сетки (1,0) и (1,1) находятся на одинаковом расстоянии от отрезка и в качестве «ближайшего» выбирается узел (1,1). Таким образом, если Yот >= 1/2, то Delta >=0, в противном случае, при Yот < 1/2, Delta < 0. Для организации вычислений удобнее пользоваться не величиной Delta, а другой, определяемой как величина d = Delta - 1/2. При изменении координаты X на 1 величина d меняется на значение углового коэффициента: d = d + dY / dX. На каждом шаге алгоритма производится вычисление координаты X, величины d и выполняется анализ знака d. При этом, если окажется, что d >= 0, то производится увеличение Y на 1, а значение d корректируется путем вычитания из нее 1.

С учетом изложенного, алгоритм аппроксимации отрезка в первом октанте можно представить в следующем виде :

```
X := X1; Y : =  Y1;
dX := X2 - X1;
dY := Y2 - Y1;
d : = - 1/2;
while X =< X2 do
 PutPixel (X, Y);
 X := X + 1;
 d := d + dY  / dX;
 if d >= 0 then
 begin
  Y := Y +  1;
  d := d -  1
 end
end while.
```

Представленный алгоритм использует вещественные числа и операцию деления. Оба недостатка можно исключить заменой величины d на другую, равную D = 2 * dX * d. В соответствии с этим арифметические выражения, в которых участвует d, модифицируются путем умножения обеих частей на величину 2 * dX.

Тогда алгоритм принимает вид :

```
X := X1; Y := Y1;
dX := X2 - X1;
dY := Y2 - Y1;
D := - dX;
while X =< X2 do
 PutPixel (X, Y);
 X := X + 1;
 D := D + 2  * dY;
 if D >= 0 then
 begin
  Y := Y +  1;
  D := D -  2 * dX;
 end
end while.
```

Последний вариант алгоритма можно еще улучшить, если операцию умножения на 2 заменить операцией сдвига влево на один разряд. Кроме того, если вычисление 2 * dX и 2 * dY выполнить перед циклом, то операцию сдвига потребуется выполнить только два раза.

Тогда алгоритм принимает окончательный вид :

```
X := X1; Y := Y1;
dX := X2 - X1;
dY := Y2 - Y1;
D := - dX;
Dx := dX shl 1;
Dy := dY shl 1;
while X =< X2 do
 PutPixel (X, Y);
 X := X + 1;
 D := D + Dy;
 if D >= 0 then
 begin
  Y := Y +  1;
  D := D -  Dx
 end
end while.
```

Оценивая достоинства полученного алгоритма, можно отметить, что он выполняет оптимальную аппроксимацию отрезка, используя при этом целочисленную арифметику и минимальное количество операций сложения и вычитания. Кроме того, алгоритм позволяет использовать его и в остальных октантах плоскости.

В алгоритме Брезенхема для всех октантов линия рассматривается как набор сегментов двух типов: тех, которые расположены диагонально, и тех, которые расположены горизонтально или вертикально. Для линий с наклоном больше 1 прямые сегменты вертикальны, <= 1 – горизонтальны. Первая задача алгоритма состоит в вычислении наклона. Затем вычисляется выравнивающий фактор, который следит, чтобы некоторое число прямых сегментов имело большую длину, чем остальные. В цикле по большему отрезку (по x или по y) BX поочередно принимает то положительные, то отрицательные значения, отмечая, какой тип сегмента выводится – диагональный или прямой.

В алгоритме проводится линия от (x1, y1) к (x2, y2). Используются некоторые регистры и переменные, назначение которых комментируется.

Начало алгоритма.
CX=1, DX=1 – начальные инкременты по X и по Y
{ вычисление вертикальной дистанции }
DI=y2-y1 > 0 ? Если да, то на шаг 4, иначе – на след.
DX=-1, DI=-DI (дистанция по Y должна быть > 0)
DIAG_Y=DX - приращение по Y для диагональных эл-тов
{ вычисление горизонтальной дистанции }
SI=x2-x1 > 0 ? Если да, то на шаг 7, иначе – на след.
CX=-1, SI=-SI (дистанция по X должна быть > 0)
DIAG_X=CX – приращение по X для диагональных эл-тов
{ вертикальны или горизонтальны прямые сегменты }
SI >= DI ? Если да, то на шаг 9, иначе шаг 10.
DX=0 – для прямых сегментов Y не меняется (т.е. они горизонтальны).
Переход на шаг 11.
CX=0 – прямые сегменты вертикальны.
SI <-> DI – обмен SWAP, чтобы больший отрезок был в SI.
{ вычисление выравнивающего фактора }
SHORT=DI – более короткий отрезок
STR_X=CX, STR_Y=DX – инкременты для прямых сегментов (один из них = 0)
STR_COUNT = 2*SHORT
DIAG_COUNT= 2*SHORT - 2*SI
BX=2*SHORT-SI – начальное значение переключателя
{ подготовка к выводу линии }

CX=x1, DX=y1 – начальные координаты
SI=SI+1 - прибавить один пиксел для конца отрезка
{ вывод линии }
SI=SI-1
SI = 0 ? Если да, то конец алгоритма – шаг 19.

Вывод точки с координатами CX, DX через BIOS или ПДП.
BX < 0 ? Если да, то шаг 17, иначе 18.
{ следующий сегмент прямой – подготовка координат для вывода }
CX=CX+STR_X
DX=DX+STR_Y
BX=BX+STR_COUNT
переход на шаг 13
{ след. сегмент диагональный }
CX=CX+DIAG_X
DX=DX+DIAG_Y
BX=BX+DIAG_COUNT
переход на шаг 13
Конец алгоритма.

Для примера рассмотрим проведение линии по алгоритму в декартовых координатах из точки (0, 0) в точку (10, 6). Наклон прямой < 1, значит, все сегменты либо горизонтальны (STR_X=1, STR_Y=0), либо диагональным (DIAG_X = 1, DIAG_Y = 1). Больше расстояние по X: SI = 10 (цикл по нему). Выравнивающий фактор: STR_COUNT = = 12, DIAG_COUNT = -8. Начальное значение BX = 2, следовательно, первый сегмент будет диагональный.

Таблица значений по шагам:

BX	CX	DX		BX	CX	DX
1) 2	1	1	6)	2	6	4
2) -6	2	1	7)	-6	7	4
3) 6	3	2	8)	6	8	5
4) -2	4	2	9)	-2	9	5
5) 10	5	3	10)	10	10	6

Первая точка будет иметь координаты (0, 0), остальные – как указано в таблице (CX, DX). Нарисуйте получившуюся линию на клетчатой бумаге.

Заполнение сплошных областей

Заполнение областей может быть выполнено двумя способами – сканированием строк и затравочным заполнением.

Метод сканирования строк решает задачи определенного порядка сканирования, принадлежности точки внутренней области многоугольника или контура. При затравочном заполнении предполагается наличие некоторой точки, принадлежащей внутренней области (затравочная точка), для которой определяют соседние точки и включают ее в список других затравочных точек. Список этих точек анализируется с точки зрения необходимости их закрашивания.

Области могут задаваться внутренне определенным и гранично-определенным способом. Внутренне определенная область состоит из точек только одного цвета или интенсивности. Пикселы, внешние по отношению к точкам внутренне определенной области, имеют отличную от них интенсивность или цвет.

Гранично-определенные области имеют контур, состоящий из пикселов с выделенным значением цвета или интенсивности. Внутренние пикселы области отличны от них по цвету или интенсивности, а внешние могут с ними совпадать.

Внутренне и гранично-определенные области могут быть четырех- и восьмисвязными. В четырехсвязных областях любой пиксел достигается движением по четырем взаимно перпендикулярным направлениям, а восьмисвязных – по восьми: горизонтальным, вертикальным и диагональным.

Можно сформировать простейший алгоритм затравочного заполнения четырехсвязной гранично-определенной области в следующем виде:

```
var
 X,Y: integer: { координаты  затравочной точки}
 oldcolor: word: { исходное  значение цвета}
 newcolor: word: { новое  значение цвета}
 framcolor: word: { цвет  границы}
begin
 помещаем пиксел в Stack;
 while (Stack не пуст)  do
 begin
  извлекаем пиксел из  Stack;
  PutPixel (X, Y,  newcolor) ;
  для точек (Xт,  Yт), соседних с (X, Y):
  (X+1, Y); (X,  Y+1); (X-1, Y); (X, Y-1),
  проверяем условие:
  if GetPixel (Xт,  Yт) = newcolor and
   GetPixel (Xт,  Yт) = framcolor
  then помещаем (Xт,  Yт) в Stack;
 end
end.
```

Аналогично строится алгоритм и для внутренне определенной области. В этом случае меняется только условный оператор, который должен проверять принадлежность соседних точек и наличие неизмененного цвета.

Предложенные алгоритмы легко обобщаются на случай восьмисвязных областей, для которых необходимо выполнять анализ восьми соседних точек.

Процедура затравочного заполнения областей, описанная выше, обладает двумя недостатками: стек заполняется большим количеством повторяющихся точек, что приводит к существенным временным затратам при их анализе и, как следствие, требуется стек большого объема. Однако эти проблемы легко разрешить изящным методом перестановки местами операций помещения точек в стек и их закраски.

Тогда процедура будет записана следующим образом :

```
begin
 PutPixel (X, Y, newcolor)  ;
 помещаем пиксел в Stack;
 while (Stack не пуст)  do
 begin
  извлекаем пиксел (X,  Y) из Stack;
  для точек (Xт,  Yт) , соседних с (X,  Y) :
   (X+ 1,Y); (X,  Y+1); (X-1, Y); (X, Y-1),
  проверяем условие:
  if GetPixel (Xт,  Yт) = newcolor and
   GetPixel (Xт,  Yт) = framcolor
  then
  begin
   PutPixel (Xт,  Yт, newcolor);
   помещаем (Xт,  Yт) в Stack
  end
 end
end.
```

Другой метод повышения эффективности затравочного заполнения областей заключается в организации построчного заполнения непрерывного интервала на сканирующей строке с одной затравочной точкой.

5.2. Задания для выполнения

1. Используя цифровой дифференциальный анализатор (ЦДА) и алгоритм Брезенхема, составьте блок-схему алгоритма и напишите программу для вычерчивания отрезков из одной произвольной точки

в другую на псевдорастре 32x32. Используйте псевдобуфер кадра в виде одномерного массива сначала для хранения изображения, а затем для вывода его из псевдобуфера на экран. Демонстрационный тест должен состоять, по крайней мере, из 16 отрезков с началом в центре окружности и концами, равномерно расположенными по окружности. Обеспечьте возможность устанавливать центр окружности в произвольную точку растра. Визуально сравните результаты. Напечатайте список активированных пикселов для отрезка из (0, 0) в (-8, -3) для обоих алгоритмов.

2. Окружность, разложенную в растр, можно получить с помощью алгоритма Брезенхема для генерации окружности. Ее можно также сгенерировать путем разложения в растр ребер вписанного многоугольника с помощью алгоритма Брезенхема для отрезка. Составьте блок-схему алгоритма и напишите программу, реализующую оба метода, разложите в растр окружность радиуса R=15 на псевдорастре 32x32. Сравните результаты для вписанных многоугольников с 4, 8, 16, 32, 64 и 128 сторонами для алгоритма генерации окружности. Используйте псевдобуфер кадра в виде одномерного массива сначала для хранения изображения, а затем для вывода его из псевдобуфера на экран. Предусмотрите выдачу списка растровых точек, используя формат «строка-колонка» для каждого алгоритма в предположении, что начало координат (0, 0) находится в левом нижнем углу. Результаты сравните визуально и численно.

3. Пусть задан многоугольник с внешней частью, определяемой точками (4, 4), (4, 26), (20, 26), (28, 18), (21, 4), (21, 8), (10, 8) и (10, 4) и внутренним отверстием, описываемым точками (10, 12), (10, 20), (17, 20), (21, 16) и (21, 12) на псевдорастре 32x32. Составьте блок-схему алгоритма и напишите программу, использующую простой алгоритм с упорядоченным списком ребер, для развертки и изображения сплошной области, лежащей внутри многоугольника. Предполагается, что начало координат (0, 0) находится в левом нижнем углу растра. Выведите список заполненных пикселов в формате «строка-колонка» в порядке сканирования сверху вниз и слева направо.

4. Для многоугольника из задачи 3 составьте блок-схему алгоритма и напишите программу, реализующую более эффективный алгоритм с упорядоченным списком ребер. Преобразуйте в растровую форму и изобразите сплошную область внутри многоугольника. Используйте список активных ребер и напечатайте содержимое этого списка для строки 18 в растре 32x32. Напечатайте пикселы в порядке сканирования изображения.

5. Составьте блок-схему алгоритма и напишите программу, реализующую простой гранично-заполняющий алгоритм с затравкой для заполнения внутренней части многоугольника из задачи 3. Обеспечьте выдачу списка граничных пикселов. Сгенерируйте и выдайте на печать список заполненных пикселов, если затравка – пиксел (14, 20). Обеспечьте возможность просмотра содержимого стека в любой момент.

ЛИТЕРАТУРА

1. Ахо, Альфред В. Структуры данных и алгоритмы / Альфред В. Ахо, Джон Е. Хопкрофт, Джеффри Д. Ульман. – М.: Издательский дом «Вильяме», 2003. – 341 с.

2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – М.: Мир, 1989. – 360 с.

3. Грин, Д. Математические методы анализа алгоритмов / Д. Грин, Д. Кнут. – М.: Мир, 1987. – 230 с.

4. Гудман, С. Введение в разработку и анализ алгоритмов / С. Гудман, С. Хидетниеми. – М.: Мир, 1981.

5. Гуц, А.К. Математическая логика и теория алгоритмов: учебное пособие / А.К. Гуц. – Омск: Наследие: Диалог-Сибирь, 2003. – 178 с.

6. Иванов, Б.Н. Дискретная математика. Алгоритмы и программы: учебное пособие / Б.Н. Иванов. – М.: Лаборатория Базовых Знаний, 2003. – 267 с.

7. Касьянов, В.Н. Графы в программировании: обработка, визуализация и применение / В.Н. Касьянов, В.А. Евстигнеев. – СПб.: БХВ-Петербург, 2003. – 237 с.

8. Кнут, Д. Искусство программирования для ЭВМ: в 3 т. / Д. Кнут. – М.: Мир, 1978. – 890 с.

9. Кнут, Д.Э. Искусство программирования. Т 3: Сортировка и поиск / Д. Кнут. – М.: Издательский дом «Вильямс», 2000. – 281 с.

10. Кубенский, А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ / А.А. Кубенский. – СПб.: БХВ-Петербург, 2004. – 246 с.

11. Курейчик, В.М. Математическое обеспечение конструкторского и технологического проектирования с применением САПР: учебник для вузов / В.М. Курейчик. – М.: Радио и связь, 1990. – 360 с.

12. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл. – М.: Техносфера, 2004. – 412 с.

13. Морозов, К.К. Автоматизированное проектирование конструкций радиоэлектронной аппаратуры: учебное пособие для вузов / К.К. Морозов, В.Г. Одиноков, В.М. Курейчик. – М.: Радио и связь, 1983. – 524 с.

14. Окулов, С.М. Программирование в алгоритмах / С.М. Окулов. – М.: БИНОМ: Лаборатория знаний, 2002. – 321 с.

15. Павлидис, Т. Алгоритмы машинной графики и обработки изображений / Т. Павлидис. – М.: Радио и связь, 1986. – 254 с.

16. Роджерс, Д. Алгоритмические основы машинной графики / Д. Роджерс. – М.: Мир, 1989. – 311 с.

Учебное издание

Лобанова Валентина Андреевна
Воронина Оксана Александровна
Лобанова Наталья Геннадьевна

ТЕОРИЯ АЛГОРИТМОВ

Учебное пособие

Редактор Г.В. Карпушина
Технический редактор Т.П. Прокудина

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Орловский государственный университет имени И.С. Тургенева»

Подписано к печати 05.09.2017 г. Формат 60×90 1/16.
Усл. печ. л. 5.9. Тираж 100 экз.
Заказ №_____

Отпечатано с готового оригинал-макета
на полиграфической базе ОГУ имени И.С. Тургенева
302026, г. Орел, ул. Комсомольская, 95.