**Тема 1. Полное построение алгоритма**

Основными этапами полного построения алгоритма являются:

1. Технические требования для решения задачи или постановка задачи.

2. Построение модели.

3. Разработка алгоритма.

4. Проверка правильности алгоритма.

5. Реализация алгоритма.

6. Анализ алгоритма и его сложности.

7. Проверка программы.

8. Составление документации.

**Алгоритмы**

Каждый, кто решал задачи с помощью ЭВМ, имеет некоторое интуитивное представление о значении слова «алгоритм». Кое-кто даже утверждает, что это одно из важнейших понятий вычислительной математики. «Официальное» определение из последнего издания Оксфордского словаря английского языка гласит, что «algorithm» — это ошибочное написание слова «algorism», в свою очередь algorism счи¬тается более или менее синонимичным алгебре и арифметике, и это определение датируется девятым веком. Очевидно, современное употребление термина пока еще ограничено кругом людей, связанных с ЭВМ.

Попытаемся сначала выразить наши интуитивные соображения. Мы можем нестрого определить алгоритм как однозначно трактуемую процедуру решения задачи. Процедура — это конечная последовательность точно определенных шагов или операций, для выполнения каждой из которых требуется конечный объем оперативной памяти и конечное время. Дополнительно потребуем, чтобы алгоритм работал конечное время для любых входных данных.

Одно из неудобств этого определения состоит в том, что термин «однозначная трактовка» весьма неоднозначен. «Однозначен» для кого? Или по отношению к чему? Поскольку ничто не является абсолютно ясным или абсолютно неясным, должен быть указан, хотя бы неявно, исполнитель. Алгоритм вычисления производной кубического полинома может быть вполне ясен тем, кто знаком с анализом, но для прочих он может оказаться совершенно непонятным. Таким образом, следует также указать вычислительные возможности исполнителя.

Существуют и другие трудности с определением. Может случиться, что алгоритм заведомо существует для конкретной задачи, но его трудно или невозможно описать в некоторой заданной форме. Человечество разработало эффективный алгоритм завязывания шнурков на ботинках. Многие дети с пятилетнего возраста могут зашнуровывать свои ботинки. Но дать чисто словесное описание такого алгоритма без картинок и демонстрации — очень трудно (попробуйте).

Очевидно, что в нашем определении есть некоторые недостатки. Можно избежать большинства этих недостатков, определив «математические машины» с очень точно указанными возможностями. Тогда мы скажем, что алгоритм — это некоторая процедура, которая может выполняться такой машиной. Подобные попытки определения алгоритма очень глубоки и трудны математически, и они слишком негибки для наших целей.

Нам хотелось бы сохранить некоторую гибкость и интуитивную привлекательность первого определения и в то же время, хотя бы частично, устранить некоторые из его неопределенностей. Это легко сделать, описав «типичный» современный компьютер и язык общения с ним и определив затем алгоритм как процедуру, которая может быть реализована на этом компьютере при помощи данного языка.

Этот компьютер будет иметь неограниченную память с произвольным доступом, в которой могут храниться действительные и целые числа, а также логические константы. В одном слове этой памяти можно хранить произвольное конечное число, и можно извлечь любое слово за фиксированное, постоянное время (это удобное предположение, может быть, немного нереалистично, но на практике это почти так). Такой компьютер может выполнять хранимую программу, которая состоит' из допустимой последовательности команд, охватывающих все стандартные арифметические операции, операции сравнения, переходы и т. д. Обычно мы будем считать,, что .каждая такая команда выполняется за единицу времени. Простыми описаниями часть памяти может быть организована в одно-, двух- и трехмерные массивы (матрицы).

По сути дела, мы утверждаем, что только тогда имеется алгоритм для решения задачи, когда можно написать программу для ЭВМ, решающую эту задачу. Это спорный вопрос. Программы на описанном выше компьютере не могут зашнуровывать ботинки. Среди точно определенных шагов не содержатся шаги, необходимые для завязывания шнурков. Имеются весомые аргументы в пользу того, что мы на самом деле оперируем ограниченным понятием алгоритма. Человек, как машина, способен выполнять множество тонких операций, которые лежат за рамками возможностей нашего типичного компьютера. Однако ограниченное определение — это как раз то, что нам требуется в данной книге.

Следующий пример алгоритма иллюстрирует уровень детализации, согласующийся с нашим определением. Рассмотрим простую задачу нахождения максимального числа в списке из N действительных чисел R (1), R (2), . . R (N). Основная идея алгоритма заключается в том, чтобы перебирать по очереди все числа списка и запоминать наибольшее до сих пор встретившееся число. К тому моменту, когда весь список будет проверен, запомнится наибольшее число. Запись Л^—В означает оператор присваивания, т. е. переменной А присваивается текущее значение В.

Algorithm МАХ. Даны N действительных чисел в одномерном массиве R (1), R (2), …, R(N), найти такие М и J, что

В случае когда два или более элементов R имеют наибольшее значение, запоминается наименьшее значение J.

Шаг 0. [Установка в начальное состояние, или инициализация]

**Set М←R(1); and J←1.**

Шаг 1. [N=l?] If N=l then STOP fi/

В этом алгоритме fi и od используются соответственно для обозначения конца конструкций if и do.

Шаг 2. [Проверка каждого числа] For K←2 to N do шаг 3 od; and STOP.

Шаг 3. [Сравнение] If M<R(К) then set M←R (K) and J←K fi (теперь M — наибольшее число из проверенных, а К — его номер в массиве).

Алгоритм МАХ не закодирован на каком-то языке, а записан в форме, которую легче воспринять; он выражен в виде шагов, которые легко реализуются в каждом общепринятом языке программирования. От такого представления нетрудно перейти к кодовой форме. Однако так бывает не всегда. Некоторые алгоритмы слишком сложны, чтобы перейти за один шаг от предварительного словесного описания к кодам машины. Может потребоваться ввести по крайней мере один промежуточный этап разработки.

**Основные этапы полного построения.**

**Постановка задачи**

Прежде чем мы сможем понять задачу, мы должны ее точно сфор­мулировать. Это условие само по себе не является достаточным для понимания задачи, но оно абсолютно необходимо.

Обычно процесс точной формулировки задачи сводится к поста­новке правильных вопросов. Перечислим некоторые полезные воп­росы для плохо сформулированных задач:

Понятна ли терминология, используемая в предварительной фор­мулировке? Что дано?

Что нужно найти? Как определить решение?

Каких данных не хватает и все ли они нужны? Являются ли какие-то имеющиеся данные бесполезными? Какие сделаны допущения?

Возможны и другие вопросы в зависимости от конкретной задачи. Часто после получения полных пли частичных ответов на некоторые из вопросов их приходится ставить повторно.

Для еще лучшего понимания как правильно сформулировать ТЗ рассмотрим пример о так называемой транспортной задачи.

**Пример.**

Виктор — агент по продаже компьютеров (коммивояжер); на его территории 20 городов, разбросанных по всей Воронежской области. Компа­ния возмещает ему только 50% стоимости деловых автомобильных поездок. Виктор вычислил, сколько ему будет стоить переезд на машине между каждыми двумя городами на его территории. Ему, естественно, хотелось бы снизить свои дорожные расходы.

Что дано? Исходная информация задана в виде перечня городов на территории Виктора и соответствующей матрицы стоимостей, т. е. двумерного массива с элементами Cij, равными стоимости переезда из города i в город j. В данном случае матрица стоимостей имеет 20 строк и 20 столбцов.

Что мы хотим найти? Мы хотим помочь Виктору снизить его дорож­ные расходы. Это звучит несколько туманно. Действительно, вопрос о том, каким будет решение, выглядит неуместным. Обдумав ситуа­цию, мы придем к выводу, что ничего не можем сделать без дополни­тельной информации от Виктора. Имеет ли Виктор в одних городах боль­ше покупателей, чем в других? Если да или если есть какие-то особые покупатели, то Виктор, возможно, захочет посещать какие-то города чаще. Могут быть и такие города, в которые Виктор специально не поедет, а заедет туда, когда окажется в соседнем городе. Другими словами, нам надо знать больше о приоритетах Виктора и учитывать предпочтения при составлении графика поездок.

Поэтому мы возвращаемся к Виктору и требуем у него дополнитель­ную информацию. Он сообщает, что хотел бы иметь маршрут, начи­нающийся и оканчивающийся в его базовом городе и проходящий по одному разу через все остальные города на его территории. Следо­вательно, нам требуется список городов, содержащий каждый город только один раз, за исключением базового города, который стоит в списке первым и последним. Порядок городов в этом списке пред­ставляет собой маршрут, по которому Виктор должен объезжать города на своей территории. Сумма стоимостей проезда между каждыми двумя последовательными городами списка - это общая стоимость маршрута, представленного списком. Мы решим задачу Виктора, если представим ему список с наименьшей возможной общей стоимостью.

Это хорошая исходная постановка задачи. Мы знаем, что мы имеем и что хотим найти.

**Построение модели**

Задача четко поставлена, теперь нужно сформулировать для нее математическую модель. Это очень важный шаг в процессе реше­ния, и его надо хорошо обдумать. Выбор модели существенно влияет на остальные этапы в процессе решения.

Как вы можете догадаться, невозможно предложить набор правил, автоматизирующих стадию моделирования. Большинство задач долж­но рассматриваться индивидуально. Тем не менее, существует не­сколько полезных руководящих принципов. Выбор модели - в боль­шей степени дело искусства, чем науки, и, вероятно, эта тенденция сохранится. Изучение удачных моделей - это наилучший способ приобрести опыт в моделировании.

Для математической модели можно дать такое определение - это совокупность математических зависимостей, описывающих функционирование объекта.

Математические модели могут быть получены теоретическим и экспериментальным путем. Однако на практике математическая модель объекта чаще всего получается при сочетании теоретических и практических методов.

Приступая к разработке модели, следует задать по крайней мере два основных вопроса:

1. Какие математические структуры больше всего подходят для задачи?

2. Существуют ли решенные аналогичные задачи?

Второй вопрос, возможно, самый полезный во всей математике. В контексте моделирования он часто дает ответ на первый вопрос. Действительно, большинство решаемых в математике задач, как пра­вило, являются модификациями ранее решенных. Большинство из нас просто не обладает талантом Ньютона, Гаусса или Эйнштейна, и для продвижения вперед нам приходится руководствоваться накоп­ленным опытом.

Сначала нужно рассмотреть первый вопрос. Мы должны описать математически что мы знаем и что хотим найти. На выбор соответст­вующей структуры будут оказывать влияние такие факторы, как:

1) ограниченность наших знаний относительно небольшим количест­вом структур,

2) удобство представления,

3) простота вычисления,

4) полезность различных операций, связанных с рассматриваемой структурой или структурами.

Сделав пробный выбор математической структуры, задачу следует переформулировать в терминах соответствующих математических объектов. Это будет одна из возможных моделей, если мы можем ут­вердительно ответить на такие вопросы, как:

Вся ли важная информация задачи хорошо описана математиче­скими объектами?

Существует ли математическая величина, ассоциируемая с иско­мым результатом?

Выявили мы какие-нибудь полезные отношения между объектами модели?

Можем мы работать с моделью? Удобно ли с ней работать?

Для лучшего понимания вопросов составления модели, рассмотрим транспортную задачу, которую мы формулировали выше, когда составляли ТЗ.

**Пример.**

Возвращаемся к задаче агента по продаже компьютеров, рассмотренной ранее в этом разделе. Начинаем с постановки задачи, данной в конце примера.

Решали ли мы ранее аналогичные задачи? В математическом смысле, вероятно, нет. Однако все мы сталкивались с задачами вы­бора пути по дорожным картам или в лабиринтах. Можем ли мы прийти к удобному представлению нашей задачи наподобие карты?

Очевидно, нужно взять лист бумаги и нанести на нем по одной точке, соответствующей каждому городу. Мы не собираемся изобра­жать точки так, чтобы расстояние между каждой парой точек, соот­ветствующих городам i и j, было пропорционально стоимости проезда Cij. Расположим точки любым удобным способом, соединим точки i и j линиями и проставим на них «веса» Cij*.*

Схема, которую мы только что изобразили,— это частный случай известного в математике *графа,* или *сети.* В общем случае сеть — это множество точек (на плоскости) вместе с линиями, соединяющими некоторые или все пары точек; над линиями могут быть проставлены веса.



**Рис. 1.1** Задача коммивояжера с пятью городами

Для простоты предположим, что у Виктора только пять городов, для которых матрица стоимостей показана на рис. 3.2 a. Тогда сете­вая модель может быть изображена, как на рис. 3.2 б. Предполо­жим также, что стоимость проезда из города i в город j такая же, как и из j в i*,* хотя это и необязательно.

Что мы ищем в задаче? В терминах теории сетей список городов (который мы ранее описали) определяет замкнутый цикл, начинаю­щийся с базового города и возвращающийся туда же после прохож­дения каждого города но одному разу. Такой цикл соответствует не­отрывному движению карандаша вдоль линий сети, которое проходит через каждую точку только одни раз и начинается и оканчивается в одной и той же точке. Обход такого рода назовем *туром.* Стоимость тура определяется как сумма весов всех пройденных ребер. Задача решена, если мы можем найти тур с наименьшей стоимостью.

Па рис. 3.2, обход 1—5—3—4—2—1 есть тур со стоимостью 5+2+1+4+1=13. Является ли он туром с минимальной стоимостью? Рассмотренная задача известна в литературе как *задача комми­вояжера,* она стала в какой-то мере классической. Это один из наибо­лее известных примеров таких задач, которые очень легко поставить и промоделировать, по очень трудно решить. Время от времени мы будем возвращаться к этой задаче в целях иллюстрации.

**Разработка алгоритма**

Как только задача четко поставлена и для нее построена модель, мы должны приступить к разработке алгоритма ее решения. Выбор метода разработки, зачастую сильно зависящий от выбора модели, может в значительной степени повлиять на эффективность алгоритма решения. Два разных алгоритма могут быть правильными, но очень сильно отличаться по эффективности.

При разработки алгоритма важно оценить такие его характеристики как скорость, точность, требуемая память, средства ввода и вывода. При этом исходя из того, что важнее для конкретного случая выбирается подходящий алгоритм.

Следует отметить, что алгоритмы характеризуются пятью характеристиками:

1. Вход алгоритма.

2. Выход алгоритма

3. Определенность.

4. Выполнимость.

5. Конечность.

**Пример.**

Вернемся к коммивояжеру из предыдущего раздела. По­становка задачи и модель, описанные ранее, наводят на мысль о сле­дующем алгоритме.

Сначала произвольно перенумеруем N городов целыми числами от 1 до *п,* присваивая каждому городу свой номер. Базовому городу приписываем номер *п.* Заметим, что каждый тур однозначно соответст­вует перестановке целых чисел 1, 2, . . ., *п—*1. Действительно, каждый тур соответствует единственной перестановке, и каждая перестановка соответствует единственному туру. Такое соответствие называется *взаимно-однозначным.* Таким образом, для любой данной перестановки мы можем легко проследить соответствующий тур на сетевой модели и в то же время вычислить стоимость этого тура.

Можно решить задачу, образуя все перестановки первых *п—*1 целых положительных чисел. Для каждой перестановки строим соот­ветствующий тур и вычисляем его стоимость. Обрабатывая таким образом все перестановки, запоминаем тур, который к текущему моменту имеет наименьшую стоимость. Если мы находим тур с более низкой стоимостью, то производим дальнейшие сравнения с этим туром.

Проиллюстрируем предложенный алгоритм следующей условной схемой, которая в ряде учебников называется как исчерпывающий алгоритм:

*Исчерпывающий алгоритм решения транспортной задачи.* Решить задачу ком­мивояжера с *N* городами, последовательно рассматривая все переста­новки из *N—*1 положительных целых чисел. Таким образом мы рас­смотрим каждый возможный тур и выберем вариант TOUR с наимень­шей стоимостью MIN. Исчерпывающий алгоритм требует в качестве входных дан­ных число городов *N* и матрицу стоимостей *С.*

*Шаг 0.* [Инициализация, т. е. установка в начальное состояние]

**Set TOUR←∅;** МIN←∞.

*Шаг 1.* [Образование всех перестановок]

**For** I←1 to (N—1)! do throughшаг 4 od; and STOP.

*Шаг 2.* [Получение новой перестановки]

Set P=1-я перестановка целых чисел 1, 2, . . ., *N—*1. (Заметим, что здесь нужен подалгоритм.)

*Шаг 3.* [Построение нового тура] Строим тур Т(Р), соот­ветствующий перестановке Р и вычисляем стои­мость COST (Т (Р)). (Заметим, что здесь нужны два других подалгоритма.)

*Шаг 4.* [Сравнение] If COST (Т(P))<MIN **then set** TOUR←***T(P),* and** MIN=COST(T(P)) fi*.*

*Ш****аг 5* [**Вывод результатов и окончание программы]

Исчерпывающий алгоритм — неплохое первое приближение к точному алго­ритму. Ему недостает некоторых важных подалгоритмов, и он недостаточно близок к окончательной программе. Эти недостатки будут устранены позже.

Похоже, что существует тенденция: программисты затрачивают относительно небольшое время на стадию разработки алгоритма при создании программы. Проявляется сильное желание как можно быст­рее начать писать саму программу. Этому побуждению не надо под­даваться. На стадии разработки требуется тщательное обдумывание, следует также уделить внимание двум предшествующим и первым трем следующим за стадией разработки этапам. Как и следует ожи­дать, все восемь основных этапов, перечисленных выше, нельзя рассматривать независимо друг от друга. В особенности, первые три сильно влияют на последующие, а шестой и седьмой этапы обеспечи­вают цепную обратную связь, которая может заставить нас пере­смотреть некоторые из предшествующих этапов.

**Правильность алгоритма**

Доказательство правильности алгоритма — это один из наиболее трудных, а иногда и особенно утомительных этапов создания алгоритма.

Вероятно, наиболее распространенная процедура доказательства правильности программы — это прогон ее на разных тестах. Если выданные программой ответы могут быть подтверждены известными или вычисленными вручную данными, возникает искушение сделать вывод, что программа «работает». Однако этот метод редко исключает все сомнения; может существовать случай, в котором программа не сработает.

Мы предложим следующую общую методику доказательства правильности алгоритма. Предположим, что алгоритм описан в виде последовательности шагов, скажем, от шага 0 до шага т. Постараемся предложить некое обоснование правомерности для каждого шага. В частности, может потребоваться лемма об условиях, действующих до и после пройденного шага. Затем постараемся предложить доказательство конечности алгоритма, при этом будут проверены все подходящие входные данные и получены все подходящие выходные данные.

**Пример.**

Алгоритм транспортной задачи настолько прост, что его правильность легко доказать. Поскольку проверяется каждый тур, должен быть проверен и тур с минимальной стоимостью; как только до него дойдет очередь, он будет запомнен. Он не будет отброшен — это может случиться только в том случае, если существует тур с меньшей стоимо¬стью. Алгоритм должен закончить работу, так как число туров, которые нужно проверить, конечно. Подобный метод доказательства известен как «доказательство исчерпыванием»; это самый грубый из всех методов доказательства.

Подчеркнем тот факт, что правильность алгоритма еще ничего не говорит о его эффективности. Исчерпывающие алгоритмы редко бы¬вают хорошими во всех отношениях. Доказательство правильности алгоритма - это один из наиболее трудных, а иногда и особенно утомительных этапов создания алго­ритма.

**Реализация алгоритма**

Как только алгоритм выражен, допустим, в виде последователь­ности шагов и мы убедились в его правильности, настает черед реали­зации алгоритма, т. е. написания программы для ЭВМ.

Написание программы зависит от многих качеств, как объективных, так и субъективных. Но среди всех аспектов целесообразно выделить такие, как выбор языка программирования, выбор структуры данных, типы оборудования, на которых будет эксплуатироваться программа, тип интерфейса, который необходим для работы этой программы и многое другое.

Реализация алгоритма - этот существенный шаг может быть трудным. Во-первых, трудность заключается в том, что очень часто отдельно взятый шаг алгоритма может быть выражен в форме, которую трудно перевести непосредственно в конструкции языка программирования. Например, один из шагов алгоритма может быть записан в виде, требующем це­лой подпрограммы для его реализации. Во-вторых, реализация может оказаться трудным процессом потому, что перед тем, как мы сможем начать писать программу, мы должны построить целую систему струк­тур данных для представления важных аспектов используемой мо­дели. Чтобы сделать это, необходимо ответить, например, на такие вопросы:

Каковы основные переменные?

Каких они типов?

Сколько нужно массивов и какой размерности?

Имеет ли смысл пользоваться связными списками?

Какие нужны подпрограммы (возможно, уже записанные и реализованные)?

Каким языком программирования пользоваться?

На каких типах оборудования будет использоваться данный программный продукт?

Каковы основные аспекты взаимодействия человек - ЭВМ?

На какого пользователя рассчитано программное обеспечение?

Для какого класса задач и продолжительности по времени разрабатывается данное программное обеспечение?

Как будет проводится верификация?

Конкретная реализация может существенно влиять на требования к памяти и на скорость алгоритма.

Другой аспект построения программной реализации — это про­граммирование сверху-вниз. Объяснение этого понятия будет дано в позднее, а пока укажем, что программирование сверху-вниз— это подход к разработке и реализации, который состоит в преобразо­вании алгоритма в такую последовательность все более конкретизи­рованных алгоритмов, что окончательный вариант представляет собой программу для ЭВМ.

Сделаем одно важное замечание. Одно дело — доказать правиль­ность конкретного алгоритма, описанного в словесной форме. Другое дело — доказать, что данная машинная программа, предположи­тельно являющаяся реализацией этого алгоритма, также правильна. Таким образом, необходимо *очень* тщательно следить, чтобы процесс преобразования правильного алгоритма (в словесной форме) в машин­ную программу «заслуживал доверия».

**Анализ алгоритма и его сложности**

Существует ряд важных практических причин для анализа алго­ритмов. Одной из них является необходимость получения оценок или границ для объема памяти, или времени работы, которое потребуется алгоритму для успешной обработки конкретных данных. Машинное время и память — относительно дефицитные (и дорогие) ресурсы, на которые часто одновременно претендуют многие пользователи. Всегда следует избегать прогонов программы, отвергаемых системой из-за нехватки запрошенного времени, которое указывается на рабо­чей карте. Поразительно, скольким программистам приходится слиш­ком дорогим способом выяснять, что их программа не может обрабо­тать входные данные раньше, чем через несколько дней машинного времени. Лучше было бы предугадать такие случаи с помощью каран­даша и бумаги для того, чтобы избежать ненужных прогонов. Хоро­ший анализ способен выявить узкие места в наших программах, т. е. разделы программы, на которые расходуется большая часть времени.

Существуют также важные теоретические причины для анализа алгоритмов. Хотелось бы иметь некий количественный критерий для сравнения двух алгоритмов, претендующих на решение одной и той же задачи. Более слабый алгоритм должен быть улучшен или отброшен. Желательно также иметь механизм для выявления наиболее эффек­тивных алгоритмов и замены устаревших. Иногда невозможно соста­вить четкое мнение об относительной эффективности двух алгоритмов. Один может в среднем лучше работать, к примеру, на случайных входных данных, в то время как другой лучше работает на каких-то специальных входных данных. Хотелось бы иметь возможность де­лать аналогичные выводы о сравнительных достоинствах двух алго­ритмов.

Важно также установить абсолютный критерий. Когда можно считать решение задачи оптимальным? Иными словами, когда алгоритм настолько хорош, что *невозможно* (независимо от умственных способностей) значительно его улучшить?

Пусть *А —* алгоритм для решения некоторого класса задач, а n *—* размерность отдельной задачи из этого класса. Во многих задачах этого методического пособия *n* — просто скаляр, равный числу вершин графа. В об­щем случае *n* может быть массивом или длиной вводимой последова­тельности. Определим fa(n) как *рабочую функцию,* дающую верхнюю границу для максимального числа основных операций (сложения, сравнения и т. д.), которые должен выполнить алгоритм *А* для реше­ния любой задачи размерности *n.* Будем пользоваться следующим критерием для оценки качества алгоритма: чем меньше растет функция fa(n) на каком-то участке области n, тем алгоритм будет более эффективным.

В зависимости от роста функции fa(n) говорят, что алгоритм A *полиномиальный,* если fa(n) растет не быстрее, чем полином от n, в противном случае алгоритм A *экспоненциальный.* Этот критерий основан на времени работы в худшем случае, но аналогичный крите­рий может быть также определен для среднего времени работы. «Экспериментальная» основа для этого критерия качества заключается в том, что, по-видимому, последовательные или параллельные машины более или менее способны воспринимать полиномиальные алгоритмы для больших задач, а на экспоненциальных алгоритмах они доволь­но быстро «задыхаются».

Поэтому сделаем общее утверждение: необходимо стремиться к тому, чтобы рост алгоритма был как можно меньше от n на той области участка n, на котором требуется получить решение.

**Проверка программы**

Программа написана, настает время для ее отладки и тестирования, Этот процесс предшествует ее эксплуатации.

Отладка программы включает в себя проверку на синтаксические, логические и подобные ошибки. Отладка делится в свою очередь на два этапа: отладка синтаксиса и отладка семантики. Синтаксическая ошибка - это нарушение правил записи на данном языке. Эти ошибки обычно диагностируются транслятором, и их исправление трудности не вызывает. Специальной подготовки программы для отладки синтаксиса не требуется. Семантическая (смысловая) ошибка - это применение операторов, которые не дают нужного эффекта (простейший пример: написать a-m вместо a+m). Эти ошибки ЭВМ самостоятельно найти, естественно, не может.

Первым процессом отладки является исправление синтаксических ошибок, что транслятор делает самостоятельно. Следующим этапом является отладка на семантические ошибки. С этой целью на схеме выделяются места, где будут использоваться средства отладки, и принимаются решения о том, какие средства будут применены. Отладочные средства ставятся в узловых точках схемы, на входах в процедуры; на длинных линейных участках ставят промежуточные печати. Если есть возможность организуются аварийные выдачи, используются различные отладочные режимы системы.

Таким образом, основа отладки - это отладка семантики. Из вышесказанного ясно, что основным инструментом отладки служат тесты и отладочные печати. Подготовка тестов и расстановка отладочных печатей - это такой же необходимый этап, как и само программирование.

После того как исправлено множество синтаксических, ло­гических ошибок и ошибок редактора, программу, наконец, можно прогнать на простом примере (таком, который может быть проверен вручную). Таким образом начинается процесс тестирования программы.

Тест - это специально подобранные исходные данные в совокупности с теми результатами, которые должна выдать программа при обработке этих данных. Несовпадение результатов программы с результатами тестов - признак наличия ошибки. Но иногда и неправильная программа может по некоторым тестам выдать правильный результат, поэтому необходимо контролировать и промежуточные результаты, чтобы не упустить взаимное уничтожение ошибок в данном варианте работы программы. Для такого контролирования и локализации ошибок ставятся отладочные печати промежуточных результатов. Они позволяют проследить логический след программы («где она ходила») и ее арифметический след («что она вычисляла»).

Процесс проверки программы должен включать в себя значительно больше, чем было указано выше. Было бы преувеличением сказать, что проверка программы в вычислительной математике аналогична экспериментированию в естественных науках, подумается, что между этими процессами есть что-то общее. Проверка программы может быть охарактеризована как экспериментальное подтверждение того факта, что программа делает именно то, что должна делать. Проверка программы является также экспериментальной попыткой установить границы использования алгоритма (программы).

Все мы можем сделать ошибки при доказательстве и при переводе правильного алгоритма в программу. Каждый может забыть или не учесть некоторый частный случай задачи. Недостаточно доказать правильность алгоритма. Окончательная программа должна быть тща­тельно проверена и оттестирована. Мельчайшие особенности вашей операционной системы могут вызвать для некоторых входных данных такое действие какой-то части вашего алгоритма, о котором вы не подозревали. Программа должна быть проверена для широкого спектра допустимых входных данных. Этот процесс может быть продолжи­тельным, утомительным и сложным.

Как выбрать входные данные для тестирования? На этот вопрос невозможно дать общего ответа. Для любого алгоритма ответ зависит от сложности программы, имеющегося ресурса времени, а также от персонала, занимающегося проверкой, числа вводов (т. е. вариантов входных данных), для которых можно установить правильность вы­водов, и т. д. Обычно множество всех вводов огромно, и полная про­верка практически невозможна. Мы должны выбрать множество вводов, которые проверяют каждый участок программы. Надо обязательно достаточно полно проверить случаи, которые с большой ве­роятностью встретятся в практике. Редко можно гарантировать пра­вильность программы, но мы можем и должны провести соответст­вующую проверку, чтобы быть достаточно уверенными в этом.

Дальнейшая проверка также необходима для того, чтобы устано­вить качество алгоритма. Анализ, описанный в предыдущем разделе, не всегда надежен. Сделанные в ходе анализа упрощающие допущения должны быть экспериментально проверены. Многие большие, сложные алгоритмы трудно или невозможно математически исследовать. В та­ких случаях особенно важно проверить алгоритм в действии, трудоем­кости, так как это единственная возможность оценить его качество.

Опыт показывает, что анализ среднего функционирования алго­ритма более ценен и трудоемок, чем анализ наилучшего и наихудшего случаев. Если возможен анализ худшего случая, то очень важно экспериментально установить, работает ли алгоритм значительно лучше в среднем, чем в худшем случае.

Аналитический и экспериментальный анализ дополняют друг друга. Аналитический анализ может быть неточным, если сделаны слишком сильные упрощающие допущения. В этом случае могут быть получены только грубые оценки. С другой стороны, получить доста­точное экспериментальное подтверждение для гарантий какой-либо статистической достоверности может оказаться невозможным или непрактичным. Экспериментальные результаты, особенно когда ис­пользуются случайно сгенерированные данные, могут оказаться слишком односторонними. Чтобы получить достоверные результаты, нужно там, где это возможно, провести как аналитическое, так и экс­периментальное исследование.

Программы следует тестировать также для того, чтобы определить их вычислительные ограничения. Многие программы для некоторых входных данных работают хорошо, а для других плохо. Желательно, чтобы характеристика работы алгоритма «плавно» менялась от хоро­шей к плохой при переходе от входных данных, на которых алгоритм хорошо работает, к входным данным, для которых это не так. Алго­ритм ETS хорошо работает для n<7 и очень плохо для n>15. К сожа­лению, в данном случае переход не плавный, н алгоритм имеет тен­денцию плохо работать в промежуточных случаях. Желательно поль­зоваться как аналитическими, так и экспериментальными методами, чтобы охарактеризовать те входные данные, которые считаются или «хорошими», или «плохими».

Итак, в идеальном случае, программа должна быть оттестирована для всех возможных комбинаций входных данных, которые сравниваются с предварительно рассчитанными правильными результатами. Однако почти для всех программ, даже очень простых, существует бесконечное число возможных комбинаций входных данных, или по меньшей мере это число очень велико. Поэтому, после тестирования, как правило, остается относительно небольшое количество ошибок и усилия направленные на их выявление не оправданы. Следовательно, устранение этих ошибок откладывается на этап сопровождения программы в ходе эксплуатации.

**Документация**

На самом деле, этап документации не является последним шагом в процессе полного построения алгоритма. В частности, он не заклю­чается в том, чтобы добавить карты с комментариями, когда вы за­кончили все остальное. Процесс документации должен переплетаться со всем процессом построения алгоритма, и особенно с этапами раз­работки и реализации.

Трудно читать чужую программу. Наиболее очевидный мотив для документации — дать возможность людям понять про­граммы, которые написаны другими. Конечно, лучший способ — это составить программу настолько понятно, чтобы она сама себя пояс­няла. Но это невозможно осуществить ни для каких программ, кроме простейших; и программа в коде должна быть дополнена другими фор­мами пояснений. Обычно для этого используются операторы с коммента­риями.

Но, в действительности, это только надводная часть айсберга. До­кументация включает в себя всю информацию и помогает объяснить, что делается в программе, т. е., в частности, блок-схемы, описания ступеней в вашем построении сверху-вниз, вспомогательные доказа­тельства правильности, результаты тестирования, детальные описания формата и требований к вводу/выводу и т.д.

В простейшем случае документацию можно разбить на 3 составляющие:

- функциональную часть, где определяется назначение программы, метод решения, требования к входным и выходным данным, ссылки на литературу;

программная часть, описывающая подробное описание метода решения, блок-схему, текст программы, детали реализации языка и компилятора, использование библиотеки или внешних модулей, требования к точности представления чисел, детали работы с файлами, эксплуатацию работы в интерактивном режиме и т.п.

- эксплуатационные процедуры, состоящие из требований к техническому обеспечению, типу ЭВМ, размеру памяти, периферийным устройствам и другое.

В последующих главах проводится более детальное обсуждение документации. В настоящий момент пред­лагаем вам золотое правило: оформляйте ваши программы в таком виде, в каком вам хотелось бы видеть программы, написанные дру­гими.

**Тема 2. Некоторые основные приемы и алгоритмы**

**Структурное программирование сверху-вниз и правильность программ**

В основу структурного программирования положены следующие достаточно простые положения:

1. Программа должна создаваться мелкими шагами. Размер шага определяется количеством решений, применяемых программистом на этом шаге.

2. Сложная задача должна разбиваться на достаточно простые, легко воспринимаемые части, каждая из которых имеет только один вход и один выход.

3. Логика программы должна опираться на минимальное количество достаточно простых базовых управляющих структур, подобно тому как любая функция алгебры логики может быть выражена через функционально полную систему (например, дизъюнкцию, конъюнкцию, отрицание).

От алгоритма требуется (и это требование становится все более весомым) такая реализация, чтобы она была легка для понимания, проста для доказательства правильности и удобна для модификаций в случае изменения спецификаций функции.

Последнее требование важно главным образом потому, что по мере того, как алгоритмы становятся все более и более сложными, соответственно растет трудность понимания, как они работают, исправления обнаруженных в них ошибок, доказательства их правильности и при необходимости правильного внесения в них изменений. В последние годы обратили внимание на то, что от 50 до 100% времени программист тратит на решение задач по исправлению, эксплуатации и модификации программ. Пытаясь помочь программистам, индустрия программного обеспечения предлагает более систематичные подходы к программированию, т. е. предлагает методики, использование которых уменьшает вероятность ошибок в программах, упрощает их понимание и облегчает модификацию.

На сегодняшний день самой популярной методикой, по-видимому, следует считать структурное программирование сверху-вниз. В данном разделе мы введем основные идеи этой методики и проиллюстрируем ее на примере.

**Основные правила структурного программирования**

Наше объяснение структурного программирования мы начнем с определения блок-схемы. Блок-схема — это ориентированная сеть, вершины которой могут быть одного из трех типов, представленных на рис. 2.1.

Функциональная вершина используется для представления функции f: X→Y. Предикатная вершина используется для представления функции (или предиката) р: Х→{Т, F}, т. е. логического выражения, передающего управление по одной из двух возможных ветвей. Объединяющая вершина представляет передачу управления от одной из двух входящих ветвей к одной выходящей ветви.

Структурная блок-схема — это блок-схема, которая может быть выражена как композиция из четырех элементарных блок-схем, изображенных на рис. 2.2. Сравнительно просто доказать, что любая программа для вычислительной машины может быть «представлена» блок-схемой. Как было показано Бомом и Якопини, верно (хотя далеко не очевидно) и то, что любая блок-схема может в свою очередь быть представлена структурной блок-схемой. Из этого результата немедленно следует, что для разработки любого алгоритма достаточно четырех элементарных блок-схем, приведенных на рис. 2.2.

Когда структурная блок-схема служит как представление программы, В интерпретируется как булевское выражение, а S1 и S2 интерпретируются как программные операторы (или процедуры).

Блок-схемы на рис. 2.2 а, б, в и г называют структурами управления программы. Схема на рис. 2.2 а называется композицией и записывается в виде do S1; S2 od (или просто S1; S2). Схема на рис. 2.2, б называется выбором (или альтернативой) и записывается в виде if В then S1 else S2 fi (символ fi можно опускать при записи этой конструкции; он является признаком окончания конструкции if). Блок-схемы на рис. 2.2 в и г назы-ваются итерацией (или повторением) и записываются соответ-ственно как while В do S1 od и do S1 while B od.

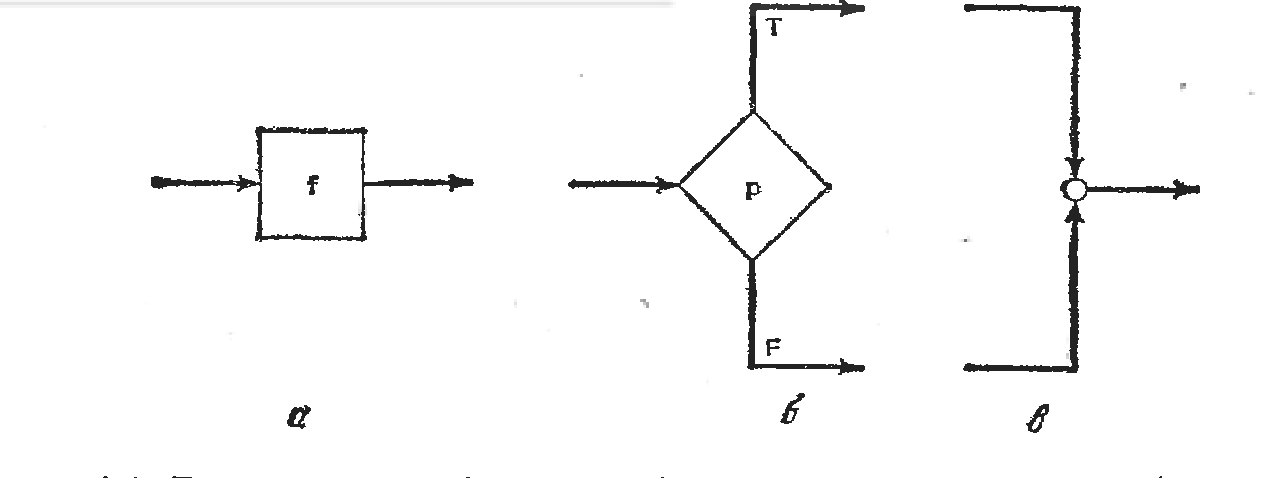


Рис. 2.1. Типы вершин на блок-схеме: а - функциональная вершина, б - предикатная вершина, в - объединяющая вершина.

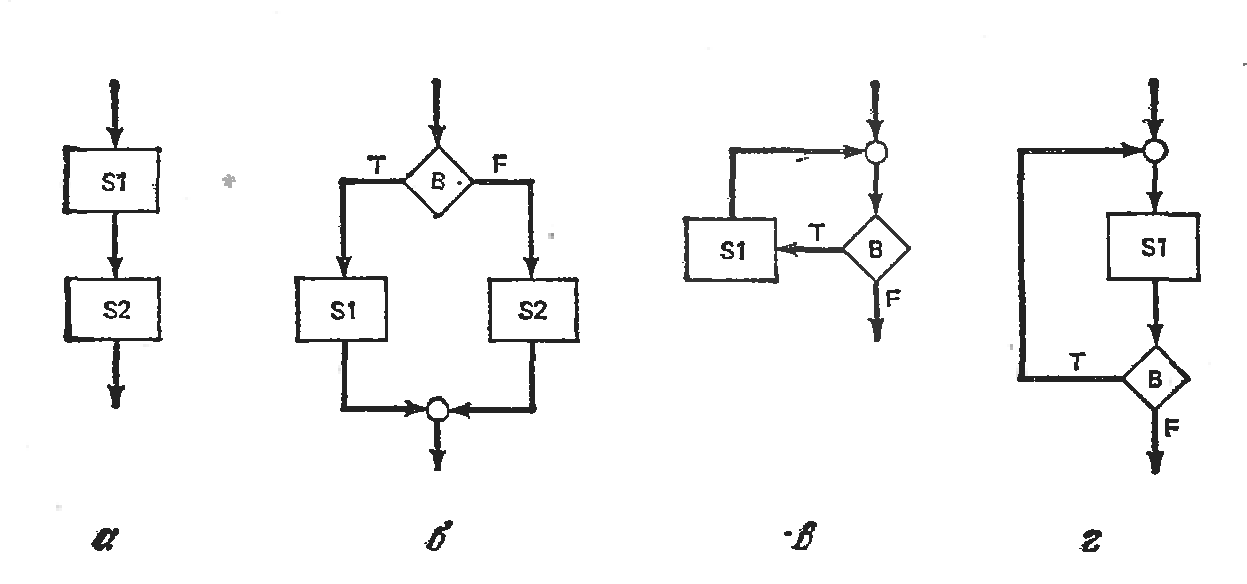


Рис. 2.2. Четыре элемента структурной блок-схемы.

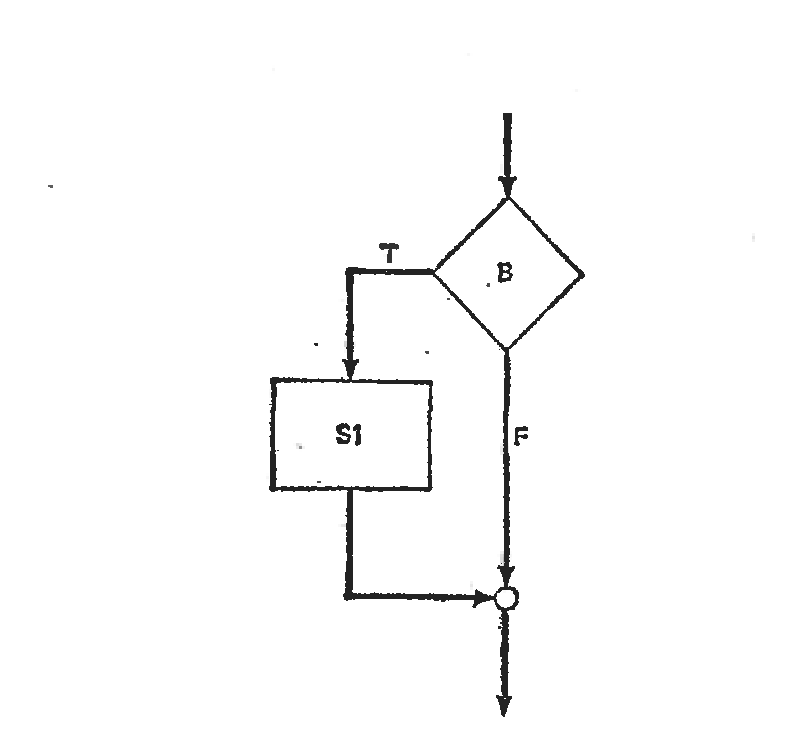


Рис. 2.3. Блок-схема для оператора if В then SI.

Следует отметить, что часто употребляемый упрощенный оператор if В then SI, блок-схема которого приведена на рис. 2.3, представляет собой частный случай оператора if - then-else.

Важная особенность заключается в том, что каждая из этих четырех программных структур управления имеет один вход и один выход. Отсюда следует, что у любой блок-схемы, составленной из этих элементов, также будет один вход и один выход. На рис. 2.4 в качестве иллюстраций приводится несколько примеров структурных блок-схем, построенных из данных элементов.

Лингвистически эти блок-схемы могут быть описаны следующим образом:

(а) lf B then doS1;S2 od

else if С than S3

else S4 fi fi

(б) if В then if С then do S1; S2 od

else while D do S3 od fi

else do S4;S5;S6 od fi

В приложении А приведены соглашения принятые для описания алгоритмов.

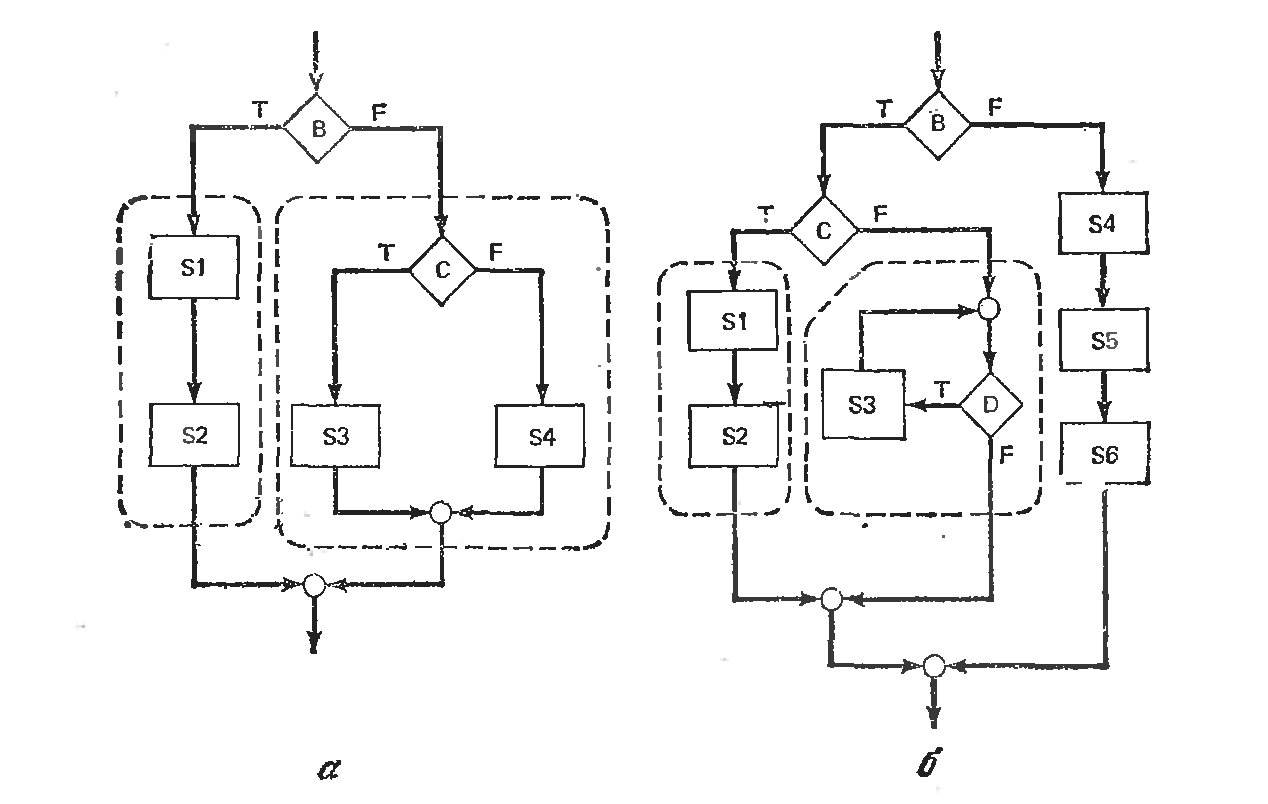


Рис. 2.4. Две структурных блок-схемы.

Обратите внимание на то, что в предыдущих блок-схемах совершенно отсутствует оператор go to.

Строго говоря, под структурным программированием понимается процесс разработки алгоритмов с помощью структурных блок-схем.

Однако в более широком плане структурное программирование допускает большее разнообразие элементарных структур управления, чем те, которые были предложены Бомом и Якопини (т. е. те, которые изображены на рис. 2.2). Дело в том, что, хотя эта совокупность структур управления достаточна для построения любой программы для вычислительной машины, само построение не обязательно окажется простейшим или наиболее естественным. Проиллюстрируем это на некоторых примерах.

Первый пример (рис. 2.5) соответствует ситуации, обрабатываемой на С++ оператором SWITCH .

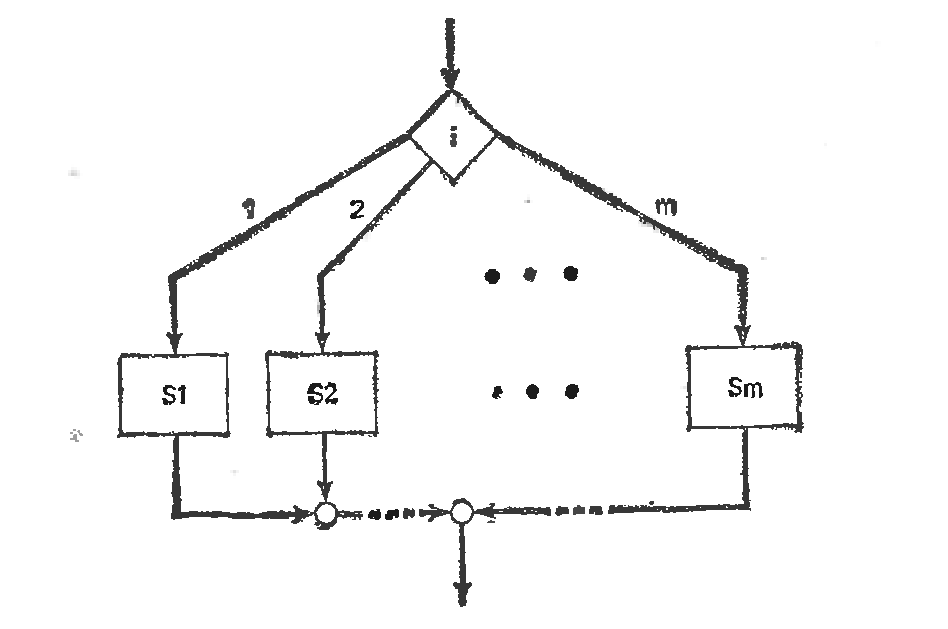


Рис. 2.5. Блок-схема для оператора case

Лингвистически эту блок-схему можно выразить в виде

case i of S1; S2; . . .; Sm fo

что эквивалентно

if i=l then S1 else

if i=2 then S2 else

.

.

.

if i=m then Sm fi . . . fi fi

Вторая запись по сравнению с первой выглядит менее естественно. Второй пример (рис. 2.6) соответствует ситуации, когда необходимо преждевременно завершить выполнение итераций, или цикла DO, вводя дополнительный выход.

Один из возможных лингвистических операторов для блок-схемы на рис. 2.6 имеет вид

while В do S1; if С then S2

else goto OUT fi od

Хотя более строгие формы структурного программирования исключают применение оператора goto, блок-схемы, в которых оператор goto используется достаточно аккуратно, могут сохранить свойство структурных блок-схем: один вход и один выход.

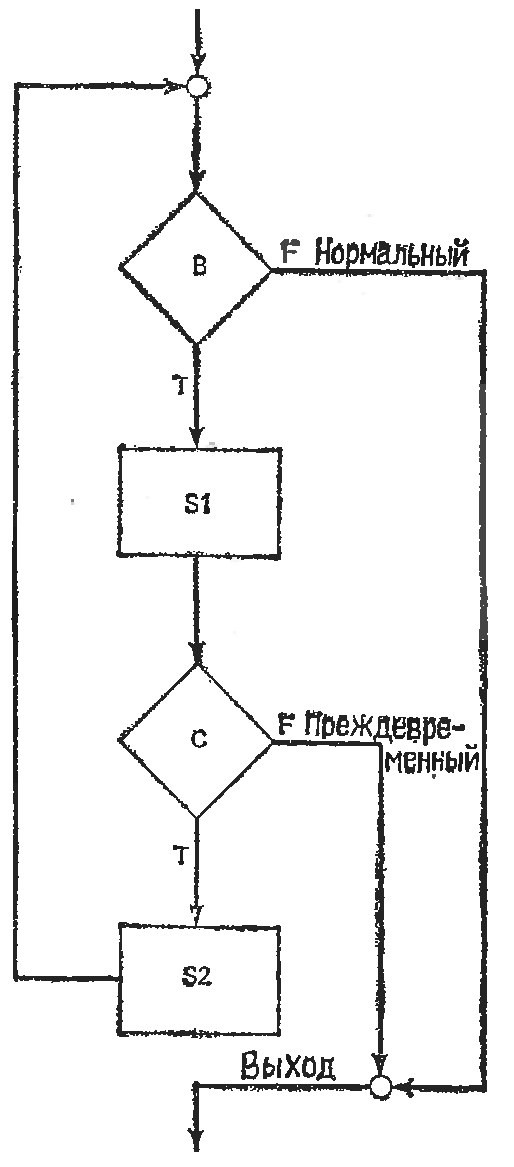


Рис. 2.6. Блок-схема для преждевременного окончания цикла.

Еще один пример простой неструктурированной блок-схемы, часто встречающийся в программировании, приведен на рис. 2.7 а. На рис. 2.7 б указан один из способов структурирования этой блок-схемы, хотя получаемый результат и нельзя считать вполне естественным.

Заметим, что лингвистический оператор для неструктурированной блок-схемы на рис. 2.7 а требует применения оператора goto. Например,

k: S1; if В then S2; goto k fi

Тогда как лингвистический оператор для структурированной блок-схемы на рис. 2.1.7, б его не содержит. Например,

do S1; while В do S2; S1 od od

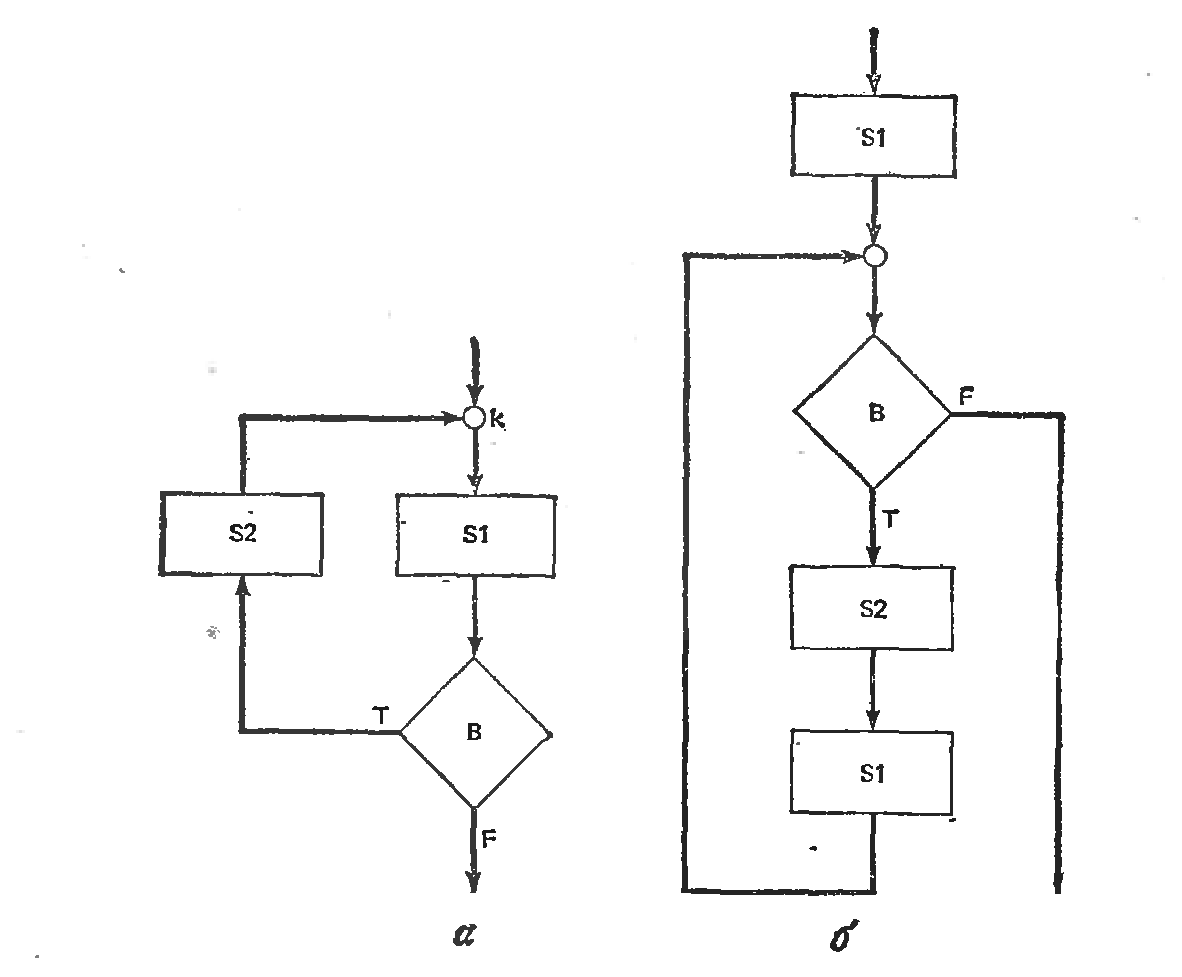


Рис. 2.7 а -неструктурированная блок-схема; б - эквивалентная структурированная блок-схема.

Вообще говоря, структурное программирование опреде-ляет процесс разработки алгоритмов в терминах «квазиструктурированных» блок-схем, в которых элементарные структуры управления являются, как правило, структурами Бома и Якопини. Впрочем, допускаются и другие структуры «простого типа» при условии, что они обладают свойством «один вход и один выход».

Как отмечалось выше, программирование сверху-вниз—это процесс пошагового разбиения алгоритма на всё более мелкие части с целью получить такие элементы, для которых можно легко написать конкретные команды.

Теперь попытаемся дать определение структурного программирования

Структурное программирование сверху-вниз — это процесс про­граммирования сверху вниз, ограниченный использованием струк­турных блок-схем. Идея весьма проста. Предположим, что требуется разработать алгоритм для некоторой конкретной функции f. Пусть далее можно доказать, что f есть композиция двух других (надо пола­гать, более простых) функций g и h, т.е. f(x)=h(g(x)). Тогда проблема разработки алгоритма для f сводится к проблемам разработки алгоритмов для g и h. Пусть далее можно доказать, что функция g равна некоторой функции i, когда заданный параметр х неотрицателен, или равна некоторой другой функции j, когда х отрицателен. Тогда алгоритм для вычисле­ния g можно выразить в форме конструкции if-then-else .

Поэтому, если алгоритмы для функций i и j построены, то правильный алгоритм для функции g строится автоматически.

Разработанные таким методом «сверху-вниз» алгоритмы обладают в некотором смысле свойством правильности, вследствие встроенного в них павила «шаг за шагом». В связи с этим в них, как правило, меньше ошибок, и их правильность доказывается легче.

В структурном программировании сверху-вниз на каждом шаге процесса разработки спрашивается, нельзя ли текущую функцию (подфункцию) выразить как композицию двух (или более) других функций.

Мы не собираемся утверждать, что структурное программирование сверху-вниз исчерпывается тем, что было сказано в предыдущем раз­деле, Оно включает достаточно много других аспектов, один из кото­рых можно выразить лозунгом «Упрощай структуры управления».

Другие аспекты — это требование соответствующей документации, наглядной записи программных операторов с использованием отсту­пов в начале строк, разбиения программы на обозримые части.

Рекомендуется, например, составлять программные модули так, чтобы их распечатка не превышала одной-двух страниц. Страница -это смысловая единица, которую можно представить себе в любой момент как единое целое и которая сводит к минимуму необходимость переворачивать страницы при прослеживании передач управления в программе.

Конечно, для разработки структурированной сверху-вниз про­граммы потребуется затратить больше усилий, чем для получения неструктурированной программы. Впрочем, опыт показывает, что дополнительные затраты с лихвой вознаграждаются. Структурироанные программы, как правило, легче читать, и в них легче разби­раться, так как их можно читать сверху-вниз, не прыгая по тексту из конца в начало. Главным образом это достигается благодаря тому, что общая структура управления в структурированной программе является деревом, а не сетью со многими циклами.

По этой причине такие программы иногда называют программами со структурой дерева.

Прежде чем окончить наше обсуждение структурного программирования сверху-вниз, хотелось бы сделать несколько предостережений. Во-первых, этот метод разработки алгоритмов не гарантирует отсутствие ошибок в программе. Опыт показывает, например, что, когда глубина вложения структур управления равна трем или более трех, вероятность совершения ошибки возрастает очень сильно.

Второе предостережение связано с термином «программирование без goto», которое иногда используется как характеристика структурного программирования. Чем бы ни было структурное программироание, это не метод программирования, запрещающий использование операторов goto, ни процесс преобразования неструктурированной программы в программу без операторов goto. Помимо того, что само по себе это не приводит к структурированной программе, текст программы становится менее понятным.

**Сети**

Тот факт, что сети — это одно из самых полезных математических средств, подтверждается тем, что они применяются во многих областях самого различного направления; сюда входят химия, физика, строительство и электротехника, исследование операций, психология, социология, лингвистика и много разделов математики (включая численный анализ, теорию матриц, теорию групп, топологию, дискретные вероятностные процессы, теорию игр и комбинаторику). Более того, сети, и в частности деревья, являются, по-видимому, наиболее часто используемыми математическими структурами в вычислительной математике. Сети и деревья используются при изучении структур данных, в компиляции, языках программирования, операционных системах, теории вычислений, сортировке, теории поиска и искусственном интеллекте. В этом разделе мы представим основы теории сетей, обсудим различные способы представления сетей в памяти вычислительной машины и сформулируем некоторые алгоритмы определения элементарных свойств сетей.

Сеть G=(V, Е) состоит из конечного множества V=V(G) М вершин (M⩾1) и конечного множества E=E(G) N неупорядоченных пар различных вершин (*u*, *v*) (N⩾0), называемых ребрами G. Сеть G1=(V1, Е1) на рис. 2.9, а состоит из семи вершин V1{*t, u, v, w, х, у, z*} и семи ребер E1={(*u, v*), (*u, w*), (*w, x*), (*w, z*), (*z, u*), (*u, x*), *(w, y*)}.

Очень важно понять, что сеть может быть изображена многими различными способами', например, сети, приведенные на рис. 2.9, б и в, те же, что и на рис. 2.9, а. Это следует из определения сети (убедитесь сами). В конце этого раздела задача, состоящая в том, чтобы определить, представляют ли два чертежа одну и ту же сеть, будет рассмотрена более подробно.

Две вершины *u* и *v* являются смежными, если в G существует ребро (*u, v*); в противном случае *u* и *v* независимы. Про ребро (*u, v*) также говорят, что оно инцидентно вершинам и *u* и *v*.

Обратите внимание на то, что ребра состоят, как мы сказали, из неупорядоченных пар различных вершин. Таким образом, ребра неориентированы в том смысле, что ребра (u, v) и (*u, v*) считаются одним и тем же ребром. Более того, ребра вида (*u, v*), называемые петлями, не допускаются. Мы также не разрешаем наличия в сети мультиребер, т.е. двух или более копий данного ребра.

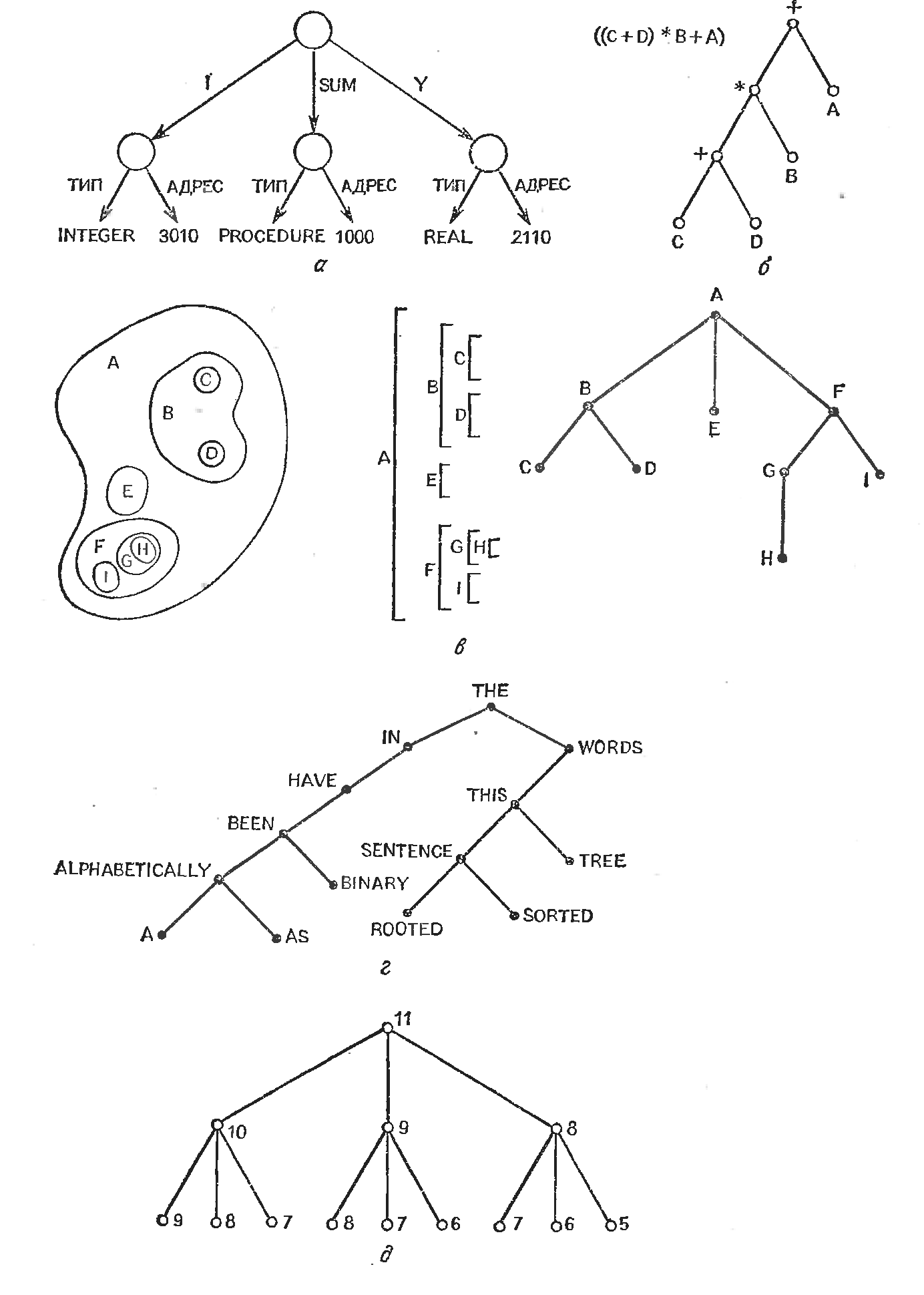


Рис. 2.8. Некоторые применения деревьев в вычислительной математике: (а) таблицы символов; (б) арифметические выражения; (в) подмножества множеств и вложенные структуры; (г) деревья сортировки— слова в этом предложении были отсортированы по алфавиту как двоичное корневое дерево; (д) игровые деревья;

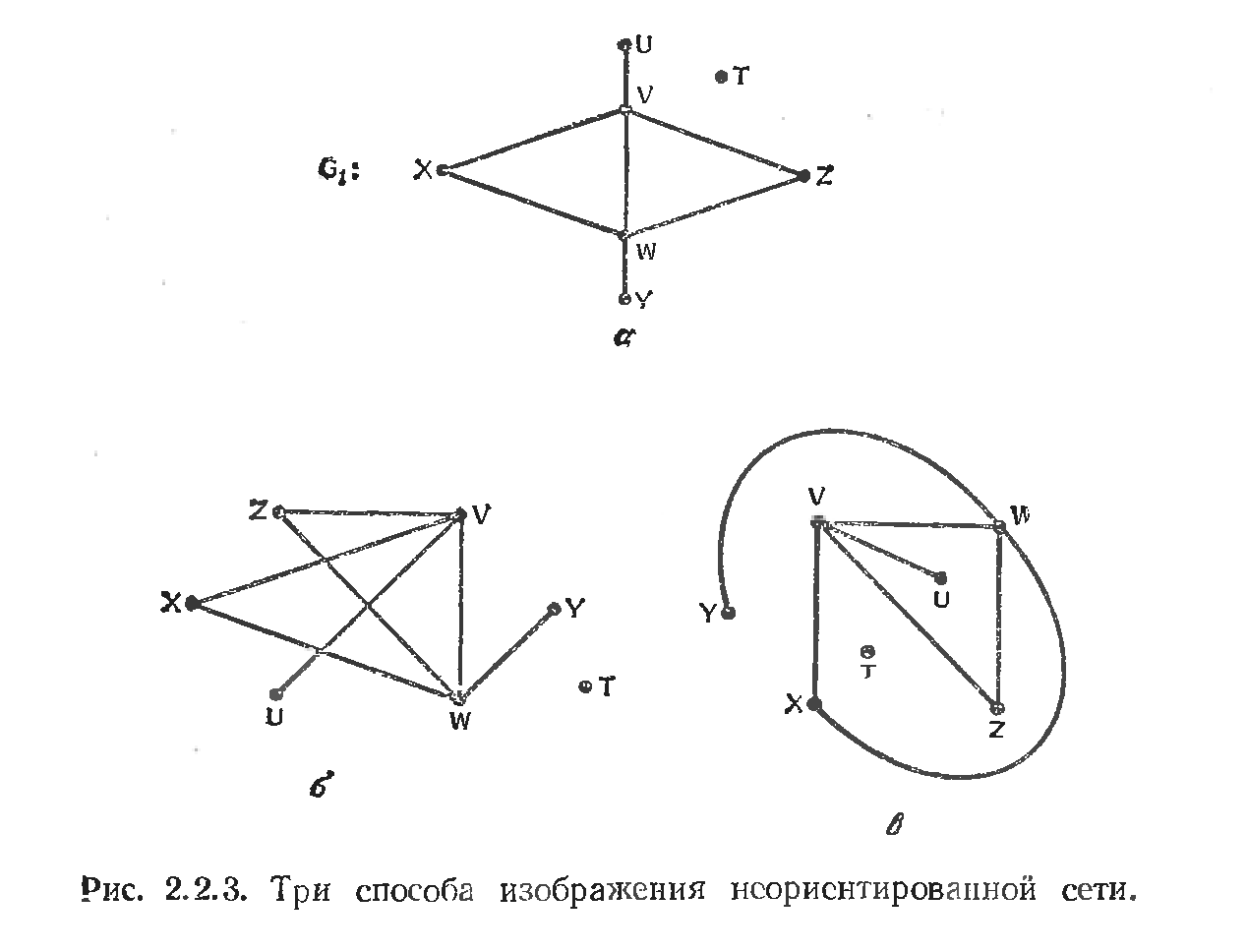


Рис. 2.9 Три способа изображения неориентированной сети

**Тема 3. Сложные структуры данных**

В этом разделе мы введём несколько структур данных (связанные списки, стеки и очереди) и процедуры для работы с ними, которые часто оказываются полезными.

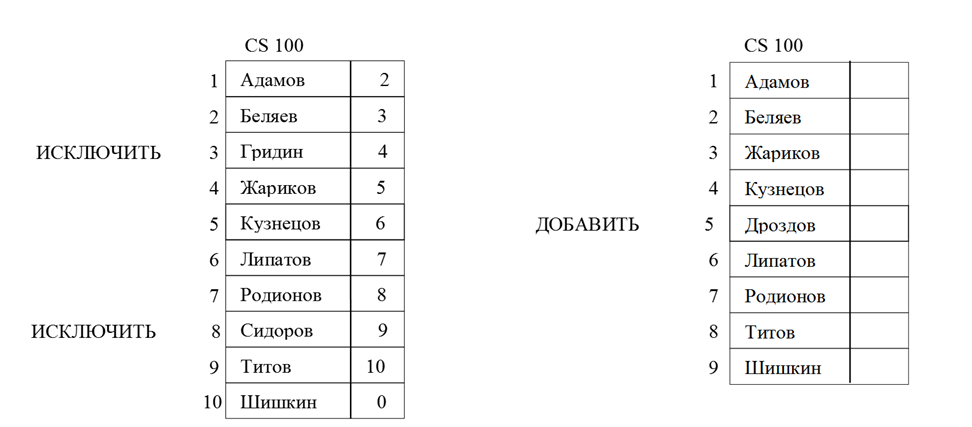
Рассмотрим преимущества и недостатки использования массивов.

Среди преимуществ отметим следующие:

* 1. Массивы помогают объединять множества данных в осмысленные группы.
  2. Имена массивов с индексами минимизируют потребность в слежении за многими элементами данных с различными именами.
  3. Использование индексов обеспечивает непосредственный и автоматический доступ к любому элементу в массиве.
  4. Индексация позволяет также производить с помощью циклов DO и FOR автоматическую, быструю и эффективную обработку всех данных или выделенных подмножеств данных, хранимых в массивах. В эту работу входит инициализация, поиск, хранение и модификация.

Недостатки массивов не так очевидны. Лучше всего массивы годятся для данных, значения которых не изменяются, порядок которых (важен он или не важен) также не изменяется. Если порядок элементов в массиве подвергается изменению, то каждый раз, когда порядок меняется, перестановка элементов требует очень много времени.

Рассмотрим, например, N-элементный массив CS100(N), в котором содержатся лексикографически упорядоченные имена студентов, слушающих в данный момент курс программирования CS100 (рис. 3.1, а).



а б

Рис. 3.1. Переорганизация линейного списка.

Если студенты Гридин и Сидоров перестают посещать курс, а новый студент Дроздов добавляется, то нам бы хотелось получить новый список слушателей, приведённый на рис. 1,б. Подумайте о трудности и стоимости написания и выполнения программы, которая смогла бы перестроить массив CS100 в соответствии с этими изменениями.

С другой стороны, связанный список представляет собой структуру данных, которая требует дополнительной памяти, но позволяет легко вносить такие изменения.

**Связанные списки**

На рис. 3.2,а массив CS100 преобразован из одномерного в двумерный массив CS100L. В столбце 1 массива CS100L по-прежнему содержатся фамилии студентов, зачисленных на курс CS100, хотя, как явствует из рис. 2,б, теперь уже не требуется, чтобы эти фамилии были упорядочены по алфавиту. В столбце 2 массива CS100L содержатся неотрицательные целые числа, называемые связями или указателями, значениями которых являются номера строк массива, содержащих фамилию следующего студента (в алфавитном порядке) в текущем списке. Звездочкой (\*) мы отметили те указатели, значения которых изменились в связи с удалением фамилий ГРИДИН и СИДОРОВ и добавлением ДРОЗДОВ. Заметим, что на рис. 3.2,б:



а б

Рис. 3.2. Переорганизация связанного списка.

АДАМОВ указывает на БЕЛЯЕВ {CS100L (1, 2)=2 и CS100L(2 ,1)= БЕЛЯЕВ}.

БЕЛЯЕВ указывает на ЖАРИКОВ {CS100L(2, 2) =4 и CS100L (4, 1)= ЖАРИКОВ}

КУЗНЕЦОВ указывает на ДРОЗДОВ.

ДРОЗДОВ указывает на ЛИПАТОВ и т.д.

Массив CS100L(I,J) размера N×2 работает как линейный связанный список. Линейный связанный список – это конечный набор пар, каждая из которых состоит из информационной части INFO (ИНФО) и указующей части LINK (СВЯЗЬ). Каждая пара называется ячейкой. Если мы хотим расположить ячейки в порядке *Ci1*,*Ci2,...*,*Cin,то* СВЯЗЬ (ij)=ij+1 для j=1,...,n-1,а СВЯЗЬ (in)=0 и указывает на конец списка.

На рис. 3,а приведена стандартная диаграмма линейного связанного списка; на рис 3,б приведена диаграмма связанного списка с рис. 3.2,б. Заметим, что ГРИДИН и СИДОРОВ отсутствуют в списке на рис 3.3,б хотя они присутствуют на рис 3.2,б как затемненные элементы. При реализации связанных списков участвует переменная FIRST(ПЕРВЫЙ) или HEAD (ГОЛОВА), значение которой есть адрес первой ячейки списка (рис 3.3,а).

Как было указано выше, одно из главных преимуществ связанных списков заключается в том, что можно легко удалять и добавлять элементы списка. Далее мы приводим два алгоритма, которые можно использовать при выполнении модификаций, требуемых для преобразования рис. 3.2,а в 3.2,б.

АДАМОВ 2

БЕЛЯЕВ 4

ЖАРИКОВ 5

Ряд 1 Ряд 2 Ряд 4

КУЗНЕЦОВ 11

РОДИОНОВ 9

ТИТОВ 10

ШИШКИН 0

ЛИПАТОВ 7

ДРОЗДОВ 6

Ряд 5 Ряд 11 Ряд 6

Ряд 7 Ряд 9 Ряд 10

ПЕРВЫЙ ИНФО СВЯЗЬ ИНФО СВЯЗЬ ИНФО СВЯЗЬ ИНФО СВЯЗЬ ИНФО СВЯЗЬ СВЯЗЬСВЯЗЬИИИнфоИНССВЯЗЬСВЯЗЬ

0

С1 С2  С3  СN-1 CN

а

б

Рис. 3.3. Диаграммы линейных связанных списков.

Первый алгоритм будет удалять заданный элемент из связанного списка. Этот процесс иллюстрируется на рис. 3.4, где мы ищем ячейку *С*i, у которой ИНФО (i)=ТРИ, передвигая указатели PREV и PTR до тех пор, пока такая ячейка не будет найдена. Затем мы заменяем значение СВЯЗЬ в ячейке, на которую указывает PREV, на значение СВЯЗЬ в ячейке, на которую указывает PTR. Далее следуют блок-схема (рис. 3.5), формальное описание и реализация этого алгоритма.

**Algorithm DELETE** (УДАЛЕНИЕ). Удаляется ячейка *Сi*, у которой INFO(i)=VALUE из связанного списка, первая ячейка которого задается переменной FIRST.

*Шаг 1 .*[Выбор первой ячейки] **Set** PTR ←FIRST; PREV← 0/

*Шаг 2*. [Ячейка пуста ?] **While** PTR ≠0 **do** шаг 3 **od.**

*Шаг 3.*[Это она?] **If** INFO (PTR)=VALUE

**then**[Это первая ячейка?]

**if** PREV=0 **then** [удалить спереди]

**set** FIRST←LINK (PRT); **and** STOP

**else** [удалить изнутри]

**set** LINK (PREV)←LINK(PTR);

**and** STOP **fi**

**else**[Выбор следующей ячейки]

**set** PREV←PTR; PTR←LINK(PTR) **fi**.

*Шаг 4* . [Не в списке ] НАПЕЧАТАТЬ «ЗНАЧЕНИЕ НЕ В СПИСКЕ»;

**and** STOP.

ОДИН

0

↑

PREV

↑

PTR

ДВА

ТРИ

ЧЕТЫРЕ 0

а

ОДИН

ДВА

ТРИ

ЧЕТЫРЕ 0

↑

PREV

↑

PTR

б

ТРИ

ДВА

ОДИН

ЧЕТЫРЕ 0

ОДИН

ДВА

ТРИ

ЧЕТЫРЕ 0

↑

PTR

↑

PREV

в

г

Рис. 3.4 Удаление ячейки из связанного списка.

Следующий алгоритм будет вносить в связанный список новую ячейку. Предположим, что в списке ячейки упорядочены в порядке возрастания содержимого поля ИНФО(INFO). Этот алгоритм аналогичен алгоритму DELETE (УДАЛЕНИЕ) в том отношении, что при выполнении операции внесения используются указатели PREV и PTR.

Начало

Выбрать первую ячейку

Ячейка

пуста?

Это

она?

Выбор

следующей

ячейки

Удалить спереди

Это

первая

ячейка? ??

Удалить изнутри

Напечатать

что не в списке

Окончание

Да

Да

Да

Нет

Нет

Нет

Рис. 3.5. Блок-схема DELETE (УДАЛЕНИЕ).

На рис. 3.6 проиллюстрирован процесс внесения, причем мы предполагаем, что:

ОДИН<ДВА<ТРИ<ЧЕТЫРЕ

****

Рис. 3.6. Внесение ячейки в связанный список.

На рис. 3.7 приведена блок-схема следующего алгоритма:

**Algorithm INSERT** (ВНЕСЕНИЕ). Внести новую ячейку ROW (РЯД), где INFO (ROW)=VALUE, в упорядоченной связанный список, первая ячейка которого задаётся в FIRST (ПЕРВЫЙ).

*Шаг 1*. [Выбор первой ячейки] **Set** PTR **←** FIRST; **PREV**←0

*Шаг 2*[Ячейка пуста ?]**While** PTR **≠ do** шаг 3 **od.**

*Шаг 3.*[Предшествует ли новая ячейка выбранной?] **If** VALUE≤ INFO (PTR)

**then**[вносится спереди?]

**if** PREV=0 **then**[внести новую ячейку спереди]

**set** FIRST←ROW;

LINK (ROW)←PTR;

**and** STOP

**else** [внести новую ячейку внутрь]

**set**  LINK(PREV)←ROW; LINK (ROW)←PTR;**and** STOP **fi**

**else** [Выбрать очередную ячейку]

**set** PREV← PTR;PTR←LINK(PTR) **fi.**

*Шаг 4*.[Список пуст?]  **If** PREV=0

**then**[внести новую ячейку как единственную ячейку в списке ]

**set** FIRST← ROW;LINK (ROW)←0;**and** STOP

**else** [внести новую ячейку в конец списка]

**set** LINK(PREV)←ROW; LINK (ROW)←0; **and** STOP **fi.**

Начало

Выбрать первую ячейку

Ячейка пуста?

Новая ячейка предшествует?

Выбор следующей ячейки

Добавлять спереди?

Добавить новую ячейку

спереди

Добавить новую ячейку внутрь

Список пуст?

Добавить

ячейку как единственную

Окончание

Да

Да

Да

Да

Нет

Нет

Нет

Нет

Добавить ячейку в конце списка

Рис. 3.7. Блок-схема INSERT (ВНЕСЕНИЕ).

**Стековые списки и стеки**

Мы увидели, что (линейные) связанные списки – это эффективная структура данных для моделирования ситуаций, в которых подвергаются изменениям упорядоченные массивы элементов данных. В частности, это справедливо для случая, когда модификациями являются главным образом внесение элементов в середину массивов или удаление элементов из середины массива . Когда модификации касаются лишь начала и конца, необходимость в связанных списках исчезает, и становятся достаточными простые линейные (одномерные) массивы.

В качестве примера рассмотрим следующую задачу. Во всех компиляторах с языков программирования требуется узнавать, является ли произвольное выражение правильно построенным. в частности, нужно определять, правильно ли расставлены в выражении скобки. Например, последовательность ( ) (( ) ( )) представляет правильную последовательность скобок, а ( ) ((( ) ( )) неправильную (есть лишние левые скобки). В более широком математическом контексте в арифметическом выражении обычно встречаются также квадратные скобки [ ] и фигурные скобки { }.

Предположим, что вас попросили разработать алгоритм определения того, что произвольная последовательность круглых, квадратных и фигурных скобок является правильно построенной. Что мы понимаем под правильно построенной последовательностью? Обычное определение состоит в следующем:

1. Последовательности ( ), [ ] и { } являются правильно построенными.
2. Если последовательность *х* правильно построенная, то правильно построены и последовательности (*х*), [*x*] и{*x*}.
3. Если последовательности *x* и *y* правильно построенные, то такова же и последовательность *xy.*
4. Правильно построенными являются лишь те последовательности, правильность которых следует из конечной последовательности применений правил 1,2 и ,3.

Это определение определяет правильно построенные последовательности конструктивно. Например, следующие последовательности являются правильно построенными:

Обычно, чтобы установить, являются ли выражения такого типа правильно построенными, используют ***стековую память***  (или просто стек), которая представляет собой бесконечную в одну сторону последовательность слов памяти, как изображено на рис. 3.8.

Для запоминания элементов данных в стеке элемент заносят в верхнее слово ТОР (ВЕРШИНА), сдвигая тем самым вниз (по одному слову) все другие слова, хранящиеся в стеке (совершенно аналогично тому, как в столовой складывают подносы в стопку). Выбор элементов данных из стека возможен лишь считыванием по одному за раз из слова ТОР, причем все остальные элементы данных сдвигаются вверх.

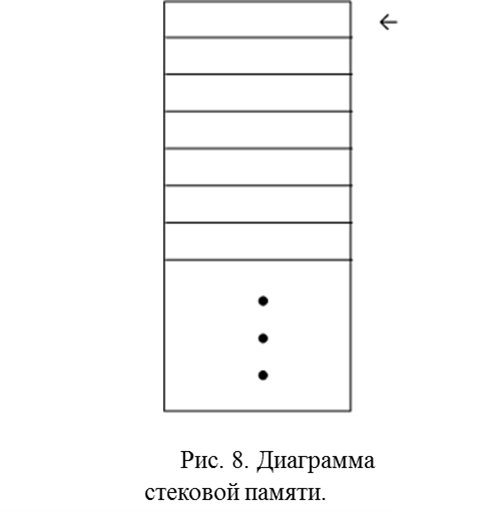


Рис. 3.8 Диаграмма стековой памяти

Мы не имеем права выбирать элементы данных внутри стека. (В столовой мы обычно не выбираем двадцатый поднос из стопки подносов.) Иногда термин «стек» обозначает стековую память, когда нам разрешено считывать элементы, находящиеся ниже слова ТОР, но не разрешается изменять их значения или добавлять новые элементы ниже слова ТОР.

Покажем на примере, как стековая память помогает нам легко определить, является ли такая последовательность как g={[ ]({[( )]}{ })}, правильно построенной. Элементы g будем обозначать  *х1, х*2,… , *х*n, где *x*i есть одна из скобок {, } , ( , ), [ или ]. Элементы {, ( и [ назовем ЛЕВЫМИ символами ; скажем, что *xi - левый партнер хj,* если *xi=*[ и *хj*=] , или *xi*=( и *xj*=), или *x*i ={ и *xj* =}/

**Algorithm WELLFORMED** (ПРАВИЛЬНО ПОСТРОЕННАЯ). Определяет, является ли произвольная последовательность символов *xi x2...xn,,*где каждый *xi -* одна из скобок {, }, (, ), [, или ], правильно построенной .

*Шаг 0.* [Инициализация ]**Set** TOP ←0; **I←1.**

*Шаг 1*.[ Читается последовательность слева направо]

**While I≤**N **do through**  шаг 3 **od/**

*Шаг 2.*[Записывается в стек ЛЕВЫЙ символ]

**If** *х*i ЛЕВЫЙ символ **then**[ записывается *xi*]

**else** [выбирается символ из стека ]

**if** TOP левый партнер *хi*

**then**  выбирать элемент ТОР из памяти

**else** НАПЕЧАТАТЬ « НЕПРАВИЛЬНО ПОСТРОЕННАЯ»;  **and**  STOP **fi fi**

*Шаг 3.*[Прочесть следующий символ ] **Set I ←I** +1.

*Шаг 4 .* [Память пуста?] **If** TOP=0  **then** PRINT « ПРАВИЛЬНО ПОСТРОЕННАЯ»;

**else**  PRINT « НЕПРАВИЛЬНО ПОСТРОЕННАЯ» **FI ; and** STOP.

Применим теперь алгоритм ПРАВИЛЬНО ПОСТРОЕННАЯ к последовательности

g={ [ ] ( { [ ( ) ] } { } ) }

1 2 3 4 5 6 7 8 9 10 11 12 13 14

На рис. 3.9. приведено содержимое стека, соответствующее операциям чтения элементов из g слева направо. Целое значение над **I-**й конфигурацией стека указывает на то, что в данный момент читается *xi-*й символ.

Для реализации стека на Фортране мы возьмем одномерный массив STORE [описанный как STORE (M)] и целую переменную с именем ТОР. Всегда, когда нам нужно поместить на вершину стека элемент Х(I), мы выполняем следующие простые команды:

ТОР ++;

if (TOP < M)

STORE (TOP)=X(I);

tlse cout << «Переполнение стека»;

Далее для выбора элемента выполняются следующие команды:

if (TOP)

{

X(I)=STORE(TOP);

TOP - -;

}

tlse cout << «Cтек пуст»;

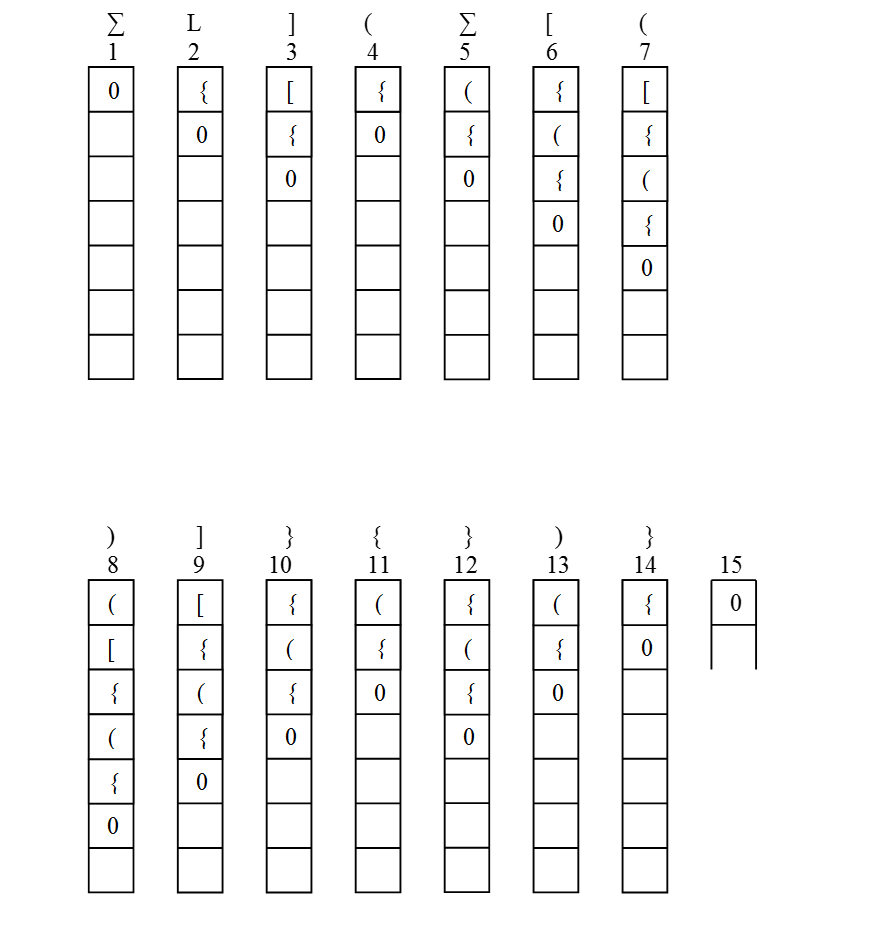


Рис. 3.9. Конфигурации стековой памяти при работе алгоритма

WELL-FORMED (ПРАВИЛЬНО ПОСТРОЕННАЯ).

**Очереди**

В стековой памяти для внесения и удаления элементов можно пользоваться только словом ТОР. В очереди элементы добавляются с одного конца, а выбираются с другого. Эти элементы обычно называются началом (FRONT) и концом (REAR) очереди. Термин «очередь» выбран потому, что эти структуры данных часто используются для моделирования обслуживающих линий, например люди в очереди к парикмахеру, автомобили у светофора, очередь заданий операционной системы.

Для моделирования очереди, как правило, используют линейный массив [например, QUEUE(500)] и две целые переменные FRONT и REAR , которые указывают на первые и последние элементы очереди соответственно. Вначале REAR≥FRONT очереди добавится свыше 500 элементов, то, по-видимому, некоторые записи будут удалены из начала очереди. Если это так, то, чтобы не допустить переполнения массива, мы присваиваем REAR=1 и продолжаем заполнять очередь с начала массива. Впрочем, REAR никогда не должен перегонять FRONT.

На рис.3.10 показано то, что называется ситуацией  *одна очередь - одно обслуживание.* В этом примере емкость очереди 5, конкретные клиенты помечены метками от А до Н; первый ожидающий в очереди клиент обслуживается за три единицы времени, после чего он покидает очередь. В момент Т=0 очередь пуста, и значения FRONT (F) и REAR(F) равны нулю. В момент Т=1 прибывает А, ждет 3 единицы времени и покидает очередь в момент Т=4. Клиенты B,C,D,E,G и H прибывают в моменты времени 2,3,4,7,8 и 9 соответственно. С ждет 4 единицы, прежде чем продвинуться в начало очереди, и покидает ее в момент Т=10. Когда прибывает G, в момент Т=8, конец очереди находится в массиве в положении 5. Здесь мы помещаем G в положение 1 очереди и присваиваем R=1. Когда в момент Т=9 прибываем Н, в R засылается 2, в этот момент очередь полностью заполнена. Теперь конец очереди сравнялся с ее началом, и, пока С не покинет очередь, в нее становится нельзя.

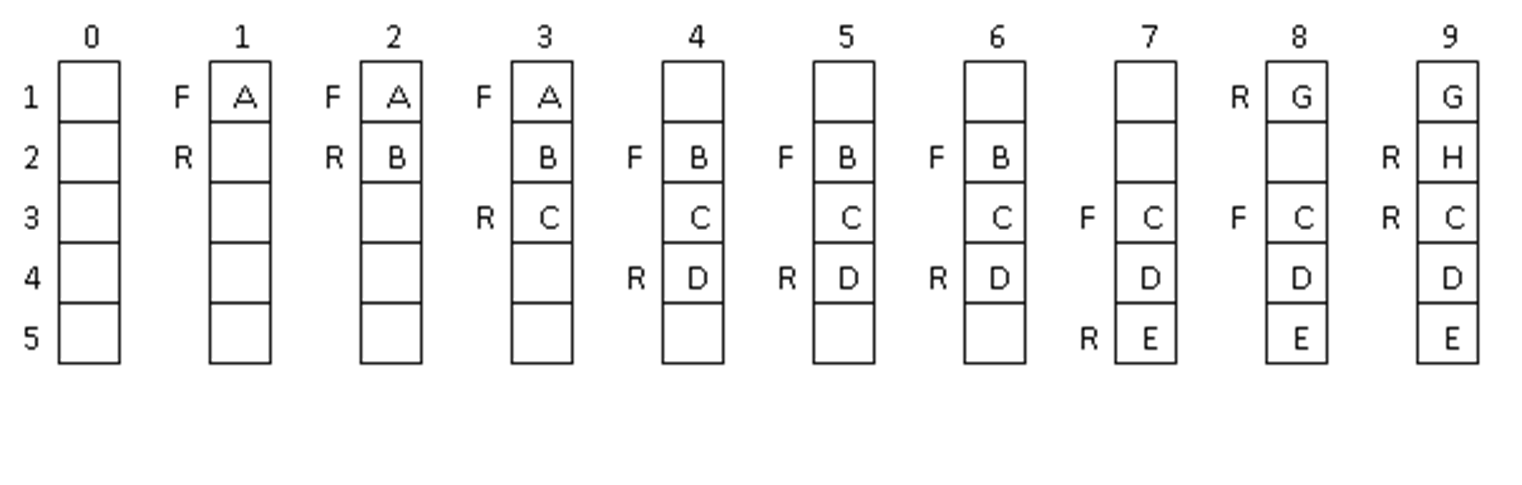


Рис. 3.10. Реализация очереди с использованием линейного массива.

**Тема 4. Методы разработки алгоритма**

Для разработки какой-либо программы прежде всего необходимо определить каким методом будет решаться поставленная задача. Для ответа на этот вопрос целесообразно — просмотреть свой запас общих алгоритмических методов для того, чтобы проверить, нельзя ли с помощью одного из них сформулировать решение новой задачи. Таких методов решения задач существует множество. В этой главе мы остановимся на основных из них:

- частных целей, подъема и отрабатывания назад - три самые общие метода решения задач;

- метод отхода и метод ветвей и границ - методы решения больших комбинаторных задач;

- рекурсия и итерация - инструменты, которые может в зна­чительной степени упрощают логическую структуру многих алго­ритмов;

- эвристика и имитация - два ши­роко используемых «житейских» метода.

Эти методы необязательно действуют обособленно, как правило решение задачи основывается на нескольких методах, один из которых доминирует. Кроме того возникает ситуация когда один метод основывается на другом. Например, метод эвристики очень часто использует алгоритм решения задачи с помощью метода частных целей.

Цель этой главы — построить и пояснить вышеназванные фундаментальные методы разработки алгоритмов и решения задач. Для студентов изучающих данный курс необходимо прежде всего овла­деть этими основными методами и понятиями.

**Методы частных целей, подъема и отрабатывания назад**

Как разработать хороший алгоритм? С чего начать? У всех нас есть печальный опыт, когда смотришь на задачу и не знаешь, что делать. Прежде чем самому предлагать свой алгоритм, необходимо подумать какой метод, из уже разработанных, подходит для решения поставленной задачи. В этом разделе кратко описаны три общих метода решения задач, по­лезных для разработки алгоритмов.

Для начала дадим определения каждому методу, а потом их поясним

**Метод частных целей** - когда решение первоначальной задачи разбивается из решения более простых задач.

**Метод подъема** - метод постепенного приближения к цели от начального предположения к такому, при котором дальнейшее приближение не возможно.

**Метод отрабатывания назад** - метод, использующий путь решения от результата к начальным условиям.

Теперь поясним эти методы более подробно.

Первый метод связан со сведением трудной задачи к последова­тельности более простых задач. Конечно, мы надеемся на то, что более простые задачи легче поддаются обработке, чем первоначальная зада­ча, а также на то, что решение первоначальной задачи может быть получено из решений этих более простых задач. Такая процедура на­зывается методом частных целей.

Этот метод выглядит очень разумно. Но, как и большинство общих методов решения задач или разработки алгоритмов, его не всегда легко перенести на конкретную задачу. Осмысленный выбор более простых задач — скорее дело искусства или интуиции, чем науки. Более того, не существует общего набора правил для определения класса задач, которые можно решить с помощью такого подхода. Размышление над любой конкретной задачей начинается с постановки вопросов. Частные цели могут быть установлены, когда мы получим ответы на следующие вопросы:

1. Можем ли мы разбить данную задачу на отдельные подзадачи?

2. Можем ли мы решить часть из этих подзадач?

Можно ли, игнорируя не­которые условия, решить оставшуюся часть задачи?

3. Можем ли мы решить задачу для частных случаев?

Можно ли разработать алгоритм, который дает решение, удовлетворяющее всем условиям задачи, но входные данные которого ограничены некоторым подмножеством всех входных данных?

4. Есть ли что-то, относящееся к задаче, что мы не достаточно хо­рошо поняли? Если попытаться глубже вникнуть в некоторые особенности задачи, сможем ли мы что-то узнать, что поможет нам подойти к решению?

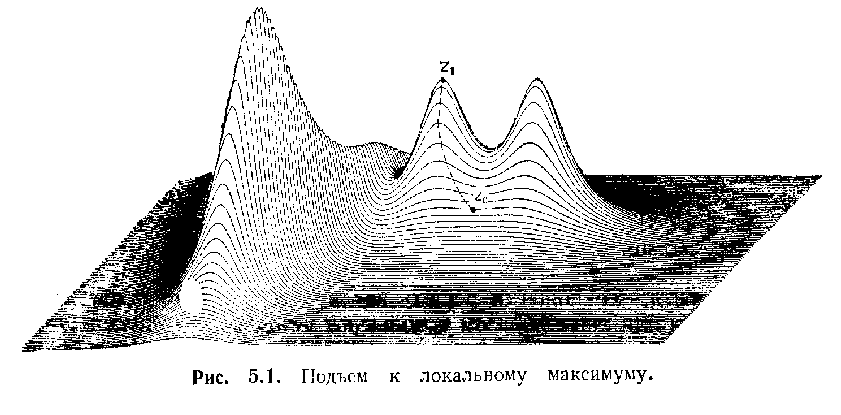
5. Встречались ли мы с похожей задачей, решение которой извест­но. Можно ли видоизменить ее решение для решения нашей задачи? Возможно ли, что эта задача эквивалентна известной нерешенной задаче?

Второй метод разработки алгоритмов известен как метод подъема. Алгоритм подъема начинается с принятия начального предположения или вычисления начального решения задачи. Затем начинается на­сколько возможно быстрое движение «вверх» от начального решения по направлению к лучшим решениям. Когда алгоритм достигает такую точку, из которой больше невозможно двигаться наверх, алгоритм ос­танавливается. К сожалению, мы не можем всегда гарантировать, что окончательное решение, полученное с помощью алгоритма подъема, будет оптимальным. Этот «дефект» часто ограничивает применение метода подъема.

Название «подъем» отчасти происходит от алгоритмов нахождения максимумов функций нескольких переменных. Предположим, что f(x, у) — функция переменных х и у и задача состоит в нахождении максимального значения этой функции. Проиллюстрируем данную задачу рисунком 4.1. Функция не нем представлена по­верхностью (имеющей холмы и впадины) над плоскостью х у. Алгоритм подъема может начать работу в любой точке Zî этой поверхности и проделать путь вверх к вершине в точке Z1. Это значение является «локальным» максимумом в отличие от «глобально­го» максимума, и метод подъема, как мы видим, не дает оптимальною решения. Т.е. мы нашли, максимальное значение функции - что требовалось в задаче, но это «локальный» максимум, а не «глобальный», т.е. не самое максимальное значение. Вообще методы подъема являются «грубыми». Они запоминают некоторую цель и стараются сделать все, что могут и где могут, чтобы подойти ближе к цели. Это делает их несколько недальновидными. Как показывает наш пример, алгоритмы подъема могут быть по­лезны, если нужно быстро получить приближенное решение.

Третий метод, рассматриваемый в этом разделе, известен как отрабатывания назад, т. е. начинаем с цели или решения и движемся обратно по направлению к начальной постановке задачи. Затем, если эти действия обратимы, движемся обратно от постановки задачи к ре­шению. Многие из нас делали это, решая головоломки в развлекатель­ных разделах воскресных газет.

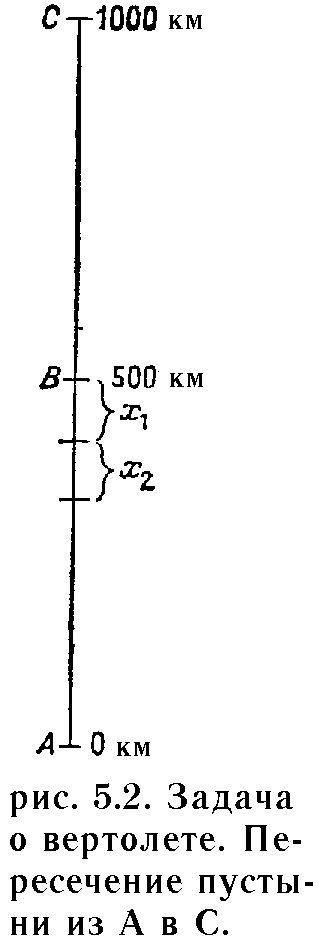
Давайте постараемся применить все три метода этого раздела в довольно трудном примере.



**Рис. 4.1** Подъем к локальному максимуму

**Задача о вертолете**

Мы хотели бы сделать перелет на легком вертолете через пустыню, которая тянется на 1000 километров, израсхо­довав при этом минимум горючего. Максимальный объем горючего, которое может захватить с собой вертолет, предположим, составляет 500 литров, горючее расходуется равномерно, по 1 литру на километр. В точке старта имеется неограниченный резервуар с топливом. Так как в пустыне нет складов горючего, мы должны установить свои соб­ственные хранилища и наполнить их топливом. Задача будет решена, если мы ответим на следующие вопросы. Где расположить эти хранилища? Сколько горючего нужно залить в каж­дое из них?



**Рис. 4.2** Задача о вертолете. Пересечение пустыни из А в С.

Подойдем к этой задаче с помощью метода отрабатывания назад. На какое расстояние от конца мы сможем пересечь пустыню, имея за­пас горючего в точности на *k* баков? Мы будем задавать этот вопрос для k=1, 2, 3, ..., пока не найдем такое целое *п,* что *п* полных баков позволят пересечь всю 1000-километровную пустыню.

Для k=1 ответ равен 500 милям, как и показано на рис. 4.2. Можно заправить машину в точке В и пересечь оставшиеся 500 километров пустыни. Ясно, что это наиболее отдаленная точка, стартуя из которой можно преодолеть пустыню, имея в точности 500 литров горючего.

Мы поставили перед собой частную цель, потому что не смогли бы решить сразу исходную задачу. Мы не задаем вопрос: сколько топлива нужно вертолету, чтобы преодолеть заданную дистанцию? Вместо этого задаем более простой, но родственный вопрос: какое расстояние можно пролететь на заданном количестве топлива? Ответить на первый вопрос становится воз­можным, когда ответом на второй является: не мень­ше 1000 километров.

Предположим, что k=2, т. е. имеется два полных бака (1000 литров). Будем рассматривать этот слу­чай, опираясь на результат для k=1. Данная ситуа­ция иллюстрируется на рис. 4.2. Каково максималь­ное значение X1, такое, что, отправляясь с 1000 литрами горючего из точки 500— X1, можно перевезти горючего в точку В столько, чтобы из точки В мы вылетели с 500 литрами горючего (т.е. как в случае k=1).

Один из способов определения приемлемого зна­чения Xi состоит в следующем. Заправляемся в точ­ке 500—X1, едем X1 километров до В и переливаем в хранили­ще все горючее, кроме X1 литров, которые потребуют­ся для возвращения в точку 500-X1. В этой точке бак становится пус­тым. Теперь наполняем его снова, проезжаем X1 километров до В, забираем в В горючее, оставленное там, и из В едем в С с полным ба­ком. Общее пройденное расстояние состоит из трех отрезков по X1 километров и одного отрезка ВС длиной 500 километров. Мы должны израсходовать каждую каплю топлива, чтобы сделать значение X1 как можно боль­шим. Поэтому X1 находим из уравнения

3 Õ1 + 500 =1000 (литров);

его решение: Õ1=500/3.

Таким образом, два бака (1000 литров) позволяют нам пролететь расстояние

D2=500+X1=500 (l+1/3) километров.

Заметим, что исходная предпосылка является недальновидной и грубой. Когда имеются в распоряжении k баков горючего, мы просто стараемся продвинуться назад как можно дальше от точки, найденной для k—1 баков.

Рассмотрим k=3. Из какой точки мы можем вылететь с 1500 литрами топлива так, что вертолет сможет доставить 1000 литров в точку 500—X1? Возвращаясь к рис. 5.2, мы ищем наибольшее значение Х2, такое, что, выезжая с 1500 литрами топлива из точки 500—X1—Х2, мы можем доставить 1000 литров в точку 500—X1. Мы выезжаем из точки 500—X1—Х2, доезжаем до 500—X1, переливаем все горючее, кроме Х2 литров, и возвращаемся в точку 500—X1—Х2 с пустым баком. Повторив эту процедуру, мы затратим 4Õ2 литров на проезд и оставим 1000 — 4Õ2 литров в точке 500—X1. Теперь в точке 500—X1—Х2 осталось ровно 500 литров. Заправляемся послед­ними 500 литрами и едем в точку 500—X1, израсходовав на это Х2 литров.

Теперь мы находимся в точке 500— X1, затратив на проезд 5Õ2 литров топлива. Здесь оставлено в общей сложности 1500—5Х2 литров. Это количество должно быть равно 1000 литрам, т. е. Х2==500/5. Из этого заключаем, что 1500 литров позволяют нам пролететь

D = 500 + X1+Õ2 = 500 (l+1/3+1/5) километров.

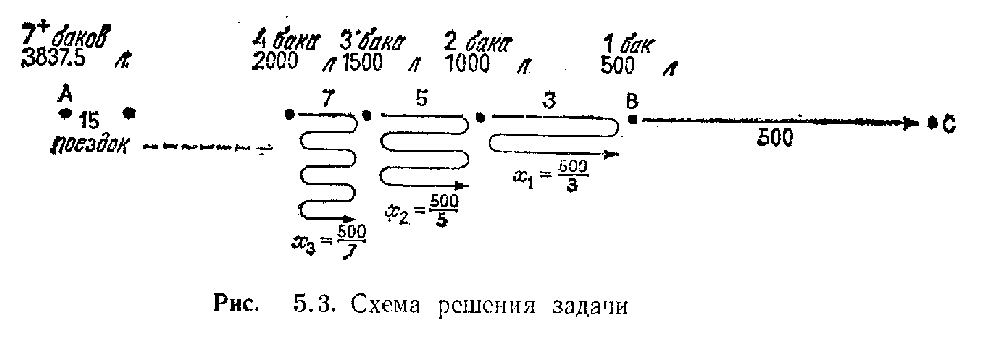
Продолжая индуктивно процесс отрабатывания назад, получаем, что n баков горючего позволяют нам пролететь Dn километров, где

D =500 ( 1+1/3+1/5+...+1/(2n-1) ).

Нужно найти наименьшее значение п, при котором Dn = 1000. Простые вычисления показывают, что для n=7 имеем D7=977,5 километра, т. е. семь баков, или 3500 литров, топлива дадут нам возможность пролететь 977,5 мили. Полный восьмой бак — это было бы уже больше, чем нам надо, чтобы перевезти 3500 литров из точки А в точку, отстоящую на 22,5 мили (1000—977,5) от А. Читателю представляется самостоятельно убедиться в том, что для доставки 3500 литров топ­лива к отметке 22,5 мили достаточно 337,5 литра. Таким образом, для того чтобы пересечь на вертолете пустыню из А в С, нужно 3837,5 литра горючего.

Теперь алгоритм транспортировки горючего может быть пред­ставлен следующим образом. Стартуем из А, имея 3837,5 литра. Здесь как раз достаточно топлива, чтобы постепенно перевезти 3500 литров к отметке 22,5 мили, где мы в конце концов окажемся с пустым баком и запасом горючего на семь полных заправок. Этого топлива достаточно, чтобы перевезти 3000 литров к точке, отстоящей на 22,5+500/13 километров от А, где бак машины будет опять пуст. После­дующие перевозки приведут нас к точке, отстоящей на 22,5+500/13+500/11 километров от А, с пустым баком машины и 2500 литрами.

Продолжая таким образом, мы продвигаемся вперед благодаря анализу, проведенному методом отрабатывания назад. Вскоре мы окажемся у отметки 500 (1—1/3) = 1000/3 километров с 1000 литрами топ­лива. Затем мы перевезем 500 литров в В, зальем их в бак вертолета и долетим без остановки до С. Рис.4.3 иллюстрирует весь этот процесс.



**Рис. 4.3** Схема решения задачи

Для тех, кто знаком с бесконечными рядами, заметим, что Dn есть n-я частная сумма нечетного гармонического ряда. Поскольку этот ряд расходится, алгоритм дает возможность пересечь любую пустыню. Как можно модифицировать этот алгоритм для того, чтобы оставить достаточно горючего в различных точках пустыни для воз­вращения в А?

Возникает вопрос, можно ли пролететь 1000 километров, затратив меньше чем 3837,5 литра горючего. Оказывается, что нельзя. Доказатель­ство этого факта довольно сложно. Однако можно высказать следую­щий, довольно правдоподобный довод. Очевидно, мы действуем наилучшим образом для k=1. При k=2 мы используем наш план для k=1 и затем вводим в действие второй бак горючего для того, чтобы оказаться как можно дальше от В. Исходная предпосылка для k баков заключается в том, что мы знаем, как действовать наилучшим образом в случае с k—1 баками, и отодвигаемся как можно дальше назад с помощью k-ro бака.

**Метод эвристики**

Метод решения задачи на интуитивном уровне, правильность которого доказать не возможно, но тем не менее, который дает хорошее, но не обязательно оптимальное решение называют **эвристический метод**.

Или другими словами эвристический алгоритм, или эвристика, определяется как алго­ритм со следующими свойствами:

1. Он обычно находит хорошие, хотя не обязательно оптимальные решения.

2. Его можно быстрее и проще реализовать, чем любой известный точный алгоритм (т. е. тот, который гарантирует оптимальное решение).

Понятия «хороший» н «обычно» в свойстве 1 меняются от задачи к задаче.

Возникает вопрос почему решение многих задач основано на этом методе, который дает не обязательно оптимальное решение. Дело в том, что иногда оптимальное решение достичь не возможно, либо процесс достижения оптимального решения настолько трудоемок, что проще получить пусть не оптимальное, но достаточно хорошее решение за значительно меньшее время. Например, если для решения задачи все известные точные алгоритмы требуют нескольких лет машинного времени, то мы можем охотно принять любое нетривиальное приближенное решение, которое может быть получено за разумное время. С другой стороны, имея быстрое, близкое к оптимальному решение, мы можем стремиться все же к точному решению.

Хотя не существует универсальной структуры, которой можно описать эвристические алгоритмы, многие из них основываются или на методе частных целей, или на методе подъема.

Рассмотрим при каких случаях возникает эвристический алгоритм.

- когда решение задачи не достижимо, или очень трудоемко традиционными методами;

- когда не возможно доказать правильность используемого алгоритма

Итак, для первого случая общий подход к построению эвристических алгоритмов заключается в перечислении всех требований к точному решению и разделении требований на два класса, например:

1. Те, которые легко удовлетворить.

2. Те, которые не так легко удовлетворить.

Или

1. Те, которые должны быть удовлетворены.

2. Те, по отношению к которым мы могли бы пойти на компромисс.

Тогда цель построения алгоритма — создать алгоритм, гарантиру­ющий выполнение требований 1-го класса, но не обязательно 2-го. Это не означает, что для удовлетворения требований 2-го класса не делается никаких попыток, это просто означает, что не может быть дано никаких гарантий, что они будут удовлетворены.

По отношению ко второму случаю можно сказать, что часто очень хорошие алгоритмы должны рассматриваться как эвристические. Например, предположим, что мы построили быстрый алгоритм, который, кажется, работает на всех тестовых задачах, но мы не можем доказать, что алгоритм правильный. Пока не дано такое доказательство, алгоритм следует рассматривать как эвристику.

В качестве примера рассмотрим следующий «грубый» алгоритм решения задачи коммивояжера.

.

**Транспортная задача.**

*Приближенный алгоритм решения транспортной задачи.* Решить задачу ком­мивояжера с *N* городами, последовательно двигаясь из города n в город n+1, соблюдая условие: стоимость проезда между городом n и новым городом n+1 минимальна. Индекс n изменяет значение от 1 до N.

*Шаг 0.* [Инициализация, т. е. установка в начальное состояние]

**TOUR=0;** МIN=∞.

*Шаг 1.* [Посещение всех городов]

**For** I=1 to (N—1)

*Шаг 2.* [Выбор следующего пути]

Определение минимальной стоимости проезда из города n во все

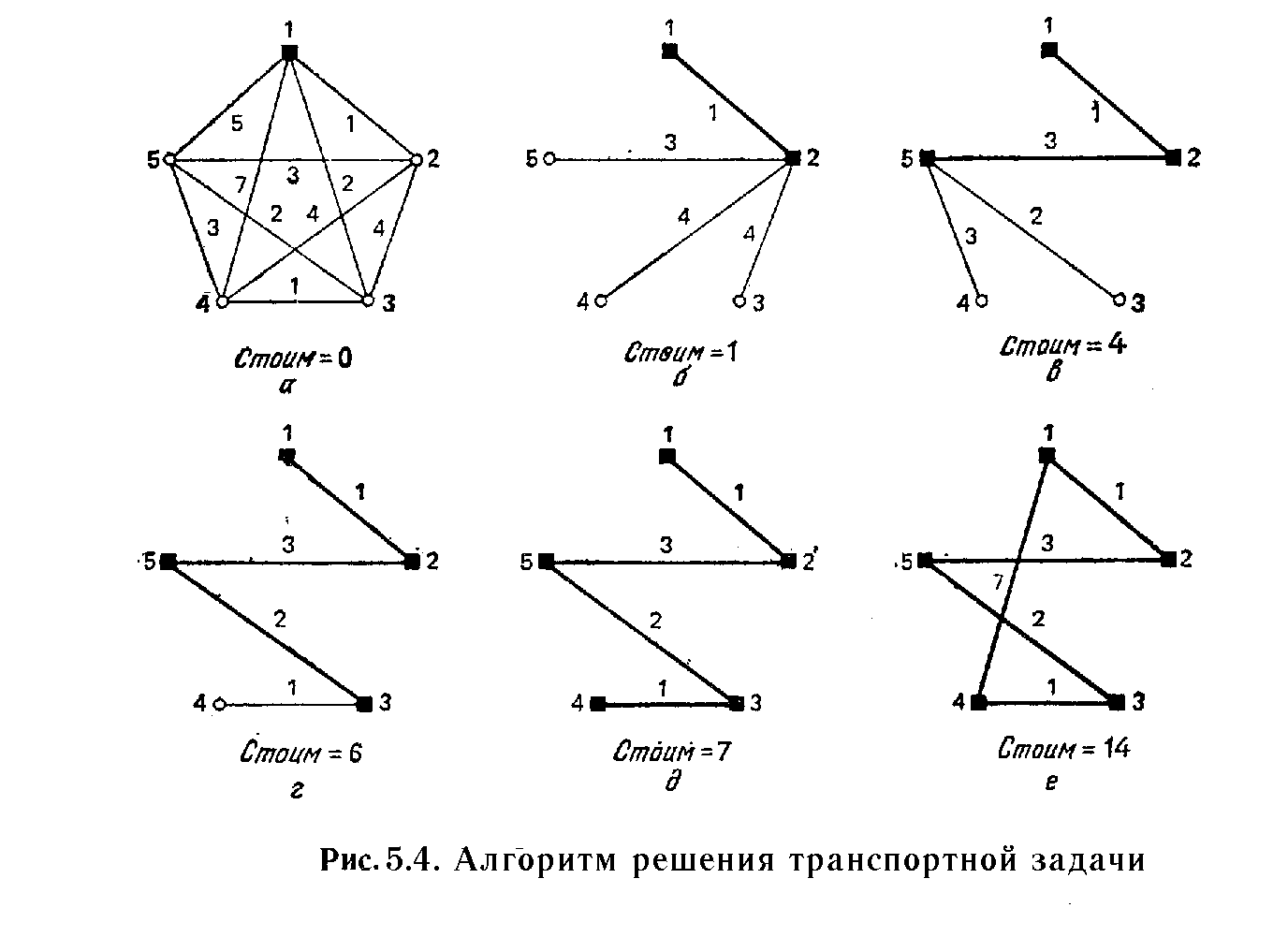
оставшееся города и определение города n+1.

*Шаг 3.* [Вычисление стоимости проезда]

Вычисляем стои­мость COST (Т (Р)) как сумму всех стоимостей от начальной точки до точки V+1.

*Шаг 4.* [Окончание цикла]

*Ш****аг 5***[Вывод результатов и окончание программы]



На рис. 5.4а изображена сеть городов с соответствующими стоимостями проезда, а рис.5.4б - 5.4е иллюстрируют построение тура коммивояжера для эвристического алгоритма, начинающегося с вершины 1; пройдённые вершины обозначены черным квадратиком, а непройденные — кружком.

Алгоритм эвристики дает общий проезд со стоимостью 14, а в гл.3 решение исчерпывающего алгоритма дает общий проезд со стоимостью 13. Ясно, что эвристический алгоритм не всегда находит общий проезд с минимальной стоимостью.

Обычно грубые алгоритмы очень быстры и интуитивно привлека­тельны, но, как мы заметили, они не всегда дают оптимальное решение задачи. Нам повезло, и мы смогли легко найти контрпример для алгоритма эвристического метода транспортной задачи. Однако не всегда можно легко показать, что грубый алгоритм не работает. Разумеется, если мы считаем, что наш грубый алгоритм работает всегда, то мы должны доказать, что это так.

Эвристический алгоритм транспортной задачи основан на идее подъема. Цель — найти общий проезд с минимальной стоимостью. Задача сведена к набору частных целей — найти на каждом шаге «самый дешевый» город, чтобы посетить его следующим. Алгоритм не строит плана вперед; текущий выбор де­лается безотносительно к последующим выборам.

Оптимальное решение задачи коммивояжера имеет два основных свойства:

1. Оно состоит из множества ребер, вместе представляющих общий проезд.

2. Стоимость никакого другого общего проезда не будет меньше данного.

Эвристический алгоритм рассматривает свойство 1 как «обязательное», или «легкое», требование, а свойство 2 — как «трудное», относительно которого можно пойти на компромисс.

Пример с пятью городами на рис. 5.4 сразу показывает, что ал­горитм приведенный выше не гарантирует свойство 2. Однако на шаге 2 делается некоторая попытка снизить стоимость общего проезда*.*

Конечно, для алгоритма с помощью метода эвристики легко написать программу, но яв­ляется ли он быстрым? Для произвольной задачи коммивояжера с *п* городами требуется операций *пропорционально N2*, чтобы прочесть или построить матрицу стоимостей *С.* Поэтому нижняя граница сложности *любого* алгоритма, способного дать нетривиальное возможное решение этой задачи по существующему методу равна N2*.* Поэтому представленный алгоритм настолько быстрый, насколько возможно.

По-видимому, представленный алгоритм — это хороший эвристический алго­ритм. Конечно, понятие «хороший» относительно. Поскольку алго­ритм «исчерпывающий коммивояжер» слишком неэффективен, эпитет «хороший» может просто указывать на тот факт, что представленный алгоритм — наилучший из имеющихся для больших (в разумных пределах) значений N.

**Программирование с отходом назад**

Часто в программировании встречаются задачи в которых необходимо перебрать все возможные комбинации некоторой величины, или ситуация поиска некоторого значения из списка параметров. Встает задача как правильно организовать перебор комбинаций или осуществить поиск, не пропустив при этом ни одного значения. Причем, очень часто складывается ситуацию в которой, при таком переборе необходимо учесть те комбинации на которые условиями задачи поставлен запрет.

Один из методов такой организации вышеназванных процессов является программирование с отходом назад.

Метод разработки алгоритма, известный как **программирование с отходом назад**, можно описать как организованный исчерпывающий поиск, который часто позволяет избежать исследования всех возмож­ностей.

Этот метод особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений.

Для более полного представления этой задачи ее целесообразно посмотреть на примере задачи о замке с шифр.

**Задача о замке**

В качестве примера рассмотрим комбинационный замок допустим для вело­сипеда, состоящий из набора N переключателей, каждый из которых может быть в положении «вкл» или «выкл». Замок открывается только при одном наборе положений переключателей, из которых не менее [N/2] (целая часть от N/2) находятся в положении «вкл». Предпо­ложим, что мы забыли эту комбинацию, а нам надо отпереть замок. Предположим также, что мы готовы перепробовать (если необходимо) все комбинации. Нам нужен алгоритм для систематического генери­рования этих комбинаций. Если проигнорировать условие [N/2], то для замка существует 2N возможных комбинаций. Однако условие [N/2J позволит отбросить (или лучше не генерировать) многие комбинации.

Промоделируем каждую возможную комбинацию вектором из N нулей и единиц. На i-м месте будет 1, если i-й переключатель нахо­дится в положении «вкл», и 0, если i-й переключатель — в положении «выкл». Множество всех возможных N-векторов хорошо моделируется с помощью двоичного дерева. Каждая вершина k-го уровня этого дерева будет соответствовать определенному набору первых k компонент N-вектора. Две ветви, идущие вниз из вершины этого уровня, соответствуют двум возможным значениям (k+1)-й компоненты в N-векторе. У дерева будет N уровней. Рис. 5.5 на примере N=4 поясняет основную конструкцию.

Условие, заключающееся в том, что число переключателей в положении «вкл» должно быть не меньше [N/2], позволяет нам не образовывать части дерева, которые не могут привести к правильной комбинации. Например, рассмотрим вершину 00 на рис. 5.5. Так как правая ветвь (к 000) не может привести к допустимой комбинации, нет нужды ее формировать. Если какие-то вершины, следующие за рассматриваемой вершиной, не удовлетворяют ограничению задачи, то эти вершины не надо рассматривать. В данном случае никакие из вершин, находящихся внутри пунктирных линий, не нужно исследо­вать и даже формировать.

Теперь, воспользовавшись этой моделью двоичного дерева, можно изложить процедуру отхода назад для образования только тех ком­бинаций, в которых по крайней мере [N/2] переключателей нахо­дятся в положении «вкл». Алгоритм сводится к пересечению дерева.

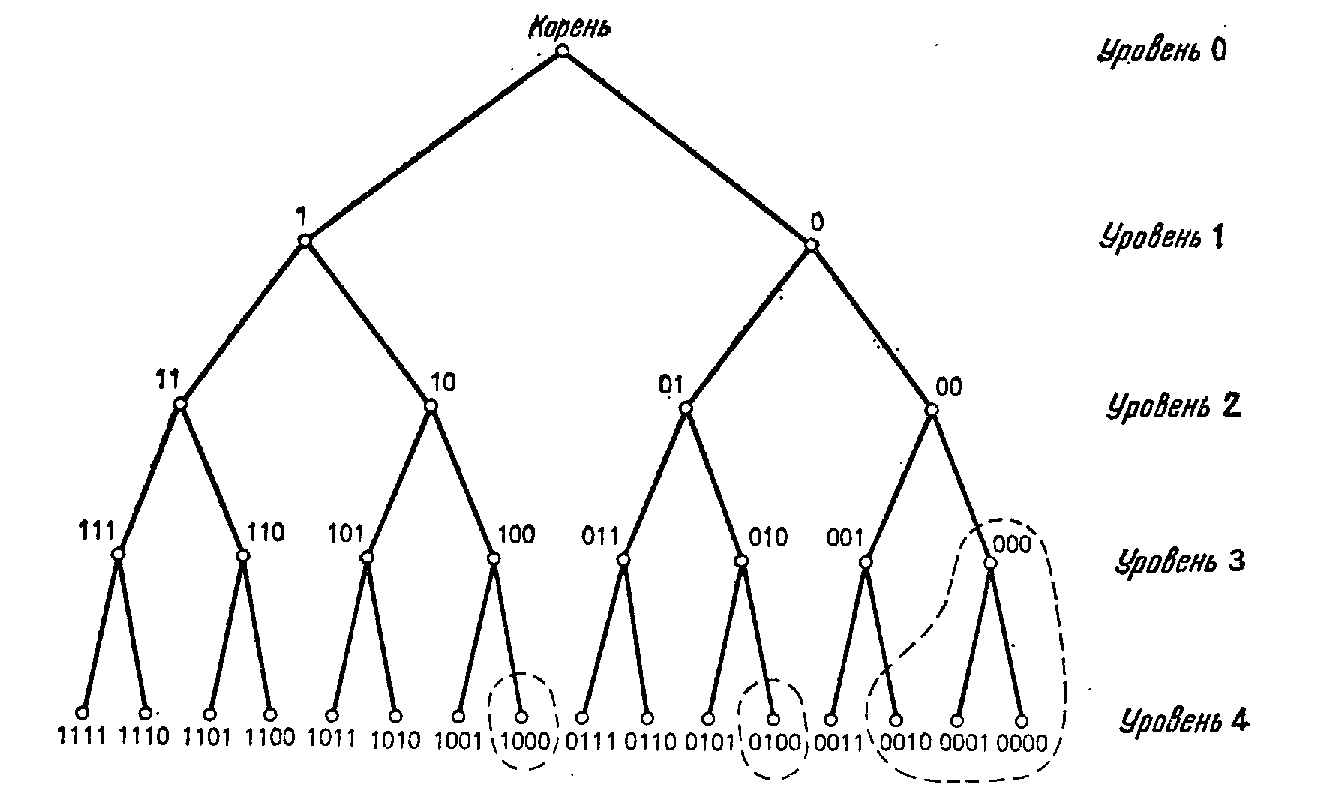


Рис. 5.5 Двоичное дерево, представляющее N-векторы из нулей и единиц.

Двигаемся вниз по дереву, придерживаясь левой ветви, до тех пор, пока это возможно. Достигнув конечной вершины, опробуем соот­ветствующую комбинацию.. Если она не подходит, поднимаемся на один уровень и проверяем, можем ли мы спуститься опять по другой ветви. Если это возможно, берем самую левую из неисследованных ветвей. Если нет, отходим вверх еще на один уровень и пытаемся спуститься из этой вершины. Перед спуском проверяем, можно ли удовлетворить условие об [N/2] включенных переключателях в последующих вершинах. В ситуации, изображенной на рис. 5.5, этот алгоритм просмотрит следующую последовательность вершин:

0001

0011

0111

1111 Проверка этой комбинации.

0111 Отход, так как 1111 —конечная вершина.

1110 Спуск по единственной непросмотренной ветви вниз, про­верка этой комбинации.

0111 Снова отход.

0011 Отход дальше, так как из 0111 все ветви просмотрены.

0110 Спуск по единственной непросмотренной ветви.

1101 Проверка комбинации.

0110 Отход из 1101

1100 Проверка комбинации.

0110 Отход из 1100.

0011 Отход, так как все ветви просмотрены.

0001 Отход, так как все ветви просмотрены.

0010 Спуск по единственной непросмотренной ветви.

0101 Спуск по самой левой непросмотренной ветви.

1011 Проверка комбинации.

0101 Отход.

1010 Проверка комбинации.

0101 Отход.

0010 Отход.

0100 Спуск по единственной непросмотренной ветви.

1001 Проверка комбинации.

0100 Отход.

0010 Отход; заметим, что мы не опускаемся к 1000, так как эта вершина нарушает условие о том, что должно быть по край­ней мере две единицы.

0001 Отход.

Корень Отход.

0 Спуск по единственной непросмотренной ветви.

.

.

.

и т. д.

Алгоритм останавливается, когда мы возвращаемся к корню и не остается непросмотренных ветвей.

Этот простой пример иллюстрирует основные свойства, общие для всех алгоритмов с отходом назад. Если можно сформулировать задачу так, что все возможные решения могут быть образованы построением N-векторов, тогда ее можно решить при помощи процедуры с отходом.

**Метод ветвей и границ**

Метод, известный как метод ветвей и границ, похож на методы с отходами назад тем, что он исследует древовидную модель простран­ства решений и применим для широкого круга дискретных комбина­торных задач. Алгоритмы с отходами нацелены на то, чтобы найти одну или все конфигурации, моделируемые N-векторами, которые удовлетворяют определенным свойствам. Алгоритмы ветвей и границ ориентированы в большей степени на оптимизацию.

В этом методе определяется некая функция, как весовая числовая функция для различных вариантов решения задачи и вычисляется численные значения этой функции для различных вариантов решения. Цель — найти конфигурацию, на которой эта функция достигает максимального или минималь­ного значения. Сам же метод отыскания и построения такой функции получил название метод ветвей и границ. Такое название возникло в результате того, что в нем используется ветвление и вычисляются границы.

Рассмотрим более подробно что понимается под ветвлением и что понимается под границей.

Ветвление - это процесс разбиения всех вариантов решения задачи на два множества, таким образом, что какая-либо конкретная подзадача принадлежит только одному из этих множеств. В этом случае говорят, что ветвление происходит на этой подзадаче. Если пользоваться терминологией древовидной структуры, то говорят, что ветвление происходит на определенной ветви древовидной структуры.

Поясним это на примере.

Как конкретно определить подзадачу ветвления рассмотрим на примере транспортной задачи с коммивояжером, в которой очень хорошо работает алгоритм ветвей и границ. В данном разделе изложение будет вестись на примере этой задачи. Сделаем замечание, что алгоритмы ветвей и границ, как правило, бывают довольно сложны и представленный здесь — не исключение. Тем не менее, в некоторых случаях, его использование бывает очень эффективным. При помощи алгоритмов ветвей и границ удалось решить большой круг разнообразных серь­езных практических задач, но эти алгоритмы редко оказыва­ются простыми.

Напомним, что задача заключается в нахождении в торговом участке коммивояжера пути из N городов с наименьшей стоимостью. Каждый город входит в путь только один раз. В терминах сетей в сети городов надо найти покрывающий цикл наименьшей стоимости. Имеется матрица С, каждый элемент которой Сij равен стоимости (обычно в единицах времени, денег или расстояния) прямого проезда из города I в город J. Задача называется симметричной, если Сij=Cji для всех i и j, т. е. если стоимость проезда между каждыми двумя городами не зависит от направления. Предположим, что Сii = ∞ для всех i.

Алгоритмы ветвей и границ для задачи коммивояжера могут быть сформулированы разными способами. Авторы излагаемого алгорит­ма — Литл, Мерти, Суини и Карел. Это своего рода классика.

Во-первых, рассмотрим ветвление. На рис. 5.7а показана мат­рица стоимостей для асимметричной (несимметричной) задачи комми­вояжера с пятью городами, представленной на рис.5.7б. Обратите внимание на то, что мы пользуемся направленной сетью, чтобы ука­зать стоимости, так как стоимость проезда из города i прямо в город j не обязательно такая же, как стоимость проезда из города j в город i. Корень нашего поискового дерева будет соответствовать множеству «всех возможных вариантов», (в дальнейшем назовем каждый вариант решения задачи туром) т. е. эта вершина представляет множество всех 4! возможных туров в нашей задаче с пятью городами. В общем случае для любой асимметричной задачи с N городами корень будет представлять полное множество R всех (N—1)! возможных туров. Ветви, выходящие из корня, определяются выбором одного ребра, скажем (i, j).

Наша цель состоит в том, чтобы разделить множество всех туров на два множества: одно, которое, весьма вероятно, содер­жит оптимальный тур, и другое, которое, вероятно, не содержит. Для этого выбираем ребро (i,j); оно, как мы надеемся, входит в оп­тимальный тур, и разделяем R на два множества {i,j} и{i,j}. В мно­жество {i, j} входят туры из R, содержащие ребро (i, j), а в {i,j} — не содержащие (i,j).

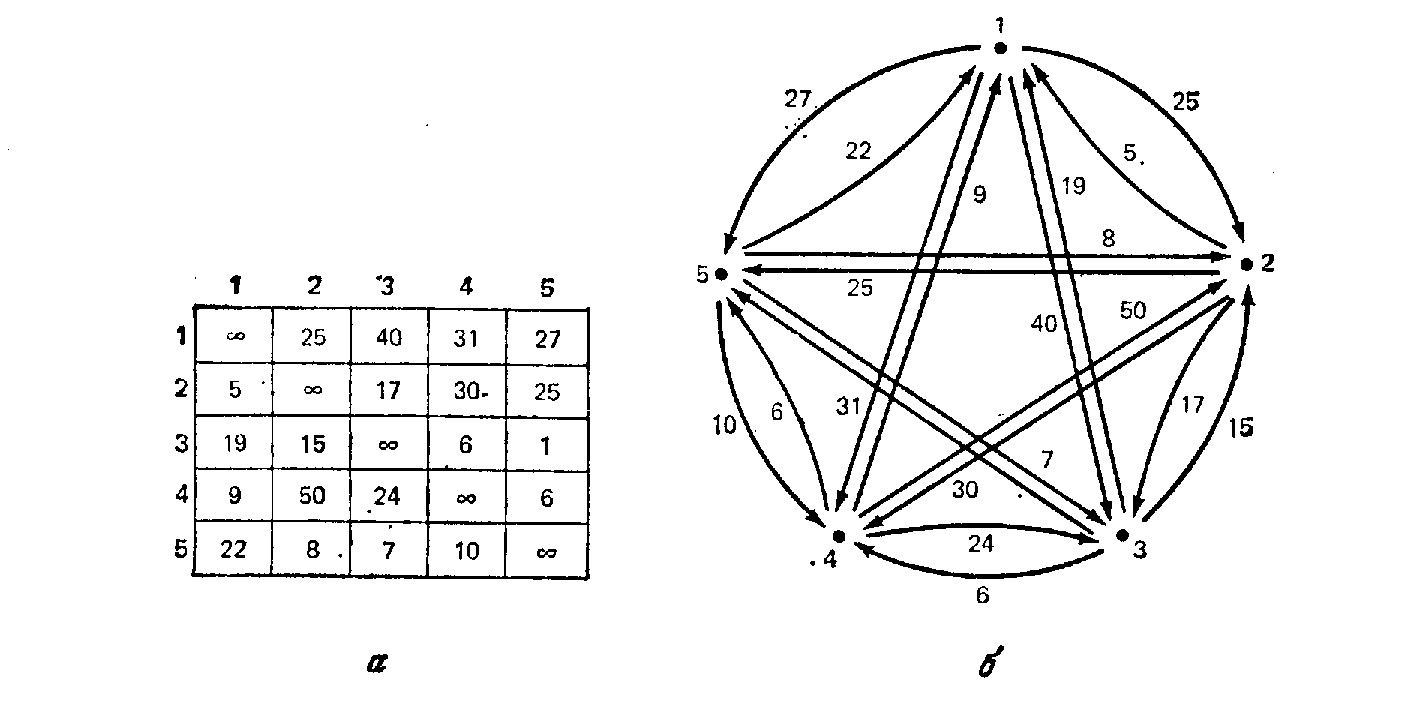


Рис. 5.7 Задача коммивояжера: (а) матрица стоимостей; (б) сеть из пяти городов.

Предположим, в нашем примере мы производим ветвление на ребре (i, j) = (3,5), имеющем наименьшую стоимость во всей матрице. Корень и первый уровень дерева пространства решений будут тогда такими, как показано на рис.5.8 Заметим, что каждый тур из R содержится только в одном множестве уровня 1. Если бы мы как-то могли сделать вывод, что множество {3,5} не содержит оптимального тура, то нам нужно было бы исследовать только множество {3, 5}.

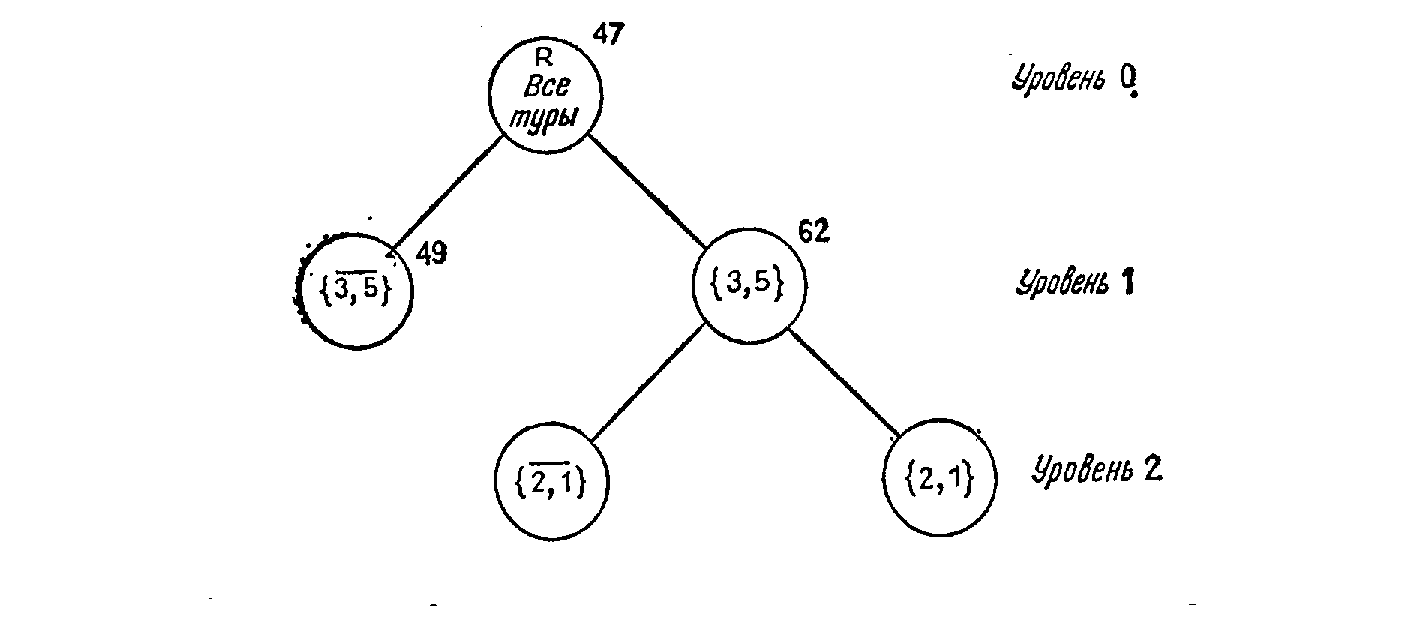


Рис. 5.8. Построение дерева поиска по методу ветвей и границ.

Затем разделяем множество {3, 5} таким же образом, как и множе­ство R. Следующее по дешевизне ребро в матрице — это ребро (2, 1) со стоимостью 5. Поэтому можно разделить множество {3, 5} на туры, включающие ребро (2, 1), и туры, не включающие этого ребра; это показано на уровне 2 на рис. 5.8. Путь от корня к любой вершине дерева выделяет определенные ребра, которые должны или не должны быть включены в множество, представленное вершиной дерева. Например, левая вершина уровня 2 на рис. 5.8 представляет множе­ство всех туров, содержащих ребро (3, 5) и не содержащих ребра (2, 1). Вообще, если Х — вершина дерева, a (i, j) — ребро ветвления, то обозначим вершины, непосредственно следующие за X, через Y и Y. Множество Y обозначает подмножество туров из Х с ребром (i,j), а множество Y — подмножество Х без (i, j).

Теперь, когда на примере мы пояснили, что такое ветвление, поясним, на этом же примере что подразумевается под вычислением границ.

Для каждой вершины дерева существует определенное множество решений, это множество укладывается в некоторые границы. Решения задачи мы связывает с весовой функцией. Для нашего случая эта функция представляет собой общую стоимость проезда. И поскольку для каждой вершины дерева может быть определенное множество решений, следовательно, для каждой вершины имеется определенное множество значений функций стоимости. Это множество значений имеет границы - верхнюю и нижнюю. Для нашей задачи о минимальной стоимости проезда, нас будет интересовать нижняя граница функции стоимостей.

Вычис­ление этих нижних границ — основной фактор, дающий экономию усилий в любом алгоритме типа ветвей и границ. Поэтому особое внимание следует уделить получению как можно более точных границ. Причина этого следующая. Предположим, что мы получили нижнюю границу стоимостей на одной вершине m, а на другой М и выполняется условие М>m, то исследованию должна быть подвергнута вершина стоимости m. и все следующие за ней.

Основной шаг при вычислении нижних границ известен как при­ведение. Оно основано на следующих двух соображениях:

1. В терминах матрицы стоимостей С каждый полный тур содержит только один элемент (ребро и соответствующую стоимость) из каждого столбца и каждой строки. Заметим, что обратное утверждение не всегда верно. Множество, содержащее один и только один элемент из каждой строки и из каждого столбца С, не обязательно представ­ляет тур. Например, в задаче, изображенной на рис. 5.7, множе­ство {(1, 5), (5, 1), (2, 3), (3, 4), (4, 2)} удовлетворяет этому условию, но не образует тура.

2. Если вычесть константу h из каждого элемента какой-то строки или столбца матрицы стоимостей С, то стоимость любого тура при новой матрице С' ровно на h меньше стоимости того же тура при матрице С. Поскольку любой тур должен содержать ребро из данной строки или данного столбца, стоимость всех туров уменьшается на h. Это вычитание называется приведением строки (или столбца).

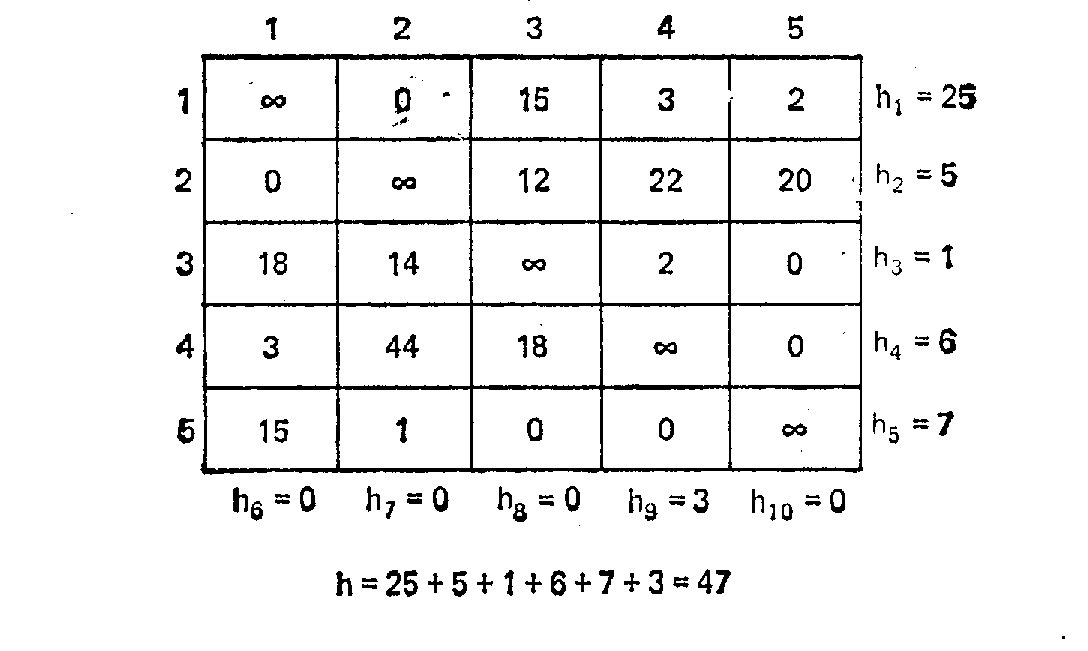


Рис. 5.9. Приведение матрицы стоимостей, показанной на рис.5.7 а.

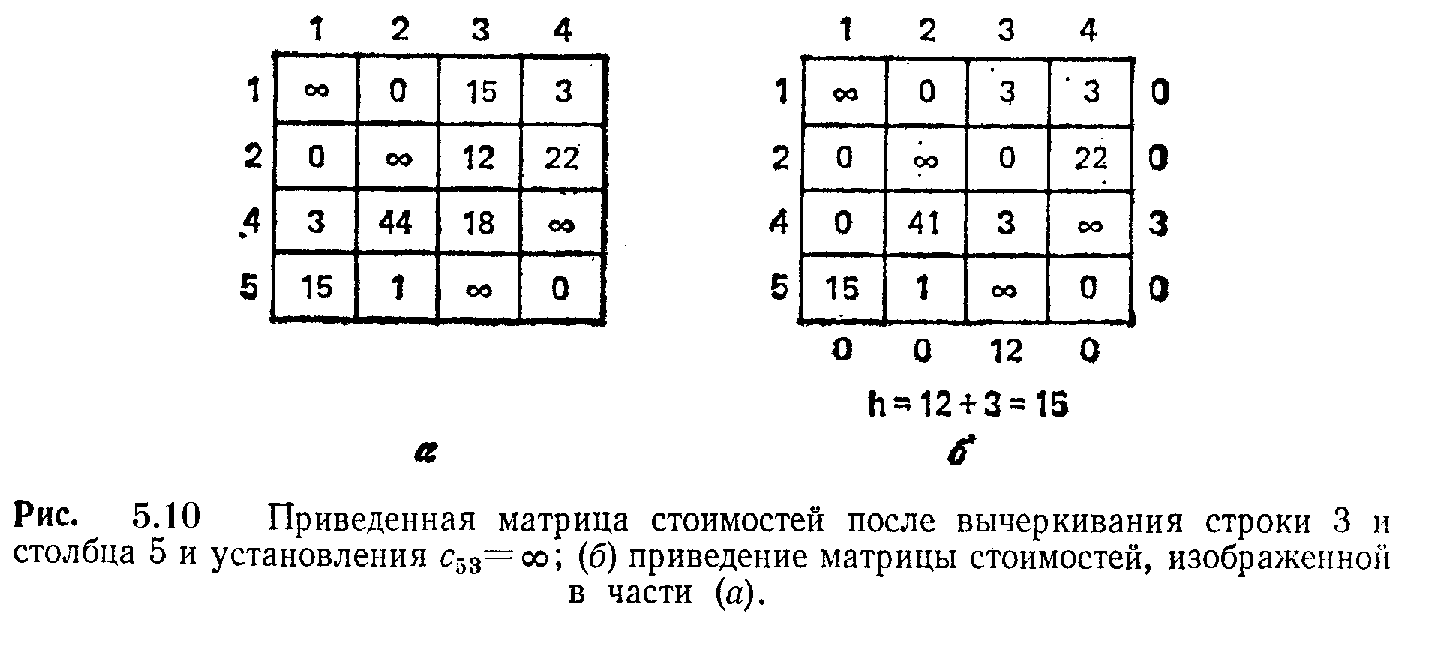
Под приведением всей матрицы стоимостей С понимается следую­щее. Последовательно проходим строки С и вычитаем значение наи­меньшего элемента Hi каждой строки из каждого элемента этой строки. Потом то же самое делаем для каждого столбца. Если для некоторого столбца или строки Hi=0, то рассматриваемый столбец или строка уже приведены, и тогда переходим к следующему столбцу или строке.

Сумма всех Hi есть нижняя граница стоимостей, обозначим ее как H

Полученную в результате матрицу стоимостей назовем приведенной из С. На рис.5.9 показано приведение матрицы стоимостей, изобра­женной на рис. 5.7а. Значения Hi, даны в конце каждой строки и столбца (строки и столбцы последовательно перенумерованы).

Общее приведение составляет H=47 единиц. Следовательно, ниж­няя граница стоимости любого тура из R также равна 47. Эта граница указана около корня дерева на рис. 5.8. Рассмотрим нижние границы для вершин уровня 1, т. е. для множеств {3,5} и {3,5}. Вычислим их нижние границы.

По определению ветвления путь (3,5) содержится в каждом туре множества {3,5}, следовательно путь (5,3) в нем не может содержаться, исходя из условий задачи (поездка осуществляется по одному пути только один раз) и, поэтому, значение (5,3) в матрице стоимостей равно бесконечности. Строку 3 и столбец 5 также можно исключить из дальнейшего рассмотрения потому что путь (3,5) уже существует. (строка 3 предполагает выезд из 3, а столбец 5 приезд в 5). Таким образом матрица приведенная на рис. 5.9. может быть нарисована в виде, изображенном на рис. 5.10. На рис.5.10а показана матрица после вычеркивания из нее строки 3 и столбца 5 и соблюдения условия, что стоимость (5,3) равна бесконечности, а на рис.5.10б показана уже приведенная матрица.



Нижняя граница для множества {3,5} равна сумме Н для предыдущего значения 47 и полученного 15, т.е. нижняя граница равна 62.

Нижняя граница множества {3,5} получается несколько иным способом. Путь (3,5) не может находится в этом множестве, поэтому полагаем его равным бесконечности (для матрицы расположенной на рис.5.9). Но в любой вариант решения задачи будет входить путь из 3 в какой либо другой город и путь входящий в город 5 из какого либо другого города. Выбираем минимальные значения стоимостей для этих путей. Самое меньшее из города 3 равно 2 (полагая (3,5) бесконечности), а самое меньшее входящее в 5 равно 0. Следовательно нижняя граница равна 47+2=49.

Значения 62 и 49 указаны на рис. 5.8. Естественно, что теперь необходимо рассматривать множество {3,5}.

В основных чертах блок-схема этого алгоритма ветвей и границ показана на рис.5.11.

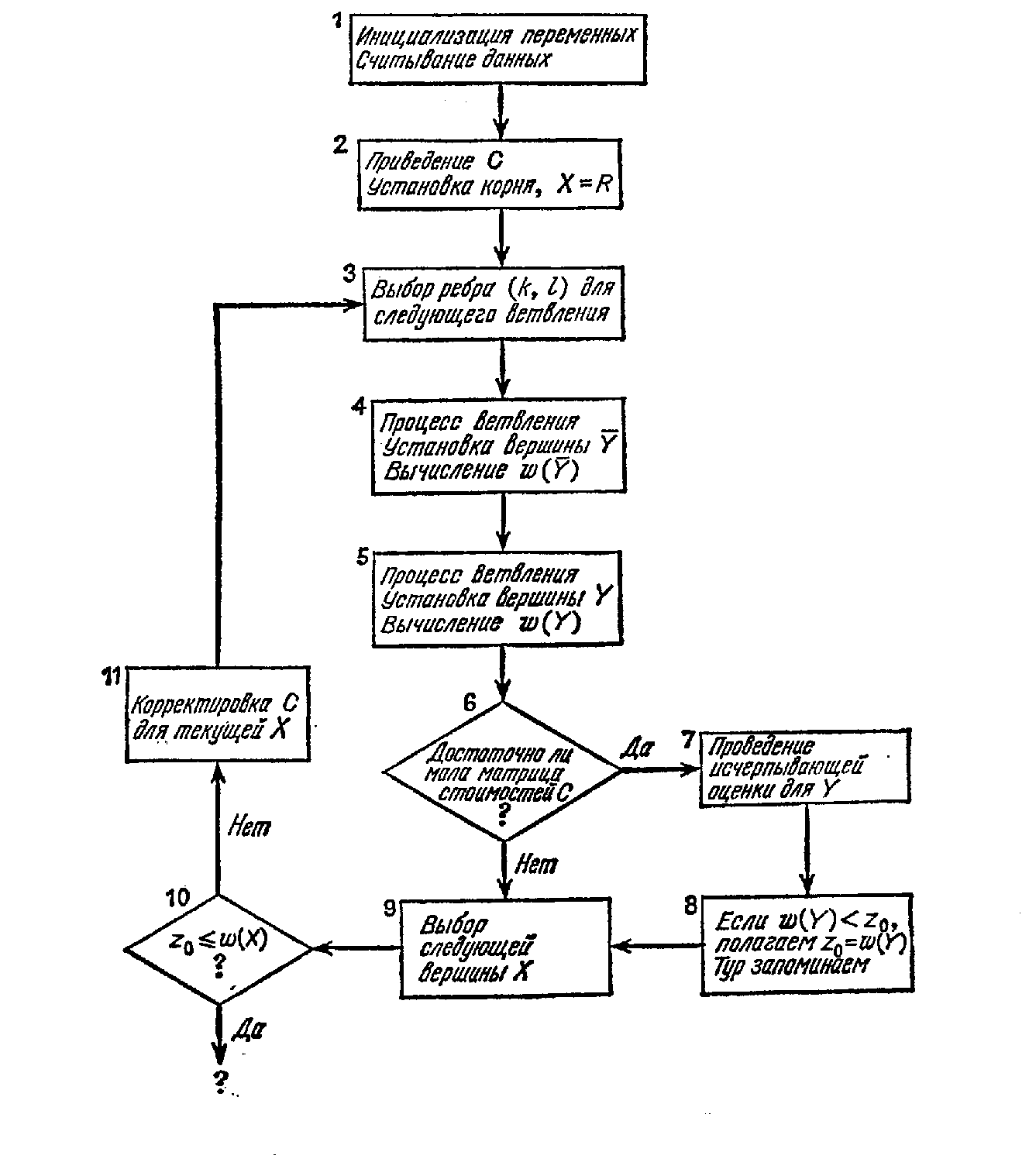


Рис. 5.11 Блок-схема алгоритма ветвей и границ.

Здесь используются следующие обозначения. Буква Х обо­значает текущую вершину на дереве поиска, a w (X) — соответствующую нижнюю границу. Вершины, следующие непосредственно за X, назовем Y и Y, они выбираются ветвлением по некоторому ребру (k, l). Символ Zo обозначает стоимость самого дешевого варианта, извест­ного на данный момент. В начальный момент Zo =oo.

Теперь раскроем более детально содержание некоторых блоков этой схемы.

Блок 1. Установление начальных значений переменных, или ини­циализация.

Блок 2. Первое приведение — это непосредственная реализация описанной ранее процедуры.

Блок 3. Выбор следующего ребра ветвления (k, 1) определяет множества Y и Y, непосредственно следующие за текущим X. Ребро (k, 1) нужно выбирать так, чтобы попытаться получить большую по величине нижнюю границу на множестве Y={k, 1), что облегчит проведение оценки для множества Y. Обычно предпочтительнее про­вести оценку для Y, так как размер этого множества и соответству­ющая ему матрица стоимостей обычно больше, чем у Y (из матрицы для Y вычеркнуты строка k и столбец l). Можно надеяться также, что Y с большей вероятностью содержит оптимальный вариант.

Как применить эти идеи к выбору конкретного ребра ветвления (k, l). В приведенной матрице стоимостей С, связанной с X, каждая строка и столбец имеют хотя бы по одному нулевому элементу (если нет, то С не полностью приведена). Можно предположить, что ребра, соответствующие этим нулевым стоимостям, будут с большей веро­ятностью входить в оптимальный вариант, чем ребра с большими стои­мостями. Поэтому мы выберем одно из них. Но какое? Пусть ребро (i,j) имеет Cij=0 в С. Мы хотим, чтобы у Y={i, j) была как можно большая нижняя граница. Вспоминая метод вычисления нижней границы для {3, 5} в нашем примере, мы видим, что для Y эта граница задается в виде

W(Y)=w(X)+ (наименьшая стоимость в строке i, не включая Cij)+(наименьшая стоимость в столбце j, не включая Сij).

Следовательно, из всех ребер (i, j) с Cij=0 в текущей матрице С мы выбираем то, которое дает наибольшее значение для W(Y).

Учитывая, что данный метод довольно трудный для понимания ниже приводится решение транспортной задачи с указанными выше данными.

**Решение задачи коммивояжера.**

1. Приведение матрицы стоимостей

1 2 3 4 5 1 2 3 4 5 **Н**

1 ∞ 25 40 31 27 1 ∞ 0 15 3 2 **25**

2 5 ∞ 17 30 25 2 0 ∞ 12 22 20 **5**

3 19 15 ∞ 6 1 3 18 14 ∞ 2 0 **1**

4 9 50 24 ∞ 6 4 3 44 18 ∞ 0 **6**

5 22 8 7 10 ∞ 5 15 1 0 0 ∞ **7**

**Н 0 0 0 3 0**

**Нс=47**

2. Выбор ребра ветвления

Пути соответствующие значениям 0 следующие (1,2), (2,1), (3,5), (4,5), (5,3), (5,4)

Значения на которое предстоит увеличить суммарную Нс = 47 соответственно равны: 2+1=3; 12+3=15; 2+0=2; 3+0=3; 0+12=12; 0+2=2.

Привило по которому производится это вычисление, следующее:

(наименьшая стоимость в строке i, не включая Cij)+(наименьшая стоимость в столбце j, не включая Сij),

где Сij соответствует 0 значениям приведенной матрицы.

Выбирает ребро (2,1), как максимальный элемент из списка.

3. Исключение из матрицы стоимостей строки 2 и столбца 1 и ее приведение.

2 3 4 5 2 3 4 5 **Н**

1 ∞ 15 3 2 1 ∞ 13 1 0 **2**

3 14 ∞ 2 0 3 13 ∞ 2 0 **0**

4 44 18 ∞ 0 4 43 18 ∞ 0 **0**

5 1 0 0 ∞ 5 0 0 0 ∞ **0**

**Н 1 0 0 0**

**Нс=3**

4. Определение нижних границ стоимостей вершин

Для множества включающего ребро (2,1) 47+3=50

Для множества не включающего ребро (2,1) 47+15=62

Рассмотрение проводим для варианта имеющего нижнюю границу 50

5. Определяем ребро ветвления для вершины с нижней границей 50

Пути соответствующие значениям 0 следующие (1,5), (3,5), (4,5), (5,2), (5,3), (5,4)

Значения на которое предстоит увеличить суммарную Нс = 47 соответственно равны: 1+0=1; 2+0=2; 18+0=18; 13+0=13; 13+0=13; 0+1=1.

Выбирает ребро (4,5), как максимальный элемент из списка.

6. Исключение из матрицы стоимостей строки 4 и столбца 5 и ее приведение.

2 3 4 2 3 4 **Н**

1 ∞ 13 1 1 ∞ 12 0 **1**

3 13 ∞ 2 3 11 ∞ 0 **2**

5 0 0 0 5 0 0 0 **0**

**Н 1 0 0**

**Нс=3**

7. Определение нижних границ стоимостей вершин

Для множества включающего ребро (4,5) 50+3=53

Для множества не включающего ребро (4,5) 50+18=68

Рассмотрение проводим для варианта имеющего нижнюю границу 53

8. Определяем ребро ветвления для вершины с нижней границей 53

Пути соответствующие значениям 0 следующие (1,4), (3,4), (5,2), (5,3).

Значения на которое предстоит увеличить суммарную Нс = 47 соответственно равны: 12+0=12; 11+0=11; 11+0=11; 12+0=12.

Выбирает ребро (1,4).

9. Исключение из матрицы стоимостей строки 1 и столбца 4 и ее приведение.

2 3 2 3 **Н**

3 11 ∞ 3 0 ∞ **11**

5 ∞ 0 5 ∞ 0 **0**

**Н 1 0**

**Нс=11**

Примечание: путь (5,2) мы положили равным ∞, так как ребро (1,4), вместе с ребрами (2,1), и (4,5) образует путь и следовательно пути где участвуют точки 5 и 2 уже присутствуют в рассмотрении, следовательно путь (5,2) не осуществим по условию задачи.

7. Определение нижних границ стоимостей вершин

Для множества включающего ребро (1,4) 53+11=64

Для множества не включающего ребро (1,4) 53+12=65

8. Таким образом, у нас получился один из оптимальных вариантов со стоимостью 64. Ему соответствует общий путь (1,4) - (4,5) - (5,3) - (3,2) - (2,1). Ребра (2,1), (4,5) и (1,4) выбраны как ребра ветвления, а ребра (3,2) и (5,3) - остаются в матрице (п.9).

Естественно заметить, что одна из рассмотренных ранее нижних границ для вершины исключающий вариант с ребром (2,1) равна 62, поэтому теперь проводим решение для этой вершины и вычисляем нижние границы для нее. Решение провидится аналогично.

1. Исследование вершины исключающей путь (2,1)

9.1 Матрица для нее записывается также как в п.1 за исключением того, что значение (2,1) приравнивается бесконечности, так как путь (2,1) для этих вариантов не возможен.

1 2 3 4 5 1 2 3 4 5 **Н**

1 ∞ 25 40 31 27 1 ∞ 0 15 3 2 **25**

2 ∞ ∞ 17 30 25 2 ∞ ∞ 0 10 8 **17**

3 19 15 ∞ 6 1 3 15 14 ∞ 2 0 **1**

4 9 50 24 ∞ 6 4 0 44 18 ∞ 0 **6**

5 22 8 7 10 ∞ 5 12 1 0 0 ∞ **7**

**Н 3 0 0 3 0**

**Нс=62**

9.2 Определяется ребро ветвления, это будет (4,1).

9.3. Для него определятся нижние границы. Это будет 74 исключающее ребро (4,1), и 62 включающее ребро (4,1).

9.4. Необходима еще одна итерация. Оставим это в качестве упражнения.

Весь проделанный ход рассуждения иллюстрирует схема представленная на рис. 5.12.

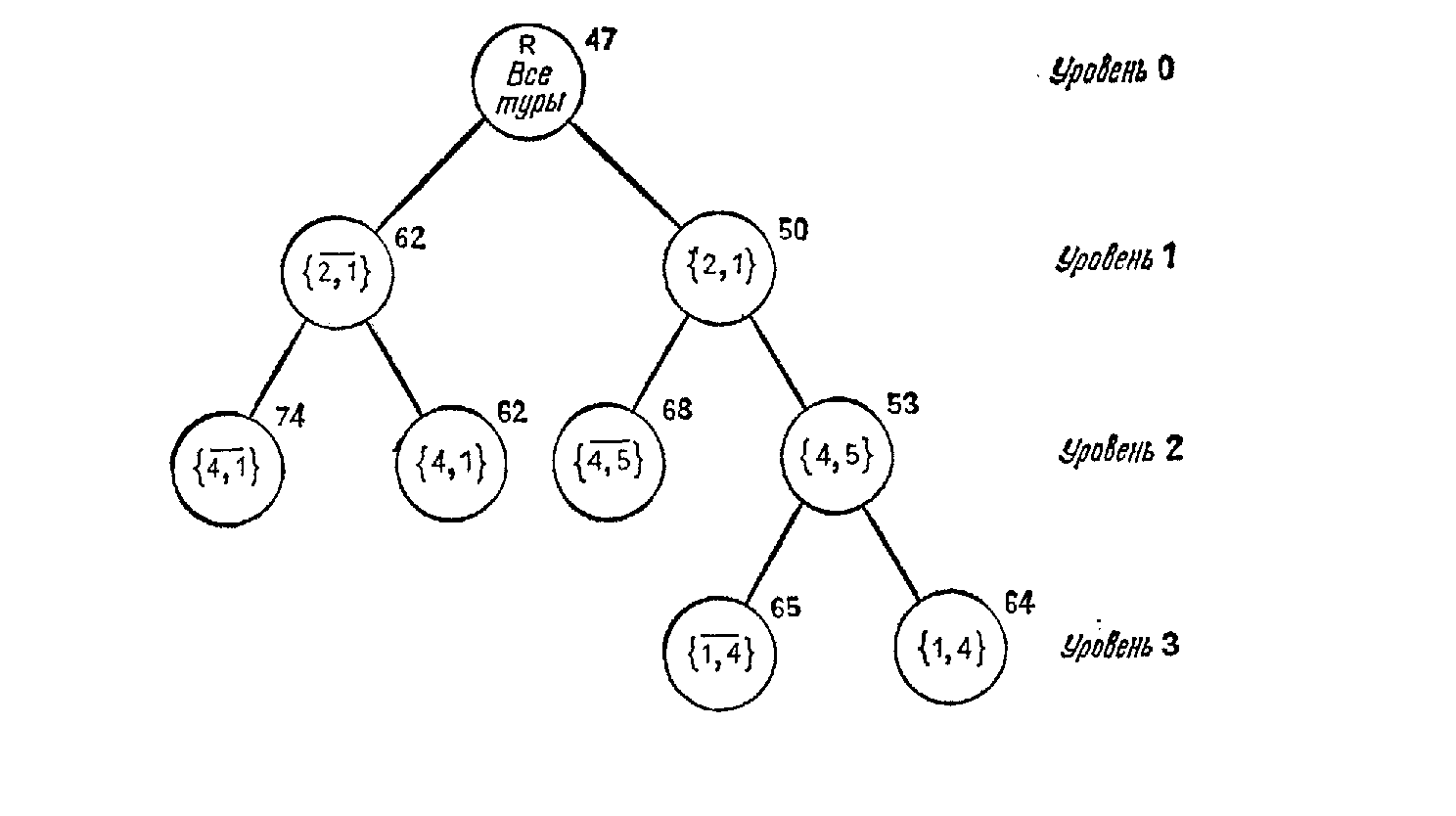


Рис. 5.12. Дерево поиска

Таким образом мы полностью рассмотрели алгоритм решения задачи - названный как метод ветвей и границ. И хотя, этот метод довольно сложен, эффективность его решение в ряде случае бесспорна. В отношении его сложности можно сказать, что существуют различные эвристические приемы, которые ускоряют этот сложный алгоритм и в ряде случаев еще и упрощают его.

**Рекурсия**

Рекурсивным называется объект, частично состоя­щий или определяемый с помощью самого себя. Ре­курсия встречается не только в математике, встре­чается она и в повседневной жизни. Кто не видел рекламной картинки, содержащей свое собственное изображение?

В частности, рекурсивные определения представ­ляют собой мощный аппарат в математике.

Примером рекурсии может является вычисление факториала Функция «факториал» n! (для неотрицательных целых чисел) рекурсивно определяется как

0! = 1

n! = n\*(n—1)! Еслси n>0

Уравнение вида n! = n\*(n-1)! называется рекуррентным соотношением. Рекуррентные соотношения выражают значения функции при помощи других значений, вычисленных для меньших аргументов.

Уравнение О!=1 — нерекурсивно определенное начальное значение функции. Для каждой рекурсивной функции нужно хотя бы одно такое начальное значение, в противном случае ее нельзя вычислить в явном виде,

Аналогично числа Фибоначчи определяются следующей беско­нечной последовательностью целых чисел: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . .. Проверка показывает, что N-н элемент этой последова­тельности равен сумме двух непосредственно предшествующих эле­ментов. Таким образом, значение FIB(N) для N-того числа может быть определено из рекуррентного соотношения:

FIB(N)=FIB(N—1)+FIB(N—2).

Так как FIB(N) определено через два разных значения для мень­ших аргументов, необходимы два начальных значения. Ими служат

FIB(1)=1, FIB(2)=1.

Определенные здесь числа Фибоначчи являются рекурсивным решением следующей задачи:

Каждый месяц самка из пары кроликов приносит двух кроликов (самца и самку). Через два месяца новая самка сама приносит пару кроликов. Нужно найти число кроликов в конце года, если в начале года была одна новорожденная пара кроликов и в течение этого года кролики не умирали,

Эта последовательность названа в честь итальянского математика Фибоначчи, который в 1202 г, сформулировал изложенную здесь задачу.

Функция Аккермана

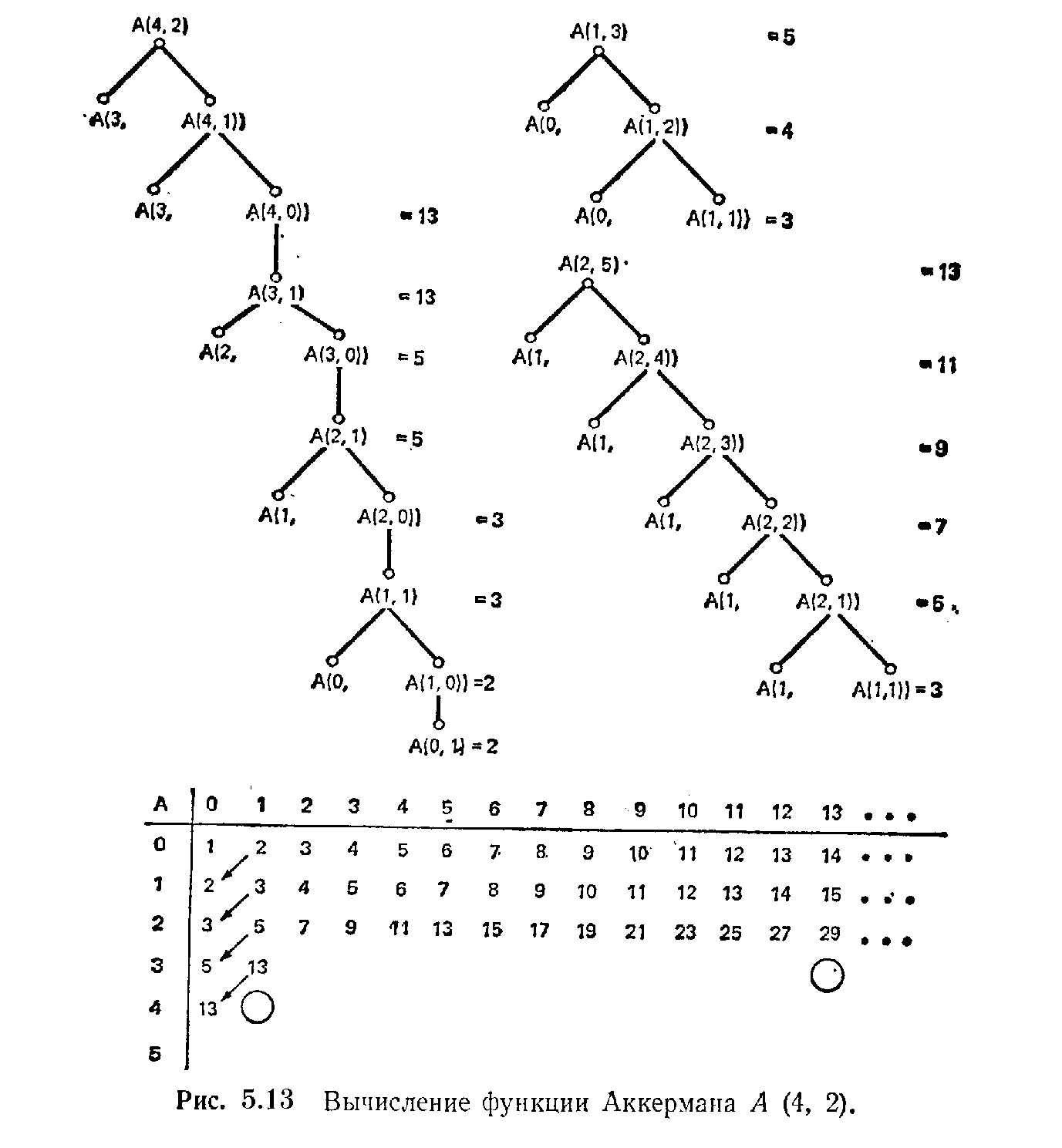
Две рекурсивные функции, рассмотренные до сих пор, были до­статочно простыми. Поэтому, чтобы не сложилось неправильное впе­чатление о том, насколько сложны рекурсивные функции, представим следующую, довольно простую на первый взгляд, дважды рекурсив­ную функцию, известную как функция Аккермана. Функция дважды рекурсивна, если сама функция и один из ее аргументов определены через самих себя.

N + 1, если М = 0;

А (М, N) = А (М—1, 1), если N = 0;

A [М—1, А (М, N—1)] в остальных случаях.

Беглый просмотр рис. 5.13 показывает, как трудно вычислить эту функцию даже для таких малых аргументов, как М = 4 и N=2. Заметьте, например, что А (4, 1)=A(3, 13).



**Тема 5. Алгоритмы машинной математики**

**Сортировка массивов**

Методы сортировки массивов можно условно разбить на две группы:

- сортировка когда элементы массива а передаются в результирующий массив b;

- сортировка когда элементы массива а перегруппируются в самом массиве а, который и будет результирующий.

Выбор методов сортировки зависит от условий задачи, и хотя в ряде случаев предпочтительнее выглядят методы второй группы, так как они экономят память, нередко в условиях сортировки требуется помимо отсортированного списка сохранять неотсортированный.

Рассмотрим эти методы.

**Сортировка методом прямого включения**

Сортировка методом прямого включения работает со списком не­упорядоченных чисел (обычно называемых клю­чами), сортируя их в порядке возрастания или убывания. Это делается примерно так же, как большинство игроков упорядочивают сданные им карты, под­нимая каждый раз по одной карте. Покажем работу общей процедуры на примере следующего неотсортированного списка из восьми целых чисел:

27 412 71 81 59 14 273 87,

который надо отсортировать по возрастанию.

Отсортированный список создается заново; вначале он пуст. На каж­дой итерации первое число неотсортированного списка удаляется из него и помещается на соответствующее ему место в отсортированном списке. Для этого отсортированный список просматривается, начиная с наименьшего числа, до тех пор, пока не находят соответствующее место для нового числа, т. е. пока все отсортированные числа с меньшими значениями не окажутся впереди него, а все числа с большими значениями — после него. Другими словами новое число неотсортированного списка включается (устанавливается в соответствующее место) в отсортированный список. Следующая последовательность списков показывает, как это делается:

Итерация 0

Неотсортированный 412 71 81 59 14 273 87

Отсортированный 27

Итерация 1

Неотсортированный 412 71 81 59 14 273 87

Отсортированный 27 412 Включение 412 в отсортированный

список на соответствующее место

Итерация 2

Неотсортированный 71 81 59 14 273 87

Отсортированный 27 71 412 Включение 71

Итерация 3

Неотсортированный 81 59 14 273 87

Осортированный 27 71 81 412 Включение 81

Итерация 4

Неотсортированный 59 14 273 87

Отсортированный 27 59 71 81 412 Включение 59

Итерация 5

Неотсортированный 14 273 87

Отсортированный 14 27 59 71 81 412 Включение 14

Итерация 6

Неотсортированный 273 87

Отсортированный 14 27 59 71 81 273 412 Включение 273

Итерация 7

Неотсортированный 87

Отсортированный 14 27 59 71 81 87 273 412 Включение 87

В данном алгоритме присутствуют два списка последовательностей, что естественно отрицательно сказывается на использование памяти. Другими словами для сортировки списка из N элементов необходимо задействовать 2N ячеек памяти. Поэтому, целесообразнее пользоваться методом прямого включения, который работает с одним списком последовательности. Рассмотрим его также на примере.

Допустим надо отсортировать последовательность из 10 чисел:

10, 20, 5, 45, 12, 2, 46, 48, 23, 32

Укрупненно алгоритм может быть представлен следующим образом: вначале некоторому числу R присваивается второй элемент последовательности, который сравнивается с первым и располагается с ним в порядке возрастания, затем выбирается третий элемент присваивается числу R и сортируется с первыми двумя в порядке возрастания. Эта операция продолжается до тех пор, пока не дойдет очередь до последнего элемента.

**Algorithm   SIS**

**Algorithm   SIS** ( Сортировка Прямым включением). Отсортировать на старом  месте последовательность целых чисел I(1), I(2), . . . ,I (N)  в порядке возрастания.

*Шаг 1.* [ Основная  итерация ]

**For J←** 2 **to** N **do through**шаг 4 **od** ; **and** STOP.

*Шаг 2.*[ Выбор  следующего целого ]  **Set**  K← I(J);  **and**

L←J−1.

*Шаг 3.* [ Сравнение с отсортированного  целыми ] **While**

K<I(L)

                           AND L≥1 **do set** I (L+1) **←**I(L); **and** L←L−1 **od.**

*Шаг 4.*  [ Включение ] **Set** I(L+1)←K.

Блок схема этого алгоритма представлена на рис.6.1.

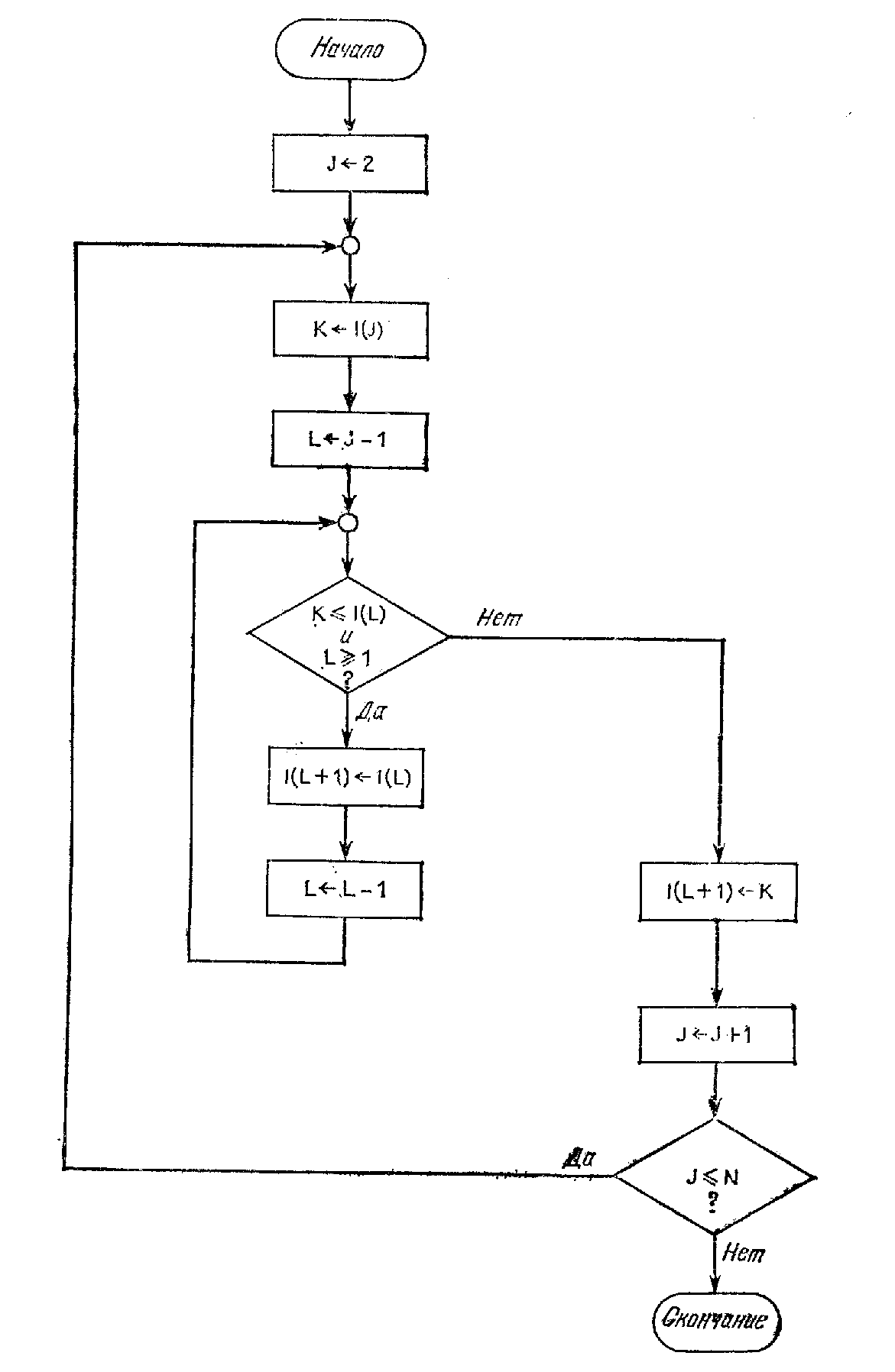


Рис.6.1. Блок схема алгоритма сортировки прямым включением.

Текст программы на языке Турбо-Бейсик может быть записан следующим образом:

CLS

DIM I(10)

DATA 10, 20, 5, 45, 12, 2, 46, 48, 23, 32

READ I(1), I(2), I(3), I(4), I(5), I(6), I(7), I(8), I(9), I(10)

FOR n = 1 TO 10: PRINT I(n); : NEXT n: PRINT

J = 1

5 R = I(J)

L = J - 1

9 IF R <= I(L) AND L >= 1 THEN

I(L + 1) = I(L): L = L - 1: GOTO 9

ELSE

I(L + 1) = R

J = J + 1

IF J <= 10 THEN

GOTO 5

ELSE

FOR n = 1 TO 10: PRINT I(n); : NEXT n

END IF

END IF

В представленном алгоритме заводится только один список, и переорга­низация чисел производится в старом списке.

**Метод быстрой сортировки.**

Основная причина медленной работы алгоритма сортировки простыми включениями заключается в том, что все сравнения и обмены между ключами в последователь­ности элементов происходят для пар из соседних элементов. При таком способе требуется довольно большое время, чтобы поставить ключ, находящийся не на месте, в нужную позицию в сортируемой последовательности.

К.Хоор изобрел и весьма эффективно применил идею сравнения пары элементов, находящихся далеко друг от друга в последовательности. Это значительно сократило время сортировки. Для того чтобы понять этот алгоритм целесообразно рассмотреть следующий пример.

Предположим, что мы хотим отсортировать последовательность чисел из первой строки на рис.6.2. Начнем с предположения, что первый ключ в этой последовательности (38) служит хорошей аппрок­симацией ключа, который в конечном счете появится в середине от­сортированной последовательности. Используем это значение в ка­честве ведущего элемента, относительно которого ключи могут ме­няться местами, и продолжим следующим образом. Устанавливаем два указателя I и J, из которых I начинает отсчет слева (I=1), а J — справа в последовательности (J=N). Сравниваем аi и аj. Если ai<=aj, то устанавливаем J=-J—1 и проводим следующее сравнение. Продолжаем уменьшать J до тех пор, пока не достигнем ai>=aj. Тогда поменяем местами аi и аj (рис. 6.2, строка 5, обмен ключей 38 и 04), устанавливаем I=I+1 и продолжаем увеличивать I до тех пор, пока не получим ai>aj. После следующего обмена (строка 10, 79,38 поменялись местами) снова уменьшаем J. Чередуя уменьшение J и увеличение I, продолжаем этот процесс с обоих концов последовательности к «сере­дине» до тех пор, пока не получим I=J.

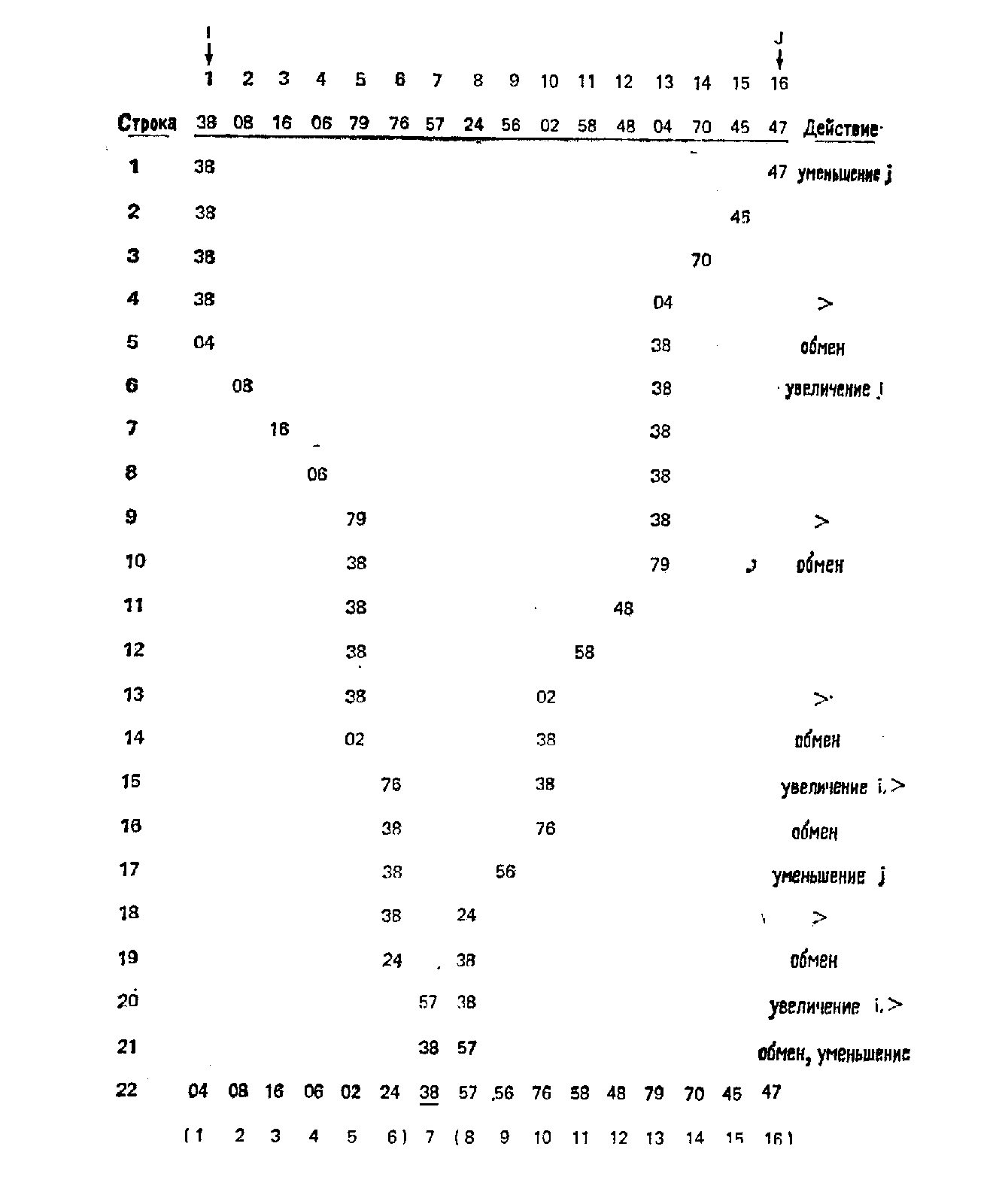


рис.6.2 Начальные шаги алгоритма метода быстрой сортировки.

Теперь имеют место два факта. Во-первых, ключ (38), который сначала находился в первой позиции, к этому времени занимает над­лежащее место в сортируемой последовательности. Во-вторых, все ключи слева от этого элемента будут меньшими, а все ключи справа — большими.

Ту же процедуру можно применить к левой и правой подпоследо­вательностям для окончательной сортировки всей последователь­ности. Последняя строка (с номером 22) рис.6.2 показывает, что когда будет получено I=J, то I=7. После этого процедура снова применяется к подпоследовательностям (1,6) и (8,16).

Рекурсивный характер алгоритма наводит на мысль, что следует значения индексов крайних элементов большей из двух неотсортиро­ванных подпоследовательностей (8,16) поместить на стек и затем пе­рейти к сортировке меньшей подпоследовательности (1,6).

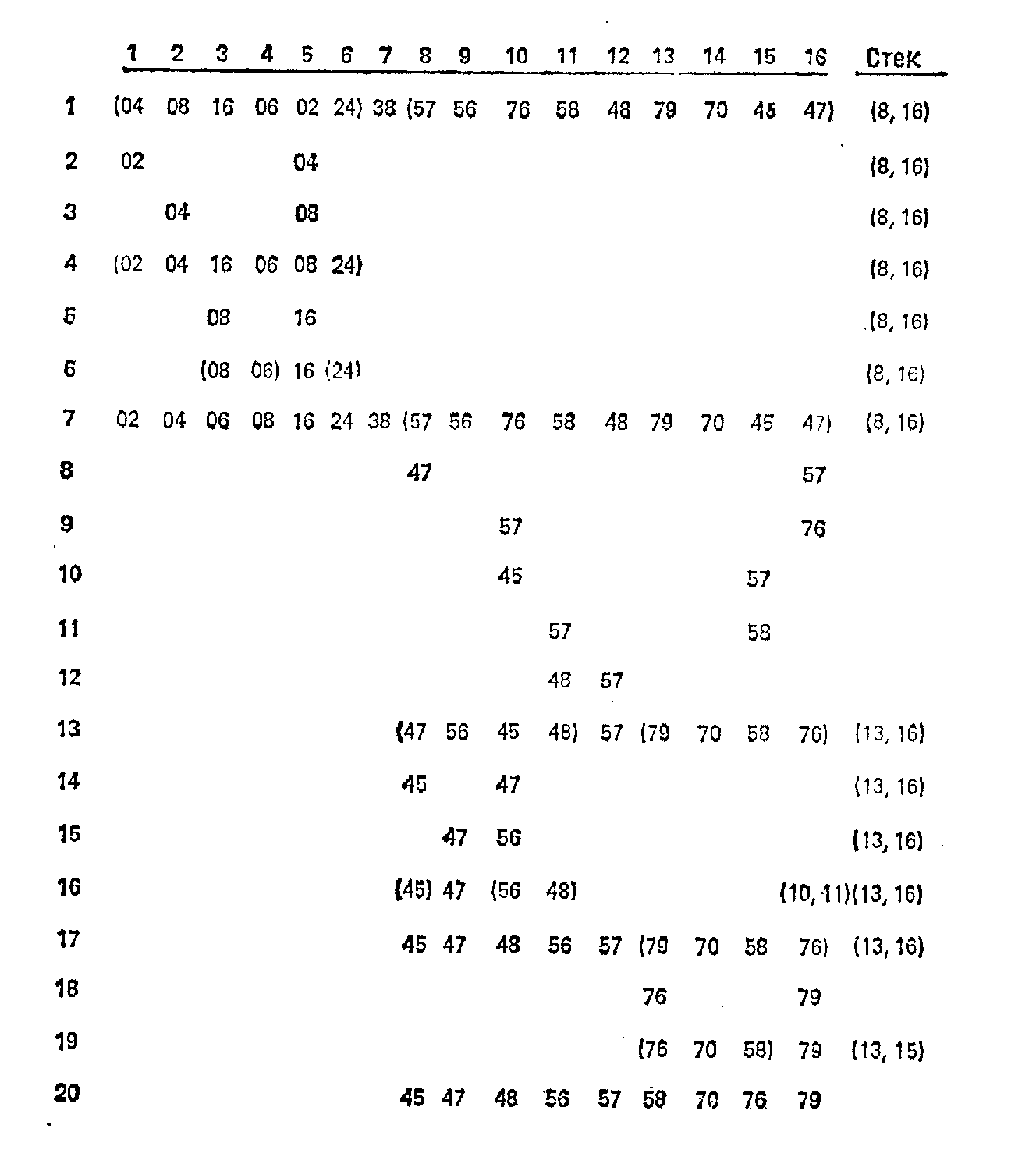


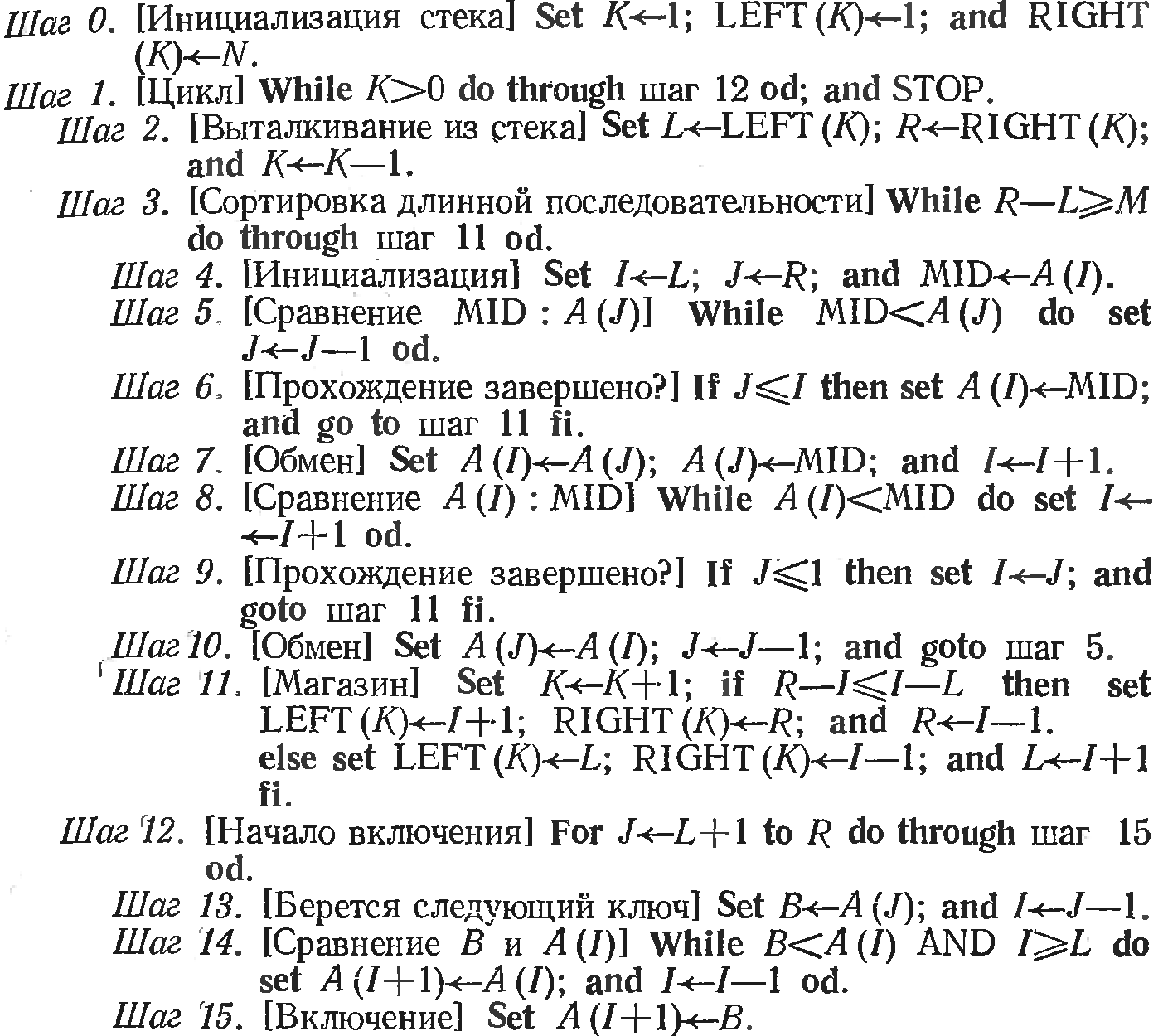
рис. 6.3. Завершение работы алгоритма быстрой сортировки.

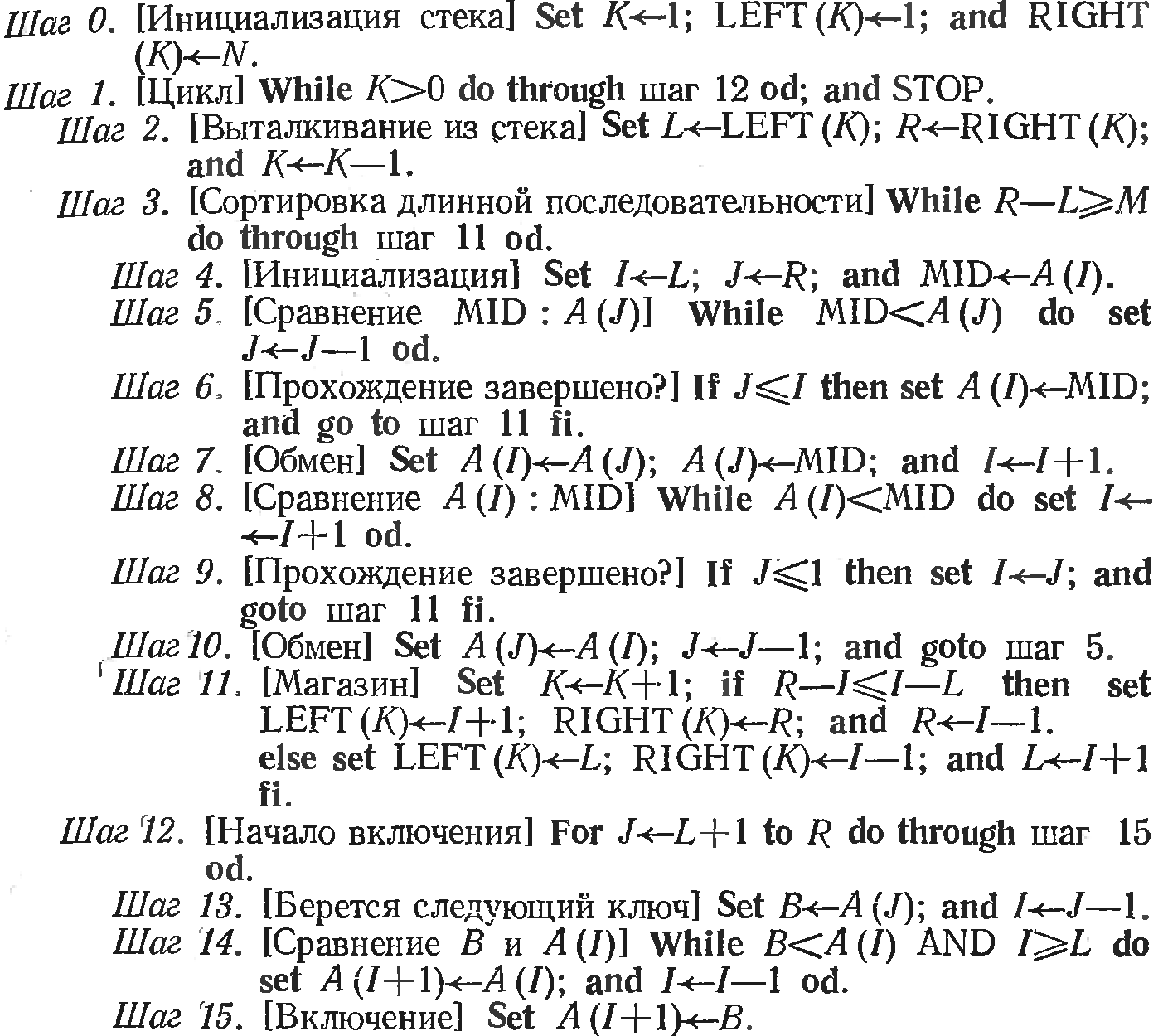
В строке 4 на рис. 6.3 число 04 перешло в позицию 2 и сорти­ровке подлежат подпоследовательности (1,1) и (3,6). Так как (1,1) уже отсортирована (число 02), сортируем (3,6), что в свою очередь ведет к строке 6, в которой подлежат сортировке (3,4) и (6,6). В строке 7 подпоследовательность (1,6) отсортирована. Теперь извле­каем (8,16) из стека и начинаем сортировку этой подпоследователь­ности. В строке 13 находятся подпоследовательности (8, II) и (13, 16), которые надо отсортировать. Помещаем (13, 16) на стек, сортируем (8,11) и т. д. В строке 20 последовательность целиком отсортирована.

Математически доказано, что последовательность с большим числом элементов таким алгоритмом сортируется быстрее, чем методом сортировки простым включением, но если число элементов мало, то быстродействие рассмотренного метода значительно падает. В данном разделе мы не будем исследовать почему это происходит, а только укажем это. Строгие математические расчеты показывают, что для последовательностей с числом элементов больше 9 метод быстрой сортировки оказывается более быстродействующим. Если же число элементов меньше 9, то более быстродействующим оказывается метод простым включением.

Следовательно, большие массивы данных целесообразно обрабатывать методом быстрой сортировки, а для массивов данных с числом менее 9 с помощью метода прямого включения. На практике, для больших последовательностей, выгоднее применять оба эти метода для одной последовательности данных. Смысл такого применения заключается в том, что вначале последовательность обрабатывается методом быстрой сортировки, а потом когда размер подпоследовательностей станет меньше 9 данных, обработка продолжается методом прямого включения.

**Алгоритм быстрой сортировки**





**Алгоритм, использующий специальную структуру дерева, называемую пирамидой**

**HEAPSORT: Алгоритм сортировки, имеющий в худшем случае порядок *0 (N* log *N)***

Существует не много алгоритмов сортировки, которые в худшем случае могут гарантировать не более *О* *(N* log *N)* сравнений. Одним из них является предложенный Вильямсом остроумный алгоритм, использующий специальную структуру дерева, называемую *пирамидой.* Пирамида состоит из помеченного корневого двоичного дерева заданной высоты *h,* обладающего следующими тремя свойствами:

1. Каждая конечная вершина имеет высоту *h* или *h*—1.

2. Каждая конечная вершина высоты *h* находится слева от любой конечной вершины высоты *h*—

3. Метка любой вершины больше метки любой следующей за ней вершины.

Из свойства 3 непосредственно вытекает, что метка корня пира­миды является наибольшей меткой дерева. На рис. 7.1 изображены три помеченных корневых двоичных дерева, не являющихся пирамидами, и одно (Т4) — пирамида. Заметим, что для дерева Ti не выполняется свойство i при i=1, 2, 3.

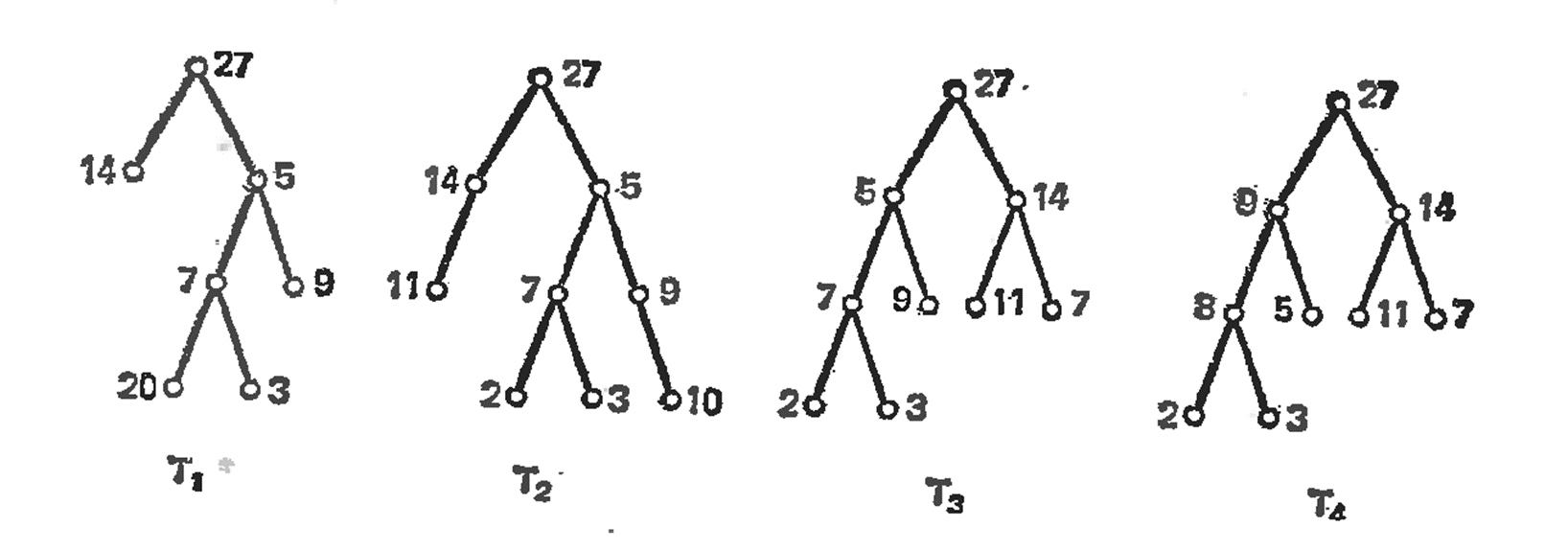
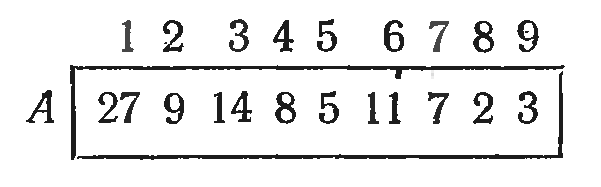


Рис. 7.1 Только дерево Т4 является пирамидой

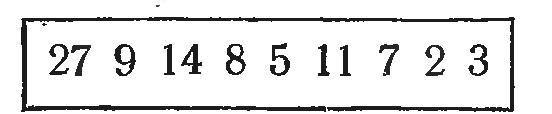
Специальная структура пирамид позволяет компактно размещать их в памяти. В частности, мы можем разместить пирамиду с *п* вер­шинами в массиве *А,* в котором две непосредственно следующие за вершиной из *A(i)* вершины помещаются в *А* (2i) и *А(*2i+1). Напри­мер, пирамиду Г4, изображенную на рис. 7.1 можно разместить в массиве *А* из девяти элементов следующим образом:



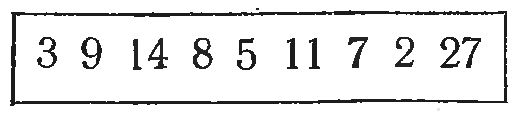
Заметим, например, что элементы 11 и 7 в А(6) и А(7) — это те два элемента, которые следуют за элементом 14 в *А*(3). Если 2*i>n,* тогда за вершиной *А*(i) не следуют другие вершины, и она является конечной вершиной пирамиды. Этот порядок расположения элемен­тов пирамиды такой же, как и порядок при методе поиска в ширину по дереву.

Используя такое представление массива, легко отсортировать вершины пирамиды. Проиллюстрируем это на пирамиде *Т4.*

1. Переставляем *А* (1) с *А (п).* Таким образом,



преобразуется в



что эквивалентно замене Т4 деревом Т5 на рис. 7.5.

2. Устанавливаем *п←п*-1. Это эквивалентно вычеркиванию вершины 27 из *Т5* на рис. 7.2.

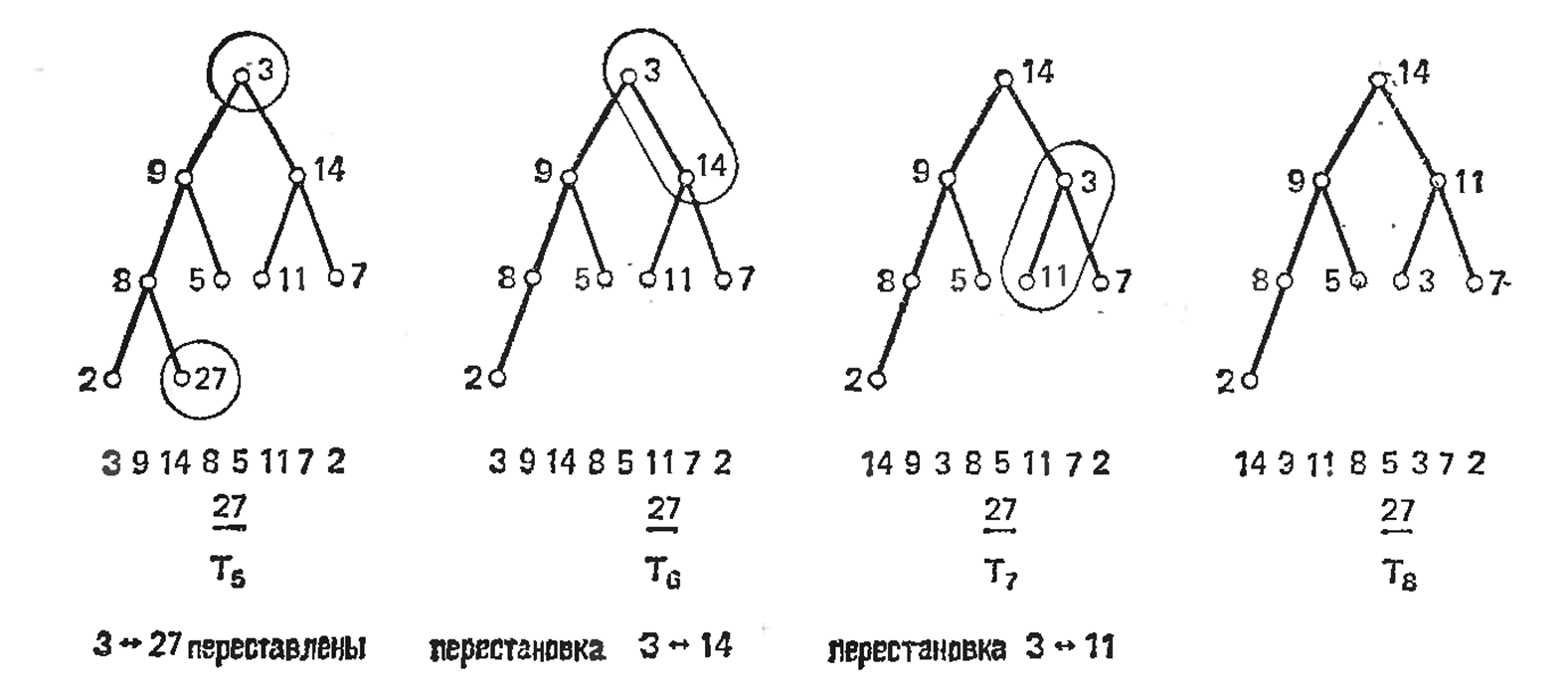


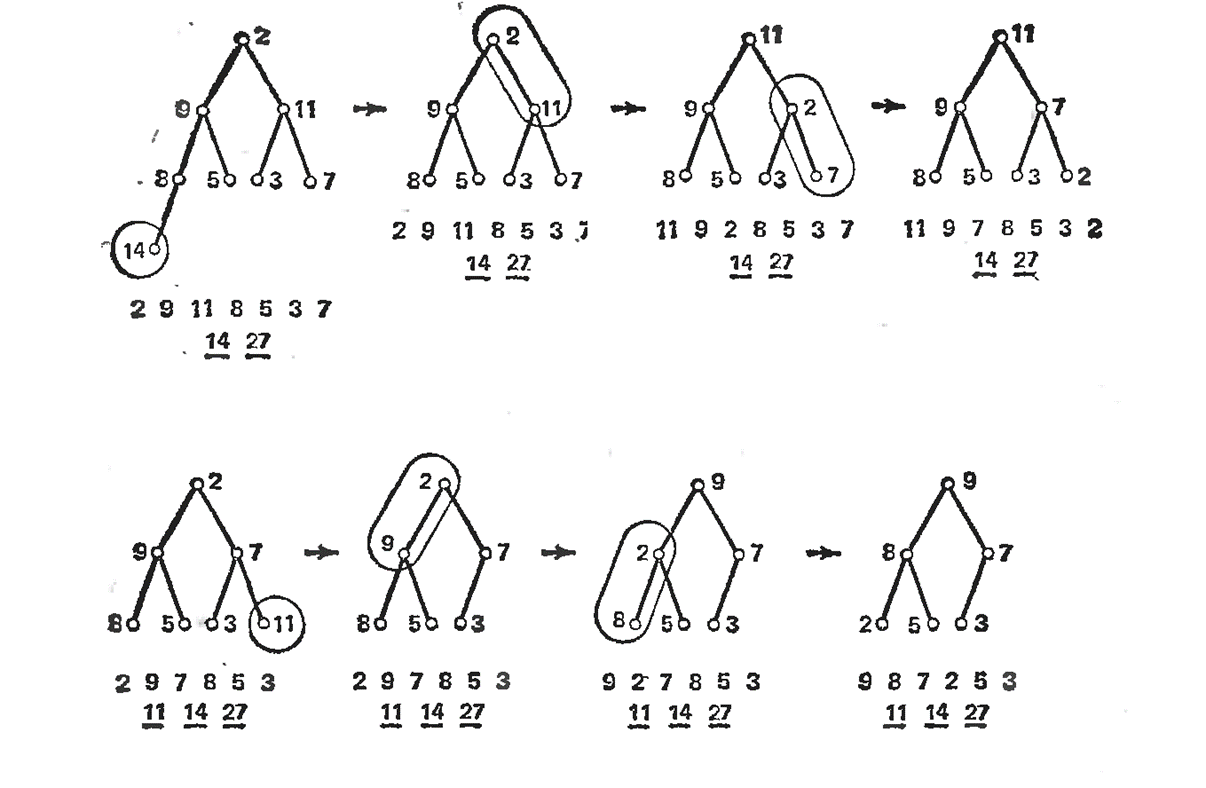
Рис. 7.2 Первые шаги при сортировки вершин пирамиды Т4, представленной на Рис. 7.1.

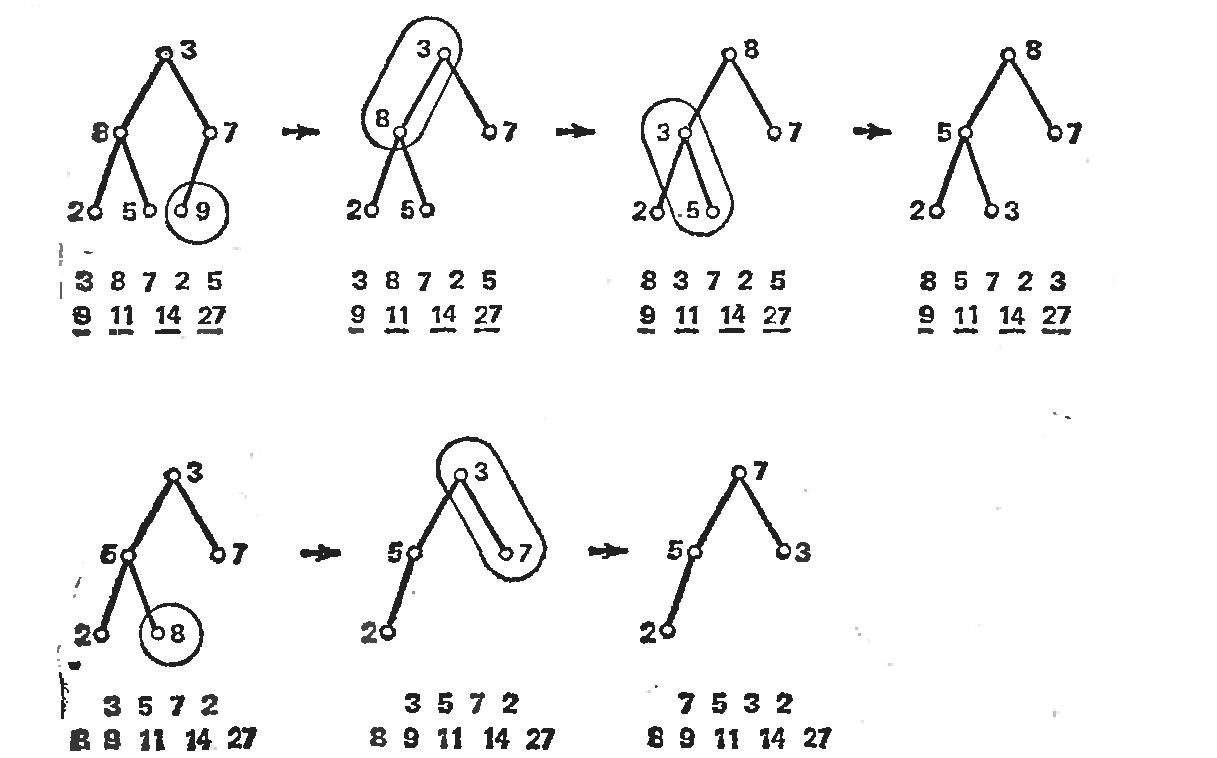
3. Преобразуем модифицированное дерево в другую пирамиду. Это достигается повторением перестановки нового корня с большей из двух новых непосредственно следующих за ним вершин до тех пор, пока он не станет больше, чем обе вершины, непосредственно за ним следующие. В Т7 переставлены вер­шины 3 и 14, а в *Т8* в соответствии с этой процедурой перестав­лены вершины 3 и 11. Поэтому Ts —преобразованная пирамида.

4. Повторяем шаги 1, 2 и 3 до тех пор, пока не получим *п=* 1.

Рис. 7.3 иллюстрирует завершение этого процесса сортировки

для пирамиды Т4 на рис. 7.1.





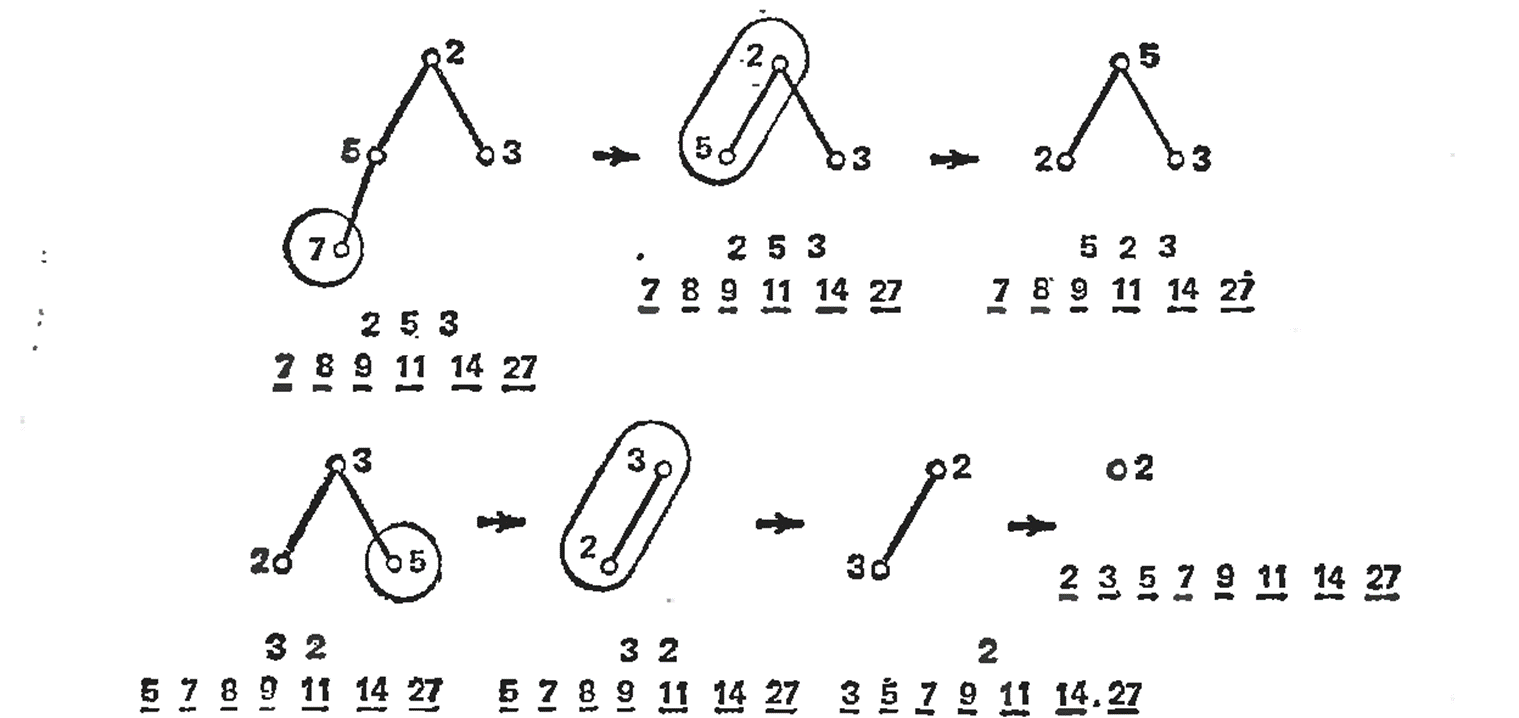


Рис. 7.3 Полная последовательность шагов при сортировке вершин пирамиды Т8

Поскольку для такой процедуры сортировки на входе требуется пирамида, нам нужен алгоритм для преобразования произвольного массива целых чисел в пирамиду. Такой алгоритм использует шаг 3 ранее описанного процесса. Это иллюстрируется на массиве целых чисел — рис. 7.4 а. Начнем с предположения, что целые числа этого массива расположены в корневом двоичном дереве, как было бы в случае, если в массиве была бы размещена пирамида — рис. 7.4 б. Затем проходим справа налево весь массив, проверяя, больше ли А (i), чем A (2i) и A (2i+l) для i=[n/2], . . ., 1 (рис. 7.5). Если для какого-то значения i А (i) не больше, чем А (2i) и A (2i+l), то применяем процедуру шага 3 и переставляем A(i) с большим из элементов A (2i) и А (2i+1).

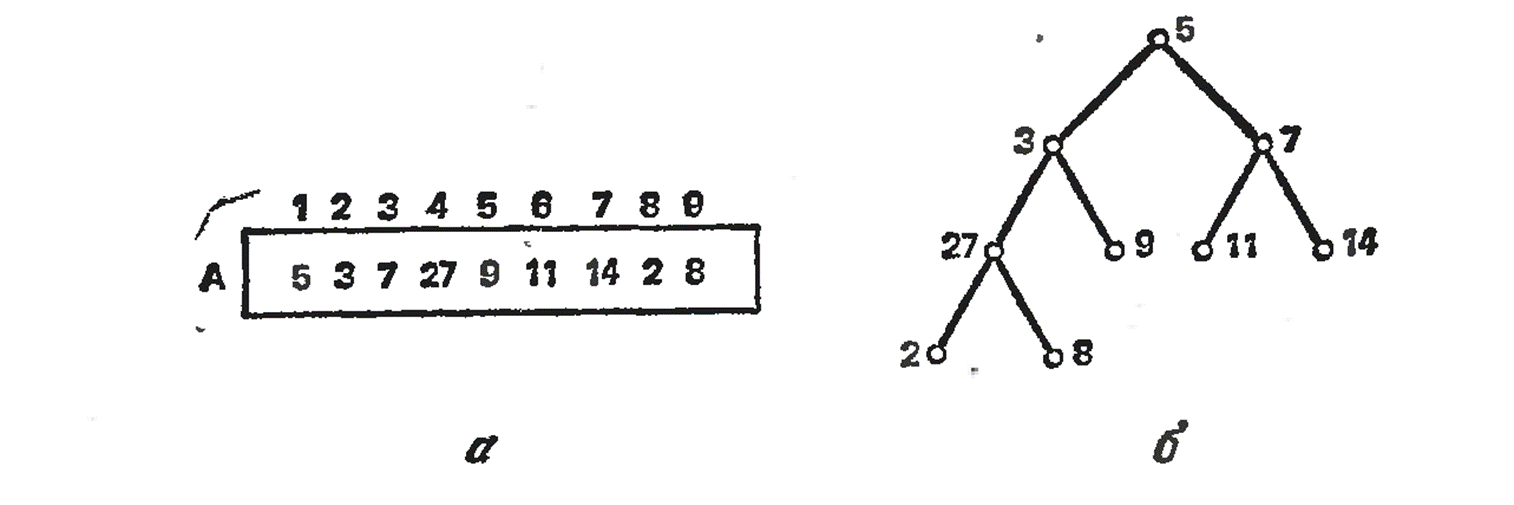


Рис. 7.4 Создание дерева из вводимой в произвольном порядке последовательности целых чисел

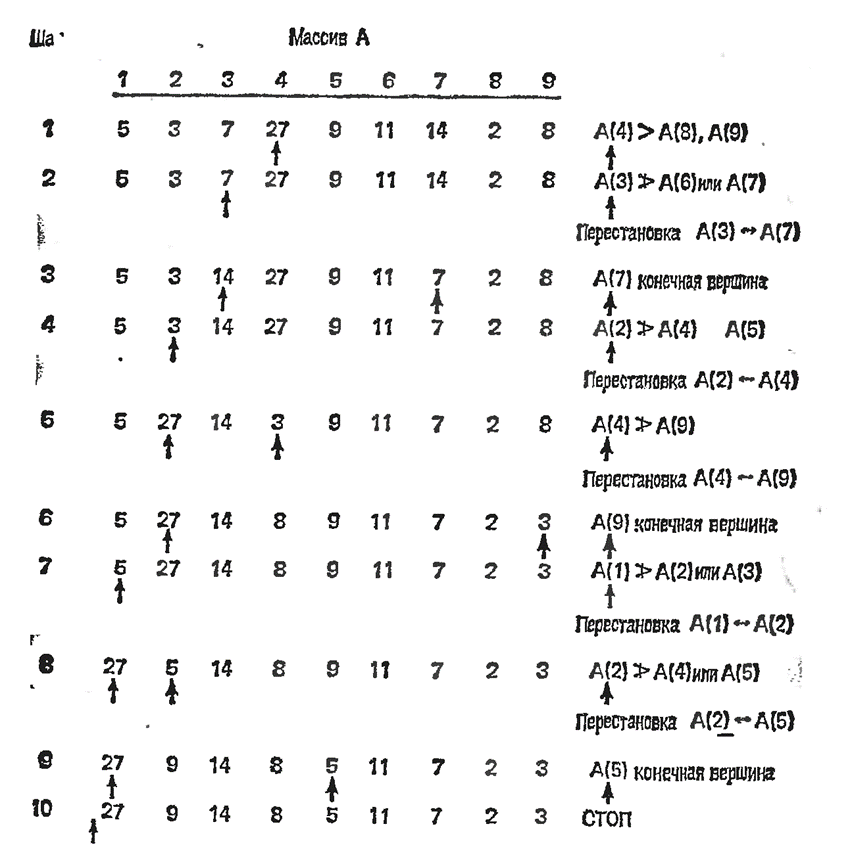
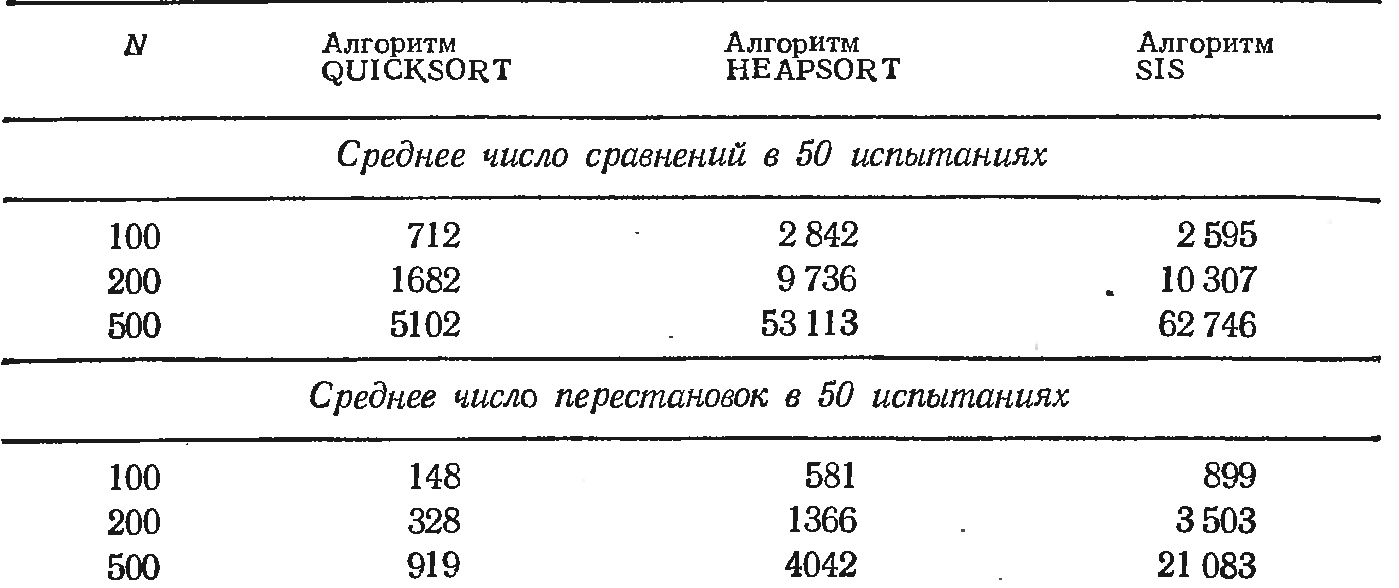


Рис. 7.5 Создание пирамиды из последовательности, представленной на Рис. 7.4

Средние значения, полученные в этих местах для каждого зна­чения *N* и каждого алгоритма, сведены в табличку. Мы будем анали­зировать только число сравнений для алгоритма QUICKSORT; ос­тальные данные обрабатываются аналогично и оставлены для инте­ресующегося читателя.

Результаты проверки сложности алгоритмов SIS, QUICKSORT и HEAPSORT



**Поиск**

Теперь обратимся к исследованию некоторых основных проблем, относящихся к поиску информации на стуктурах данных. Как и в предыдущем разделе, посвященному сортировки, будем предполагать, что вся информация хранится в записях, которые можно идентифицировать значениями ключей, т.е. записи Ri  соответствует значение ключа, обозначаемое Ki.

Предположим, что в файле расположены случайным образом N записей в виде линейного массива. Очевидным методом поиска заданной записи будет последовательный просмотр ключей. Если найден нужный ключ, поиск оканчивается успешно; в противном случае будут просмотрены все ключи, а поиск окажется безуспешным. Если все возможные порядки расположения ключей равновероятны, то такой алгоритм требует O(N) основных операций как в худшем, так и в среднем случаях. Время поиска можно заметно уменьшить, если предварительно упорядочить файл по ключам. Эта предварительная работа имеет смысл, если файл достаточно велик и к нему обращаются часто.

Далее будем предполагать, что у нас есть N записей, уже упорядоченных по ключам так, что К1<К2<…<КN. Пусть дан ключ К и нужно найти соответствующую запись в файле (безуспешный поиск).

Предположим, что мы обратились к середине файла и обнаружили там ключ Ki. Сравним К и Кi. Если К=Кi, то нужная запись найдена. Если К<Кi, то ключ К должен находиться в части файла, предшествующей Кi (если запись с ключом К вообще существует). Аналогично, если Кi<К, то дальнейший поиск следует вести в части файла, следующей за Кi. Если повторять эту процедуру проверки ключа Кi из середины непросмотренной части файла , тогда каждое безуспешное сравнение К с Кi будет исключать из рассмотрения приблизительно половину непросмотренной части.

Блок-схема этой процедуры, известной под названием *двоичный поиск*, приведена на рис.8.1.

**Algorithm BSEARCH** (Binary search - двоичный поиск) поиска записи с ключом К в файле с N≥2 записями , ключи которых упорядочены по возрастанию К1<К2…<КN.

*Шаг 0.*[Инициализация] **Set** FIRST←1; LAST← N. (FIRST и LAST- указатели первого и последнего ключей в еще не просмотренной части файла.)

*Шаг 1.*[Основной цикл] **While** LAST≥FIRST **do through** шаг 4 **od.**

*Шаг 2.* [Получение центрального ключа] **Set** I←|\_(FIRST + LAST)/2\_| .(Кi- ключ, расположенный в середине или слева от середины еще не просмотренной части файла.)

*Шаг 3.*[Проверка на успешное завершение ] **If** К=КI  **then** PRINT «Успешное окончание, ключ равен КI»; **and** STOP **fi.**

*Шаг 4.* [Сравнение] **If** K**<**KI **then set**  LAST←I-1 **else set**

FIRST←I+1 **fi.**

*Шаг 5.* [Безуспешный поиск] PRINT «безуспешно»; **and** STOP.

Алгоритм BSEARCH используется для отыскания К=42 на рис. 14.

Метод двоичного поиска можно также применить для того, чтобы представить упорядоченный файл в виде двоичного дерева. Значение ключа, найденное при первом выполнении шага 2 (К(8)=53), является корнем дерева. Интервалы ключей слева (1,7) и справа (9,16) от этого значения помещаются на стек. Верхний интервал снимается со стека и с помощью шага 2 в нем отыскивается средний элемент (или элемент слева от середины). Этот ключ (К(4)=33) становится следующим после корня элементом влево, если его значение меньше значения корня, и следующим вправо в противном случае.

Алгоритм BSEARCH используется для отыскания К=42 на рис.8.2.

Подынтервалы этого интервала справа и слева от вновь добавленного ключа [(1,3), (5,7)] помещаются теперь на стек. Эта процедура повторяется до тех пор, пока стек не окажется пустым. На рис.8.3 показано двоичное дерево, которое было бы построено для 16 упорядоченных ключей с рис. 8.2.

Инициализация

FIRST←1

LAST←N

LAST≥

FIRST ?

Получение центрального ключа

Кi

К:Кi

?

Результат:

Кi

Начало

Окончание

LAST←I-1

FIRST ←I+1

Результат :

«безуспешно»

Окончание

Нет

Да

<

>

Рис. 8.1. Блок-схема двоичного поиска.

Двоичный поиск можно теперь интерпретировать как прохождение этого дерева от корня до искомой записи. Если достигнута конечная вершина, а заданный ключ не найден, искомая запись в данном файле отсутствует. Заметим, что число вершин на единственном пути от корня к заданному ключу К равно числу сравнений, выполняемых алгоритмом BSEARCH при попытке отыскания К.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

7 14 27 33 42 49 51 53 67 70 77 81 89 94 95 99

**↑**FIRST=1

**↑**I=8

**↑**FIRST=1

LAST=16**↑**

7 14 27 33 42 49 51

**↑**I=4

**↑**LAST=7

42 49 51

FIRST=5**↑**

**↑**LAST=7

↑

I=6

42

**↑**FIRST=LAST=1=5,K=K5,ВОЗВРАТ

Рис. 8.2. Использование алгоритма BSEARCH для отыскания записи с ключом 42.

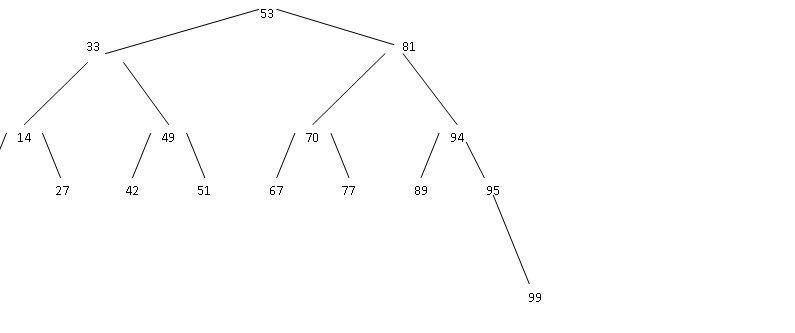


Рис. 8.3. Представление файла в виде двоичного дерева,

построенного с помощью алгоритма BSEARCH.