Темы курсовых работ
по «Технология программирования (Python)»

Тема 1: Раскрашивание карты методом исчерпывающего поиска

Чтобы на географической карте было удобно различать регионы, ее раскрашивают по следующему правилу: два региона должны быть окрашены в разные цвета, если их границы имеют более чем конечное число общих точек. (Обычно составители карт не страдают топологическими патологиями и не ищут вырожденных примеров, противоречащих здравому смыслу.) С другой стороны, картографам предстоит оплачивать типографские счета, поэтому, чем меньше цветов будет использовано, тем лучше. В частности, картографы, расписывающие карту как попало, распишутся лишь в своем легкомыслии: им придется использовать больше красок, чем это необходимо. Свои действия нужно планировать заранее. Итак, задача о раскрашивании карты сводится, в сущности, к определению минимального числа красок.

Для решения этой задачи обратимся к помощи компьютера. Тут нас подстерегают трудности: большинство ЭВМ лишено зрения, поэтому они не могут посмотреть на карту; к счастью, им нужно знать лишь, какие регионы являются соседями, т. е. смежны друг другу. Размер и форма регионов не влияют на раскраску, важно лишь наличие нетривиальных контактов между ними. Для представления отношения смежности полезно воспользоваться неориентированным графом.

Неориентированный граф состоит из конечного множества вершин и конечного множества ребер, связывающих вершины. Любые две вершины связаны не более чем одним ребром; не должно быть двух дублирующих друг друга ребер; кроме того, для рассматриваемой задачи мы запрещаем ребру связывать вершину с самой собой. На рис. 1.1 изображен неориентированный граф, представляющий первые 49 американских штатов. Ввести граф в ЭВМ несложно: достаточно перечислить все вершины, сопроводив каждую списком смежных ей вершин. Граф может не иметь вершин, а значит, и ребер; такой граф называется пустым. Вершина может быть изолированной, если нет ребер, связывающих ее с другими вершинами (примером тому могли бы служить Аляска и Гавайи); точно так же две части графа окажутся изолированными друг от друга, если нет ребер, их связывающих. Аналогия между картами и неориентированными графами столь тесна, что мы будем использовать эти понятия как равнозначные. Ну, а польза, приносимая графами, столь велика, что всем программистам следует иметь представление об их основных свойствах.

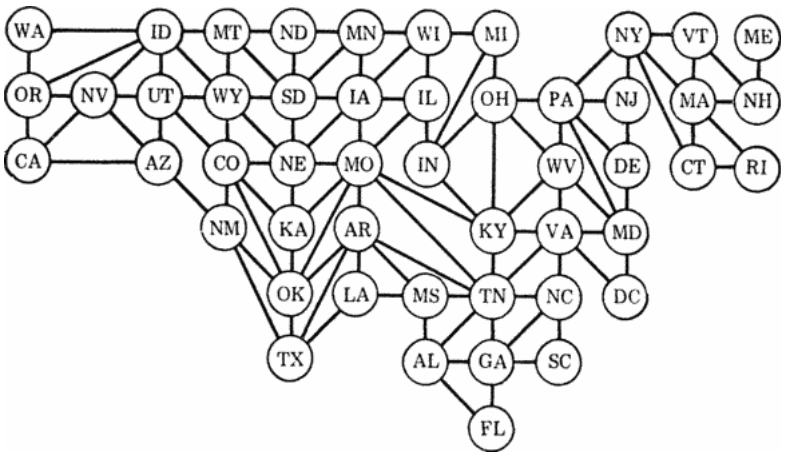


Рисунок 1.1. Топологическая карта Соединенных Штатов. Для нее достаточно четырех цветов. (WA — Вашингтон, OR — Орегон, CA — Калифорния, NV — Невада, ID — Айдахо, UT — Юта, AZ — Аризона, MT — Монтана, WY — Вайоминг, CO — Колорадо, NM — Нью-Мексико, ND — Северная Дакота, SD — Южная Дакота, NE — Небраска, KA — Канзас, OK — Оклахома, TX — Техас, MN — Миннесота, IA — Айова, MO — Миссури, AR — Арканзас, LA — Луизиана, WI — Висконсин, IL — Иллинойс, IN — Индиана, MS — Миссисипи, AL — Алабама, MI — Мичиган, OH — Огайо, KY — Кентукки, TN — Теннесси, GA — Джорджия, FL — Флорида, PA — Пенсильвания, WV — Западная Виргиния, VA — Виргиния, NC — Северная Каролина, SC — Южная Каролина, NY — Нью-Йорк, NJ — Нью-Джерси, DE — Делавэр, MD — Мэриленд, DC — округ Колумбия, VT — Вермонт, MA — Массачусетс, CT — Коннектикут, WE — Мэн, NH — Нью-Гэмпшир, RI — Род-Айленд.)

Тема. Напишите программу, раскрашивающую карту в минимальное число цветов. Исходными данными служит список регионов с указанием соседей каждого региона. Результатом должен быть список регионов с приписанными им цветами и общее число использованных цветов. Обычно проще всего для обозначения регионов и цветов применить положительные числа, но куда приятнее (и полезнее для отладки), если допускается ввод более привычных названий. Исходные данные должны проверяться на непротиворечивость; выявляйте нелепые номера вершин и связанные с собой вершины. Постарайтесь сделать программу по возможности эффективной, иначе раскраска тяжелых случаев окажется для вас слишком дорогим удовольствием.

Указания исполнителю. Исходная карта не обязана быть планарной. В самом деле, вполне допустимыми крайними случаями служат карты, в которых любые два региона — соседи, и карты, в которых никакие два региона не являются соседями. Последний случай соответствует раскраске множества раздельных шаров, когда достаточно только одного цвета. Проверка планарности — важная тема информатики, ей посвящено немало статей. Возможно, вас заинтересует проверка гипотезы о четырех красках, утверждающей, что для любой планарной карты требуется не более четырех красок. Если вам удастся подтвердить или опровергнуть ее, вы сделаете себе имя.

Из ресурсов, требуемых данной задачей, самый важный — время. Конечно, нет смысла перебирать все возможные решения, поскольку их число быстро

увеличивается с ростом числа регионов, а доля правильных решений (даже если таковых несколько) мала. Лучше воспользоваться методом перебора с возвратами. Начните с выбора некоторого региона и приписывания ему цвета. В дальнейшем переходите к соседнему нераскрашенному региону и пытайтесь приписать ему какой-нибудь из использованных цветов, совместимый с уже сделанной раскраской. (Может случиться, что раскрашивать больше нечего, тогда задача решена. Возможен и случай, когда не осталось нераскрашенных регионов, соседних с раскрашенными, т. е. попалась **несвязная** карта.) Если в некоторый момент новый регион не удается раскрасить, отступайте от уже раскрашенных регионов (в соответствии с порядком раскраски) до тех пор, пока не найдется регион, цвет которого можно изменить. Раскрасьте его в цвет, которого он ранее не имел, и снова продвигайтесь вперед. Если при отступлении вы возвратились в регион, раскрашенный первым, добавьте к своей палитре новый цвет и начните сначала.

Развитие темы. При использовании метода перебора с возвратами огромное влияние на время работы оказывает порядок выбора регионов. Учитывая этот эффект, можно заранее упорядочить регионы или использовать некоторые эвристики для выбора очередного региона. По-видимому, те регионы, у которых много соседей, раскрасить труднее, поскольку на их цвет накладывается больше ограничений. Из тех же соображений почти изолированную группу регионов следует рассмотреть отдельно, так как если ее не удастся раскрасить некоторым набором цветов, то и всю карту — тоже. Идея в обоих случаях состоит в том, что, если раскраска какого-либо региона может вызвать затруднения, ее нужно выполнить пораньше, чтобы не тратить время на разрушение почти законченной раскраски. Разумеется, полное решение такой «предварительной» задачи равносильно решению исходной задачи, но ведь и небольшой вклад может принести вполне ощутимую прибыль. Сравните несколько стратегий предварительного упорядочения по стоимости и эффекту.

Тема 2: Автоматическое форматирование текста

Известно, вам или нет, но с недавних пор еще одно тяжкое бремя свалилось с плеч человечества. Заботу о создании и размещении опечаток в тексте взяли на себя компьютеры. Там, где раньше линотипы отливали горячий свинец в строки, теперь небольшие, вполне доступные по цене компьютеры методами фотонабора выдают нескончаемые потоки готовых текстов. Жаль только, что с появлением новых эффективных методов уходит очарование доброго старого времени. Ну какой, скажите, интерес выискивать опечатки в воскресном номере Нью-Йорк Таймс, в которых и заключается весь юмор этого обширного собрания важных скучностей, если вы знаете, что компьютер способен делать ошибки в сотни раз быстрее, чем человек? Такова цена, которую приходится платить за прогресс.

Конечно, реальный прогресс заключен в том, что в издательском деле компьютер привлекается в качестве подмастерья, некоего чудесного помощника, способного выполнять черную работу быстро и — при аккуратном программировании — почти бесплатно. Программисты уже пользуются руководствами по вычислительной технике, изданными при помощи ЭВМ. Такие руководства часто очень неудобны для чтения из-за неудачного шрифта, которым снабжено печатающее устройство машины. Однако большинство людей и не подозревает, что многие журналы, газеты и книги также печатаются с помощью ЭВМ. Они выглядят гораздо привлекательнее благодаря тому, что машина не только редактирует и соответствующим образом располагает текст, но и управляет специальными периферийными фотонаборными устройствами. Последние, обладая десятками шрифтов различной гарнитуры, выдают готовую к изданию продукцию. Черновик настоящей книги также был подготовлен при помощи такой системы, и первые читатели были уверены, что держат в руках фотокопию реальной книги, а вовсе не некий аналог обычного машинописного экземпляра.

Система подготовки публикаций состоит из четырех компонентов. Во-первых, необходима хорошая **файловая система**, в которой можно хранить готовящиеся и архивные текстовые файлы. Обычно память для хранения файлов предоставляется операционной системой, но известен случай, когда в качестве такой памяти использовался шкаф для перфокарт в кабинете автора. Конечно, перфокарты не самый практичный носитель, когда речь идет об операциях над большими объемами информации, например при издании газет. Во-вторых, нужен **редактор текстов**, для того чтобы вносить изменения и поправки в файлы перед выдачей на печать. Редакторы текстов также имеются, в большинстве операционных систем, но может понадобиться специальный редактор издания, обладающий именно теми возможностями, которые требуются при подготовке публикаций. Третий элемент — **форматор**, который умеет размещать заголовки, выбирать размер страницы, располагать материал в таблицах, выделять абзацы и т. п. Форматор работает с такими элементами текста, как слова, предложения, абзацы, т. е. уже на том уровне, на котором текст воспринимается человеком. Наконец, имеется **программа-наборщик**, которая преобразует форматированный текст в его образ на внешнем носителе. Работа этой программы связана в первую очередь с особенностями шрифтов, физическими размерами, командами выводного

устройства, отдельными литерами и тому подобными вещами. Программа-наборщик, так же как и оператор линотипа, готова выдать на печать любой вздор, лишь бы он поместился в отведенное ему место. Функционально файловая система и редактор текстов заботятся о содержании текста, а форматор и наборщик — о том, как он будет выглядеть. Эта тема посвящена форматированию текстов.

Форматор

Процесс форматирования текста вручную проходит несколько этапов. Вначале автор создает черновик рукописи, и он перепечатывается набело. Затем автор вместе с редактором (по крайней мере, когда речь идет о больших публикациях) принимаются терзать эту рукопись, пока там не останется живого места, после чего автор начинает работу над новым вариантом рукописи. Этот цикл повторяется до тех пор, пока и автор, и редактор не будут удовлетворены. Затем рукопись еще раз перепечатывается (как правило, через два интервала) и передается техническому редактору. Он размечает рукопись, давая всевозможные указания относительно наборных шрифтов, размера и расположения заголовков, полосы набора, курсива и прочих деталей, определяющих в конечном счете внешний вид издания. Разметка делается при помощи специальных обозначений, и каждый значок ставится в то место рукописи, к которому он относится. Размеченная рукопись отправляется в наборный цех, где текст набирают и делают корректурный оттиск в нескольких экземплярах, называемый версткой. Верстка возвращается в редакцию, где редактор и корректор сверяют ее с окончательным вариантом рукописи. Мелкие ошибки легко исправить в наборном цехе, заменив одну строку набора другой. Но как быть, если автор вдруг решит, что вся четвертая глава никуда не годится, или художнику покажется, что гарнитура бодони будет выглядеть лучше литературной? Такие изменения повлекут за собой новый набор и обойдутся недешево. Можно только диву даваться, насколько по-разному воспринимаются типографский текст и тот же текст, напечатанный на машинке.

Система подготовки публикаций с помощью ЭВМ исключает из этого цикла большую часть работы и множество людей. Как и прежде, автор должен подготовить первоначальный вариант рукописи. Но затем рукопись поступает не в машинописное бюро, а в файловую систему машины. Текст рукописи можно ввести, как и любую информацию для ЭВМ, либо с перфокарт, либо непосредственно через терминальное устройство машины. (Большая часть этой рукописи была отперфорирована.) Автор исполняет также и функции технического редактора, сопровождая текст простейшими командами для форматора. Текстовый файл с рукописью обрабатывается форматором и наборщиком, в результате чего получается черновая верстка окончательного печатного текста. Эта черновая верстка выглядит куда как более чисто, чем машинописный вариант, — она оформлена в виде отпечатанных типографским способом страниц с правильными номерами, радующим глаз шрифтом и т. п. Заметим, что все это происходит еще до начала какого-либо пересмотра рукописи.

Затем автор и редактор начинают работать над рукописью. Интеллектуальная часть работы точно такая же, как и раньше, но теперь им значительно проще представить себе конечный результат, поскольку рукопись выглядит почти как готовое печатное издание. Да и процесс редактирования уже не такой трудоемкий. Для того чтобы добавить или убрать фразу, не нужно ничего перепечатывать — все изменения вносятся при помощи редактора текстов, подобно тому, как заменяются строки в программах. Переупорядочение больших разделов, а также вызов текстов, временно отсутствующих в основной памяти, осуществляется средствами файловой системы. Поскольку текст в любом случае придется переформатировать, то можно поменять и команды форматора, тоже просто изменив содержимое текстового файла. Наконец, выполнение программы форматора на ЭВМ стоит такие пустяки, что все множество сеансов форматирования текста обойдется наверняка несравненно дешевле, чем одна перепечатка его на машинке при старом способе работы. Имеется, правда, единственное опасение — авторы, зачарованные столь аккуратно оформленной рукописью, будут неохотно вносить в нее изменения; ведь в течение долгих лет за всякое исправление в верстке, противоречащее рукописи, им приходилось расплачиваться из авторского гонорара. Поэтому если мы хотим правильно использовать ЭВМ для подготовки публикаций, то и авторов необходимо должным образом перестроить.

Команды форматирования

Как работает типичный форматор? В **исходном файле** текст, предназначенный для редактирования, оформлен как обычная машинопись (с той разницей, что здесь не нужно заботиться об интервале, полях и т. п.) с добавленными командами форматирования. Команды должны располагаться с первой позиции записи и начинаться со знака «?», чтобы их можно было отличить от обычного текста, по крайней мере в нашем примере. Для самого простого вывода достаточно иметь команды для установки размера страницы и для разбиения текста на абзацы. В пределах одного абзаца исходный текст можно вывести в одном из трех режимов:

Неплотный — строки исходного текста передаются на вывод в том виде, в котором они записаны в исходном файле. Этот режим обычно используется для выдачи таблиц и других заранее оформленных материалов без каких бы то ни было изменений.

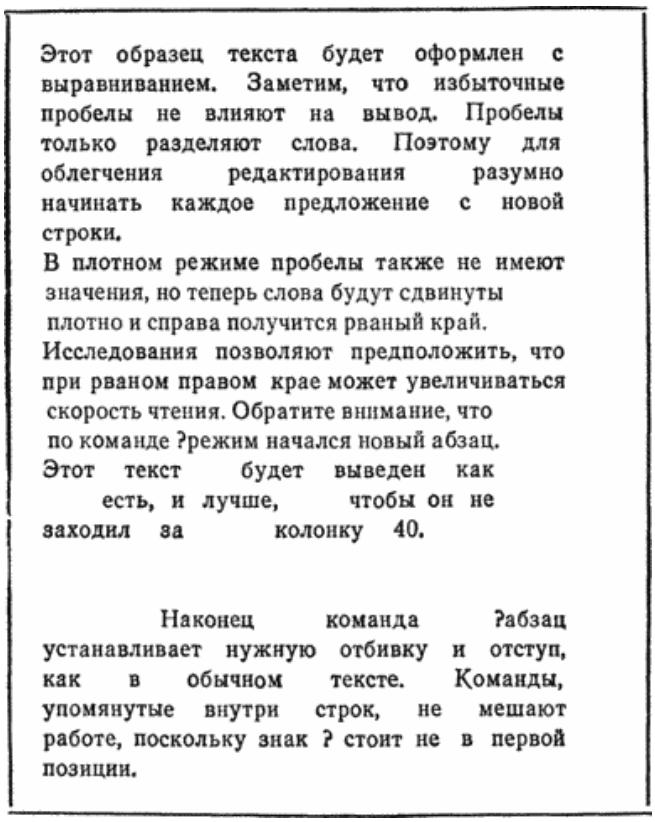
Плотный — строки вывода формируются из исходного текста слева направо наиболее плотным образом, переход на следующую строку происходит только тогда, когда очередное слово исходного текста не помещается в предыдущей строке вывода. Между словами оставляется один пробел, а после символов конца предложения, т. е. после точки, вопросительного и восклицательного знаков, дается два пробела. Именно в этом режиме обычно печатается текст на машинке. Заметим, что в плотном режиме избыточные пробелы между словами исходного текста игнорируются, пробелы служат только для разделения слов исходного текста.

Выравнивание — сначала из исходного текста формируется полный абзац в плотном режиме. Затем в каждую строку, кроме последней, добавляются пробелы между словами так, чтобы последнее слово заканчивалось у правого края страницы. Ни в один промежуток нельзя добавить (n + 1)-й пробел, пока во всех остальных промежутках данной строки не стало по n пробелов, а пробел после символа конца предложения можно добавить, лишь если во всех других промежутках строки уже есть по два пробела. Пробелы следует добавлять в случайно выбираемые промежутки между словами; если пробелы вставлять по какому-нибудь заранее выбранному правилу, то в выводном тексте образуются неприятные для глаза белые полосы. Выровненный текст по внешнему виду приближается к книжному, но не так совершенен, поскольку не учитываются неодинаковые размеры букв.

Для обработки простого текста достаточно иметь команды ?размер, ?абзац и ?режим. Действие этих команд продемонстрировано на рис. 2.1 и 2.2.

```
?размер 42 40
?режим выравнивание
Этот образец текста будет оформлен с выравниванием. Заметим,
     что      избыточные пробелы      не влияют
на         вывод.
Пробелы только разделяют слова.
Поэтому для облегчения редактирования разумно начинать каждое
предложение с новой строки.
?режим плотный
В плотном режиме  пробелы также не имеют значения,
но теперь слова будут сдвинуты плотно и справа получится
рваный край.
Исследования позволяют предположить, что при рваном
правом крае может увеличиваться скорость чтения.
Обратите внимание, что по команде ?режим начался новый абзац
?режим неплотный
Этот текст     будет  выведен  как
     есть, и лучше,     чтобы он  не
заходил  за        колонку  40.
?режим выравнивание
?абзац 10 2
Наконец команда ?абзац устанавливает нужную отбивку и
отступ, как в обычном тексте.
Команды, упомянутые внутри строк, не мешают работе,
поскольку знак ? стоит не в первой позиции.
```

Рисунок 2.1. Пример необработанного исходного текста.


Этот образец текста будет оформлен с выравниванием. Заметим, что избыточные пробелы не влияют на вывод. Пробелы только разделяют слова. Поэтому для облегчения редактирования разумно начинать каждое предложение с новой строки.
В плотном режиме пробелы также не имеют значения, но теперь слова будут сдвинуты плотно и справа получится рваный край.
Исследования позволяют предположить, что при рваном правом крае может увеличиваться скорость чтения. Обратите внимание, что по команде ?режим начался новый абзац.
Этот текст будет выведен как есть, и лучше, чтобы он не заходил за колонку 40.

Наконец команда ?абзац устанавливает нужную отбивку и отступ, как в обычном тексте. Команды, упомянутые внутри строк, не мешают работе, поскольку знак ? стоит не в первой позиции.

Рисунок 2.2. Тот же текст после форматирования.

?размер высота ширина

Команда ?размер устанавливает размер страниц текста; страница измеряется аргументами высота, равным количеству строк, и ширина, равным количеству литер в каждой строке. Как только выведены очередные строки в количестве высота штук, форматор начинает новую страницу. Выводные строки могут заполнять все пространство между колонками с номерами 1 и ширина. Новую команду ?размер можно выдать в любом месте текста, но она приводит к автоматическому завершению текущего абзаца. Формирование прерванного абзаца завершается со старыми значениями высота и ширина, а затем начинают действовать новые значения. Изменение размера страницы может привести также к переходу на новую страницу, если новое значение высота меньше прежнего. В начале сеанса форматирования значение высота равно 40, а ширина — 72, и если пользователя эти значения устраивают, то команда ?размер необязательна.

?режим тип заполнения

Команда ?режим устанавливает режим обработки выводимого текста. Аргумент тип заполнения может принимать в качестве значения одну из цепочек: неплотный, плотный или выравнивание (другие значения не допускаются). По команде ?режим текущий абзац прерывается, но его обработка завершается в прежнем режиме. В начале работы установлен плотный режим; если пользователя это устраивает, то команда ?режим необязательна.

?абзац отступ отбивка

По команде ?абзац начинается новый абзац. Первая строка нового абзаца начинается на отступ позиций правее левого поля (отступ может быть нулевым, а

позже вы увидите также, что он может быть отрицательным), а между предыдущим и новым абзацем оставляются пустые строки, количество которых задает аргумент отбивка. Если не указана отбивка или отбивка и отступ, то их значения берутся из последней команды ?абзац, где они были указаны. Начальное значение отступ равно 3, а отбивка — 0; если эти значения удовлетворительны, то в команде ?абзац можно не указывать аргументы. Заметим, что при значении отступ, равном 3, первая строка нового абзаца начинается в колонке 4.

Но команд ?размер, ?режим и ?абзац недостаточно. Полный форматор должен включать по меньшей мере еще следующие команды.

?поле слева справа

Команда ?поле указывает, что выводимый текст будет иметь левое и правое поля, начинающиеся в колонках слева и справа. Естественно, что левое поле должно начинаться в колонке с номером 1 или более, а правое — в колонке с номером не больше текущего значения ширина страницы. По команде ?поле начинается новый абзац. С введением полей приобретает смысл отрицательный аргумент отступ в команде ?абзац; первая строка нового абзаца начинается с выступом относительно левого края страницы.

?интервал отбивка

Команда ?интервал устанавливает, что между строками вывода нужно оставлять отбивка -1 пустых строк. Установка значения отбивка, равного 1, соответствует указанию для машинистки печатать через один интервал. Отбивка 2 соответствует печати через два интервала, отбивка 3 — через три интервала и т. д. Эта команда прерывает текущий абзац.

?пусто n

По команде ?пусто завершается текущий абзац, выводится n пустых строк с текущим значением интервала между строками. Эта команда по своему действию эквивалентна (n + 1) возвратам каретки на пишущей машинке. Если из-за вывода пустых строк происходит переход на следующую страницу, то новая страница действительно начинается, но пустые строки в начале страницы не выводятся. По умолчанию значение n нулевое.

?пропуск n

Команда ?пропуск работает так же, как ?пусто, но выводится точно n пустых строк; текущее значение аргумента команды ?интервал не учитывается. Это действие эквивалентно повороту валика пишущей машинки на n + 1 интервалов.

?центр

Команда ?центр выбирает из входного текста очередную строку, убирает из нее лишние пробелы и центрирует то, что получилось, между левым и правым полями следующей выводной строки. Текущий абзац не заканчивается, но перед центрируемой строкой может получиться более короткая строка. Центрируемая строка выводится с текущим интервалом. Если центрируемая строка слишком длинная и не помещается между установленными полями, то имеет место ошибка.

?страница

По этой команде прерывается текущий абзац и после вывода последней строки абзаца происходит переход на новую страницу выводного текста.

?остаток n

По этой команде текущий абзац завершается и выводится. Если в текущей странице осталось меньше чем n пустых строк, то команда ?остаток действует как ? страница. В противном случае она игнорируется. Таким образом, эта команда проверяет, осталось ли еще достаточно места в текущей странице.

?колонтитул глубина место позиция

Команда ?колонтитул устанавливает текст колонтитула, который будет печататься сверху на каждой странице, начиная со следующей. Последующие глубина строк исходного текста запоминаются без изменений и выводятся в качестве колонтитула в верхние глубина строк каждой новой страницы. В строке номер место печатается номер страницы слева, справа или в центре, в зависимости от значения аргумента позиция, который может быть одной из цепочек: слева, справа или центр. Страницы нумеруются числами, начиная с единицы, при переходе к следующей странице номер увеличивается на 1. При выводе колонтитула используются те значения полей, которые действовали в момент задания колонтитула. Колонтитул можно отменить при помощи команды ? колонтитул с нулевым значением аргумента глубина. Команда ?колонтитул не прерывает текущий абзац.

?номер n

По команде ?номер номер текущей страницы устанавливается равным n; текущий абзац не прерывается.

?прерывание

Команда ?прерывание означает переход к новому абзацу.

?сноска глубина

По команде ?сноска следующие глубина строк, включая команды, помещаются в конце страницы в качестве сноски. Значения управляющих параметров форматора — поля, интервал и т. д. — сохраняются и затем используются в качестве начального состояния при обработке сноски. Из исходного файла после сноски выбирается достаточное количество слов для заполнения той строки, которая обрабатывалась, когда встретилась команда ?сноска. Затем обрабатывается сноска и помещается в конец страницы. Если в текущей странице уже были сноски, то они выталкиваются в верхние строки, освобождая место для новой сноски. Если при этом сноски начинают наезжать на уже сформатированные строки текущей страницы, то страница завершается, а остаток сноски попадает на следующую страницу (именно поэтому сначала заполняется текущая строка основного текста, а уж потом начинается обработка сноски). После вывода глубина строк сноски продолжается обработка основного текста с прежними значениями управляющих параметров форматора (хотя номер страницы мог уже измениться), Очевидно, что команда ?сноска не должна прерывать текущий абзац и не может находиться внутри другой сноски.

?имя фиктивное настоящее

Эта команда сообщает форматору, что впредь до следующей команды ?имя вместо литеры, имеющей настоящее имя будет использоваться литера, имеющая фиктивное имя. Каждый раз перед выдачей строки на печать

все фиктивные литеры заменяются соответствующими настоящими литерами. Например, пробелы используются специальным образом для разделения слов; при помощи команды ?имя можно включить в выводной текст пробелы, не разрывая при этом слов. Команда ?имя не прерывает текущий абзац. Все переименования можно отменить, выдав команду ?имя без агрументов.

Несколько слов о словах, буквах и аргументах

Для того чтобы правильно заполнять строки и выравнивать текст, форматор должен уметь распознавать слова и предложения. Со **словами** все просто — любая цепочка литер без пробелов, заканчивающаяся пробелом или концом записи, является словом. Заметим, что по этому определению знаки препинания входят в состав предшествующего слова. **Предложение** обычно заканчивается точкой, а в конце предложения, как правило, вместо одного пробела оставляется два. Но ведь точка может стоять внутри скобок или кавычек, а после двоеточия правилами предусматривается два пробела. Поэтому слова, заканчивающиеся литерами

. ? ! .) ?) !) ." ?" !" .") ?") !") :

следует считать концом предложения. Могут быть также и другие варианты, которые здесь не упомянуты; авторы часто весьма вольно обращаются с пунктуацией.

Если ваш форматор будет работать в системе разделения времени, которая умеет вводить прописные и строчные буквы и допускает вывод на терминал, то, несомненно, алфавит языка, на котором реализован форматор, должен включать большие и малые буквы. Но если вы работаете в системе, ориентированной на ввод с перфокарт, то у вас возникнут трудности с чтением букв двух видов, поскольку на перфораторах, как правило, отсутствует переключатель регистров (лучше, если системе все-таки удастся каким-то образом печатать буквы обоих видов, иначе ваше начинание обречено на провал). Для ввода с перфокарт выберите какую-нибудь литеру, например ↑, которая будет служить признаком прописной буквы. Так, текст

Машина БЭСМ-6

нужно перфорировать как

↑машина ↑б↑э↑с↑м-6

Прописные буквы отмечаются специальным образом, поскольку они встречаются значительно реже строчных. Заметим, что буквы, отперфорированные обычным образом, считаются строчными, хотя на перфокартах они выглядят как прописные.

Аргументы команд могут быть двух видов. Некоторые аргументы представляют собой целые числа и задают либо значения управляющих параметров для форматора, либо число строк исходного текста, относящихся к этой команде. Другие аргументы являются словами или отдельными литерами, которые непосредственно используются в команде. Аргументы обоих видов разделяются пробелами, избыточные пробелы игнорируются, В команде ?имя второй аргумент может отсутствовать, тогда считается, что он равен пробелу (иначе при данных соглашениях пробел представить трудно). Следует позаботиться о том, чтобы для неправильных команд выдавались сообщения об ошибках.

Тема. Напишите для вашей системы форматор текстов, понимающий описанные выше команды. Поскольку форматирование текста не имеет большого смысла без возможности вывода прописных и строчных букв, то следует использовать выводное устройство с буквами обоих видов.

Указания исполнителю. Вы обнаружите, вероятно, что ваша программа тратит большую часть времени на ввод и вывод и совсем немного времени — на перемещение слов в строке. Значительная часть времени обработки будет уходить, по-видимому, на поиск пробелов между словами. С учетом всего этого ясно, что львиную долю усилий по оптимизации программы следует направить на центральный алгоритм сканирования и на взаимодействие форматора с внешним миром. Обработка команд и алгоритм размещения слов должны быть запрограммированы так, чтобы все было понятно. Как правило, для ввода/вывода следует пользоваться стандартными языковыми средствами, но в данной задаче мы сталкиваемся с тем случаем, когда особенности вашей операционной системы можно употребить с пользой для дела. Важно помнить только, что использование этих особенностей должно быть сконцентрировано в пределах подпрограмм ввода-вывода, а не рассеяно по всему форматору.
Набор команд был подобран с таким расчетом, чтобы требуемый вывод можно было получить за один просмотр входных данных. Ни для одной команды алгоритм не должен требовать повторного просмотра ввода. Если для некоторых алгоритмов потребуется рабочее пространство, как, например, для алгоритма обработки сноски, то попробуйте применить двойную буферизацию вывода и использовать свободный буфер в качестве рабочего пространства.
Развитие темы. В книгах можно встретить полужирный шрифт, курсив, греческие буквы, латинские рукописные и другие специальные символы. Все это имелось на выводных устройствах, но, как нетрудно догадаться, ни перфораторы, ни файловая память подобными возможностями не обладают. Для представления таких специальных литер используются специальные соглашения. Пусть, например, слова “et cetera” требуется набрать курсивом. Для этого нужно ввести текст “&i+ et cetera &i-”, и тогда на выводе получится “*et cetera*”. Тройка литер, начинающаяся значком “&”, называется **переключателем шрифта**. В данном примере вы видели включение и выключение курсива. Рассматривая подчеркивания, верхние и нижние индексы и т. п. как специальные начертания шрифтов, можно таким образом обеспечить доступ ко всем дополнительным средствам, имеющимся на вашем устройстве вывода. Разумеется, можно включить одновременно несколько переключателей, например чтобы вывести подчеркнутые греческие верхние индексы. (Возможно, вам понадобится также переключатель шрифта для возвратов по тексту вида & × n, где n — цифра от 1 до 9.)

Тема 3: Составление и оценка турнира

Едва ли не каждый из нас в свое время был болельщиком местной, чуть ли не самой сильной команды. Состоявшийся в конце сезона турнир должен был выявить чемпиона города, округа, штата, страны, мира или Вселенной. Но какое невезение — местные герои проиграли будущему победителю уже в первом круге турнира с немедленным выбыванием. Игра оказалась малоинтересной — никто даже не успел размяться. И ведь как обидно: самые настоящие слабаки в итоге занимают место, которое по праву должно принадлежать нашим парням, а болельщиков вместо волнующей борьбы в финале ждет убогое зрелище.

А виноват во всем **турнир с немедленным выбыванием**. Пусть имеется 2^n команд, $n > 0$. Тогда в первом круге команда 1 играет с командой 2, команда 3 с командой 4, ..., команда $2^n - 1$ с командой 2^n. Проигравшие вылетают, а победители выходят в следующий круг.

Рисунок 3.1. Простой турнир с немедленным выбыванием. Окончательное упорядочение, как это определено в тексте, имеет вид 1, 3, 5, 2, 8, 6, 4, 7.

На рис. 3.1 изображен турнир восьми команд. Если предположить, что более сильная команда всегда выигрывает (т. е. что не бывает срывов), лучшая команда, очевидно, завоюет первое место. Однако второй участник финальной игры может занимать в общей табели о рангах лишь место $2^{n-1} + 1$ при условии, что все более сильные команды оказались в одной **группе** с победителем. Победитель по мере своего продвижения выведет из розыгрыша хорошие команды, и слабой команде достанутся совсем никудышные соперники. Избежать подобной ситуации можно несколькими способами. Во-первых, команды (в дальнейшем будем называть их *соперниками*) можно *рассеять*, чтобы сильные соперники (оценка дается по итогам предыдущих выступлений) разместились по всей турнирной сетке. Например, самый сильный соперник попадает в позицию 1, второй по силе — в $2^{n-1} + 1$, третий — в $2^{n-1} + 2^{n-2} + 1$, четвертый — в $2^{n-2} + 1$ и т. д. Если предварительная оценка была достаточно точной, сильные соперники не выбьют друг друга в первых кругах. Во-вторых, можно устроить **турнир с отложенным выбыванием**, когда выбывают после двух поражений. Но на самом деле идеальным решением (хорошо бы еще и практичным!) был бы **круговой турнир**, в котором все соперники играют друг с другом ровно один раз. В предположении отсутствия срывов сильнейший соперник выиграет $2^n - 1$ встреч и проиграет 0, второй по силе

соответственно 2^n — 2 и 1 (уступит лишь сильнейшему), ..., а самый слабый — 0 и 2^n - 1 (проиграет всем). Трудность в том, что в круговом турнире нужно провести $2^{n-1}(2^n - 1)$ встреч, в то время как в турнире с немедленным выбыванием лишь 2^n - 1.

Таблица 3.1. Пример турнира по швейцарской системе

Круг 1 Пары	Победители	Круг 2 Пары	Победители	Круг 3 Пары	Победители	Итоговые результаты
1	1	1	1	1	1	1 (3-0)
8		2		3		3 (2-1)
2	2	3	3	5	2	2 (2-1)
7		5		2		4 (2-1)
3	3	8	8 (Срыв)	4	4	5 (1-2)
6		7		8		6 (1-2)
4	5 (Срыв)	6	4	6	6	8 (1-2)
5		4		7		7 (0-3)

Этот турнир недостаточно велик, чтобы показать достоинства швейцарской системы.

Оказавшись между двумя крайностями, выберем компромиссное решение — **швейцарскую систему**. В первом круге соперник, «посеянный» первым, встречается с последним, второй — с предпоследним и т. д. После каждого круга соперники упорядочиваются в соответствии с набранными очками. Внутри каждой группы (с равным количеством очков) соперники упорядочиваются по среднему числу очков у побежденных ими противников (тем самым ничья не учитывается). В следующем круге соперник, стоящий в описанной классификации на первом месте, встречается с соперником, занимающим наиболее высокое место из тех, с кем он еще не играл. Остальные пары определяются аналогичным образом: соперники должны иметь почти равное количество очков, причем повторные встречи не допускаются. В табл. 3.1 показан возможный трехкруговой турнир по швейцарской системе с восемью участниками. Крупный шахматный деятель Харкнесс утверждает, что турнир по швейцарской системе в $\sqrt{N + 2k}$ кругов, где N — число игроков, правильно расставит k + 1 первых игроков (и, из соображений симметрии, k + 1 последних игроков). Швейцарская система справедливее немедленного выбывания и гораздо быстрее круговой. Она позволяет всем соперникам играть в каждом круге. Вопрос состоит в том, как ведут себя подобные турниры в условиях реальных соревнований. Предположим, имеется 2^n соперников. Соперник 1 — сильнейший, соперник 2 — второй по силе, ..., соперник 2^n — слабейший. Для начала проведем круговой турнир, записывая результаты каждого матча. Если встречаются соперники i и j, i < j, положим вероятность победы игрока i равной $1/2 + (j - i)/2^{n+1}$.

Тем самым более сильный соперник побеждает с вероятностью, превышающей половину. Упорядочим соперников в соответствии с набранным в круговом турнире количеством очков. Внутри каждой группы команд с равным количеством очков упорядочим их по среднему числу очков, набранных побежденными ими соперниками. Если и здесь наблюдаются совпадения, соперники упорядочиваются по исходным номерам. В результате получается **круговая классификация**,

которую мы будем считать самой «справедливой»; она используется для оценки других способов организации турниров.
Следующий шаг состоит в том, чтобы с одной и той же базой данных провести турниры по швейцарской системе и с немедленным выбыванием. Для разбиения соперников на пары в каждом из этих турниров берутся результаты кругового турнира. Заметьте, что в обоих турнирах два соперника могут встретиться лишь однажды. **Швейцарская классификация** — это упорядочение после заключительного круга (всего n кругов), причем все оставшиеся неясности разрешаются в соответствии с начальным упорядочением. Затем начните турнир с немедленным выбыванием, составив пары для первого круга случайным образом. В **классификации по выбыванию** победитель финальной встречи идет первым, побежденный — вторым, и, вообще, проигравшие в i-м круге располагаются перед ранее выбывшими и после всех победивших в i-м и следующих кругах. Внутри группы побежденных в i-м круге соперники располагается в соответствии с итоговыми местами победивших их команд.
Чтобы сравнить эти классификации, используем новую и старую статистики, Старая статистика — это **корреляция мест** определяемая как

$R = 1 - 6 \sum_{i=1}^{N} (x_i - y_i)^2/(N^3 - N)$,

где x_i — место соперника i в одной классификации, y_i — место в другой классификации, N — общее число соперников (в данном случае 2^n). Другая статистика подсчитывает совпадения и определяется как $M = \max_i (\forall j) (j \leq i \supset x_j = y_j)$. Тем самым M равно максимальному числу мест (считая от сильнейших к слабейшим), в которых обе классификации в точности совпадают. Статистика R характеризует близость двух классификаций в целом, а M — совпадение верхних частей классификаций.

Тема. Напишите программу, читающую исходное значение n, проводящую каждый из трех турниров для 2n соперников и вычисляющую статистики R и M для каждой из трех пар классификаций. Проведите эксперимент большое число раз с постоянным значением n и подсчитайте средние значения M и R. Сравните, какая из двух систем — швейцарская или с немедленным выбыванием — лучше повторяет результаты кругового турнира.

Указания исполнителю. Разумеется, нужно досконально разбираться в разных системах проведения турниров, нужно эффективно программировать подбор пар. Но не упустите из виду еще один момент. Размеры кругового турнира заставляют эффективно запрограммировать внутренний цикл и экономно расходовать память для хранения результатов встреч. Разумеется, вам понадобится хороший генератор случайных чисел для определения результатов встреч. Наконец, при швейцарской системе возможны попытки дважды свести одну и ту же пару соперников, поэтому либо докажите, что такого не произойдет, либо измените алгоритм, избегая повторных встреч, но подчиняясь общему правилу: старайтесь сводить в пары соперников, набравших почти равное количество очков.

Развитие темы. Большинство расширений включает более подробный анализ и сравнение систем проведения турниров. Во-первых, заметьте, что нижняя часть классификации по итогам турнира с немедленным выбыванием носит довольно произвольный характер. Кроме того, соперникам, попавшим в эту часть классификации, весьма тоскливо, ибо они рано вылетели. Для утешения можно организовать турниры с немедленным выбыванием среди неудачников каждого круга. Результаты этих турниров, а не приведенное выше правило, расставят неудачников по местам. Поскольку и в этих побочных турнирах будут проигравшие, организуйте турниры неудачливых неудачников, и так до посинения. Заметьте, что турнир по-прежнему пройдет в n кругов, но теперь все соперники будут участниками всех кругов. Если во всех встречах побеждают сильнейшие, этот, более тщательно организованный турнир превращается в законченный алгоритм сортировки.

Вообще, турниры — это сортировка участвующих в них соперников, хотя правило сравнения и носит вероятностный характер. На основе любого метода сортировки, не нарушающего двух основных правил турниров, можно организовать состязание. Вот основные правила:

1. Ни один из соперников не должен участвовать более чем в одном матче одного круга, а число кругов должно примерно равняться логарифму числа участников.
2. Никакие два соперника не должны встречаться больше одного раза.

Используя изложенные идеи, вы можете оценить и классические способы проведения турниров, такие, как отложенное выбывание, и способы, придуманные вами.

В голову приходит также несколько статистических вопросов. Как влияет частичное или полное рассеивание на турниры с немедленным выбыванием? Как влияет случайная **жеребьевка** (т. е. случайное составление начальных пар) на ход турниров по швейцарской системе? Каков будет эффект введения иной функции превосходства? Наконец, поскольку для получения итоговой статистики по нескольким экспериментам, видимо, нельзя просто усреднять две наши статистики, спрашивается: какая статистическая операция должна быть использована?

Тема 4: Управление предприятиями и машинное моделирование

Моделирование нередко оказывается удобным средством для изучения тех или иных ситуаций, которые характерны для реальной жизни. Процесс моделирования дает нам кроме множества курьезов некоторые знания об изучаемом объекте. В самом деле, многие так увлекаются исследованием модели, что уже не возвращаются к объекту как таковому. Попробуйте, что получится у вас.

Деловая игра Менеджмент[12]

Решением совета директоров крупного промышленного концерна вы назначены президентом компании. Компания владеет несколькими фабриками. Каждый месяц компания покупает сырье, обрабатывает его и продает изготовленную продукцию публике, ждущей ее с нетерпением. Вам теперь придется решать, сколько и каких товаров выпускать, стоит ли и когда именно расширять производственные мощности, как финансировать их расширение и как принять скромно-застенчивый вид, отчитываясь о незаконных прибылях. Перед тем как приступить к работе, вы строите модель промышленности в целом, чтобы в частном порядке отработать свою линию поведения. И вот какую игру вы в результате изобрели.

Начальная ситуация

Моделирование ведется с шагом по времени в один месяц. В начале игры каждый игрок (президент компании) получает две **обычные фабрики**, четыре **единицы сырья и материалов** (сокращенно ЕСМ), две **единицы готовой продукции** (сокращенно ЕГП) и 10000 долл. наличными. Игроки занумерованы от 1 до N, и в первом круге игрок 1 — **старший**. С каждым кругом (т. е. ежемесячно) роль старшего переходит к следующему по порядку номеров игроку, после N-го старшим становится опять первый (так что номер старшего в круге T вычисляется по формуле (T mod N) + 1). На торгах при прочих равных условиях выигрывает самый старший игрок (тот, кто будет старшим в следующем круге).

Ежемесячные операции

Описанные ниже сделки происходят каждый месяц и именно в таком порядке. Если в какой-то момент компания не может выполнить своих финансовых обязательств, она немедленно объявляется **банкротом**. Ее имущество пропадает, и она выходит из игры (так что лучше иметь наготове запас наличных). Все денежные расчеты происходят между отдельными игроками и одним общим банком. Невозможна передача денег прямо от игрока к игроку, что исключает сговор, направленный против какой-либо компании. Банк, кроме того, контролирует источники сырья и скупает всю готовую продукцию.

1. Постоянные издержки. Каждый игрок (в порядке убывания старшинства, начиная со старшего) платит 300 долл. за каждую имеющуюся у него ЕСМ, 500 долл. за каждую наличную ЕГП, 1000 долл. за владение каждой обычной фабрикой и 1500 долл. — за владение автоматизированной. Это постоянные ежемесячные издержки каждого игрока, даже если он в этом круге не предпринимает никаких других действий.

2. Определение обстановки на рынке. Банк решает и сообщает игрокам, сколько ЕСМ продаст в этот раз и какова их *минимальная* цена. Объявляется также, сколько ЕГП в общей сложности будет закуплено и какова *максимальная* цена.

Таблица 4.1. Уровни цен на ЕСМ и ЕГП

Уровень	ЕСМ	Минимальная цена	ЕГП	Максимальная цена
1	1.0P	$800	3.OP	$6500
2	1.5P	650	2.5P	6000
3	2.0P	500	2.0P	5500
4	2.5P	400	1.5P	5000
5	3.OP	300	1.0P	4500

В табл. 4.1 приведены пять уровней предложения ЕСМ и спроса на ЕГП (обратите внимание, что с ростом одной из этих величин другая убывает), а также верхние и нижние границы цен для каждого случая. В число игроков P не включены те, кто обанкротился, и P может, таким образом, быть меньше N. Произведения 1.5P и 2.5P округляются до ближайшего целого *с недостатком*. В табл. 4.2 приведена матрица вероятностей перехода, в соответствии с которой банк определяет новый месячный уровень спроса и предложения, исходя из прежнего. Предполагается, что в нулевом месяце уровень равен 3.

Таблица 4.2. Вероятности перехода уровней цен друг в друга

Старый уровень	Новый уровень				
	1	2	3	4	5
1	1/3	1/3	1/6	1/12	1/12
2	1/4	1/3	1/4	1/12	1/12
3	1/12	1/4	1/3	1/4	1/12
4	1/12	1/12	1/4	1/3	1/4
5	1/12	1/12	1/6	1/3	1/3

3. Заявки на сырье и материалы. Каждый игрок тайно от других готовит заявку на ЕСМ на текущий месяц. В заявке указывается требуемое число ЕСМ и предлагается цена не ниже банковской минимальной (запрос нуля ЕСМ или предложение цены ниже минимальной автоматически исключает игрока из торгов в этом месяце). Все заявки раскрываются одновременно, и имеющиеся ЕСМ распределяются между игроками в порядке убывания предложенной цены. Если сырья не хватает на всех, заявки с предложением более низкой цены не удовлетворяются. При прочих равных условиях сырье достается самому старшему игроку. Игроки платят за сырье при его получении. Банк *не* сохраняет оставшееся после удовлетворения заявок сырье на следующий месяц.

4. Производство продукции. Все игроки по очереди (по убыванию старшинства, начиная со старшего) объявляют, сколько ЕСМ они собираются переработать в ЕГП в текущем месяце и на каких фабриках. Каждый игрок обязан тут же покрыть расходы на производство. Обычная фабрика может за месяц переработать одну

ЕСМ при затратах в 2000 долл. Автоматизированная фабрика может либо сделать то же, либо переработать 2 ЕСМ при затратах в 3000 долл. Конечно, чтобы переработать ЕСМ, их надо иметь.

5. Продажа продукции. При покупке банком у игроков ЕГП организуются примерно такие же торги, как и при продаже ЕСМ. Заявленные цены не должны превышать *максимальную цену*, установленную банком, причем банк покупает ЕГП в первую очередь у тех, кто заявил *более низкую* цену. При прочих равных условиях предпочтение отдается старшему игроку. Если предложение превышает спрос, наиболее дорогие ЕГП остаются непроданными. Игроки получают деньги за продукцию при ее продаже.

6. Выплата ссудного процента. Каждый игрок платит один процент от общей суммы непогашенных ссуд, в том числе и тех, которые будут погашены в текущем месяце.

7. Погашение ссуд. Каждый игрок, получивший ссуду сроком до текущего месяца, должен ее погасить. Поскольку возвращение ссуд предшествует получению новых, платить надо наличными.

8. Получение ссуд. Теперь каждый игрок может получить ссуду. Ссуды обеспечиваются имеющимися у игрока фабриками; под обычную фабрику дается ссуда 5000 долл., под автоматизированную — 10000 долл. Общая сумма непогашенных ссуд не может превышать половины гарантированного капитала, но в этих пределах можно свободно занимать. Банк немедленно выплачивает ссуду игроку. Срок погашения ссуды истекает через 12 месяцев — например, ссуду, взятую в 3-м месяце, возвращать надо в 15-м. Нельзя погашать ссуды раньше срока.

9. Заявки на строительство. Игроки могут строить новые фабрики. Обычная фабрика стоит 5000 долл. и начинает давать продукцию на 5-й месяц после начала строительства; автоматизированная фабрика стоит 10 000 долл и дает продукцию на 7-й месяц после начала строительства. Обычную фабрику можно автоматизировать за 7000 долл., реконструкция продолжается 9 месяцев, все это время фабрика может работать как обычная. Половину стоимости фабрики надо платить в начале строительства, вторую половину — за месяц до начала выпуска продукции в этой же фазе цикла. Общее число имеющихся и строящихся фабрик у каждого игрока не должно превышать шести.

Окончание игры и подсчет результатов

Игра оканчивается после некоторого фиксированного числа кругов (13 или более) или когда обанкротятся все игроки, кроме одного. Чтобы подсчитать общий капитал компании, надо сложить стоимость всех фабрик (по цене, по которой их можно было бы построить заново), стоимость имеющихся у нее ЕСМ (по минимальной банковской цене текущего месяца), стоимость имеющихся ЕГП (по максимальной банковской цене текущего месяца) и имеющиеся у компании наличные. После этого надо вычесть общую сумму ссуд и предстоящих расходов по уже начатому строительству. Если к концу игры приходит несколько игроков, результаты считаются по их капиталам.

Любой игрок в любой момент может узнать состояние дел любого другого игрока — его капитал, наличные, взятые ссуды, все, что касается готовой продукции, имеющихся и строящихся фабрик. Во время торгов игроки ничего не знают о заявках, сделанных другими, но, как только банк собрал все заявки, они обнародуются и количество купленных или проданных каждым игроком единиц становится известно всем. Игроки могут сами вести любые записи, но банк не предоставляет им никакой информации, кроме той, которая предусмотрена правилами игры.

Тема. Задача состоит из двух частей. Первая заключается в том, чтобы написать программу, которая управляет ходом моделирования — программу-банкир. Эта программа должна полностью контролировать игру: устанавливать цены, закупать продукцию и продавать сырье, проводить торги, вести учет и т. д. Эта программа должна в соответствующие моменты опрашивать игроков и добиваться соблюдения ими всех правил. В частности, банкир защищает от несанкционированного доступа всю информацию, как свою, так и чужую, касающуюся учета и состояния дел отдельных игроков. Программа-банкир периодически (например, ежемесячно) выдает сводный финансовый отчет. Поскольку отчеты эти предстоит читать людям, они должны быть понятны и приемлемы с эстетической точки зрения.
Вторая часть задачи — написать **программы поведения игроков**. Каждая программа-игрок должна быть в состоянии отвечать на любые запросы программы-банкира по ходу игры: она должна уметь предлагать цену на сырье, принимать решения, продавать готовую продукцию и т. п. Если для моделирования используется диалоговая система, реализуйте одну из программ-игроков таким образом, чтобы она просто передавала свои запросы игроку-человеку, находящемуся за терминалом. Такая программа должна уметь отвечать на запросы человека о состоянии игры.
После того как будет написано несколько программ-игроков, их надо объединить с программой-банкиром, чтобы получилась полная игровая система. Проведите с этой системой несколько игр и изучите результаты. Заметим, что несколько экземпляров одной и той же программы-игрока вполне могут выступать в качестве противников (т. е. мы считаем людей в соответствующей реальной игре изначально одинаковыми). Но для полной гарантии надо написать хотя бы две нетривиальные программы-игрока.

Указания исполнителю. Эта игра — пример **последовательного**, или **пошагового**, моделирования, при котором все события (кроме банкротств) происходят в строго определенном, заранее известном порядке. Цикл по месяцам — удобная структура для ведущей программы. Редко можно встретить задачу на программирование, прикладную или научную, столь удобную для хорошо структурированной реализации, как эта. Не премините воспользоваться такой возможностью.
В формулировке первой части задачи имеется некий подводный камень. Программа-банкир должна защищать всю существенную информацию от

несанкционированного доступа программы-игрока. Иначе говоря, программа-банкир должна сохранять в тайне все счета, как свои, так и игроков, обеспечивать секретность торгов и в то же время предоставлять игрокам информацию в ответ на их запросы. Один из моментов, который должен найти отражение в документации, — средства сохранности и их надежность.

Развитие темы. Дополнительное удовольствие от программирования игр — возможность поиграть с программой-игроком. Иногда при применении совсем простых эвристических методов может получиться удивительно сложное поведение. В программу, реализующую стратегию игрока, несложно включить элементы самообучения, чтобы ее поведение со временем совершенствовалось. Проведите несколько тренировочных турниров с участием как людей, так и программ (люди тоже обучаемы). Имеется стандартный прием обучения интеллектуальных программ новым стратегиям. Один экземпляр (Альфа) обучается в тренировочной серии игр, второй (Бета) остается на том же уровне знаний, какой он имел перед этой серией. Затем устраивается сравнительная серия игр между ними; если Альфа побеждает, его знания передаются всем копиям программ-игроков, в противном случае Альфа эти знания забывает как бесполезные и с ним проводится новая тренировочная серия.
Игру можно сделать интереснее, если добавить новые правила, которые, видимо, приблизят ее к реальной жизни. Ниже перечислены дополнительные правила. Если какое-либо из правил включается, все правила с меньшими номерами также надо включать.
1. Чрезвычайные ссуды. В случае финансовых затруднений каждый игрок может взять чрезвычайную ссуду. Такая ссуда стоит 2% в месяц вместо обычного 1% и предоставляется сроком на 4 месяца (ссудный процент выплачивается в обычном порядке). Общая сумма непогашенных задолженностей по-прежнему не может быть больше половины суммы, обеспечиваемой всеми фабриками данного игрока. Нельзя брать чрезвычайные ссуды для спасения от банкротства в момент, когда банк уже требует платежа по какому-либо обязательству. Брать чрезвычайную ссуду можно не позже начала цикла, в котором подходит срок платежа.
2. Чрезвычайные происшествия. В начале каждого цикла, непосредственно перед выплатой постоянных издержек, банк объявляет обо всех чрезвычайных происшествиях на этот месяц. На рис. 4.1 приведены вероятности различных чрезвычайных происшествий. Эффект приведенных ниже изменений в расценках накапливается на каждом шаге: так, 10%-ный рост с последующим 10%-ным спадом дает в результате 99% исходного уровня. К чрезвычайным происшествиям относятся:

```
.01 — У игрока i забастовка.
.01 — Игрок i пострадал от транспортного кризиса.
.02 — С игрока i причитается чрезвычайный налог.
.01 — На одной из фабрик игрока i авария.
.02 — Игрок i пожинает плоды внедрения новой техники.
.02 — У игрока i неожиданная удача.
.91 — У игрока i на этот раз ничего чрезвычайного не произошло.
```

Рисунок 4.1. Вероятности чрезвычайных происшествий. Испытываются на каждом игроке в каждом цикле.

Забастовка — игрок, у которого началась забастовка, может прекратить производство на 3 месяца, начиная с текущего, или вплоть до конца игры увеличить на 10% все издержки (как постоянные, так и производственные). Игрок, прекративший производство, может участвовать во всех прочих действиях и должен по-прежнему нести постоянные издержки.

Транспортный кризис — те, кого он затронул, не могут в этом месяце ни покупать сырье, ни продавать продукцию.

Чрезвычайный налог — те, кого это касается, должны немедленно сделать однократный взнос, исходя из расчета 500 долл. за фабрику. Запрещается для выплаты этой суммы прибегать к чрезвычайным ссудам. Возможно банкротство.

Авария — у игрока, которого это касается, одна из фабрик (по возможности обычная) в этом месяце не выпускает продукции.

Внедрение новой техники — у игроков, которых это касается, вплоть до конца игры на 10% снижаются издержки производства.

Неожиданная удача — соответствующий игрок может немедленно продать любое число из имеющихся у него ЕГП по 6500 долл.

3. Закрытие фабрики. В очередной месяц, как раз перед подачей заявки на строительство, игрок может закрыть все или некоторые свои фабрики. Начиная со следующего месяца постоянные издержки по такой фабрике сократятся вдвое, но продукции не будет совсем. Впоследствии закрытую фабрику можно открыть в той же точке очередного месячного цикла. Через два месяца после этого фабрика снова вступает в строй, и надо опять оплачивать издержки в полном размере. Например, вновь открытая в 13-м месяце фабрика вступает в строй действующих в 15-м.

4. Дробление заявок. На любых торгах любой игрок может сделать одну заявку, две или ни одной. Общий объем заявок одного игрока как на покупку, так и на продажу не должен превосходить предложений банка (для продажи — еще и имеющегося у данного игрока объема продукции). Банк рассматривает различные заявки одного игрока точно так же, как заявки разных игроков. Заявки эти конкурируют как друг с другом, так и с заявками других игроков. Удовлетворены могут быть обе, одна или ни одной. При прочих равных условиях по-прежнему побеждает старший игрок.

Тема 5: Эвристическое составление головоломки

Многие считают кроссворды слишком трудной головоломкой, потому что отгадать слово им не под силу. Но вписывать буквы в клетки нравится. Для подобных людей существует более простая головоломка — крисс-кросс.

Каждый крисс-кросс состоит из списка слов, разбитых для удобства на группы в соответствии с длиной и упорядоченных по алфавиту внутри каждой группы, а также из схемы, в которую нужно вписать слова. Схема подчиняется тому же правилу, что и в кроссворде, — в местах пересечения слова имеют общую букву, однако номера отсутствуют, поскольку слова известны заранее, требуется лишь вписать их в нужные места. Обычно в схемах крисс-кросса гораздо меньше пересечений по сравнению с кроссвордами, а незаполняемые клетки не заштриховываются, если это не приводит к путанице. Крисс-кросс всегда имеет единственное решение, в котором используются все перечисленные слова. Пример головоломки, правда очень маленький, приведен на рис. 5.1. Заметьте, что длина слова служит важным ключом к разгадке.

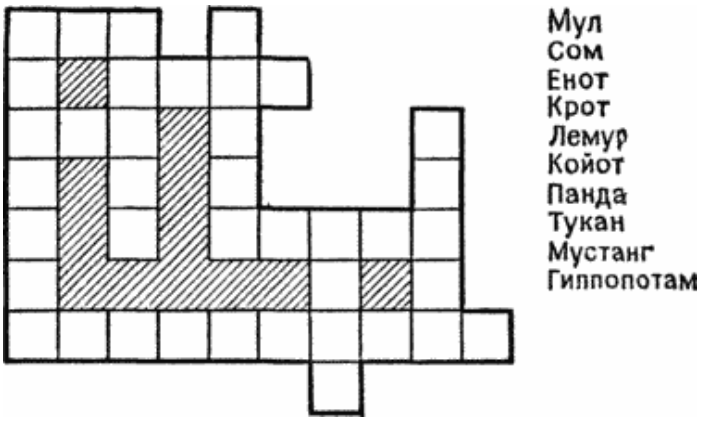


Рисунок 5.1. Пример головоломки крисс-кросс.

Тема. Напишите программу, читающую список слов и строящую для этого списка правильную схему крисс-кросса. Представьте заполненную схему как доказательство того, что она правильная. Возможно, хотя и маловероятно, что для данного списка слов не существует решения (как и в кроссворде, схема должна быть связной). Ваша программа должна сообщать о всех неудачах при построении схемы и о всех ситуациях, нарушающих однозначность (таких, например, как наличие повторяющихся слов). Попутно решите еще одну задачу — получите красивый графический вывод.

Указания исполнителю. Качество схем крисс-кросса пропорционально их «связанности», т. е., чем теснее в среднем слова переплетены с соседями, тем интереснее головоломка. Связанность можно измерять по-разному: как отношение площади схемы к площади наименьшего объемлющего прямоугольника; как среднее число пересечений на слово; как среднее число пересечений на букву; как минимальное число пересечений на слово. При генерации головоломок крисс-кросс для массовых изданий использовалась коммерческая программа, но

головоломки получались неинтересные — слишком длинные и извилистые. Когда ваша программа заработает, позаботьтесь об увеличении связанности.

Предложенная задача — классическая для метода перебора с возвратами. Начните с вписывания слов в фиксированную схему, пока в списке есть подходящие слова. Когда они кончатся, вернитесь на шаг назад, удалив последнее вписанное слово, и попытайтесь вписать другое слово. Необходимо разработать эвристику для выбора очередного кандидата из списка неиспользованных слов. Контроль однозначности должен включать проверку того, что в схеме нельзя поменять местами никакие два слова равной длины. Достаточна ли такая проверка? Нет ли более изящной? Полное алгоритмическое решение, максимизирующее связанность, несомненно, представит значительный теоретический интерес.

Тема 6: Автоматическое построение лабиринтов

Тезей должен был найти выход из Критского лабиринта или погибнуть от руки Минотавра. Но что поразительно: найти вход в лабиринт — задача не менее трудная.

Здесь не представляется возможным описать все мыслимые лабиринты, да это и не требуется. Мы займемся простыми лабиринтами, построенными на прямоугольнике m×n, где m, n — положительные целые числа. Внутри и на границах прямоугольника поставлены стенки по ребрам покрывающей его единичной квадратной сетки. Чтобы построить из прямоугольника лабиринт, выбьем одну единичную стенку на одной из сторон прямоугольника (получится вход в лабиринт); выбьем одну единичную стенку на противоположной стороне (получится выход) и еще удалим какое-то число строго внутренних стенок. Говорят, что лабиринт имеет решение, если между входом и выходом внутри лабиринта есть путь в виде ломаной, не имеющей общих точек со стенками. Решение единственно, если любые два таких пути проходят через одни и те же внутренние ячейки сетки. На рис. 6.1 приведен пример лабиринта 6×6.

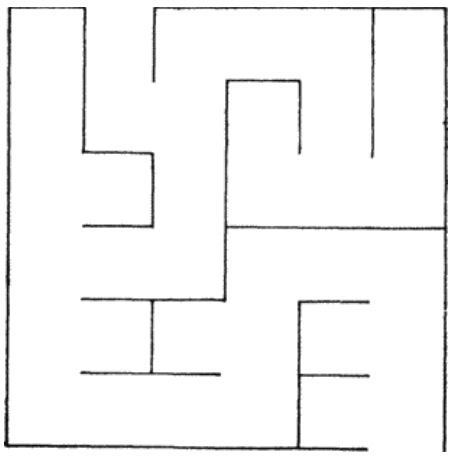


Рисунок 6.1. Пример лабиринта.

Тема. Напишите программу, которая по исходным данным m и n строит прямоугольный лабиринт m×n (проверьте, допустимы ли заданные тип). Предусмотрите, чтобы программа при каждом обращении к ней порождала разные лабиринты. Лабиринт должен иметь единственное решение, и, чтобы получившийся лабиринт был интересным, все ячейки должны быть соединены с основным путем, дающим решение.

Указания исполнителю. Теоретически нельзя удовлетворить требованию, чтобы любые два лабиринта (даже при одинаковых m и n) были различны, поскольку существует лишь конечное число лабиринтов любого наперед заданного размера, а программу можно вызвать большее число раз. Однако число лабиринтов какого-нибудь размера очень велико, и поэтому вероятность повторения лабиринта можно сделать очень маленькой. Практически это достигается, если программа будет производить «случайный» выбор различных вариантов, опираясь на какое-либо доступное ей, но неуправляемое значение (обычно берут дату и время вызова

программы). Варианты, между которыми выбирает программа, это, например, положение входа и выхода и положение хотя бы нескольких внутренних разрушаемых стенок. При отладке разумно будет отключить механизм случайного выбора, чтобы изменения результата работы вызывались только изменениями самой программы.

Один из возможных подходов к решению таков. Выбираем вход; затем, начав от него, добавляем по одной ячейке к главному пути-решению, пока он не достигнет выходной стороны. После этого удаляем некоторые внутренние стенки так, чтобы все клетки оказались соединенными с главным путем. Чтобы главный путь не получился прямым коридором, следует при его построении предусмотреть случайные повороты. Программа должна также следить за тем, чтобы при построении главного пути или при открытии боковых ячеек не нарушалась единственность решения. Наблюдательный читатель заметит, что определение единственности решения не годится в случае, когда путь заходит в боковой тупик и затем возвращается. Вы можете попробовать разработать в том же духе формально корректное определение.

Тема 7: Программа, печатающая собственный исходный текст

В философии интроспекция (или самонаблюдение) считается одним из важных элементов мышления. Все здравомыслящие люди должны внимательно отнестись к названию темы. Если человек может достичь самопознания, то почему этого не может сделать программа? Ну а чтобы познать себя, лучше всего написать автобиографию.

Тема. Напишите программу, печатающую копию собственного исходного текста. Вывод не должен содержать «управляющих» карт или другой информации, зависящей от системы. Печатается только то, что перфорируется для компилятора. Однако ваша программа ничего не должна вводить; ей не следует опираться на системные «штучки», например на знание того, что конкретный компилятор оставляет копию исходной программы в непомеченном COMMON-блоке. Проследите, чтобы программа давала одинаковый результат независимо от места и времени выполнения.

Указания исполнителю. Не поддавайтесь отчаянию и страху, даже если тринадцатая попытка оказалась неудачной! Подобные программы называются **интроспективными**, и существует теорема, в которой утверждается, что интроспективную программу можно написать на любом «достаточно мощном» языке. Все обычные языки программирования — достаточно мощные. Для решения требуется лишь взглянуть на язык под соответствующим углом зрения. Возможно придется сделать свой собственный редактор.

Тема 8: Расчет дохода от вложенного капитала

Самым разным людям — финансистам, биржевым дельцам, банкирам и даже обыкновенным труженикам, вроде казначея пенсионного фонда Тимстеров, — хотелось бы знать, какой доход принесут им вложенные средства. Если деньги лежат на срочном вкладе, то особых сложностей не возникает, ибо банки в каждом рекламном проспекте трубят о своих процентах. Даже если ваши средства вложены в облигации, по которым не только выплачиваются проценты, но которые можно впоследствии еще и с выгодой продать, то, чтобы определить свой доход, достаточно взять разницу курсовой стоимости облигаций, прибавить проценты, и вы получите сумму, которую должен выплатить банк. Результатом этих вычислений, если их выразить в процентах годовых, приносящих при условии непрерывного их начисления известную прибыль, является инвестиционный **доход**.

Ситуация, однако, не столь проста, если инвестиции связаны, скажем, с инвестиционным фондом, счетом капитала или небольшим собственным делом, когда имеют место нерегулярные поступления и платежи и текущие показатели меняются изо дня в день. Хорошим, в этом смысле, примером служит инвестиционный фонд. Действительно, новые акции могут приобретаться по рыночной стоимости в любой момент, а купленные ранее акции точно так же могут сбываться; в процессе функционирования фонда дивиденды все время меняются (и даже исчезают), однако, как правило, вкладываются в дополнительные акции; и наконец, стоимость акций фонда ежедневно меняется по мере того, как меняется курс лежащих в его основе ценных бумаг. Было бы, конечно, здорово сравнить доход, получаемый со срочного вклада, с той радужной перспективой, которую обещают проспекты инвестиционных фондов, не забывая, само собой, о том, что обычно доход пропорционален риску.

К счастью, для таких случаев имеется формула расчета дохода. Формула, к сожалению, не в замкнутом виде, а итерационная.

Предположим, что A — **текущая величина инвестиций**, что существует m операций с капиталом, причем i-я операция производилась на сумму P_i (отрицательные значения указывают на изъятие капитала) и имела место T_i лет назад, и пусть первоначальная оценка ожидаемого дохода Y_0 полагается равной нулю. Итак, определим при $j > 0$ величины

$$C_j = A - \sum_{i=1}^{m} P_i \exp(Y_{j-1} T_i)$$

и

$$D_j = \sum_{i=1}^{m} T_i P_i \exp(Y_{j-1} T_i).$$

Тогда наилучшая оценка дохода Yj дается формулой

$Y_j = Y_{j-1} + C_j/D_j$

Как только разность

$|Y_j - Y_{j-1}|$

станет достаточно малой, величина дохода считается найденной.

Таблица 8.1. Запись реальных инвестиций. Даты представлены в виде: месяц/число/год.

Дата	Величина инвестиции	Сумма операции
3/11/71	$0.00	$68.26
5/ 4/71	73.75	50.00
6/ 4/71	114.82	75.00
8/ 9/71	170.66	50.00
9/ 7/71	229.41	54.00
10/ 4/71	282.97	50.00
10/ 8/71	326.02	4.31
12/ 8/71	328.11	50.00
1/ 1/72	391.65	0.00
21 4/72	413.42	50.00
8/ 1/72	471.35	0.00
10/ 5/72	440.83	7.72
10/ 5/72	448.55	4.14
10/ 2/73	398.36	4.80
1/ 2/74	330.74	0.00
6/ 7/74	360.97	-200.00
10/ 3/74	180.42	50.00
3/13/75	253.96	-200.00

При изучении табл. 8.1 обратите внимание, что величина A получается суммированием среднего и правого столбцов таблицы. Например, для третьей строки A = 189.82 долл., P_1 = 68.26 долл., P_2 = 50.00 долл., P_3= 75.00 долл., а $T_1 \approx$ 85/365, $T_2 \approx$ 31/365, T_3 = 0. Заметим также, что для каждой строки таблицы оценка Y_0 считается равной нулю и что расчет дохода для любой текущей даты не зависит от величины доходов в предшествующие времена.

Тема. Напишите программу вычисления дохода от вложенного капитала. Исходные данные представляют собой записи о проведенных операциях, в каждой из которых указываются дата, сумма операции и величина инвестиции на день проведения операции *без учета* последней. Предполагается, что информация упорядочена по времени. Программа должна проверить, не нарушен ли хронологический порядок следования данных и нет ли где-нибудь изъятия средств, превышающего текущий счет. Программа должна отпечатать аккуратную таблицу платежных операций. При этом для каждой операции в выводимой строке должны быть указаны дата ее проведения, сумма инвестиций до операции, объем операции, сумма инвестиций после операции, доход на день проведения и сумма всех поступлений и платежей на текущий день. Каким именно образом обозначить конец вводимой информации — решать самому программисту, а вот равенство нулю суммы операции является удобным способом выяснения величины текущего дохода. Если у вас нет собственных инвестиций и вы не можете раздобыть Wall Street Journal, тогда исходными данными для программы, быть может не вполне удачными, зато реальными, может служить табл. 8.1.

Указания исполнителю. В рассматриваемой задаче существует интересный побочный вопрос. Даты проведения банковских операций задаются в обычном виде: месяц/число/год. А для решения задачи требуется иметь отрезки времени Ti, прошедшие после операции, выраженные в годах. У банкиров и юристов имеется несколько способов определения момента времени, когда прекращается начисление процентов на деньги (подозревают, что метод расчета зависит от того, кто кому должен). В программе достаточно вычислять годы в виде вещественных чисел с учетом високосных лет, предполагая, что все даты лежат в диапазоне от 1900 до 1999 включительно. Вообще говоря, перевод дат из одного календаря в другой может оказаться отнюдь не простым делом.

Тема 9: Избыточность текста и сжатие файла

Все знают, что большинству людей свойственно излишнее многословие. Гораздо менее широко известно, что даже самые лаконичные высказывания можно было бы значительно сократить. Вообще, естественные языки отличаются чрезвычайной избыточностью. Даж есл нсклко бкв вбрсть, эт прдлжн ещ мжн прчть. Языки, используемые для вычислений, обладают той же особенностью. Для экономии памяти компьютера, объем которой ограничен, имеет смысл ликвидировать избыточность текста.

Существует несколько способов уплотнения текста. Самый очевидный из них — поиск различных по длине цепочек из одной повторяющейся литеры. Такая группа может быть заменена тройкой литер mcn, где m обозначает признак повторения, специальную литеру, не используемую нигде в тексте для других целей, c — сама повторяющаяся литера и n — длина цепочки. Один такой **триграф** экономит n — 3 литер, причем значение n не может превышать максимального числа, представимого в поле одной литеры. Описанный способ обработки весьма неплохо оправдывает себя для текстов, содержащих длинные цепочки повторяющихся литер, например длинные цепочки пробелов, характерных для большинства программ. К сожалению, этот прием не столь хорош для других текстов, поскольку большинство данных не отличается такой же строгой формой записи, как программы.

Второй способ основан на том, что в различных системах кодировки литер, применяемых на ЭВМ, большинство литер практически не используется (из 256 литер обычного 8-разрядного кода, как правило, употребляется лишь около 100). Сначала в тексте отыскиваются наиболее распространенные **диграфы**, и каждому из них ставится в соответствие одна из не используемых в тексте одиночных литер. Уплотнение текста производится при просмотре его слева направо путем последовательной замены выявленных диграфов их однолитерными эквивалентами. При этом может быть достигнута значительная экономия, поскольку, например, 150 наиболее часто встречающихся диграфов уже составляют большу́ю долю текста на естественном языке. И если не ставить целью слишком высокую степень уплотнения текста, можно написать довольно эффективные программы кодирования и декодирования, работающие с машинным представлением литер.

Однако существуют все же определенные трудности. Кто сказал, что наиболее часто встречающиеся диграфы в английском тексте должны быть теми же, что и во французском, или в наборе файлов, содержащих почтовые адреса, или в тексте на Алголе? А если даже это и так, то как насчет триграфов, квадриграфов или более длинных групп? Ведь более длинные группы, даже если они и реже встречаются, дают большую экономию, а бывает, что определенный фрагмент появляется в большом куске текста намного чаще, чем можно было бы ожидать. И, возвращаясь назад, как подсчитать частоты появления диграфов?

Ответ на все эти вопросы содержится в третьем подходе к решению исходной задачи. Вместо того чтобы употреблять некоторый, заранее заданный набор кодировок, можно на ходу генерировать кодовый словарь, используя

непосредственно текст, подлежащий сжатию, или выборку из него. Поскольку при этом каждый элемент текста будет участвовать в создании своего собственного словаря, исчезнут трудности, вызванные неудачными аббревиатурами. Теперь нам надо найти способ построения такого словаря.

Опишем наш план действий в общих чертах. Начинаем с пустого словаря. Текст просматриваем слева направо. Ищем в словаре гнездо возможно большей длины, совпадающее с головной частью текста, и увеличиваем счетчик частоты соответствующего гнезда словаря. Если совпадений цепочек нет, образуем новое гнездо словаря и помещаем туда первую букву текста. Вычеркиваем обработанную цепочку из начала текста и начинаем просмотр заново. При обстоятельствах, поясняемых ниже, иногда два гнезда словаря соединяются в одно, образуя цепочку большей длины — процесс укрупнения гнезд. Когда словарь переполняется, производим его чистку, удаляя наиболее редко встречающиеся гнезда, и продолжаем просмотр. После того как частоты встречаемости гнезд словаря стабилизируются, вводим таблицу кодировок и, взяв исходный текст, полностью его кодируем.

В предложенной схеме есть два невыясненных момента: каким образом происходит укрупнение гнезд словаря и как осуществляется его чистка? Укрупнение двух гнезд словаря производится в случае, когда одно из них следует в тексте непосредственно за другим и частоты обоих гнезд превышают некоторое пороговое значение. При этом, чтобы новое гнездо словаря не подвергалось ближайшей чистке, ему может быть приписана начальная частота несколько выше обычной. Таким образом, если в словаре уже имеются, например, цепочки КОН и ТАКТ, то при условии, что содержимое их счетчиков достаточно велико, может образоваться новое гнездо словаря, содержащее цепочку КОНТАКТ. Что лее касается чистки словаря, то существует простой способ — удалять все те гнезда, значения счетчиков которых меньше среднего. Можно действовать и иначе — выбрасывать все гнезда, частота которых ниже медианы частот. Годятся и другие, подобные этому способы.

Алгоритм построения словаря

В приводимом алгоритме предполагается, что построение словаря производится с помощью некоторой выборки из текста, подлежащего сжатию. Для алгоритма существенны все литеры текста, и если табуляция, концы строк и другие аналогичные элементы имеют значение, то в тексте должны присутствовать соответствующие управляющие литеры. Предполагается, что в начале работы словарь пуст. В начальный момент переменная ***last match*** содержит пустую цепочку, а переменная ***last count*** имеет значение, равное нулю.

1. Ищем в головной части входного текста возможно более длинную цепочку ***match***, совпадающую с каким-нибудь гнездом словаря. Если переменная ***match*** пустая, засылаем в нее первую литеру входного текста, помещаем в свободное гнездо словаря и устанавливаем начальное значение счетчика этого нового гнезда равным единице. Если цепочка ***match*** не пустая,

увеличиваем на единицу счетчик соответствующего гнезда словаря. Содержимое счетчика этого гнезда записываем в ***count***.
2. Если либо ***count***, либо ***last count*** меньше значения порога укрупнения гнезд, то переходим к шагу 4. Порог укрупнения определяется как отношение максимально допустимого объема словаря к числу оставшихся в данный момент свободных гнезд.
3. Образуем новое гнездо словаря путем объединения цепочек ***last match*** и ***match***. Поскольку данное гнездо словаря возникло впервые, засылаем в его счетчик единицу. Можно применить и другие стратегии.
4. Если в словаре остались свободными менее двух гнезд, производим чистку, удаляя все гнезда с частотами меньше медианы частот. При этом, если окажется, что исключилось гнездо, содержащее ***match***, устанавливаем ***count*** равным нулю.
5. Вычеркиваем ***match*** из начала входного текста. Если текст исчерпан, то алгоритм работу заканчивает — выход. В противном случае помещаем ***last match*** в ***match***, пересылаем ***last count*** в ***count*** и возвращаемся к шагу 1.

Кодирование и декодирование
Как только построение словаря завершилось, необходимо составить таблицы для кодирования и декодирования. Образуем все возможные диграфы, начинающиеся с литеры, которая нигде в тексте не используется. Исключим из словаря все гнезда, состоящие из одной или двух литер (их уплотнение экономии дать не может). Упорядочим оставшиеся цепочки по частоте встречаемости. Поставим в соответствие гнездам словаря полученные выше кодирующее диграфы, начиная с гнезд, имеющих наибольшую частоту. Формирование таблицы кодировок завершается по исчерпании гнезд словаря или набора диграфов.
Процесс кодирования текста подобен процедуре построения словаря. На каждом этапе головная часть входного текста проверяется на совпадение в возможно большем числе позиций с гнездами словаря. Совпавшая цепочка заменяется в тексте соответствующим кодирующим диграфом, и начало просмотра входного текста сдвигается на длину выделенной цепочки. Если же в словаре не найдено нужного гнезда, в выходной текст просто переносится первая литера из головной части входного текста и начало просмотра перемещается вправо на одну позицию. Декодирование осуществляется путем простой замены кодирующих диграфов их эквивалентами из словаря.

Тема. Напишите программу, реализующую описанные выше алгоритмы построения словаря, кодирования и декодирования. Проверьте программу на достаточно больших фрагментах текста на естественном языке и языке программирования. Коэффициент сжатия данного куска текста определяется как частное от деления суммы длин сжатого текста и словаря на дайну исходного текста. Проведите небольшое исследование зависимости коэффициента сжатия от какого-нибудь из следующих параметров: языка уплотняемого текста; объема используемой для упражнения выборки из текста; длины словаря при его построении; имеющегося количества кодирующих диграфов или применимости

словаря, полученного на основании одного текста, для другого текста на том же языке.

Указания исполнителю. Данная задача интересна тем, что для ее эффективного решения требуется употребить некоторые весьма развитые алгоритмы и структуры данных. Однако пусть не столь эффективную, но правильно работающую программу можно написать, используя простые алгоритмы и структуры, которые можно, когда программа заработает, постепенно заменять более изящными конструкциями. Одним из примеров служит вычисление медианы для чистки словаря. В качестве первого варианта можно просто выбрасывать гнезда словаря с частотами, меньшими средней. При этом среднюю частоту легко вычислить за один полный просмотр всех частот словаря. А после того как такая программа в целом заработает, можно уже для нахождения порога исключения строк подключить болте сложную программу расчета медианы.
Другим примером является выбор структуры словаря на этапах его создания и кодирования. Если гнезда словаря расположить в случайном порядке, то при проверках на совпадение необходимо проходить весь словарь. Однако при такой структуре появляющиеся новые гнезда добавляются просто в конец словаря. Небольшое усложнение могло бы заключаться в группировке гнезд словаря по их длинам. Тогда поиск мог бы осуществляться в направлении от самых длинных групп к коротким и прекращаться при первом же удачном сравнении. Если каждую группу еще и лексикографически упорядочить, то можно было бы воспользоваться вместо линейного поиска внутри группы двоичным поискам, экономя таким образом время. Но зато добавление в словарь новых гнезд становится в этом случае более сложным, так как для любого нового гнезда потребуется место, скорее всего, где-то в середине группы. Не исключено, что самой выгодной структурой для организации поиска окажется какая-либо разновидность дерева. Разыскиваемую цепочку словаря могла бы тогда составить последовательность букв по пути от корня дерева к его листьям, или, иначе говоря, в узлах некоего подобия двоичного дерева поиска могли бы располагаться соответствующие строки словаря. В то же время при составлении словаря деревья потребуют намного большей обработки, нежели описанные выше более простые структуры.

Развитие темы. В описанной модели имеются три области свободы: критерий укрупнения гнезд, критерий исключения низкочастотных гнезд словаря и система их кодирования. Рассматривая их по-порядку, начнем с критерия укрупнения гнезд. Для того чтобы могло произойти укрупнение гнезд, в нашем алгоритме требуется, чтобы частоты встречаемости каждого из двух последовательных гнезд превысили один и тот же крайний предел.
Можно, однако, для каждого гнезда иметь свой порог. В другом варианте может быть у одного гнезда постоянный порог, а у другого — порог, являющийся функцией средней частоты гнезд. Аналогично может варьироваться начальная частота укрупненного гнезда, причем при любом способе начальная частота

задается большой исходя из условия повышения шансов на сохранение данного гнезда при чистке.

Точно так же может быть видоизменен образ действий при исключении гнезд словаря во время его чистки. Можно выбрасывать неизменную часть низкочастотных гнезд (используя медиану, устанавливающую эту часть равной половине). Можно исключать все гнезда с частотами, меньшими некоторой, кратной средней частоте. Или же можно вычеркивать все гнезда с частотами, меньшими заданной, и эту процедуру осуществлять до тех пор, пока словарь не будет достаточно вычищен. Сочетание различных способов укрупнения и чистки гнезд характеризуется особым показателем исключаемости. В некоторых вариантах оставляются цепочки, которые часто встречаются в одной части текста и реже в других; в иных случаях предпочтение отдается цепочкам, равномерно разбросанным по тексту. Какому показателю исключаемости отдать предпочтение, зависит от используемых особенностей как словаря, так и текста.

В алгоритме кодирования употребляются диграфы, начинающиеся с не используемых во входном тексте литер. Однако, если набор диграфов кончился, а словарь еще не доделан, можно использовать триграфы и т. д. Коль скоро частоты гнезд словаря известны, их можно употребить для организации **взвешенного кодирования переменной длины**. Этот способ будет дороже при декодировании (почему не при кодировании?), зато обеспечит даже более высокую степень сжатия текста.

Тема 10: Домашняя бухгалтерия

Кооперативы — довольно характерное явление в студенческой жизни. Иногда несколько студентов просто вместе платят за квартиру; порой они связаны друг с другом тесными и официальными общинными узами. Однако в любом случае им нужно вести и оплачивать счета. Немало общин распалось из-за денег, и, хотя более глубоких проблем ЭВМ решить не могут, честно вести расчеты они в состоянии.

Как правило, счета присылают в конце месяца, как раз после самой крупной траты — внесения платы за квартиру. В течение месяца каждый член группы платит за все из своего кармана. Пошел в магазин — плати за продукты, открыл дверь — плати разносчику газет, сел за руль — плати за бензин. При удачном стечении обстоятельств большинство членов группы заплатит примерно свою долю, но уж, конечно, точного соответствия не получится никогда.

Если расходы распределяются не поровну, расчет не сводится к простому делению. Обычно кто-нибудь не прочь платить побольше, но иметь еще одну комнату; тот, кто выходные проводит у родителей, платит за еду несколько меньше других и т. п. И разумеется, каждый может потратить деньги по своему усмотрению, например на междугородный телефонный разговор или пиво, что не будет фиксироваться в ежемесячном групповом расчете. Чтобы учесть отмеченные нами и подобные им обстоятельства, нужна устоявшаяся бухгалтерская система.

Тема. Напишите программу, обеспечивающую небольшую общину постатейно расписанными счетами. Исходные данные подразделяются на четыре части. Первая часть должна содержать фамилии тех, кто участвует в расходах в текущем месяце. Во второй части перечисляются основные статьи расходов, такие, как питание, квартплата, коммунальные услуги. За каждой статьей должен следовать список членов общины и их доли в общих расходах. Доля может выражаться как в долларах, так и в процентах. Если вся статья распределена явным образом, то остаток делится поровну между остальными членами. Например, если квартплата составляет 200 долл., студент А взялся платить 45 долл., а В — 35%, то на всех остальных членов общины приходятся равные доли от 85 долл.

Элементами третьей части исходных данных должны быть записи об общественно полезных расходах. Запись содержит дату, фамилию члена группы, уплаченную сумму, статью расхода и краткое описание. Четвертая часть содержит сходную информацию, но о расходах на личные нужды. Каждая запись в этой части имеет ту же структуру, что и в части 3, с очевидным дополнением — указывается фамилия человека, на нужды которого истрачены деньги. Исходные данные необходимо проверить на непротиворечивость, обращая особое внимание на даты, размеры платежей, фамилии и статьи расходов.

Выходная информация также должна подразделяться на несколько частей. Во-первых, каждому члену группы нужно предоставить хронологический список всех платежей и приходов в данном месяце. Во-вторых, каждый должен получить такой же список, упорядоченный по статьям и датам. В этом списке необходимо указать долговые обязательства каждого члена по каждой статье и их разложение на пай и

приход. Наконец, все должны узнать свое финансовое положение на конец месяца. Должники пусть знают, кому платить, а те, кому задолжали, пусть знают, с кого требовать деньги. Желательно, чтобы программа по возможности минимизировала число таких балансовых действий.
Заключительная часть вывода должна включать хронологический перечень всех расходов на общественные нужды и таблицу (люди/статьи), в которой приведены расходы, приходы, паи и сбалансированные долговые обязательства. Перекрестное суммирование таблицы позволит оценить точность бухгалтерии.

Указания исполнителю. Ничего особо сложного в предложенной задаче нет. Конечно, эффективная программа всегда лучше неэффективной, но в данном случае время счета мало по сравнению с временем ввода/вывода. Основного внимания требуют разнообразный формат исходных данных к элегантная организация проверки данных на непротиворечивость. А в общем это прозаическая программа, как и большинство производственных программ. Дайте «профессиональное» решение.

Тема 11: Моделирование машины Тьюринга

Задолго до появления первых универсальных цифровых вычислительных машин вопрос об ограничениях на вычисления, которые могли бы выполнять машины, заинтересовал Алана Тьюринга. Чтобы быть уверенным, что мощь его гипотетической машины не обусловлена каким-либо хитрым механизмом, Тьюринг исключил почти все возможности, которые существенны для реальных компьютеров. Осталась лишь программная память простого вида, не допускающая изменений во время выполнения, только один тип команд и простая лента для ввода и вывода. Тем не менее это устройство — **машина Тьюринга**, предмет обожания студентов-логиков в последние 40 лет — способно повторить все вычисления любого современного цифрового компьютера. Но какой мерзкой была бы задача промоделировать, скажем, IBM 370/155 на машине Тьюринга! К счастью, перед нами стоит куда более приятная обратная задача.

Машина Тьюринга состоит из **устройства управления**, которое с помощью **головки** связано с **лентой ввода/вывода**. Лента — это длинная полоска, разделенная на ячейки, каждая из которых может содержать одну **литеру**; лента простирается вправо до бесконечности (иными словами, на правом конце ленты находится небольшая фабрика, производящая по мере необходимости дополнительную ленту). Головка указывает на какую-то одну ячейку ленты и может читать содержимое ячейки, записывать и перемещаться вправо или влево. В начале работы исходные данные всегда заполняют левую часть ленты, а головка читает самую левую ячейку ленты. Когда головка, двигаясь вправо, достигает ячейки, которая *не* является частью исходных данных и никогда ранее не обозревалась головкой, считается, что в этой ячейке записан пробел, обозначаемый ø.

Устройство управления выполняет программу, подчиняясь строгим правилам. В любой момент времени устройство управления находится в некотором **состоянии**, которое записано в регистре **текущее состояние**. Состояния обозначаются положительными целыми числами. Каждая команда программы представляет собой **пятерку**, составленную из состояния, литеры, еще одного состояния, еще одной литеры и направления движения ленты. Цикл выполнения команды начинается с того, что устройство управления сравнивает текущее состояние и литеру на ленте под головкой с первыми двумя компонентами всех команд. По правилам программирования для машины Тьюринга в программе может быть не более одной пятерки с какой-либо определенной начальной парой состояние-литера (но может и не быть ни одной). Когда совпадение найдено, устройство управления выполняет три действия: в ячейку ленты под головкой записывается литера, являющаяся четвертой компонентой пятерки; головка передвигается на одну ячейку влево или вправо или остается на месте, как указано в пятой компоненте пятерки; текущее состояние заменяется на третью компоненту. После этого машина готова к следующему циклу. По соглашению, работа начинается в состоянии 1 при описанном выше состоянии ленты. Машина останавливается, если в цикле выполнения не удается найти совпадения с текущей парой состояние-литера или если головка выходит за левый край ленты; при этом результатом

работы считается все, что остается на ленте после остановки. Отметим, что программа может содержать лишь конечное число команд, так что для любой программы осмысленно только конечное число состояний и литер.

Для пояснения изложения полезно привести пример. Пусть мы хотим написать программу для машины Тьюринга, которая будет строить сумму двух целых чисел. Целое число n будет изображаться на ленте n последовательными значками * (отсутствие звездочек соответствует нулю), два исходных числа будут разделены запятой, и если исходные данные представляют n + m, то результатом должны быть n + m звездочек, расположенных у левого края ленты. Так, чтобы вычислить 7 + 4, следует записать в качестве исходных данных

*******,****

результатом должно быть

****

Структура программы проста. Сначала головка движется вправо в поисках запятой (не забывайте, что первоначально головка стоит над крайней левой ячейкой исходных данных). Запятая заменяется звездочкой, и головка продолжает движение, отыскивая пробел, ограничивающий справа исходные данные. Головка возвращается на одну ячейку назад и записывает пробел на место находящейся там звездочки, после чего программа завершается. Легко видеть, что при построении суммы одна звездочка добавляется в середине и одна убирается с правого края. Программа приведена в табл. 11.1.

Таблица 11.1. Программа для машины Тьюринга

Старое состояние	Старая литера	Новое состояние	Новая литера	Перемещение
1	*	1	*	Вправо
1	,	2	*	»
1	ø	4	ø	На месте
2	*	2	*	Вправо
2	,	4	ø	На месте
2	ø	3	ø	Влево
3	*	3	ø	На месте
4	ø	4	ø	» »

Чтобы изобразить состояние машины Тьюринга, можно напечатать все ячейки ленты, которые когда-либо рассматривались, и среди них — текущее состояние непосредственно слева от ячейки, находящейся под головкой в данный момент; такой способ мы будем считать стандартным. Мы получаем **моментальный снимок**; следующий пример показывает начало сложения 2 и 3:

1**,***

На рис. 11.1 показана последовательность моментальных снимков для всего вычисления. Отметим, что программа останавливается в состоянии 3, поскольку в ней не предусмотрены действия для пробела. Состояние 4 возникает только, если в исходных данных имеется ошибка; в этом случае машина попадает в бесконечный цикл. Убедитесь, что программа работает, если любое из исходных чисел (или оба) равно нулю.

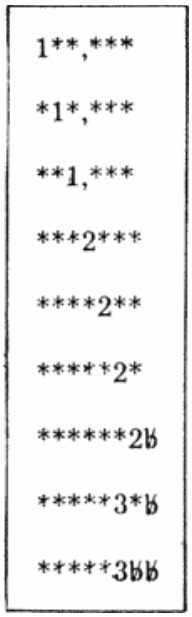


Рисунок 11.1. Последовательность моментальных снимков.

Наш пример программы может показаться слишком простым. Попробуйте изменить программу, чтобы она выполняла не сложение, а умножение. Для машины Тьюринга единичная система счисления более естественна, чем любая другая; программа сложения десятичных чисел будет длиннее и сложнее. В литературе, указанной в библиографии, можно найти гораздо более подробный материал о машине Тьюринга и обоснование того, что машина Тьюринга может выполнить любое вычисление, выполнимое на какой-либо другой машине. Вы обнаружите небольшие отличия в разных описаниях машины Тьюринга и там же — доказательства того, что эти отличия ни на что не влияют.

Тема. Напишите универсальный имитатор машины Тьюринга. Входными данными будут: программа для машины Тьюринга, ее исходные данные и (по причине, которая будет объяснена позже) начальное состояние машины. Результатом должны быть трассировка работы машины и ее окончательное состояние. Поскольку машина Тьюринга вовсе не всегда останавливается, причем заранее нельзя предсказать, остановится она или нет (если вы не понимаете, почему это так, обратите внимание на **проблему остановки**), необходимо как-то контролировать объем печати имитатора и расходуемое им время. Проверьте имитатор на нескольких программах, аналогичных рассмотренным выше.
Хотя в нашем описании именами состояний были положительные целые числа, ваш имитатор должен допускать в качестве имени состояния произвольный идентификатор. В предыдущем примере мы могли бы назвать состояния *Начало*, *ДвижениеВправо*, *Конец* и *Ошибка*, тогда одна из команд могла бы иметь вид
ДвижениеВправо ø Конец ø Влево
Теперь у нас нет выделенного первого состояния, поэтому его должен указать пользователь.

Указания исполнителю. При разработке данной программы, как и любого имитатора, перед нами стоит проблема эффективности. Если на протяжении всего выполнения используются имена состояний, то постоянно требуемый поиск займет значительное время. Скорость выполнения будет наибольшей, если представить программу для машины Тьюринга в виде двумерного массива, индексами в котором будут состояния и литеры. Элементы массива содержат команду, которую нужно выполнить; элемент специального вида указывает, что соответствующая пятерка отсутствует. Тут, конечно, определенную трудность представляет незнание необходимого размера этого массива до того, как будут прочитаны исходные данные. Отметим также, что необходимо проверить непротиворечивость исходных данных, т. е, убедиться, что никакие две пятерки не начинаются одной и той же парой состояние-литера.

Трассировочная информация должна печататься после каждого изменения состояния. Она должна включать в себя: содержимое всей ленты до самого правого непробела или до головки, в зависимости от того, что находится правее, положение головки и текущее состояние. Вероятно, содержимое ленты следует напечатать на одной строке, а указатель головки и состояние — на следующей. Руководствуйтесь соображениями красоты и ясности. **Алфавит ленты**, т. е. множество символов, которые могут появляться на ленте, есть просто набор литер, встречающихся где-либо во второй или четвертой компоненте пятерки. Программа должна позволять использовать любую нормальную литеру, имеющуюся в вашей системе. В алфавит всегда входит пробел. Его непросто изобразить в исходных данных, а появляясь в выходной строке, пробелы могут вносить неразбериху. Проблему с вводом можно обойти, если, например, разделять пять компонент команды запятыми. Работа со значащими пробелами часто вызывает затруднения. В естественных языках пробелы осмысленны, но обычно лишь как разделители слов, а не как полноправные символы. Таким образом не существует какого-либо стандартного соглашения об употреблении пробелов в качестве символов.

Тема 12: Стратегия компьютера при игре в калах

Дискуссии о «разумном» поведении компьютеров начались задолго до появления реальных вычислительных машин. Многие согласятся, что высокое мастерство в какой-либо интеллектуальной игре, не поддающейся полному анализу, должно свидетельствовать о незаурядных умственных способностях игрока. Если компьютер к тому же обучаем, то отрицать его интеллект еще труднее. Говоря о машинной игре, чаще всего имеют в виду шахматы, однако даже самые лучшие программы оказываются на уровне весьма посредственных шахматистов. Но в других играх можно достичь большего успеха.

Калах, известный также и под другими названиями (*манкала*, *вари*, *овари*), — это старинная африканская игра. По ее теории написано очень мало, тем не менее в течение столетий в калах играют с помощью камешков представители самых разных культур. Хотя игра эта — чистая борьба умов, не содержащая какого-либо случайного элемента, африканцы режутся в нее постоянно. Благодаря нехитрому инвентарю и простым правилам, калах великолепно подходит и для игры с машиной. Как показывают современные исследования, компьютер с программой наподобие предлагаемой здесь играет в калах лучше любого человека.

Игровое поле для калаха схематически изображено на рис. 12.1. Игроки (их двое) садятся друг против друга. Каждому игроку принадлежат шесть малых **лунок** вдоль длинной стороны поля и одна лунка большего размера по его правую руку, называемая **калахом**. В начале игры в каждую малую лунку помещается некоторое количество k камней (для $k \leq 3$ известно полное решение; африканцы обычно используют $k = 6$). Ход игрока заключается в том, что он забирает все камни в одной из малых лунок на своей стороне и раскладывает их по одному в остальные лунки, двигаясь против часовой стрелки. Первый камень кладется в лунку справа от той, из которых взяты камни, затем в следующие, включая свой калах и малые лунки противника, но не калах противника. Может случиться (и это допускается правилами), что раскладывая камни, мы обойдем всю доску и вернемся в исходную лунку или даже пройдем дальше. На рис. 12.2a и b, показаны позиции до и после выполнения такого циклического хода.

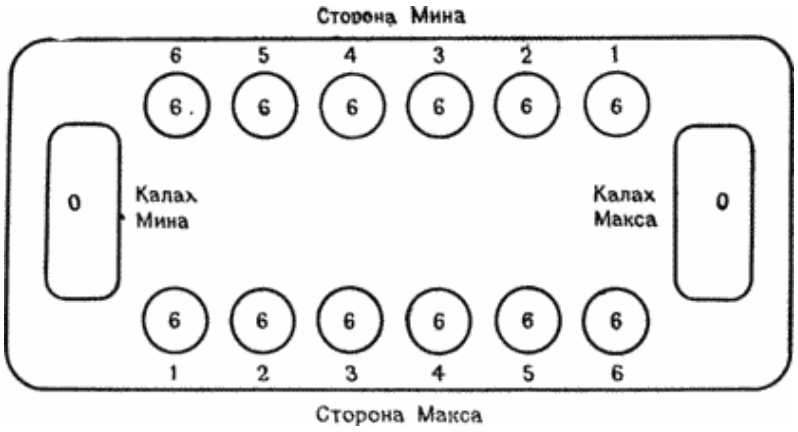


Рисунок 12.1. Игровое поле для калаха. Числа в лунках показывают количество находящихся там камней.

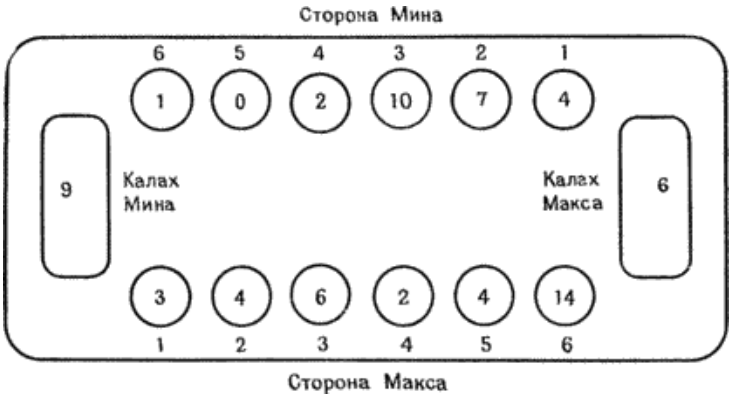


Рисунок 12.2a. Перед циклическим ходом Макса. Макс ходит из лунки 6.

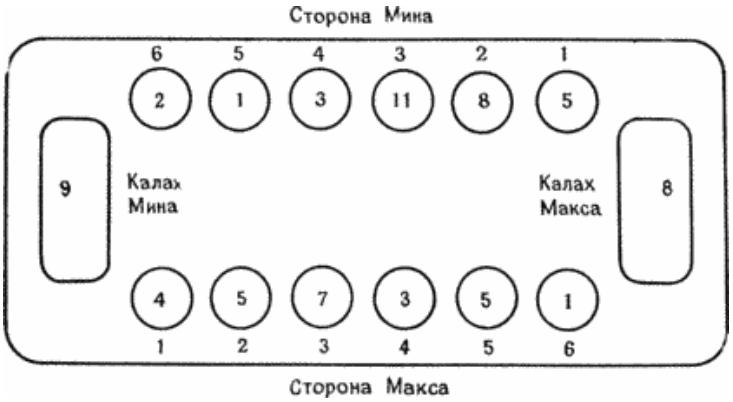


Рисунок 12.2b. После циклического хода Макса. Калах Макса пополнялся дважды.

Есть два дополнения к правилу выполнения хода. Если последний камень попал в одну из непустых малых лунок игрока, делавшего ход, причем камни клались и в лунки противника, то делается **повторный ход** из лунки, в которую попал последний камень, по тем же правилам, что и первый. Игрок может сделать сколь угодно длинную серию повторных ходов. Если последний камень попал в одну из малых лунок противника, и в этой лунке стало два или три камня, то эти камни **берутся в плен** и помещаются в калах игрока, сделавшего ход. Если при пленении в предыдущей лунке также оказалось два или три камня, то и они забираются в плен. Теоретически игрок может за один ход полностью очистить сторону противника. Игра оканчивается, как только в калахе одного из игроков окажется больше половины всех камней (заметим, что если камень попал в калах, то он уже никогда его не покинет). Если у игрока, получившего очередь хода, не осталось ни одного камня в малых лунках, то игра немедленно прекращается и все камни противника попадают в калах противника. На рис. с 12.3 по 12.5 показаны некоторые типичные ходы.

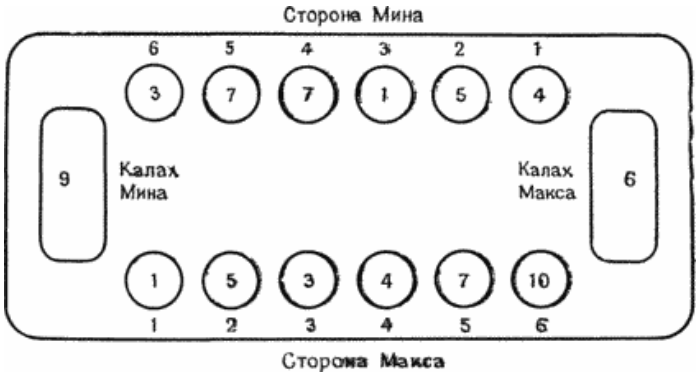


Рисунок 12.3a. Серия ходов из лунки 6 Макса. Последний камень попадает в лунку 3 Макса, и из этой лунки делается повторный ход.

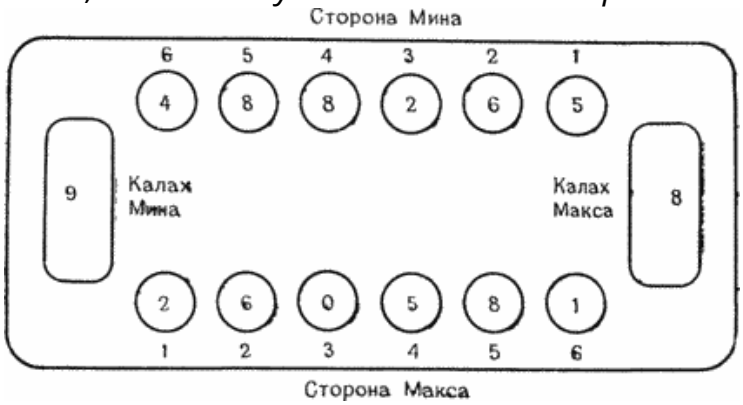


Рисунок 12.3b. После серии ходов. Камни из лунки 3 разложены в следующие лунки.

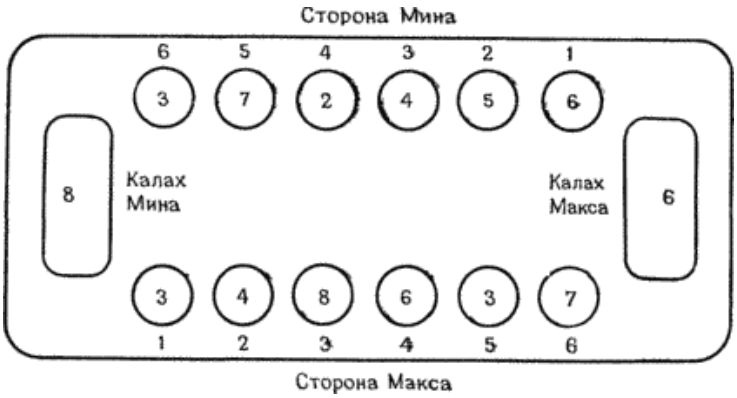


Рисунок 12.4a. Перед взятием в плен камней Мима. Ходом из лунки 3 Макс попадает в лунку 4 Мина.

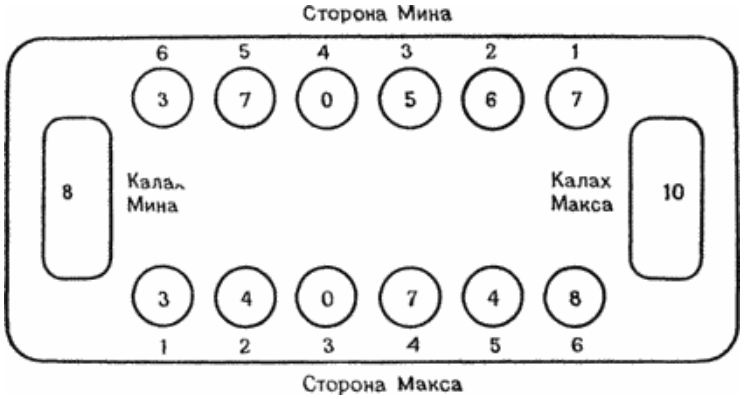


Рисунок 12.4b. После взятия в плен камней из лунки 4 Мина. В калах Макса один камень попадает при раскладывании камней и еще три — при пленении.

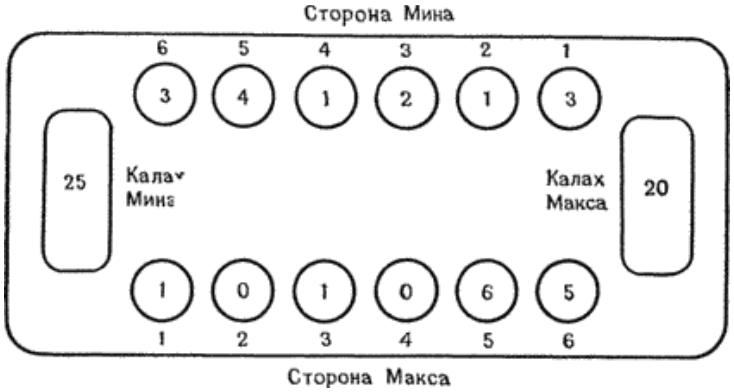


Рисунок 12.5a. Многократный захват пленных Максом. Камни из лунок Мина 2, 3 и 4 берутся в плен одним ходом из лунки 6.

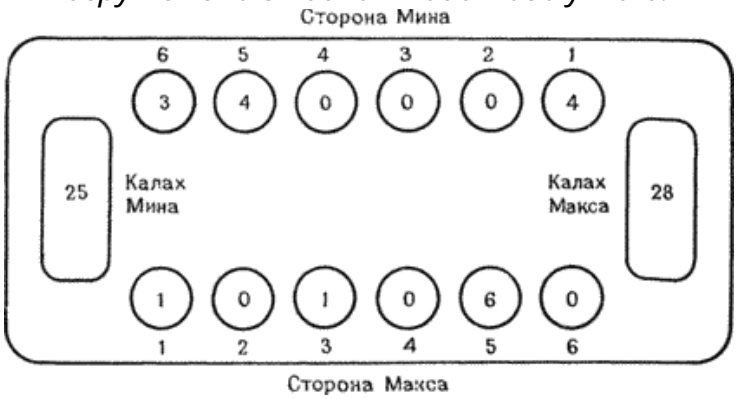


Рисунок 12.5b. Макс почти опустошил лунки Мина. Отметим, что Макс мог бы сделать ход с пленением из лунки 5 вместо лунки 6.

Программа для проверки правильности хода весьма проста. Выбор игрока ограничен максимум шестью возможностями для хода. После того как начальная лунка выбрана, легко находятся повторные ходы и взятия в плен. Для проверки окончания игры после хода надо лишь сравнить калах игрока, делавшего ход, с половиной общего числа камней. Конечно, отсюда никоим образом не следует, как находить наилучший в данной позиции ход.

Основная идея выбора хода состоит в построении **дерева** всех возможных продолжений из заданной позиции. Затем мы выбираем такую ветвь дерева, которая обеспечивает в конце концов победу. Для простоты изложения и еще по одной причине, которая вскоре станет понятной, мы будем называть компьютер Максом, а его противника — Мином. Предположим, что где-то в середине игры настала очередь хода компьютера. Макс может попытаться оценить положение, испробовав поочередно каждый из шести возможных ходов. Если один из них сразу же приводит к выигрышу, то Максу, очевидно, следует делать именно этот ход. Но что делать Максу, если ни один ход не ведет к немедленной победе? Как выбрать ход?

Максу следует проанализировать каждый ответ Мина на свои ходы. Допустим, один из этих ответов приводит к выигрышу Мина. Со стороны Макса было бы глупо делать ход, дающий Мину шанс на немедленную победу (хотя иногда Максу не избежать этого). В таком случае Макс узнает, какие ходы не делать. Однако, чтобы

выбрать, какой ход делать, Максу придется построить еще один уровень ответов на ответы Мина к исходным ходам Макса. Если можно найти выигрышный ответ для некоторого множества ответов Мина, то Максу следует выбрать тот первоначальный ход, при котором Мину остаются только ответы, для которых есть выигрышный ответ Макса (помните дом, который построил Джек?). Если все это непонятно, попробуйте найти ходы, ответы и ответы на ответы для позиции с рис. 12.6.

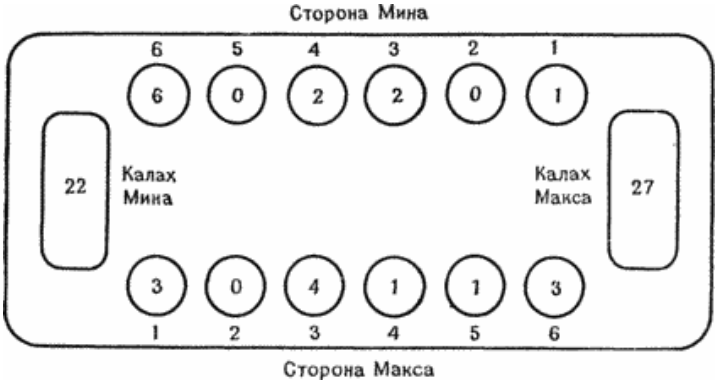


Рисунок 12.6a. Ход Макса. Макс ходит из лунки 1.

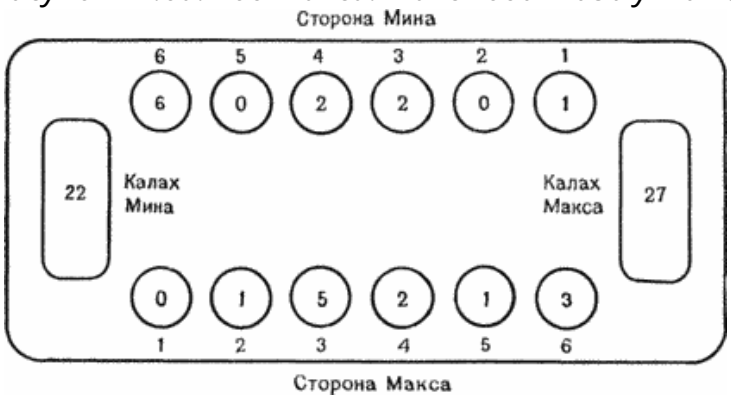


Рисунок 12.6b. Результат хода Макса. Мин отвечает ходом из лунки 6.

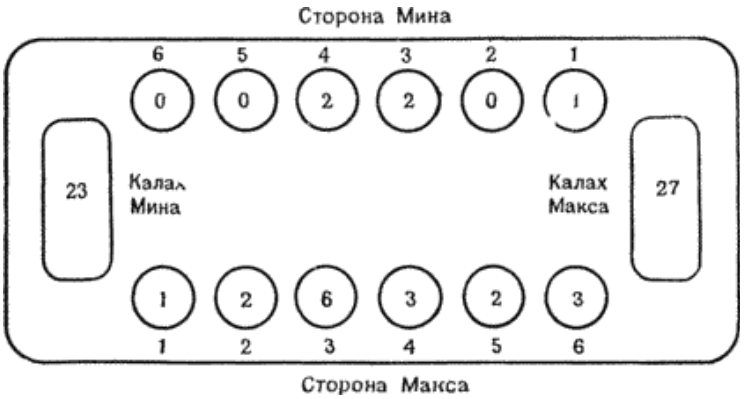


Рисунок 12.6c. Позиция после ответа Мина. Макс отвечает ходом из лунки 2.

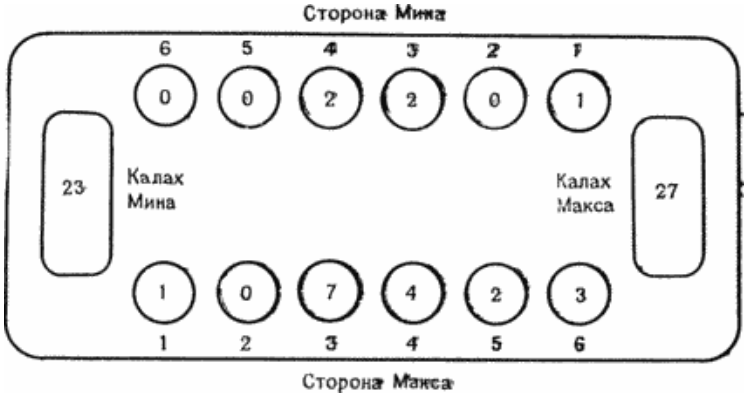


Рисунок 12.6d. Позиция после ответа Макса. Это всего лишь одна из примерно 63 подобных цепочек ходов.

Однако может оказаться недостаточным заглядывать вперед на два уровня. Несмотря на то что игра в калах обязательно кончается, предсказать, насколько далеко нужно заглянуть вперед, чтобы найти конец, оказывается весьма трудно. Каждый следующий уровень требует примерно в шесть раз больше времени и памяти, чем предыдущий. Надо что-то предпринять, чтобы остановить этот рост.

Этой цели служит **статическая оценочная функция**, дающая оценку позиции без построения дерева. В калахе мы воспользуемся **разницей очков**, получаемой как разность числа камней в калахах Макса и Мина. Если слегка изменить правила, потребовав, чтобы в калах победившего игрока сразу же перекладывались все камни, то разница очков всегда будет положительной, когда Макс имеет преимущество, а когда Макс выиграет, разница очков станет максимально возможной. Теперь Макс может выбирать тот из шести ходов, который **максимизирует** (не зря его зовут Макс) статическую оценочную функцию. Если два хода одинаковы с этой точки зрения, Макс может выбрать любой из них случайным образом.

Ну вот мы и ответили на вопрос о стратегии Макса. Так ли это? Если бы этим исчерпывалась стратегия калаха, то такая игра немногого бы стоила. Ведь Мин может ставить Максу ловушки, и, чтобы их избежать, Максу нужно смотреть вперед. Статическую опенку можно применять к позициям, лежащим глубоко в дереве и не являющимся заведомо выигрышными или проигрышными.

Пусть Макс хочет заглянуть вперед на d уровней. Будем считать, что начальная позиция лежит на уровне 0. Постройте все шесть мыслимых ходов, приводящих нас на уровень 1. Применяя в каждой позиции уровня 1 ходы Мина, получите все позиции уровня 2.

Продолжайте в том же духе, пока не построите все дерево вплоть до уровня d. Иногда может оказаться, что не все шесть ходов возможны, поскольку одна или несколько лунок пусты. Кроме того, некоторая ветвь может окончиться из-за хода, завершающего игру. Заметим, что все ходы на четных уровнях делает Макс, а на нечетных — Мин.

Теперь, чтобы перенести оценку с уровня d на уровень 0, выполните следующие действия на всех уровнях, начиная с уровня d и кончая нулевым. Примените статическую оценочную функцию ко всем **листьям** на рассматриваемом уровне.

Это дает разницу очков в листьях. Для нелистовых узлов постройте оценку разницы очков, найдя максимум оценок по всем непосредственным преемникам данного узла, если он находится на четном уровне, и минимум, если узел — на нечетном уровне. Такой способ действий отвечает стремлению Макса максимизировать разрыв и стремлению Мина минимизировать его (или сделать более отрицательным). После того как пройден весь путь до нулевого уровня и найдена разница очков в исходном узле, выберите любой из шести ходов, позволяющий получить эту разницу очков. Отметим, что, как правило, все листья будут находиться на уровне d. Кроме того, при построении дерева можно всегда проходить каждую ветвь сначала вглубь,т.е. строить дерево в порядке **перебора в глубину**, а не в порядке **перебора в ширину**, как описано[19]. На рис. 12.7 показана часть возможного дерева игры. До листьев доведена лишь одна ветвь. Изображены правильные значения, вычисленные исходя из показанной на рисунке информации. Максу следует выбрать ход из лунки 1.

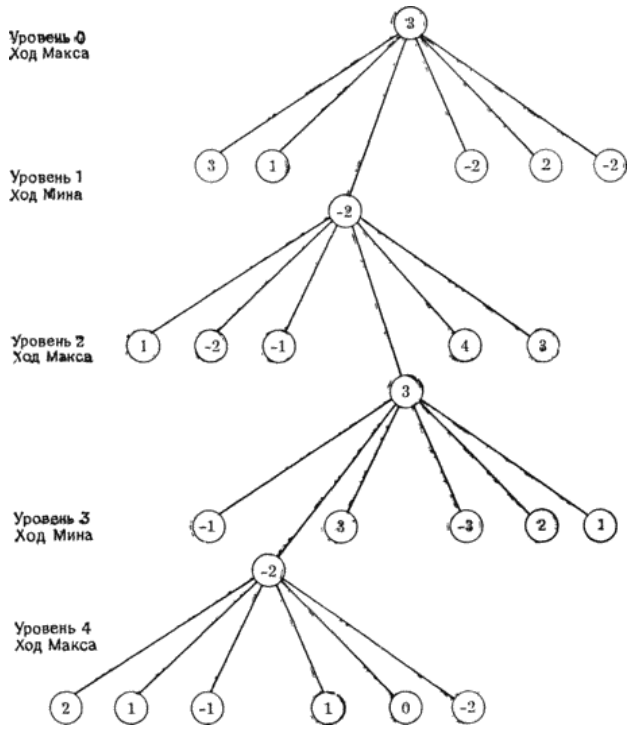


Рисунок 12.7. Возможное дерево игры. Полностью показан только один путь до низа дерева. Предполагается, что значения в кружках получены из анализа нижних уровней.

По сути дела, мы сейчас описали **минимаксную** процедуру для игры двух лиц. Как нетрудно видеть, для анализа на d уровней вперед необходимо построить около

$$\sum_{i=1}^{d} 6^i \approx 6^{d+1}$$

позиций. Ввиду очень быстрого роста этой функции желательно иметь какое-либо средство, экономящее усилия. **Альфа-бета-процедура** при том же объеме работы позволяет иногда провести анализ на вдвое большую глубину.

Идея этой процедуры обобщает рассмотренный пример. Допустим, что в некотором внутреннем узле A дерева ход должен сделать Макс и что он с помощью перебора в глубину уже построил полное дерево B для хода из лунки 1 и дерево C

для хода из лунки 2. Предположим далее, что оценка, вычисленная при анализе дерева, равна 1 для узла B и 2 — для C. Тогда можно приписать узлу A **предварительную оценку** (ПО), равную 2. Что бы ни случилось, Макс может отвергнуть любой ход из A с оценкой меньше 2. Допустим теперь, что Макс начинает строить дерево для хода из лунки 3 в узел D. В узле D ход Мина. Как только D получит ПО, равную или меньшую 2, дальнейшее построение дерева ниже D окажется уже ненужным. Действительно, Мин заведомо не выберет ход с оценкой больше 2, если доступно значение 2 или меньше. Но тогда узел D не будет интересовать Макса, коль скоро он уже имеет возможность получить 2. Итак, можно прекратить раскрытие узла D. Обсуждавшееся дерево показано на рис. 12.8.

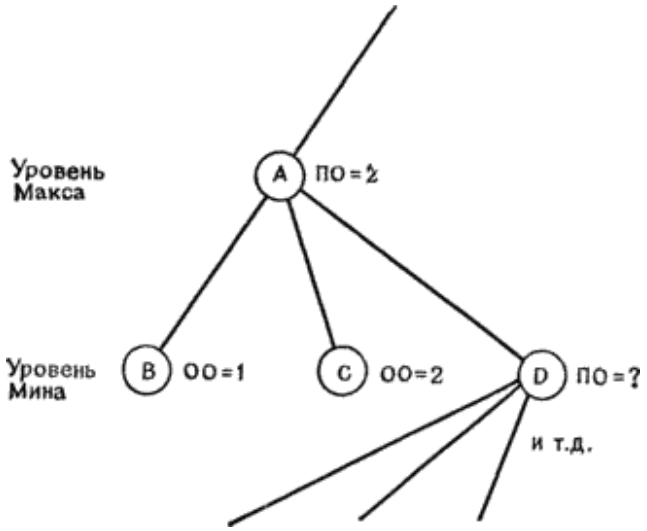


Рисунок 12.8. Часть дерева для альфа-бета-процедуры, описанного в тексте. Как только ПО в узле D опустится ниже 3, можно прекращать раскрытие узла D и его преемников.

Альфа-бета-процедура

Для выполнения альфа-бета-процедуры поиска минимакса начните с перебора дерева игры в глубину. Каждому узлу приписывается предварительная оценка (ПО) и **окончательная оценка** (ОО). Для листьев как ПО, так и ОО равна статической оценке. ПО во внутренних узлах Макса равна максимуму из ОО преемников этого узла, в узлах Мина — минимуму. Всякий раз, когда ПО меняется, мы проверяем, не следует ли прекратить раскрытие этого узла. (Первоначально ПО равна -∞ во внутренних узлах Макса и +∞ во внутренних узлах Мина). В узле Макса происходит **отсечение** всякий раз, как только ПО этого узла становится не меньше ПО какого-либо предшественника этого узла, принадлежащего Мину. Аналогично в узле Мина отсечение происходит, когда его ПО становится не больше ПО какого-либо из предшественников, принадлежащего Максу. При отсечении узла его ПО становится его ОО. Вам следует убедиться, что альфа-бета-процедура всегда выбирает тот же ход, что и обычная минимаксная процедура.

Тема. Напишите программу для игры в калах, использующую альфа-бета-процедуру. Ваша программа должна уметь играть как против человека за терминалом, так и против самой себя. Следует предусмотреть возможность

изменения глубины d просмотра, числа к камней в каждой лунке, а также замены игрока, делающего первый ход. Вывод позиций и ввод ходов следует представить в наиболее удобной форме. По требованию программа должна выдавать на печать дерево ходов. Чтобы с программой было интересно играть, она должна случайным образом выбирать ход из нескольких равноправных (как обычно, эту возможность следует отключать на время отладки).

Указания исполнителю. Несмотря на многословное описание, сама программа для игры в калах довольно проста. Основную трудность составляет построение структуры данных для представления игровых деревьев и обеспечение должного порядка создания и уничтожения этих деревьев. Вам, вероятно, придется написать свои собственные программы, которые будут выделять пространство для деревьев и собирать освобождающуюся память. Требование эффективности по времени работы накладывает ограничение на глубину просмотра; учитывайте это в программах порождения дерева. Вероятно, имеет смысл обеспечить относительную независимость минимаксной процедуры от остальной части программы, с тем чтобы изменения минимаксной процедуры не влияли на всю программу.

Развитие темы. Хотя в литературе, указанной в библиографии, содержится большое число модификаций альфа-бета-процедуры, мы обсудим здесь только две из них. В первой делается попытка повысить эффективность отсечения. Работа альфа-бета-процедуры основана на том, что хорошие ходы (за обоих игроков) прекращают анализ худших ходов. Поэтому, чем раньше мы будем находить хорошие ходы, тем чаще будут отсекаться плохие. Итак, следует попытаться в первую очередь раскрывать хорошие ходы. В методе с **фиксированным упорядочением** непосредственные преемники узла упорядочиваются с помощью статической оценочной функций до их анализа. Первым раскрывается узел с наилучшей оценкой. Статическая оценочная функция, как мы надеемся, хорошо предсказывает окончательный результат, получаемый с помощью просмотра, следовательно, эта процедура повышает вероятность того, что сначала будут рассматриваться хорошие ходы. В узлах Мина в первую очередь следует рассматривать преемники с минимальной оценкой.
Еще одно обобщение состоит в изменении статической оценочной функции. Часто используют оценку, равную разности числа всех камней на стороне Макса (в калахе и лунках) и числа камней на стороне Мина. Можно применить любую простую линейную функцию от 14 значений для всех лунок. Наилучшую такую функцию можно выбрать при помощи турнира. Не забывайте, однако, что главным фактором, определяющим силу игрока, является, до всей видимости, глубина просмотра.

Тема 13: Поиск узоров из простых чисел

Всякий, кто изучает простые числа, бывает очарован ими и одновременно ощущает собственное бессилие. Определение простых чисел так просто и очевидно; найти очередное простое число так легко; разложение на простые сомножители — такое естественное действие. Почему же тогда простые числа столь упорно сопротивляются нашим попыткам постичь порядок и закономерности их расположения? Может быть, в них вообще нет порядка, или же мы так слепы, что не видим его?

Какой-то порядок в простых числах, несомненно, есть. Простые числа можно отсеять от составных **решетом Эратосфена**. Начнем с того, что 2 — простое число. Теперь выбросим все большие четные числа (делящиеся на 2). Первое из уцелевших за двойкой чисел, 3, также должно быть простым. Удалим все его кратные; останется 5. После удаления кратных пяти останется 7. Будем продолжать в том же духе; все числа, прошедшие через решето, будут простыми. Эта регулярная, хотя и медленная процедура находит все простые числа. Мы знаем, кроме того, что при n, стремящемся к бесконечности, отношение количества простых чисел к составным среди первых целых чисел приближается к ln n/n. К сожалению, этот предел чисто статистический и не помогает при нахождении простых чисел.

Оказывается, что все известные методы построения таблицы простых чисел — не что иное, как вариации унылого метода решета. Эйлер придумал формулу $x^2 + x + 41$; для всех x от нуля до 39 эта формула дает простые числа. Однако никакая полиномиальная формула не может давать подряд бесконечный ряд простых чисел, и функция Эйлера терпит фиаско при x = 40. Другие известные функции дают длинные ряды простых чисел, но ни одна не дает сплошь простые. Исследователи проанализировали множество целочисленных функций, однако до сих пор не удалось увидеть закономерность.

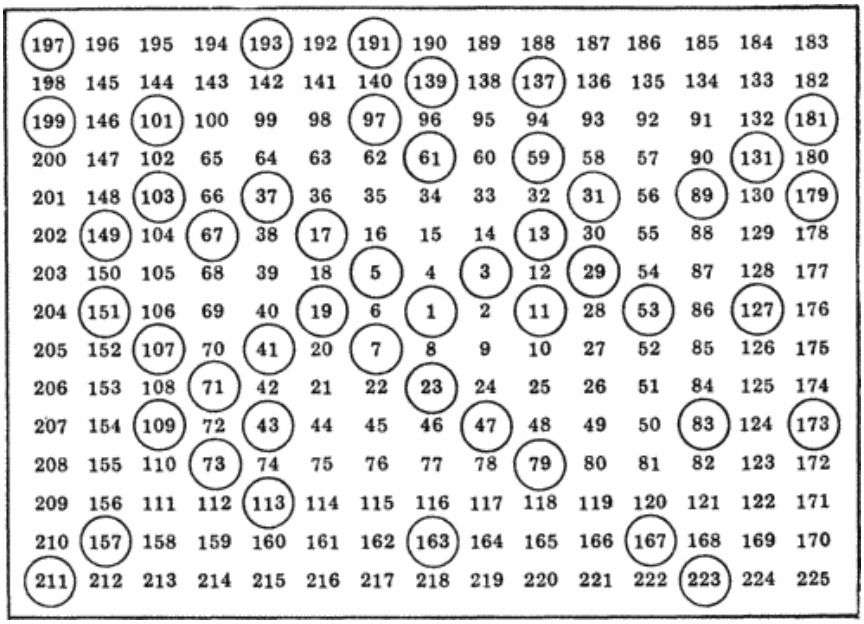


Рисунок 13.1. Числа расположены по спирали против часовой стрелки.

Закономерности проявляются, когда целые числа отображаются на плоскость (или в пространство). Одно из возможных отображений показано на рис. 13.1, где числа располагаются вокруг начальной точки по спирали против часовой стрелки. На рис. 13.2 целые числа заполняют треугольник положительного квадранта. Если достаточно далеко расширить рамки этих рисунков, то станет видно, что простые числа располагаются преимущественно вдоль отдельных прямых (в основном по диагоналям) и совершенно игнорируют другие прямые. Частично этот эффект легко объясним* В обоих расположениях целые числа, попадающие на любую диагональ, даются некоторым квадратичным многочленом. Если многочлен, соответствующий какой-либо прямой, разлагается на рациональные линейные множители, то эта прямая будет содержать одни составные числа. Таким образом, простым числам волей-неволей пришлось облюбовать неприводимые прямые. Однако некоторые неприводимые многочлены изобилуют простыми числами, и изобилие это не оскудевает, несмотря на то что плотность простых чисел среди всех целых медленно стремится к нулю. Иными словами, хотя разложение многочленов объясняет в некоторой степени скученность простых чисел, все же существуют многочлены, более богатые простыми числами, чем предсказывает обычный статистический анализ.

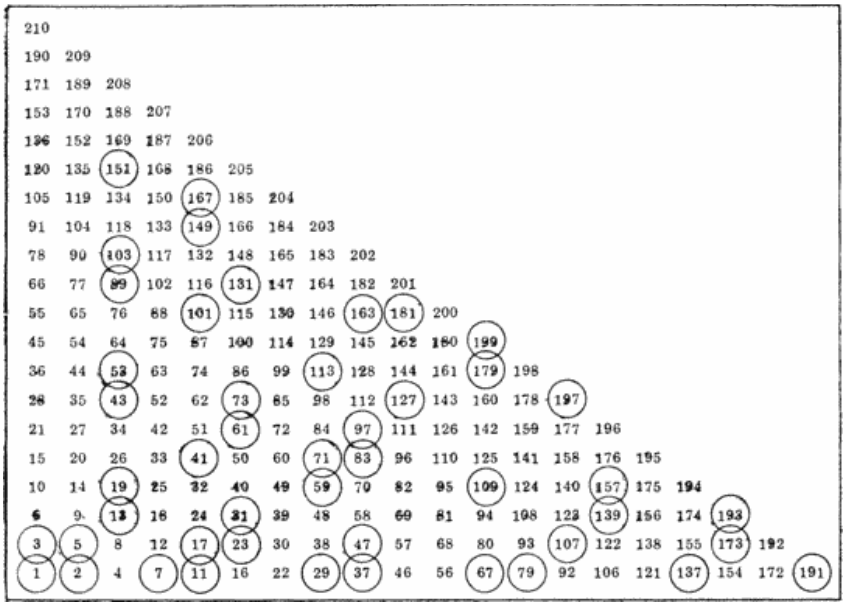


Рисунок 13.2. Числа в треугольнике.

Тема. Напишите программу, которая отображает целые числа на плоскость некоторым регулярным образом и отмечает на рисунке места, где находятся простые числа. Выведите формулы, описывающие прямые линии на вашем рисунке, и напечатайте те из них, которые особенно изобилуют простыми числами; печатайте также долю простых чисел на этих прямых. Обеспечьте высокую эффективность ваших программ проверки целых чисел на простоту, так чтобы вам хватило времени для анализа весьма отдаленных отрезков натурального ряда.

Тема 14: Учет расхода бензина

Тридцать центов за галлон — дело прошлое. Сорок центов за галлон — дело прошлое. Пятьдесят центов за галлон — дело прошлое. Сейчас галлон бензина стоит шестьдесят центов, и, возможно, вскоре мы останемся вообще без горючего. Так что на повестке дня — анализ индивидуального расходования бензина.

Многие ведут журнал покупок бензина. Обычно туда записывают дату, показания счетчика пройденного пути, марку бензина, цену одного галлона, сколько галлонов куплено и общую стоимость. Три последние величины зависят друг от друга; эта зависимость не совсем точная из-за ошибок округления, но ее все же можно использовать для проверки правильности исходных данных. С помощью ЭВМ вы можете получить разнообразную статистическую информацию. Интересно вычислить такие производные величины, как средняя стоимость одного галлона, средний пробег на галлон, средний пробег за день, средняя стоимость пробега в одну милю, среднее время расходования одного галлона. Кроме того, хорошо было бы получить такую же информацию по каждой марке бензина и посмотреть, есть ли разница между марками. Таблица 14.1 — фрагмент реального журнала покупок бензина.

Таблица 14.1. Выдержка из реального журнала покупок бензина

Дата	Марка бензина	Пробег (в милях)	Цена галлона (в центах)	Количество галлонов	Сумма
03/27/74	Texaco	24 370	59.9	13.5	$8.00
04/05/74	Texaco	24 434	59.9	5.5	$3.30
04/11/74	Texaco	24 596	59.9	8.2	$4.88
04/23/74	Mobil	24 862	58.8	12.9	$7.60
05/13/74	Mobil	25 057	61.9	13.9	$8.60
06/11/74	Arco	25 239	62.9	12.5	$7.85
07/12/74	Texaco	25 435	63.3	14.2	$8.90
07/19/74	Chevron	25 713	58.8	12.4	87.27
07/28/74	Mobil	26 135	60.9	14.1	$8.60
08/07/74	Arco	26 384	60.4	13.1	$8.00
08/08/74	Chevron	26 712	59.9	13.3	$7.90
08/16/74	Arco	26 997	60.9	13.6	$8.30
08/22/74	Mobil	27 068	60.9	4.0	$2.45
08/22/74	Shell	27 362	61.6	11.8	$7.25
08/23/74	Shell	27 606	63.4	10.3	$6.54
08/25/74	Ericson	27 913	60.9	13.6	$8.29
08/26/74	American	28 163	60.9	10.8	$6.55
08/26/74	American	28 487	57.9	14.0	$8.10
08/27/74	DX	28 771	53.9	12.2	$6.60
08/28/74	Conoco	29 114	59.9	14.8	$8.90
08/28/74	Texaco	29 337	58.9	10.2	$6.00
08/28/74	Phillips	29 661	60.9	13.9	$8.35
08/29/74	Chevron	29 912	65.9	10.8	$7.10
08/29/74	Shell	30 147	65.9	10.3	$6.70
08/30/74	Texaco	30 317	60.9	7.6	$4.60
08/31/74	Exxon	30 643	56.9	13.3	$7.60
09/06/74	Shell	30 878	59.9	13.2	$7.90

09/10/74	Shell	31 182	59.9	13.0	$7.80
09/14/74	Exxon	31467	57.9	13.1	$7.60
09/18/74	Arco	31 711	57.9	10.1	$5.85
09/24/74	Arco	31 984	57.9	12.5	$7.25
09/27/74	Arco	32 225	57.9	9.9	$5.70
10/01/74	Arco	32 455	57.9	9.8	$5.65

Будем считать в данной задаче, что каждой новой записи в журнале соответствует новая полная заправка автомобиля. Первая запись в журнале устанавливает точки отсчета дат и пройденного пути; никак иначе она не используется. Следующие записи фиксируют пробег и расходы на очередную заправку, показывая объем сожженного горючего и количество пройденных миль после предыдущей заправки. Было бы также любопытно печатать текущие средние значения за самое последнее время, чтобы заметить краткосрочные изменения.

Тема. По данным, имеющимся в журнале покупок бензина, напечатайте разнообразную контрольную статистику, показывающую водителю, во что обходится эксплуатация автомобиля. Исходные данные о каждой покупке — это дата, марка бензина, показание счетчика пройденного пути, цена одного галлона, сколько галлонов куплено и общая стоимость. Выводимая информация должна повторять исходную и, кроме того, включать в себя такие показатели, как пробег между заправками, пробег на один галлон, стоимость пробега в одну милю, стоимость одного галлона, стоимость одного дня, время расходования одного галлона. Все указанные показатели следует получать для каждой заправки и усреднять двумя способами: за небольшой срок и за все время наблюдений. Кроме того, соберите данные по каждой марке бензина и напечатайте соответствующие средние значения. Не ограничивайте число различных марок.

Указания исполнителю. Предлагаемая программа не особенно трудна. Для печати долларовых величин, как и в некоторых других задачах, нужна некоторая изобретательность. Требование неограниченности числа марок означает, что их нельзя задать заранее, поэтому нужна простая расширяемая таблица марок и связанной с ними информации.

Тема 15: Моделирование движения на автостраде

Энергетический кризис уже в своем начале привел к снижению допустимой скорости движения на автодорогах в масштабах всей страны. Большинство автомобилистов в длительных поездках не устают проклинать это ограничение. Разумеется, теперь мы знаем, что снижение скорости ежегодно сберегает тысячи жизней и миллионы долларов. Но мало кто из водителей понимает, что в условиях перегруженности автострад вблизи больших городов снижение скорости на самом деле ведет к экономии времени. Более точно парадокс формулируется так: если все тише едут, они скорее приедут.

Вспомните, как вы однажды «с ветерком» катили по шоссе, миль на 5 превышая допустимую скорость, хотя машин было много. Внезапно все стали еле ползти, и вам тоже пришлось нажать на тормоза. Затем последовала четверть, половина, а то и целая миля конвульсивного чередования остановок и движения. Наконец затор остался позади, и вы смогли вновь прибавить скорость. Но все это происходило без всякой видимой причины! Что же нарушило плавность движения? Для объяснения причины задержки необходимо привлечь гидродинамику. Движущиеся по шоссе автомобили ведут себя во многом аналогично частицам протекающей в трубе жидкости. Если плотность и скорость частиц достаточно велики, любая кратковременная задержка потока приведет к возникновению **ударной волны**. Ударная волна — это область очень высокой плотности; автомобили (или частицы) резко замедляются, попадая в эту область, и затем ускоряются, когда, преодолев довольно четко очерченный **ударный фронт**, оказываются в области с гораздо более низкой плотностью. Ударная волна продолжает существовать длительное время, медленно двигаясь навстречу потоку и медленно рассеиваясь. Отметим, что рассеяние объясняется уменьшением плотности в ударной области и может быть ускорено, если водители заранее слегка притормозят, увидев впереди затор.

Было бы любопытно провести эксперимент на автостраде в часы пик, но, несомненно, пришлось бы привлечь не одну сотню машин. Не лучше ли обойтись одной вычислительной машиной? Рассмотрим прямой однорядный участок автострады длиной 5 миль, без перекрестков. Автомобили появляются на одном конце дороги, проезжают по ней и бесследно исчезают на другом конце. Машины стремятся двигаться по дороге с постоянными скоростями (возможно, разными для разных машин). Чтобы изучать ударные волны, будем вводить в эту транспортную благодать случайные замедления.

Для проведения эксперимента нужны **генератор** автомобилей и **генератор** возмущений. В начале каждого эксперимента автострада пуста. Запустите генератор автомобилей, который поместит машину на дорогу, придаст ей скорость и выберет интервал до порождения следующего автомобиля. Начальные скорости автомобилен подчиняются равномерному случайному распределению на отрезке от 50 до 60 миль в час, а интервалы между порождениями также равномерно распределены на отрезке от 4 до 6 с. Минимальное допустимое сближение составляет одну длину автомобиля (10 футов) на каждые 10 миль в час скорости передней машины. Когда автомобиль

приближается к идущей впереди машине на утроенное допустимое расстояние, он начинает притормаживать, пока скорости не сравняются, теряя по одной миле в час за секунду. Если передний автомобиль начинает резко замедляться, идущий сзади выжидает 0,2 с и затем тормозит, снижая каждую секунду свою скорость на 15 миль в час. В результате может произойти авария, которой и закончится эксперимент.

Собственно эксперимент состоит в заполнении дороги машинами, введении искусственного замедления и наблюдении результата. Начните запускать машины на пустую дорогу; продолжайте делать это, пока не пройдет 2 минуты (модельного времени) с момента прохождения заданного участка дороги первым автомобилем. Затем, не прекращая запускать машины, выберите автомобиль, который раньше всех пересечет отметку в 4 мили, сбросьте с его скорости как можно резче 0, 10, 20, 30, 40 или 50 миль в час, удержите на новой скорости 100 ярдов, после чего придайте ему ускорение 5 миль в час за секунду, пока автомобиль не наберет свою первоначальную скорость (машины всегда стремятся сохранить первоначальную скорость). Продолжайте эксперимент еще 5 минут после того, как виновник затора начал замедляться, и подсчитайте количество машин, прошедших участок дороги за это время. Полученная величина и есть наблюдаемый результат эксперимента. Машины, следующие за виновником, также могут ускоряться на 5 миль в час за секунду, если дорога перед ними освобождается. Проведите эксперимент несколько раз для каждого значения замедления. Если произойдет авария, все машины, находящиеся позади, автоматически остановятся и не смогут пройти заданный участок дороги. В аварию может попасть не сам виновник, а машины, идущие сзади.

Тема. Напишите программу, позволяющую провести эксперимент с ударной волной на автостраде. Все исходные данные представлены одним числом — сколько раз повторять эксперимент для каждого уменьшения скорости. Обязательная часть выводимой информации — среднее количество машин, прошедших участок дороги после каждого искусственного замедления. Но для отладки и лучшего понимания физического поведения системы полезно вывести дополнительную информацию. В частности, несколько «моментальных снимков» дороги, вероятно, позволят почувствовать ситуацию лучше, чем любое количество статистики.

Указания исполнителю. Наиболее трудным в предлагаемой задаче является отслеживание всех автомобилей на дороге. Можно организовать цикл и примерно через одну сотую — одну десятую секунды модельного времени должным образом подправлять положение каждого автомобиля. Если интервал достаточно мал, заметного накопления ошибок не произойдет, а выглядеть программа будет красиво — как семейство вложенных циклов. Однако при использовании метода **пошаговой фиксации** цикл может выполняться слишком большое число раз. В нашем случае эксперимент продлится примерно 12 минут модельного времени, в каждый момент на дороге будет около 90 машин, и, даже если выбрать

большой интервал в одну десятую секунды, потребуется примерно 1200 циклов, или около 650 000 операций с отдельными автомобилями. Если программа тратит много времени на продвижение одного автомобиля, эксперимент слишком затянется. Положение можно подправить, варьируя интервал в зависимости от дорожной обстановки.

Другой подход состоит в том, чтобы подправлять положение автомобилей только в моменты критических событий. При таком подходе заводится список всех событий, ожидаемых в недалеком будущем; например, запускается или исчезает автомобиль, одна машина догоняет другую, прошло две минуты с момента исчезновения первого автомобиля, пора вновь ускорять автомобиль-виновник. Головным элементом **списка событий** всегда должно быть ближайшее событие; список в целом не обязан быть упорядоченным — его можно представить и как **очередь с приоритетами**, и как **кучу**. В основном цикле от списка отделяется головной элемент, описывающий очередное событие, все автомобили устанавливаются в позиции, соответствующие новому времени, запоминаются все события, которые следует планировать, эти события вставляются в список и список переупорядочивается, чтобы ближайшее событие оказалось в голове. Достоинство **моделирования методом критических событий** в том, что порой довольно долго, 4–5 секунд, ничего не происходит. Сэкономленное время можно употребить на более сложную обработку списка событий.

Развитие темы. Строго говоря, в предложенной задаче не изучается ситуация, описанная в нескольких первых абзацах. Вместо выяснения того, что происходит с ударной волной при различных средних скоростях движения на автостраде, в эксперименте рассматривались удары различной силы. Проделайте все еще раз, взяв диапазон начальных скоростей от 40 до 50 или от 60 до 70 миль в час. Попробуйте для некоторых переменных нормальное распределение вместо равномерного. Поварьируйте законы торможения и ускорения. Иными словами, изучите влияние всех параметров, а не только одного, выбранного нами.

Тема 16: Конструирование интерпретатора форматов

Вам, вероятно, пришлось написать по крайней мере одну программу, которая исторгала из машины бумажный поток, несущий искусно оформленные данные. Строка за строкой сомкнутыми рядами выступали из печатающего устройства целые батальоны чисел под предводительством четких заголовков. Интересовали вас только лишь два или три числа, но не напечатать всего было как-то неловко — ведь это так просто! А *вдруг*кому-то и в самом деле захочется узнать точную сумму налога для служащего 1793 на рабочем месте 907, выплаченную им в сентябре пять лет назад.

Выдать на печать такую уйму информации и не свалиться при этом от изнеможения вам удалось благодаря существованию таких вещей, как фортранные инструкции формата, которые помогли преобразовать эти неудобоваримые двоичные числа в радующие глаз цепочки из цифр, букв и знаков. По существу, то же самое происходит и при вводе. Вводные данные аккуратно пробиваются на перфокартах, и вы даже не задумываетесь о том, каким преобразованиям должны они подвергнуться для того, чтобы центральный процессор мог проделывать над ними свои несложные арифметические фокусы. А пожалуй, о вводе и выводе стоит поразмышлять чуть серьезнее. Программа, обрабатывающая большое количество данных, вполне может тратить от четверти до половины своего счетного времени на служебные подпрограммы ввода и вывода, а большая часть этого времени в свою очередь может уходить на интерпретирование форматов и преобразование данных. Вероятно, вы уже не будете так легкомысленно относиться к преобразованиям данных при вводе и выводе, если попытаетесь решить предлагаемую задачу, превратившись на время в системного программиста.

Для изучения были выбраны форматы языка Фортран, поскольку они просты, эффективны и были, в сущности, праотцами большинства других схем форматов. Всякий раз, когда в операции ввода/вывода участвует устройство, предназначенное для взаимодействия машины с человеком, связующим звеном между ними оказывается инструкция формата. Основные элементы, участвующие в **операции ввода/вывода**, — это **список переменных**, **формат** и **файл**. **Элементы данных** пересылаются из файла в переменные списка или в обратном направлении, в зависимости от того, какая операция — ввода или вывода — выполняется. При пересылке каждого элемента интерпретируется некоторая часть формата, достаточная для того, чтобы определить текстовое представление этого элемента в файле. Формат определяет лишь характер пересылки того или иного элемента данных, но объем перемещаемых данных от него не зависит.

Что такое формат?

Формат — это цепочка литер, описывающая преобразования данных, которые нужно выполнить. Поскольку формат всякий раз при использовании интерпретируется, то его можно рассматривать как маленькую программу. Формат общего вида имеет структуру

$(yf^1s^1f^2s^2...f^ns^nz)$,

где n может быть нулем,

y и z — последовательности наклонных черточек, возможно пустые,
f^i — либо одиночный код формата, либо формат общего вида, перед которым может стоять натуральное число,
s^i — разделитель, т. е. последовательность из запятых и наклонных черточек, которая в некоторых случаях может быть пустой.
Пробелы в формате игнорируются везде, кроме одного случая, специально оговоренного ниже, а числа записываются в виде цепочек из десятичных цифр.
Предположим, произошло обращение к операции ввода/вывода. Указатель текущей позиции в файле устанавливается на начало следующей записи. Курсор в формате устанавливается на начальную открывающую скобку и движется вправо либо до первого кода, которому должна соответствовать переменная в списке переменных, либо до правого края формата. Такой процесс позволяет при помощи инструкции вывода напечатать строку данных, не пересылая никаких данных из переменных. Интерпретатор форматов будет иметь некоторую внутреннюю память (организованную, как правило, в виде стека), которая будет освобождаться и на которую мы будем время от времени ссылаться, говоря, что интерпретатор что-либо «запомнил». Основной цикл просмотра формата прост. Интерпретатор получает очередную переменную из списка переменных. Курсор начинает двигаться вправо по формату в поисках такого кода, который соответствует передаче элемента данных из переменной в файл или обратно. При движении курсора вправо могут встречаться такие коды, которые влияют на содержимое файла или устанавливают новые значения параметров, управляющих работой интерпретатора. Действия, предписываемые этими кодами, выполняются непосредственно в процессе сканирования. Перед некоторыми кодами допускаются **коэффициенты повторения** — такой код используется соответствующее число раз. То есть один и тот же код может использоваться с несколькими переменными списка, значит, интерпретатор должен помнить убывающее от цикла к циклу значение счетчика повторений кода. Если курсор дошел до крайней правой закрывающей скобки, то он возвращается к последней открывающей скобке первого уровня без коэффициента повторения, а если таковая отсутствует, то к начальной открывающей скобке формата. Вот три типичные ошибки, которые могут встречаться при форматном вводе/выводе: при вводе встретился конец файла; интерпретатор дважды подряд вышел на правую закрывающую скобку формата, не переслав при этом ни одного элемента данных; нет соответствия между типом кода формата, типом переменной в списке ввода/вывода и типом элемента данных, фактически находящегося в файле (последнее относится только к вводу). При завершении операции вывода последняя частично сформированная запись пишется в файл.
Теперь о самих кодах формата. Пожалуй, единственная полезная классификация, которую здесь можно провести, — это различение кодов **самозавершающихся** и **несамозавершающихся**, которые требуют после себя запятую, наклонную черту или скобку. Интерпретатор всегда помнит текущее значение **масштабирующего множителя**, которое вначале устанавливается

равным нулю и может быть изменено при помощи спецификации масштабирующего множителя. Перечислим коды формата:

r(Открывающая скобка, возможно с коэффициентом повторения, обозначает начало **групповой спецификации формата**, которая заканчивается соответствующей закрывающей скобкой (число открывающих и закрывающих скобок в формате должно быть одинаковым). Вся групповая спецификация будет повторена столько раз, сколько указывает коэффициент повторения. Если коэффициент отсутствует, то считается, что он равен единице.

, Запятая служит признаком конца для таких кодов, которые должны обязательно отделяться от последующих кодов. Никаких других функций запятая не выполняет, допускаются избыточные запятые.

/ Наклонная черта служит признаком конца для несамозавершающихся кодов, а также означает конец обработки текущей записи файла и переход к следующей записи. Если последним обработанным кодом формата была наклонная черта, то при завершении операции ввода/вывода перехода к следующей записи уже не происходит. Несколько подряд стоящих наклонных черточек Приводят к пропуску нескольких записей при вводе и к созданию пустых записей при выводе.

nX При вводе пропускается n литер файла, при выводе в файл записывается n пробелов. Код самозавершающийся, передачи данных не происходит.

nHh1⋯hn При вводе очередные n литер файла помещаются на место литер h1⋯hn формата. При выводе n литер h1⋯hn записываются в файл. Любая из литер hi может быть пробелом, это единственный случай, когда пробел является значащей литерой в формате. Код самозавершающийся. Обмена данными между файлом и переменными не происходит.

rAw Пусть g — число литер, помещающихся в переменной, которая участвует в данном цикле интерпретации формата. Если при вводе w ≥ g, то крайние правые g из очередных w литер файла передаются в переменную, иначе в переменную попадут очередные w литер файла, дополненные справа g - w пробелами. Если при выводе w ≥ g, то в файл выводятся w — g пробелов и затем g литер переменной, в противном случае в файл попадут w крайних левых литер переменной. Коэффициент повторения r необязателен, код несамозавершающийся.

rLw При вводе очередное поле из w литер файла должно представлять собой последовательность пробелов, за которой следует одна из букв T или F, а далее произвольная последовательность литер, что воспринимается соответственно как значение *истина* или *ложь*. При выводе в файл помещаются w - 1 пробелов и одна из букв T или F. Коэффициент повторения r может отсутствовать; код несамозавершающийся.

rIw При вводе цепочка литер, состоящая из нескольких старших пробелов, знака, который может и отсутствовать, и последовательности цифр и пробелов, преобразуется в машинное представление целого числа. Поле ввода состоит из w литер; пробелы после знака воспринимаются как нули. При выводе формируется поле длины w литер, состоящее из нескольких пробелов, знака минус, если он нужен, и прижатой к правому краю цепочки цифр, представляющей данное целое

число. Коэффициент повторения r может отсутствовать, код несамозавершающийся.

sPrFw.d При вводе число с плавающей точкой читается из поля длины w литер. Если поле ввода состоит только из цифр и пробелов или если левее (w - d + 1) -й литеры (начиная счет с 1) встретился только знак, то мы получим вводимое вещественное значение лишь после того, как будет вставлена десятичная точка между (w - d)-й и (w - d + 1)-й позициями поля ввода. Если вводимая цепочка литер содержит точку, то подразумеваемая позиция точки игнорируется. Если вводимая цепочка литер имеет вид вещественного или целого числа, за которым следует либо еще одно целое число со знаком, либо буква «E» и целое число со знаком или без него, то это второе число воспринимается как порядок и значение вещественного числа умножается на десять в степени, равной порядку числа. Если присутствует только порядок числа с буквой «E» вначале, то считается, что перед ним стоит вещественное число, равное единице Если показательная часть числа отсутствует, то прочитанное вещественное число, прежде чем оно будет присвоено переменной из списка ввода, умножается на степень десяти с показателем, равным текущему значению масштабирующего множителя. При выводе число с плавающей точкой записывается в виде $x_1 \cdots x_n.y_1 \cdots y_d$. Число округлено до d знаков после точки, и, если это необходимо, снабжается знаком минус. В поле вывода обязательно присутствует точка, так что при выводе по коду F всегда выполняется неравенство $w \geq d+1$. И в этом случае тоже выводимое значение прижимается к правому краю поля вывода. Спецификация масштабирующего множителя sP, как и коэффициент повторения r, не обязательна. Новое значение s (s — любое целое число со знаком) действует до тех пор, пока не встретится еще одна спецификация масштабирующего множителя. Код F несамозавершающийся.

sPrEw.d Ввод осуществляется так же, как для кола F. Основная форма поля вывода имеет вид $0.y_1 \cdots y_d E z_1 \cdots z_m$ где перед первым нулем и после буквы E может стоять знак минус, если он нужен, а значение m достаточно для размещения максимального порядка, даже если для данного числа это не нужно. Если текущее значение масштабирующего множителя равно q, то вещественная часть основной формы умножается на 10^q, а порядок уменьшается ка q единиц. При $q > 0$ будет q цифр слева от точки и $\max(d - q + 1, 0)$ цифр справа от нее: при $q \leq 0$ слева от точки будет стоять нуль, а справа d + q цифр. Так же как и код F, код E — несамозавершающийся, а спецификация масштабирующего множителя sP и коэффициент повторения r могут отсутствовать.

sPrGw.d Ввод, а также интерпретация спецификаций sP и r осуществляется так же, как для кода F. Для вывода по коду G в зависимости от величины выводимого числа выбирается один из кодов F и E. Пусть M — выводимое значение, причем $10^{k-1} \leq M < 10^k$, где $0 \leq k \leq d$, тогда вывод производится как для кода F(w - 4).(d - k), 4X; в противном случае используется код Ew.d. Отметим, что масштабирующий множитель игнорируется в случае, когда для вывода выбирается код F. Код G несамозавершающийся.

Тема. Создайте пакет программ форматного ввода/вывода для вашей ЭВМ. В общем случае он будет иметь ряд входных точек, доступных для пользователя (в роли которого, как правило, выступает сгенерированная компилятором объектная программа), а также ряд внутренних подпрограмм, которые должны быть защищены от доступа со стороны пользователя. Среди пользовательских входов должны быть: вход для инициализации с параметрами, определяющими операцию ввода или вывода, канал ввода/вывода и формат; входы для каждого типа переменных (вещественной, целой, логической и еще любой из них, используемой для представления текстовых данных), а также вход для терминирования ввода/вывода. Проведите основательное тестирование своих программ и убедитесь, что округление и обработка особых случаев выполняется правильно, а в случае ошибок выдаются соответствующие сообщения.

Указания исполнителю. Наиболее трудная часть задачи — составить ясное представление о поведении вещественных чисел на вашей ЭВМ. Преобразование данных текстового, целого и логического типов выполняется легко, а для сканирования формата и поддержания буферов годятся весьма простые методы. Однако вы, вероятно, обнаружите, что для реализации вполне правильного округления придется серьезно поразмыслить, а быть может, и немного поэкспериментировать. Обязательно включите в свои тесты значения чуть больше и чуть меньше степеней 10, чуть меньше 10^{-d} и т. д. Не поддавайтесь соблазну выделять все увеличивающееся количество частных случаев с целью исправить допущенные ранее в работе промахи, попытайтесь вместо этого найти какой-то другой подход. Одной из наших досаднейших программистских неудач был пакет форматного ввода/вывода, разросшийся наподобие Топси до свыше 3000 строк на языке ассемблера. Как непросто теперь заменить его более ясной и эффективной программой примерно в 1000 строк, написанной еще кем-нибудь! С какой радостью мы бы навсегда избавились от этого монстра!

Развитие темы. Имеется масса возможностей расширить форматы. Можно добавлять новые коды. Например:
'x⋯x' То же самое, что nHX⋯x. Апостроф представляется парой подряд стоящих кавычек.
Bw, Ow, Zw Ввод и вывод соответственно в двоичном, восьмеричном и шестнадцатеричном коде. В этом случае внутреннее представление элемента данных воспринимается как цепочка битов, прижатая к правому краю.
Tn Переместиться в n-ю позицию текущей записи. Такое передвижение может привести к повторному чтению или записи части вводного или выводного файла.
Можно также ослабить слишком строгие требования для ширины поля ввода/вывода. Так, формат E.d может означать при выводе, что система сама подберет ширину w поля, а одиночный код I при вводе может означать, что следующее целое число будет ограничено пробелом, запятой или концом записи, а не шириной поля. Почти в каждой системе ввода/вывода для Фортрана есть подобные расширения, которые вы также можете добавить.

Тема 17: Статистика пасьянсов

У каждого программиста рано или поздно наступает момент, когда работа не идет. Без каких-либо видимых причин программа прямо-таки сопротивляется всем вашим усилиям написать ее. Каждая новая попытка тут же оборачивается грудой макулатуры, и корзина снова полна испорченными бланками. Выход один — забросить на время эту задачу. Если ваш начальник станет выражать свое неудовольствие, объясните ему, что для повышения продуктивности вам необходимо снять умственное напряжение. И пойдите в кино. Или гоняйте мяч до изнеможения. Пообсуждайте Критику чистого разума с какой-нибудь симпатичной вам особой противоположного пола. Просадите небольшую сумму на скачках. Или возьмите колоду карт и приготовьтесь убить часа три, раскладывая свой любимый **пасьянс**. (В Англии бы сказали: приготовьтесь потерять **терпение**.)

Есть две разновидности пасьянсов. В пасьянсах первого рода есть правила раскладки карт, а также правила перекладывания карт. Раскладывание такого пасьянса — это некий механический ритуал, где человек выступает в роли автомата. Такая игра, хоть она и лишена творческого элемента, поможет вам в полной мере вкусить и понять эмоциональное состояние компьютера, выполняющего одну из ваших программ. В пасьянсах второго рода игроку предоставляется некоторая свобода выбора. Человек уже не пассивный наблюдатель, он вступает в борьбу против слепого Случая, олицетворенного перетасованной колодой игральных карт. В таких играх обычно имеются некоторые искусственные условия выигрыша, однако почти ничего не известно о том, каких результатов можно достичь, играя наилучшим образом. Прибегнув к помощи компьютера, можно найти тот эталон, с которым игрок мог бы сравнивать свои результаты. Итак, не раскладывание пасьянса, а программирование игры поможет вам снять умственное напряжение.

Правила раскладывания одного пасьянса

Возьмите обычную колоду карт и тщательно ее перетасуйте. Затем разложите карты так, как показано на рис. 17.1. В середине выложите слева направо ряд из семи стопок, содержащих соответственно ноль, одну, две, ..., шесть карт рубашкой вверх и еще по одной карте сверху рубашкой вниз. На это уйдет 28 карт. Остальные 24 карты разложите в шесть столбиков из четырех перекрывающихся карт под шестью правыми стопками. Все карты в столбиках лежат рубашкой вниз, они перекрывают друг друга таким образом, что самая нижняя, последняя карта столбика лежит поверх предыдущей, которая в свою очередь лежит поверх предыдущей, и т. д. до карты, которая лежит поверх соответствующей стопки и служит начальной картой для этого столбика. Выкладывать карты следует так, чтобы старшинство и масть карты, лежащей рубашкой вниз, были хорошо видны. Наконец, в верхней части стола нужно предусмотреть место для четырех **счетных стопок**, по одной для каждой масти. На рис. 17.1 изображен общий вид первоначальной раскладки.

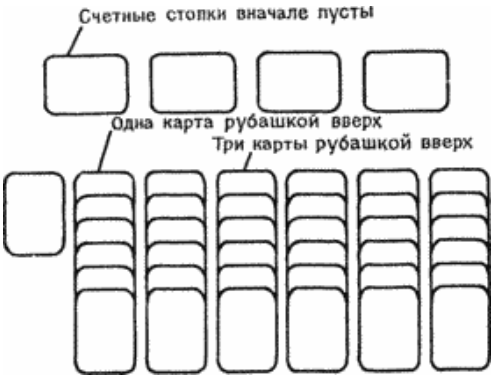


Рисунок 17.1. Сдача карт для пасьянса. Карты в стопках под первыми картами столбиков со второго по седьмой лежат рубашкой вверх, остальные карты лежат рубашкой вниз.

Один **ход** состоит в том, что выбирается произвольная карта, лежащая рубашкой вниз, вместе со всеми накрывающими ее картами, т. е. со всеми картами, лежащими ниже ее в том же столбике, и эта часть столбика пристраивается в низ какого-либо другого столбика. Перемещение возможно лишь в том случае, если выбранная карта имеет ту же масть и на единицу младше той карты, на которую она накладывается (в этой игре тузы имеют наименьшее старшинство, т. е. соответствуют единице, а наибольшее старшинство имеют короли). На рис. 17.2 изображен пример возможного хода. Если в результате такого перемещения освобождается верхняя, лежащая рубашкой вверх карта стопки, то ход завершается переворачиванием этой карты. В результате хода может также полностью опустошиться один из столбиков; тогда на любом из последующих ходов на освободившееся место можно перенести любого лежащего рубашкой вниз короля вместе со всеми накрывающими его картами. Если какой-то из тузов оказывается последней картой в одном из столбиков, то он перекладывается в верхнюю часть стола и дает начало счетной стопке для своей масти. После того как начата счетная стопка для какой-либо масти, в нее можно добавлять другие карты той же масти по мере того, как они оказываются последними в каком-нибудь столбике, но так, чтобы карты в счетной стопке шли в строго возрастающем порядке по старшинству. Заметим, что если последнюю карту столбика можно положить в счетную стопку, то медлить с этим не стоит, поскольку рано или поздно ее все равно придется туда положить, а до тех пор эта карта может лишь блокировать дальнейшие ходы с участием своего столбика.

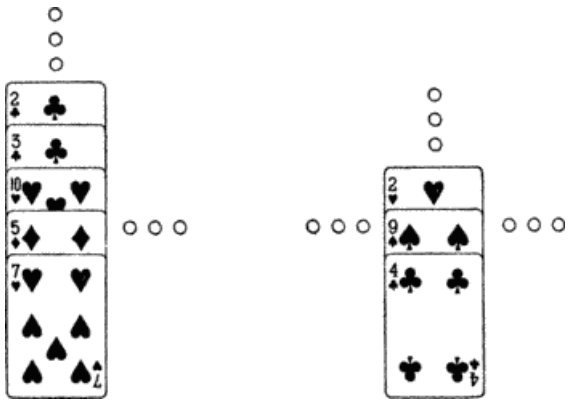


Рисунок 17.2. Пример возможного хода с перемещением карт из одного столбика в другой. Карты, начиная с тройки треф и ниже в том же порядке, накладываются поверх четверки треф. Остальные столбики не показаны.

Игра кончается, когда не остается ни одного допустимого хода и ни одну карту нельзя положить в счетную стопку. **Счет игры** равен суммарному числу карт в счетных стопках. Как выяснилось, эта игра весьма популярна в Лас Вегасе (известном также под названием Город Просаженных Получек). Колоду карт в казино можно получить по цене 1 долл. за карту (плюс 3 долл. за право начать игру), игрок же получает 5 долл. за каждую карту, вошедшую в счет. Таким образом, при счете 11 карт (за что причитается 55 долл.) игрок остается при своих, а каждая карта сверх того — его чистый выигрыш. Сомнительно, чтобы в казино действительно предлагались такие условия, но если это так, то мы вправе Подозревать, что владельцы должны иметь чудовищные прибыли. Каково в действительности ожидаемое число карт в счетных стопках? Насколько нечестными были бы при данных условиях доходы владельцев казино?

Анализ пасьянса

Каждой первоначальной раскладке соответствует в точности один оптимальный результат, хотя достичь его можно при различных последовательностях ходов. По-видимому, существует некоторая очень сложная вероятностная функция для вычисления ожидаемого значения оптимального результата. Но даже если бы удалось явно выписать эту функцию, она, несомненно, содержала бы столь большое количество членов, что вычислить ее было бы делом крайне затруднительным. Нельзя ли вместо этого попытаться сыграть достаточно много партий и извлечь интересующую статистику из полученных таким образом результатов? Эта идея применения моделирования для получения результата, который теоретически может быть непосредственно вычислен, уже встречалась в других главах книги именно потому, что, как и в данном случае, превращение компьютера в модель интересующего нас реального процесса оказывается весьма плодотворным. Какова необходимая последовательность действий при моделировании пасьянса? Во-первых, нужны подпрограммы для получения первоначальной раскладки, для проверки существования возможных ходов в данной позиции, для перемещения карт, для переворачивания верхней карты стопки, для перекладывания карты в счетную стопку — словом, процедуры, реализующие явный процесс раскладывания пасьянса. При помощи этих процедур можно вычислить результат любой заданной последовательности ходов. Для того чтобы найти оптимальный результат, реализуем на базе этих процедур **стратегию поиска**. После сдачи карт получаем некоторую начальную позицию. Стратегия поиска состоит в выполнении для каждой позиции, получающейся в ходе игры, следующих действий:

Подсчитать, сколько возможных ходов имеется для данной позиции. Их всегда не более семи.

Если возможных ходов нет, то данная последовательность ходов закончена, и можно записать ее счет. Установить новую текущую позицию, взяв верхний элемент со стека позиций, и возвратиться к началу цикла. Если стек позиций пуст, то закончить поиск.
Если есть только один ход, то выполнить его и вернуться к началу цикла.
Если ходов несколько, то упорядочить их (способ упорядочения не играет роли), записать в стек позиций текущую позицию, упорядоченный список возможных ходов, а также тот факт, что первый ход уже сделан. Выполнить первый ход и возвратиться к началу цикла. Заметим, что в шаге один, если позиция была взята со стека, неявно предполагается, что при подсчете числа возможных ходов вначале всегда ищется частично завершенный список возможных ходов.
Эта стратегия осуществляет **поиск «сначала в глубину»** по всем возможным последовательностям ходов с запоминанием еще не исследованных позиций в стеке. Поскольку проверяются все возможные последовательности ходов, то данная стратегия гарантирует нахождение некоторой, быть может не единственной, оптимальной последовательности.
Одна из неприятных для игрока особенностей этого пасьянса состоит в том, что довольно часто сразу после первоначальной раскладки не оказывается вообще ни одного возможного хода. Раскладка карт — это лишь некоторый сложный способ тасования. Несмотря на значительную вероятность быстрого окончания игры, следует ожидать все же, что дерево позиций может вырасти до очень больших размеров. Но дерево это на деле является графом, поскольку к одной и той же позиции вполне можно прийти после различных последовательностей ходов. Если некоторая позиция уже была однажды исследована, нет нужды рассматривать ее снова. Оптимальный результат игры не зависит от порядка ходов или от конкретной последовательности ходов, ведущей к нему. Если все уже исследованные позиции запоминать, то каждую вновь возникшую позицию можно во избежание повторной обработки сравнивать с этими старыми позициями. Сохранять нужно только сами позиции, без списка возможных ходов. Естественно, при этом возникает проблема поиска в множестве старых позиций.

Тема. Напишите программу для нахождения среднего значения и стандартного отклонения оптимального счета в данном пасьянсе. Покажите, что число рассмотренных игр обеспечивает достоверность полученных статистических результатов. Подсчитайте также, если сумеете, среднее число ходов и среднее число точек принятия решения на пути к оптимальному результату. Единственный входной параметр программы — число пасьянсов, которые нужно разложить. Вывод обязательно должен содержать требуемую статистику, но иногда оказываются полезными и другие данные. В частности, можно выдать информацию о распределении памяти для хранения старых позиций.

Указания исполнителю. Организация карт в представлении позиции, а также способ хранения старых позиций решающим образом определяют уровень эффективности программы. Если позиция является текущей или находится в стеке, то для нее

должно быть известно состояние всех стопок и соответствующих им столбиков, а также счетных стопок. Позиция в стеке должна содержать, кроме того, список еще не проверенных ходов. Для ускорения поиска возможных ходов нужно иметь вектор состояния для всех карт, в котором должны быть такие признаки карты: лежит ли она рубашкой вверх, находится ли уже в счете, лежит ли рубашкой вниз внутри столбика (какого именно?), лежит ли рубашкой вниз в низу столбика (какого именно?). Быть может, вы сумеете придумать другую структуру данных, обеспечивающую быстрое нахождение возможных ходов. В любом случае при запоминании старой позиции часть информации, как уже использованную, можно отбросить, экономя таким образом память.

Здесь потребуется два специальных алгоритма. Во-первых, как реализовать тасование карт в ЭВМ? Вот процедура, предложенная Кнутом. Пусть ***rand52*** — функция, генерирующая случайные целые числа, равномерно распределенные в отрезке от 1 до 52. Поместите все карты в массив ***карта*** длины 52; как в нем расположены карты вначале — не имеет значения. После этого для i от 1 до 52 поменяйте местами элементы ***карта***[i] и ***карта***[***rand52***], каждый раз заново обращаясь к функции ***rand52***. Одного такого тасования будет достаточно.

Во-вторых, как находить старые позиции? Это классическая задача поиска в растущей базе данных. Очевидным решением тут представляется хеш-таблица, где ключом поиска служит вся позиция. Поскольку полное сравнение двух позиций на равенство, скорее всего, обойдется слишком дорого, то разумно, по-видимому, будет применить виртуальный хеш-код. Пространство, отведенное для хранения старых позиций, может переполняться, поэтому вы должны уметь время от времени освобождать его. Наилучший способ освобождения памяти состоит, пожалуй, в том, чтобы иметь при каждой позиции счетчик, показывающий, сколько раз к ней обращались, и отбрасывать каждый раз те позиции, которые участвовали реже всего. Другой способ, который можно использовать и в сочетании с первым, — хранить список всех старых позиций и всякий раз, когда ищется какая-либо позиция, перемещать ее в голову списка. Когда придет время отбросить часть позиций, то кандидатами на уничтожение будут позиции в хвосте списка, поскольку к ним дольше всего не было обращений. Принятый вами способ отбрасывания старых позиций окажет влияние на выбор стратегии поиска, и наоборот. Заметим, что, хотя алгоритм отбрасывания старых позиций и не влияет на правильность программы анализа пасьянсов, тем не менее он может существенно ее замедлить.

Развитие темы. Наиболее очевидное расширение задачи — применить этот метод анализа к другим пасьянсам. Осуществление этой идеи не связано с какими-либо трудностями, если только во время игры не происходит тасования карт. На свете существуют сотни пасьянсов, и ни один из них, насколько нам известно, не подвергался мало-мальски серьезному анализу. На этой задаче можно также изучать зависимости общего числа исследуемых позиций от объема памяти и критерия отбрасывания старых позиций. Иначе говоря, чем исследовать пасьянс

при помощи методов поиска, используем задачу о пасьянсе для изучения методов поиска.

Тема 18: Пакет Для Алгебраических Вычислений

Основная трудность, с которой сталкивается программист в большинстве языков программирования, — необходимость при записи вычислений разбивать свои уравнения на мелкие части. Так, если требуется производная, то программист должен записать исходную функцию, снять с полки учебник по математическому анализу, применить изложенные там правила и затем записать получившуюся производную. Однако многие операции можно выполнить на символьном уровне, по крайней мере в случае многочленов, если представлять их подходящим образом. Некоторые программы оказались бы совсем ненужными, если бы ЭВМ могла оперировать с многочленами.

Объекты, с которыми мы будем работать, — это **рациональные функции**. Их можно определить рекурсивно.

Пусть c — любая вещественная **константа**. Тогда c — рациональная функция.

Пусть x — любая **переменная**. Тогда x — рациональная функция.

Пусть p и q — любые рациональные функции. Тогда p + q, p - q, -p, pq, p/q и (p) все суть рациональные функции. При делении рациональных функций производится упрощение, так чтобы остался только один знак деления. Правила этого упрощения хорошо знакомы школьникам, изучающим алгебру. Пусть p — любая рациональная функция, а c — целочисленная константа. Тогда p^c — рациональная функция. Если c отрицательна, образуйте рациональную функцию 1/p|c| и упростите деление как выше.

Рациональными функциями являются только те объекты, которые получаются путем применения конечного числа приведенных выше правил.

Кроме определения рациональных функций мы должны описать, как будет выглядеть их запись в качестве исходных данных и на выходе и как вызывать операции.

Рациональные функции в качестве исходных данных будут похожи на выражения в стандартном языке программирования. Константы могут изображаться любой последовательностью десятичных цифр с десятичной точкой; если десятичная точка отсутствует, то константа автоматически будет целочисленной. В силу правил образования рациональных функций константы не имеют знака, за исключением констант в показателе степени. Переменная выглядит как идентификатор и может быть любой цепочкой из больших и малых литер алфавита. Из-за ограничений на выбор литер в ЭВМ умножение будет изображаться знаком *, а возведение в степень — знаком ↑. Так, рациональную функцию

$2xy + (x^2 + y^2)^3$

можно записать как

2 * X * Y + (X ↑ 2 + Y ↑ 2) ↑ 3

Некоторые другие имена, в частности имена функций, также будут идентификаторами.

Для манипуляций с рациональными функциями нам нужны некоторые команды, чтобы пользователь мог получать ответы на вопросы, на которые не удается ответить с помощью традиционных языков программирования. Для этого нам

понадобится обозначать рациональные функции идентификаторами. Самое фундаментальное действие такое:

Установить f равным p; Эта команда приводит к тому, что имя рациональной функции f (мы будем писать сокращенно — **имя функции**) получает в качестве значения рациональную функцию p. Эта операция — символьная; она не вызывает вычисления p. Если некоторый идентификатор f использован как имя функции, то его нельзя употреблять в последующих командах в качестве переменной; надо иметь в виду, что во время интерпретации потребуется таблица имен, значений и использований. Вместо рациональной функции p может стоять имя функции; в этом случае f получает значение, которое в данный момент имеет p. Все команды заканчиваются точкой с запятой. Примеры описываемой команды:

Установить P равным z*x↑2 + 3.5; Установить fpt равным P;

Бóльшая часть остальных команд выполняет некоторую операцию над своими операндами и помещает результат в качестве значения некоторого имени функции.

Установить f равным сумме p и q; Образовать алгебраическую сумму p и q и записать полученное значение под именем f. Во всех командах исходные данные записываются в свободном формате — границы строк (или перфокарт) несущественны; единственным разделителем команд служит точка с запятой. Операндами могут быть имена функций; в таком случае в операциях используются значения, приписанные этим именам.

Установить f равным разности p минус q; Образовать алгебраическую разность p и q и записать полученное значение под именем f.

Установить f равным произведению p и q; Образовать алгебраическое произведение p и q и записать результат под именем f.

Установить f равным частному при делении p на q; Образовать алгебраическое частное p и q и записать результат под именем f. Для выполнения этой операции не нужно привлекать алгоритм деления многочленов, так как рациональная функция может включать один знак деления. Последующие знаки деления могут быть устранены при помощи школьной алгебры.

Установить f равным p в степени c; Рациональная функция p возводится в степень c, и результат записывается под именем f. Показатель степени c должен быть целым числом или именем функции с постоянным значением; если c отрицательно, результатом будет $1/p^{|c|}$.

Установить f равным p с заменой x на q; Заменить каждое вхождение переменной x в p на q и записать полученное значение под именем f. Отметим, что в результате подстановки переменная x может снова возникнуть в f, но ее не следует вновь заменять на q.

Установить f равным производной p по x; Вычислить производную dp/dx и записать полученное значение в f. Конечно, идентификатор x должен быть переменной или именем функции, состоящей из одной переменной.

Напечатать p; Напечатать рациональную функцию p в удобном для чтения виде.

Конец; Завершение последовательности команд.

При реализации команды печати мы сталкиваемся с трудностью, присущей всем программам алгебраических преобразований. При вычислениях функции, как правило, становятся очень сложными. Вместе с тем человек хотел бы получить результаты в достаточно простом виде. Рациональные функции записывают обычно в виде дроби, числитель и знаменатель которой представляют собой сумму членов, включающих только операции умножения и возведения в степень. В каждом таком одночлене все константы перемножены и образуют числовой коэффициент (первый сомножитель), переменные упорядочены (часто по алфавиту) и все степени одной переменной объединены так, чтобы каждая переменная встречалась лишь один раз. Если числовой коэффициент оказывается отрицательным, то такой одночлен должен вычитаться из предыдущих, а не прибавляться к ним. Если коэффициент окажется равным нулю или единице, то весь одночлен или коэффициент должен быть опущен. Если показатель степени отрицателен, то одночлен фактически есть дробь; в этом случае нужно освободиться от знаменателя с помощью стандартных алгебраических правил суммирования дробей. И наконец, следует приводить подобные члены, т. е. объединять одночлены, имеющие одинаковые наборы переменных и степеней, с соответствующим изменением коэффициентов.

Все эти преобразования можно выполнять путем приведения функции к некоторому каноническому внутреннему представлению. В нашем случае можно выбрать такое представление, чтобы рациональные функции были почти готовы для печати; результат каждой операции должен преобразовываться к стандартному виду. Можно и по-другому выбрать внутреннее представление, так, чтобы операция печати преобразовывала представление функции, когда это необходимо. Однако требуемый для такого преобразования объем работы может быть сколь угодно большим. Независимо от выбранного метода для целей упрощения нужно различать целые и вещественные константы, с тем чтобы погрешность машинной арифметики не помешала распознаванию нулевых и единичных значений. Заметьте также, что возведение в первую степень обычно опускают. На рис. 18.1 показаны простая программа и ее результат.

```
Установить f равным (x↑2 + y↑2)↑2 + 3*x*y;
Установить g равным (x + y)↑3 − 4;
Установить h равным произведению f и g;
Установить ааа равным f с заменой y на 2;
Установить bbb равным частному от деления ааа на h;
Напечатать g;
Напечатать bbb;
Конец;
На печать выводится следующий результат:
x↑3 + 3*x↑2*y + 3*x*y↑2 + y↑3 − 4
(x↑4 + 8*x↑2 + 6*x + 16)/(x↑7 + 3*x↑6*y + 5*x↑5*y↑2 + 7x↑4*y↑3 − 4x↑4*y − 4*x↑4 + 7*x↑3*y↑4 + 9*x↑3*y↑2 + 5*x↑2*y↑5 + 9*x↑2*y↑3 − 8*x↑2*y↑2
+ 3*x*y↑6 + 3*x*y↑4 − 12*x*y + y↑7 − 4*y↑4)
```

Рисунок 18.1. Пример программы и ее результат.

Тема. Напишите программу для работы с рациональными функциями, реализующую описанные выше возможности. Исходными данными должен быть список команд в свободном формате, а результатом — рациональные функции в формате, удобном для чтения. Переменные, константы и слова, входящие в запись команды, не должны переходить с одной строки на другую, но для самих команд и

рациональных функций это вполне допустимо. Определение «удобного» формата печати довольно туманно, зато здесь вы можете продемонстрировать свое искусство в удовлетворении тех потребностей пользователей, которые они сами не могут сформулировать. Не забывайте про доказательство правильности результатов, выдаваемых программами. Одна из важных черт программ работы с рациональными функциями — это способность точно выполнять арифметические действия над целыми числами; позаботьтесь об этом в вашей программе.

Рекомендации исполнителю. Чтение программой команд и рациональных функций требует привлечения некоторых простых методов компиляции, в частности лексического анализа для распознавания символов и синтаксического анализа для построения внутреннего представления. Необходимые сведения содержатся в литературе, указанной в других главах. В процессе выполнения программы вам придется поддерживать расширяющуюся таблицу имен и значений; для этого также имеется простой метод. Самая трудная часть реализации — это выбор внутреннего представления для рациональных функций. Они, несомненно, должны представляться с помощью некоторого варианта списочной или древовидной структуры, но какого именно?
Одним из возможных представлений является стандартное арифметическое дерево, содержащее переменные и константы в листьях, а операции — во внутренних узлах. Такая форма представления особенно подходит для подстановки и алгебраических операций, но для печати она слишком беспорядочна. Другая возможность — дерево, содержащее на верхнем уровне числитель и знаменатель, на следующем уровне — одночлены и на еще более низком уровне — сомножители. Такое дерево будет легко напечатать, но с ним трудно работать. Что бы вы ни выбрали, не забывайте копировать структуры данных при выполнении подстановки, иначе более позднее изменение в подставляемой функции повлияет также и на функцию, в которую она подставлялась.

Развитие темы. В настоящее время широко используются многие системы алгебраических преобразований. Как правило, в их основе лежат функции, подобные описанным выше. Дальнейшее развитие происходит по трем направлениям: введение новых типов данных, новых операций и эвристических процедур, предназначенных для выполнения действий с нечетко определенным результатом. Новые типы данных взаимосвязаны с новыми операциями. Можно, например, добавить к рациональным функциям тригонометрические, показательные функции и логарифмы. В таком случае надо будет изменить операцию возведения в степень, чтобы она допускала любой операнд в качестве показателя степени, кроме того, понадобится операция логарифмирования, в которой будет указываться основание логарифмов и логарифмируемая функция. Отметим, что при введении новых типов данных и операций следует убедиться в замкнутости пространства функций, которые могут быть порождены произвольной последовательностью операций. Замкнутость означает, что всякую функцию,

которую можно породить, можно также в принципе записать в команде Установить.

Для многих важных математических операций не существует методов, которые позволяли бы всегда вычислять результат в символьном виде. Важное место среди них занимает интегрирование. Хотя любая рациональная функция имеет **неопределенный интеграл**, простой пример функции 1/x (неопределенный интеграл от нее — ln x) показывает, что нам не надо далеко ходить за функциями, нарушающими границы замкнутого пространства рациональных функций. Расширение пространства функций путем добавления показательных функций и логарифмов, как предложено выше, лишь обостряет проблему. Не решает проблемы даже использование **определенного интеграла**, поскольку результат определенного интегрирования может и не быть константой, если подинтегральное выражение содержит переменные, отличные от переменной интегрирования, или если пределы интегрирования не константы. Символьные интеграторы были одними из первых программ, написанных для демонстрации «интеллектуального» поведения ЭВМ. Если вы будете работать над предлагаемой задачей в два или три раза дольше, то сможете создать примитивный интегратор.

Введение новых функций создает еще одну проблему. Для более сложных функций, которые теперь можно построить, не существует стандартного формата вызова. Кроме того, выбор применяемых законов упрощения становится нелегким делом. Поскольку теперь применимо гораздо больше алгебраических законов — тригонометрические тождества, законы, связывающие показательные и логарифмические функции, законы о константах, — может случиться, что программа будет тратить большую часть времени на упрощение внутреннего представления выражений. Упрощение с целью облегчить человеку понимание результатов — очень важная и сложная тема; от программиста требуется немалое искусство, чтобы успешно реализовать упрощение.

Тема 19: Ошибки при работе с плавающей точкой

Многие из методов, которые сейчас изучаются в средней школе, создавались величайшими математиками в течение столетий. Среди них — методы решения системы линейных уравнений, которые неявно включают методы обращения квадратных матриц. Начинающий алгебраист, изучая эти алгоритмы, может усомниться в том, что они всегда будут работать; но, испробовав метод на двух-трех примерах* наш скептик отбросит всякие сомнения. Он даже себе не представляет, какой его ждет удар: программа, написанная им в соответствии с простым и обоснованным алгоритмом, дает совершенно неверные результаты. Разве можно заподозрить, чтобы метод обращения матриц, придуманный королем математиков Гауссом, оказался несостоятельным?

Прежде всего освежим в памяти основные положения. Матрица — это квадратный массив вещественных чисел, в котором по горизонтали и вертикали располагается по $n \geq 1$ элементов. Произведение C матрицы A справа на матрицу B записывается в виде $C = AB$ и задается формулой

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad 1 \leqslant i \leqslant n, \quad 1 \leqslant j \leqslant n.$$

Здесь подразумевается, что A, B и C — матрицы размера $n \times n$. Умножение некоммутативно; можно найти такие матрицы A и B, что $AB \neq BA$. **Обратной матрицей** к матрице A будет такая матрица A^{-1}, что

$AA^{-1} = A^{-1}A = I$

где I — единичная матрица, определяемая формулами $I_{ii} = 1$ и $I_{ij} = 0$ для $i \neq j$. Большинство матриц имеет обратные, но не все. К сожалению, простейший способ обнаружить такие **вырожденные** матрицы состоит в том, чтобы попытаться вычислить обратную матрицу и потерпеть неудачу.

Как вычислить обратную матрицу? Следующий алгоритм принадлежит Гауссу.

Во-первых, положите матрицу X равной матрице I. В процессе вычислений матрица A будет в конце концов преобразована в I, матрица X, которая изначально была единичной матрицей, станет обратной к матрице A.

Для каждого столбца A, начиная со столбца 1 слева и кончая столбцом n справа, выполните следующее: Обозначим столбец, который будет обрабатываться на каждом этапе, символом j.

Пусть $M = \max_{j \leqslant i \leqslant n} |A_{ji}|$ есть наибольший по абсолютной величине элемент в столбце j ниже строки j - 1. Если M равно нулю, то A — вырожденная матрица и продолжать обращение не имеет смысла. В противном случае поменяйте местами в обеих матрицах A и X строку j и строку, в которой находится M. И наконец, разделите каждый элемент в строке j матриц A и X на новое значение A_{jj}.

Теперь для всех строк i, $i \neq j$, выполните все вычитания:

$A_{ik} = A_{ik} - A_{ij}A_{jk}$, $j \leq k \leq n$,

$X_{ik} = X_{ik} - A_{ij}X_{jk}$, $1 \leq k \leq n$.

Результатом всех этих поэлементных вычитаний будет вычитание из строки i строки j с коэффициентом A_{jj} в обеих матрицах A и X. После выполнения этого шага для всех j все элементы сверху и снизу от A_{jj} станут нулевыми, а сам элемент

A_{jj} будет равен единице. В матрице A не нужно выполнять вычитания слева от столбца j, поскольку все элементы строки j слева от A_{jj} равны нулю.

Для любого алгебраиста будет одно удовольствие доказать, что этот алгоритм всегда работает правильно и после его остановки X = A-1, если матрица A невырожденна. Вы едва ли найдете алгоритм, более приспособленный для структурной реализации. Почему бы нам, исключительно ради забавы, не провести небольшую проверку? **Матрица Гильберта H^n порядка n** определяется формулой

$$H_{ij}^n = 1/(i+j-1), \quad 1 \leqslant i \leqslant n, \quad 1 \leqslant j \leqslant n.$$

Вычислите обратную к H^n матрицу для n = 1, 2, ..., 20, 25, 30, 35, 40, 45, 50. Вы, несомненно, понимаете, что результат получится не вполне точным из-за небольших погрешностей машинной арифметики, но он должен быть очень близок к точной обратной матрице. Мерой погрешности служит **левая остаточная матрица** $L = (H^n)^{-1}H^n - I$ и **правая остаточная матрица** $R = H^n(H^n)^{-1} - I$; обе эти матрицы должны быть нулевыми, но, вероятно, не будут.

Конечно, если бы все элементы матриц L и R были порядка, скажем, 10^{-20}, то мы бы не имели забот. Для всех практических целей 10^{-20} есть нуль, если элементы исходной матрицы равны в среднем 1/50 или больше. Существует, однако, точный способ оценки величины остаточных матриц L и R. Определим **норму по строкам** матрицы A как

$$|A|_r = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |A_{ij}|.$$

Добавьте к своей программе, которая вычисляет обратную к матрице Гильберта, подпрограмму, печатающую таблицу $|L|_r$ и $|R|_r$ для каждой обратной матрицы. Проверьте вашу программу на отсутствие ошибок. Не сможете ли вы теперь объяснить, почему остаточные матрицы столь велики. *Уверены* ли вы в правильности программы?

Ваша программа правильна; причина неполадок — погрешность машинной арифметики. Матрицы Гильберта внешне выглядят вполне безобидно, однако они специально предназначены для демонстрации накопления ошибок в длинном ряду взаимосвязанных вычислений. Вы, быть может, считаете источником бед то, что ваш компьютер хранит недостаточное число цифр вещественных чисел. На многих ЭВМ имеется арифметика **двойной точности**. Предусмотрев в своем алгоритме двойную точность, вы сможете улучшить ситуацию, но заведомо не сможете полностью решить проблему. Весь этот этюд посвящен изучению влияния арифметики ограниченной точности на алгоритмы, которые являются абсолютно точными для «действительных» чисел (как их понимают математики). Прикладные математики и специалисты по численным методам в программистских лабораториях тратят большую часть времени на изменение теоретических алгоритмов, чтобы они могли работать на реальных ЭВМ.

Тема. Запрограммируйте алгоритм обращения матриц и проверьте его на матрицах Гильберта указанных выше порядков. Напечатайте таблицу или начертите график зависимости $|L|_r$ и $|R|_r$ от порядка n матрицы H^n. Если используемый вами язык допускает выбор точности чисел, то повторите

вычисление обратных матриц с большей точностью, чтобы увидеть, улучшится ли в результате таблица или график ошибок. (Мудрый программист так составит программу, чтобы изменение точности достигалось путем замены небольшого числа деклараций.) Следите также за числом фактических перестановок строк при выборе ведущего элемента; оно будет показывать, насколько плохо алгоритм согласуется с теорией.

Указания исполнителю. В этом этюде требуется прямая реализация алгоритма. Единственная трудность — это проверка соответствия программы теоретическим определениям и алгоритму. Не делайте над алгоритмом никаких оптимизирующих преобразований; вы изучаете, до чего можно дойти, если слепо следовать советам математиков, забыв о важнейшем положении — о точности вычислений.

Развитие темы. Если в достаточной степени расширить задачу, то она послужит основой семестрового курса методов вычислений. Тем не менее вы можете получить дополнительную информацию о поведении ошибок, если вычислите |L| и |R| с использованием других норм, отличных от нормы по строкам, например нормы по столбцам:

$$|A|_c = \max_{1 \leqslant j \leqslant n} \sum_{i=1}^{n} |A_{ij}|.$$

Ниже определены нормы L_1, L_2 и L_∞:

$$|A|_1 = \sum_{ij} |A_{ij}|, \quad |A|_2 = \sqrt{\sum_{ij} A_{ij}^2}, \quad |A|_\infty = \max_{i,\,j} |A_{ij}|.$$

Дополните значениями этих норм вашу таблицу анализа ошибок. Наблюдаются ли какие-либо существенные отличия одной нормы от остальных по форме или приблизительному положению кривой ошибок?

Тема 20: Арифметические вычисления с высокой точностью

Математику часто считают сухой наукой, однако и математику творили люди. Одной из самых печальных была судьба Уильямса Шенкса, жившего в девятнадцатом веке и посвятившего себя вычислению числа π с высокой точностью. Закончив многолетний труд, Шенкс в 1837 г. опубликовал значение π до 707-го десятичного знака, впоследствии исправив некоторые знаки. Может быть, надо счесть за благо, что Шенкс умер в 1882 г., поскольку в 1946 г. было показано, что его вычисления ошибочны начиная с 528-го десятичного знака. Фактически Шенкс не продвинулся дальше своих предшественников.

Полученное Шенксом значение было проверено, вероятно, с помощью механических устройств, а компьютер был впервые использован для вычисления π только в 1949 г.; это была машина ENIAC. Даже тогда проект был монументальным. Джорж У. Рейтуиснер писал: «Поскольку получить машину в рабочее время было практически невозможно, мы воспользовались разрешением выполнить эту работу за 4 выходных дня в период летних отпусков, когда ENIAC стоял без дела». Собственно вычисления (не программирование!) заняли 70 часов: было получено несколько больше 2 000 цифр. Все это время приходилось постоянно обслуживать компьютер, поскольку из-за ограниченности его возможностей требовались постоянная перфорация и ввод промежуточных результатов. Те первые программисты так же далеки от нынешних, как Шенкс далек от них.

Как бы мы стали вычислять π? Во-первых, необходимо выражение, которое можно вычислять. Ряд

$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots$

довольно прост для понимания, но он ужасно медленно сходится. Гораздо лучше ряд для арктангенса

$\operatorname{arctg} x = x - x3/3 + x5/5 - x7/7 + \ldots, |x| \leq 1$

Объединим его с формулой сложения для тангенса

$\operatorname{tg}(a + b) = (\operatorname{tg} a + \operatorname{tg} b)/(1 - \operatorname{tg} a \cdot \operatorname{tg} b)$

и выберем a и b так, чтобы $\operatorname{tg}(a + b) = 1 = \operatorname{tg} \pi/4$. (Учитывая, что $\operatorname{tg}(\operatorname{arctg} x) = x$ для $-\pi/2 < x < \pi/2$, можно взять, например, $a = \operatorname{arctg}(1/2)$, $b = \operatorname{arctg}(1/3)$.)

Тогда

$\operatorname{arctg}(\operatorname{tg}(a + b)) = a + b = \operatorname{arctg} 1 = \pi/4,$

и теперь можно использовать приведенный выше ряд для нахождения a и b. На практике чаще всего используются следующие суммы:

$\pi/4 = 4 \operatorname{arctg}(1/5) - \operatorname{arctg}(1/239),$

$\pi/4 = 8 \operatorname{arctg}(1/10) - 4 \operatorname{arctg}(1/515) - \operatorname{arctg}(1/239),$

$\pi/4 = 3 \operatorname{arctg}(1/4) + \operatorname{arctg}(1/20) + \operatorname{arctg}(1/1985).$

Теперь мы собираемся просуммировать эти ряды на ЭВМ. Как известно, все, что нужно для суммирования, — это простой итерационный цикл, но тут возникает одна проблема. Точность вычислений на ЭВМ ограничена, а весь смысл этого упражнения в том, чтобы найти много-много цифр числа π, значительно превзойдя обычную точность. Первое, что приходит в голову, — промоделировать ручные методы выполнения арифметических действий. Будем представлять числа очень большими целочисленными массивами (по одной десятичной цифре в каждом

элементе), тогда ясно, как составить программы сложения, вычитания и умножения. Запрограммировать ручной метод деления несколько сложнее, но все же возможно. Неприемлемым, однако, оказывается время выполнения алгоритмов. Хотя на это редко обращают внимание, но при ручных методах для умножения или деления *n*-значных чисел требуется время, пропорциональное n^2. Если речь идет об операциях над числами из тысяч цифр, то такие расходы будут нам не по карману. К счастью, имеются лучшие алгоритмы.

Как можно быстро умножать?

Алгоритм быстрого умножения Тоома—Кука, описываемый Кнутом, зиждется на четырех основных идеях. Вот первая из них. Пусть нам известен способ выполнения некоторой операции над исходными данными размера n за время T(n). Если эту операцию удастся разбить на r частей, выполнение каждой из которых займет менее чем T(n)/r шагов, то такое разбиение позволит улучшить общее время, если, конечно, считать, что вспомогательные организационные расходы не сведут экономию на нет. Пусть, далее, каждая из r частей есть применение того же алгоритма к исходным данным длины n/r и каждая часть может быть разбита аналогичным образом. Тогда можно продолжать это разбиение, пока мы не получим столь короткие исходные данные, что вычисления для них станут тривиальными и займут лишь небольшой фиксированный отрезок времени. Этот принцип **разделяй и властвуй** обычно дает выигрыш во времени работы алгоритма по крайней мере в log n раз; так, классический метод умножения требует времени n^2, и его можно свести к $n^{1+7/\sqrt{4\log_2 n}}$, что существенно лучше при больших n (не забывайте, что у обеих функций стоимости имеются постоянные множители).

Остальные три идеи касаются чисел и действий над многочленами. Во-первых, заметим, что, если число U имеет длину n битов и записывается в двоичном виде как

$u_{n-1}u_{n-2}\ldots u_2u_1u_0$,

причем n делится на r + 1, то U можно также записать в виде

$U_r 2^{rn/(r+1)} + U_{r-1}2^{(r-1)n/(r+1)} + \ldots + U_1 2^{n/(r+1)} + U_0$,

где каждое U_i есть блок из n/(r + 1) битов исходного представления U. Фактически $U = U(2^{n/(r+1)})$, где многочлен $U(x)$ есть

$U_r x^r + U_{r-1}x^{r-1} + \ldots + U_1 x + U_0$.

Во-вторых, мы видим, что если U и V — два n-разрядных числа, записанных в виде такого многочлена, то их произведение W дается формулой

$W = UV = U(2^{n/(r+1)})V(2^{n/(r+1)}) = W(2^{n/(r+1)})$

и если бы мы смогли найти хотя бы коэффициенты $W(x)$, то вычислить W по W было бы сравнительно просто; для этого понадобились бы только сдвиги, сложения и умножения чисел из n/r битов. В-третьих, к счастью, $W(x)$ — многочлен степени 2r и его можно найти с помощью интерполяции его значений в точках 0, 1, 2, ..., 2r-1, 2r. Эти значения равны просто $U(0)$, $V(0)$, $U(1)$, $V(1)$, ..., $U(2r)$, $V(2r)$. Более того, для вычисления всех этих многочленов и интерполяции требуется умножать числа

только из n/r битов. Представляется, что эти действия подпадают под принцип «разделяй и властвуй».

Алгоритм Тоома—Кука весьма сложен, поэтому мы не будем подробно объяснять его; за этим можно обратиться к книге Кнута. Все же необходимо сообщить основные идеи и обозначения. Длинные числа должны быть как-то представлены; будем писать [p, u] для обозначения числа u из p битов. Вероятно, внутреннее представление [p, u] будет некоторой разновидностью списка или цепочки. Кроме основного алгоритма нам понадобятся подпрограммы для сложения и вычитания длинных чисел (используйте стандартный ручной метод сложения слева направо), умножения длинного числа на короткое (небольшое) число, деления длинного числа на короткое, сдвига длинного числа путем приписывания нулей справа и для разбиения длинного числа [p, u] на более короткие длинные числа $[p/(r + 1), u_r]$, $[p/(r + 1), u_{r-1}]$, ..., $[p/(r + 1), u_0]$, как описано выше. Кроме подпрограмм, работающих непосредственно с числами, алгоритм использует четыре стека для хранения промежуточных частичных результатов и несколько временных переменных, поэтому требуются подпрограммы для выполнения некоторых действий над стеком, а также подпрограммы для выделения и освобождения памяти под длинные числа. При написании всяческих вспомогательных подпрограмм черновой работы может оказаться предостаточно.

Алгоритм быстрого умножения Тоома—Кука

Исходными данными служат два положительных длинных числа [n, u] и [n, v]; результатом — их произведение [2n, uv]. Используются четыре стека U, V, W и C, в которых при выполнении алгоритма будут храниться длинные числа, и пятый стек, содержащий коды временно приостановленных операций (имеется всего три кода, и для их представления можно воспользоваться малыми целыми числами). Массивы q и r целых чисел имеют индексы от 0 до 10; необходимо выделить память для этих двух массивов и для еще нескольких временных переменных, упомянутых в алгоритме.

1. (Начальная установка.) Сделать все стеки пустыми. Присвоить K значение 1, q_0 и q_1 — значение 16, r_0 и r_1 — значение 4, Q — значение 4 и R — значение 2.

2. (Построение таблицы размеров). Пока $K < 10$ и $q_{K-1} + q_K \geq n$, выполнять следующие вычисления. Изменить K на K + 1, Q — на Q + R; если $(R + 1)^2 \leq Q$, то изменить R на R + 1; установить q_K равным 2^Q и r_Kравным 2^R. Если цикл оканчивается из-за K = 10, то остановиться, выдав сообщение об ошибке — число битов n слишком велико, массивы q и r переполнились. В противном случае присвоить k значение K. Поместить $[q_K + q_{K-1}, v]$ и за ним $[q_{K-1} + q_K, u]$ в стек C (вероятно, потребуется добавить к [n, u] и [n, v] слева нули). Поместить в управляющий стек код стоп.

3. (Главный внешний цикл.) Пока управляющий стек не пуст, выполнять шаги с 4-го по 18-й. Если на этом шаге управляющий стек окажется пустым, то остановиться с сообщением об ошибке; в управляющем стеке должен быть по крайней мере один элемент.

4. (Внутренний цикл разбиения и и v.) Пока к > 1, выполнять шаги с 5-го по 8-й.
5. (Установка параметров разбиения.) Установить k равным k - 1, s равным q_k, t равным r_k и p равным $q_{k-1} + q_k$.
6. (Разбиение верхнего элемента стека С.) Длинное число $[q_k + q_{k+1}, u]$ на вершине С следует рассматривать как t + 1 чисел длиной s битов каждое. Разбить $[q_k + q_{k+1}, u]$ на длинные числа $[s, U_t]$, $[s, U_{t-1}]$, ..., $[s, U_1]$, $[s, U_0]$. Эти t + 1 чисел являются коэффициентами многочлена степени t, который следует вычислить в точках 0,1, ..., 2t - 1, 2t no правилу Горнера. Для i = 0, 1, ..., 2t - 1, 2t вычислить $[p, X_i]$ по формуле
$(...([s, U_t]i + [s, U_{t-1}])i + ... + [s, U_1])i + [s, U_0]$
и сразу поместить $[p, X_i]$ в стек U. Для выполнения умножений можно использовать подпрограмму умножения длинных чисел на короткие; никакой промежуточный или окончательный результат не потребует более p битов. Удалить $[q_k + q_{k+1}, u]$ из стека С.
7. (Продолжение разбиения.) Выполнить над числом [m, v], находящимся сейчас на вершине стека С, ту же последовательность действий, что на шаге 6; полученные числа $[p, Y_0]$, ..., $[p, Y_{2t}]$ поместить в стек V в порядке получения. Не забудьте удалить вершину стека С.
8. (Заполнение заново стека С.) Попеременно удалять (2t раз) вершины стеков V и U и помещать эти значения в стек С. В результате значения, вычисленные на шагах 6 и 7, будут помещены, чередуясь, в стек С в обратном порядке. После выполнения этого перемешивания верхняя часть стека С, рассматриваемая *снизу вверх*, будет иметь вид $[p, Y_{2t}]$, $[p, X_{2t}]$, ..., $[p, Y_0]$, $[p, X_0]$, на вершине будет $[p, X_0]$. Поместить в управляющий стек один код операции *интерполировать* и 2t кодов операции *сохранять* и вернуться к шагу 4.
9. (Подготовка к интерполяции.) Присвоить к значение 0. Выбрать два верхних элемента стека С и поместить их в обычные переменные и и v. Оба числа и и v будут состоять из 32 битов. Используя некоторую другую подпрограмму умножения, вычислить [64, w] = [64, uv]. Это умножение можно выполнить аппаратно или с помощью подпрограммы, как вы найдете нужным.
10. (Интерполяция при необходимости.) Выбрать вершину управляющего стека в переменную А. Если значение А есть *интерполировать*, то выполнить шаги с 11-го по 16-й, в противном случае перейти к шагу 17,
11. (Организация интерполяции.) Поместить [m, w] в стек W (это может быть значение, полученное на шаге 9 или 16). Присвоить s значение q_k, t — значение r_k, p — значение $q_{k-1} + q_k$. Обозначим верхнюю часть стека W, рассматриваемую *снизу вверх*, как
$[2p, Z_0]$, $[2p, Z_1]$, ..., $[2p, Z_{2t-1}]$, $[2p, Z_{2t}]$,
последнее из этих значений — на вершине стека.
12. (Внешний цикл деления Z.) Выполнять шаг 13 для i = 1, 2, ..., 2t.
13. (Внутренний цикл деления Z.) Присвоить $[2p, Z_j]$ значение $([2p, Z_j] - [2p, Z_{j-1}])/i$ для j=2t, 2t - 1, ..., i + 1, i. Все разности будут положительными, а все деления выполняются нацело, т. е. без остатка.
14. (Внешний цикл умножения Z.) Выполнять шаг 15 для i = 2t - 1, 2t - 2, ..., 2, 1.

15. (Внутренний цикл умножения Z.) Заменить $[2p, Z_j]$ на $[2p, Z_j] - i[2p, Z_{j+1}]$ для j = i, i + 1, ..., 2t - 2, 2t - 1. Все разности будут положительными, и все результаты поместятся в 2p битов.
16. (Образование нового w и начало нового цикла.) Присвоить значение многочлена
$(\ldots ([2p, Z_{2t}]2^s + [2p, Z_{2t-1}]2^s + \ldots + [2p, Z_i]2^s + [2p, Z_0]$
переменной $[2(q_k + q_{k+1}), w]$. Этот шаг можно выполнять, используя только сдвиги и сложения длинных чисел. Заметьте, что используется та же переменная [m, w], что и на шаге 9. Удалить $[2p, Z_{2t}]$, ..., $[2p, Z_0]$ из стека W. Присвоить k значение k + 1 и вернуться к шагу 10.
17. (Проверка окончания.) Если A имеет значение *стоп*, то в переменной [m, w], уже вычисленной на шаге 9 или 16, находится результат алгоритма. В этом случае окончить работу.
18. (Сохранение значения.) Значением A должен быть код *сохранить* (если это не так, завершить алгоритм по ошибке). Присвоить k значение k + 1 и поместить $[q_k + q_{k-1}, w]$ в стек W. Это значение w, только что вычисленное на шаге 9 или 16. Теперь вернуться к шагу 3.

Комментарии к алгоритму Тоома—Кука

Мы почти не обосновывали и не объясняли алгоритм; вам придется кое в чем поверить на слово. Причина отсутствия объяснений кроется в том, что они крайне длинны и математичны, и у нас просто не хватило бы места. Изложение алгоритма в большой степени опирается на монографию Кнута; если вы хотите ознакомиться с алгоритмом подробнее, обратитесь к этой книге. Мы все же дадим некоторые комментарии, возможно способствующие лучшему пониманию.
1. (Структура алгоритма.) Наша версия алгоритма отличается от описанной у Кнута в основном структурой циклов. На рис. 20.1 представлена общая схема верхнего уровня алгоритма Тоома—Кука.

```
begin
  long integer [n, u], [n, v];
  integer K, Q, R, q [0: 10], r [0: 10];
  long integer stack C, U, V, W;
  control stack code;
  integer k, p, s, t, i, j;
  long integer X, Y, Z, w;
  control A;
  Шаг 1;
  Шаг 2;
  while code не пуст do
    while k > 1 do
      шаг 5; шаг 6; шаг 7; шаг 8;
    end;
    шаг 9;
    pop code into A; while A = интерполировать do
      шаг 11;
      for i=1 to 2t do шаг 13;
      for i=2t - 1 downto 1 do шаг 15;
      шаг 16;
    end;
    if A = стоп then return [m, w];
    if A = сохранить then шаг 18; else abort;
  end; * главного цикла *
end; * алгоритма Тоома–Кука *
```

Рисунок 20.1. Управляющая схема алгоритма Тоома—Кука.

2. (Таблицы размеров.) Значения массивов, вычисленные на шаге 2, показаны в табл. 20.1; число в колонке n_k равно наибольшему числу битов, которое может быть обработано алгоритмом при K = k. Очевидно, что предельное значение 10 для K не является очень серьезным ограничением. При желании этот предел можно повысить.

Таблица 20.1. Значения q, r и n

k	q_k	r_k	n_k
0	16	4	
1	16	4	32
2	64	4	80
3	256	4	320
4	1024	8	1280
5	8192	8	9216
6	65536	16	73728
7	1048576	16	1114112
8	16777216	16	17825792
9	268435456	32	285212672
10	8589934592	32	8858370038

3. (Глубина стеков в первом цикле.) Максимальная глубина стеков U и V на шагах с 5-го по 8-й равна $2(r_{K-1} + 1)$. Глубина стека C может возрастать до $\sum_{i=1}^{K-1}(r_i + 1)$.

4. (Глубина стеков во втором цикле.) Общая глубина стека W может достигать $\sum_{i=1}^{K-1} 2r_i$. Управляющий стек может достигать глубины $\sum_{i=1}^{K-1} 2r_i + 1$. На шагах 14, 15 и 16 верхняя часть стека W используется как массив. Этот массив может содержать максимум $2r_{k-1} + 2$ элементов.

5. (Размер исходных данных.) Для любого числа битов n в диапазоне $n_{i-1} + 1 \le n \le n_i$ алгоритм Тоома—Кука требует одинакового времени вычислений. Таким образом, сложность вычислений весьма негладко зависит от размера исходных данных. Поэтому при выполнении длинных вычислений имеет смысл подбирать число битов вблизи верхнего конца одного из диапазонов для n. Учитывайте, что для представления одной десятичной цифры требуется примерно 3⅓ бита.

6. (Как умножить два 32-разрядных числа?) На шаге 9 требуется умножить два 32-разрядных числа, получив 64-разрядное произведение, причем оба сомножителя обязательно положительны. На многих ЭВМ имеется аппаратная возможность такого умножения, но результат нельзя получить, пользуясь языками высокого уровня. Ну и, конечно, некоторые ЭВМ не имеют подобней аппаратуры. Поэтому для выполнения этого умножения нужно написать подпрограмму, причем она должна быть эффективной, поскольку время работы алгоритма определяется главным образом временем умножения 32-разрядных чисел. Вероятно, достаточно хорошим методом будет разбиение чисел на части и моделирование ручного способа умножения. Тем не менее если нужно получить произведение uv и число и записано в виде $u_1 \cdot 2^{16} + u_0$, а v — в виде $v_1 \cdot 2^{16} + v_0$, то произведением будет $(2^{32} + 2^{16})u_1v_1 + 2^{16}(u1 - u0)(v0 - v1) + (2^{16} + 1)u_0v_0$.

Вычисление по этой формуле выполняется при помощи только 16-битовых вычитаний и умножений, а также некоторых сдвигов и сложений. Обратите внимание, что одно умножение сэкономлено.

Что можно сказать относительно деления?

При вычислении предложенных рядов наряду с умножением используется деление чисел высокой точности. К счастью, при помощи алгоритма умножения удается выполнять деление почти так же быстро, как умножение. Для нахождения частного нужно приблизительно угадать число, обратное к делителю, скорректировать его, чтобы обратное стало точным, и затем умножить на делимое. Уточнение обратного осуществляется по методу Ньютона. Даны два числа [m, u] и [n, v]; мы считаем, что u ≥ v (хотя это предположение несущественно) и что n-й бит v равен 1 (т. е. у v нет старших нулей). Чем больше разница размеров и и v, тем более точным будет частное; разницу можно увеличить, умножая и на степень двойки. Отметим, что алгоритм деления будет неоднократно вызывать алгоритм умножения. Для нескольких первых из этих умножений можно воспользоваться обычной операцией умножения коротких чисел. Кроме того, все умножения и деления на степень двойки суть фактически сдвиги влево и вправо.

1. (Выбор размера обратного.) Найти наименьшее j, такое, что $2^j \geq \max(m, 2n)$. Присвоить к значение 2^{j-1}.
2. (Нормализация v.) Присвоить [k, v] значение 2^{k-n} [n, v]. На этом шаге мы сдвигаем v влево, чтобы оно заняло k битов, причем левый бит был бы равен 1. Присвоить [2, a] значение [2, 2].
3. (Вычисление последовательных приближений к 1/v.) Выполнить шаг 4 для i = 1, 2, ..., j - 1.
4. (Вычисление приближения из 2^i битов.) Присвоить $[2^{i+1}, d]$ значение
$2^{3\cdot 2i}\,[2^{i-1} + 1, a] - [2^{i-1} + 1, a]^2\,([k, v]/2^{k-2i})$

Деление в скобках (фактически сдвиг вправо) должно выполнятъся до умножения; идея состоит в том, чтобы ускорить умножение, отбросив лишние биты v, ненужные в данном приближения. Хотя кажется, что результат d может содержать больше 2^{i+1} битов, этого никогда не произойдет. Затем присвоить $[2^i + 1, a]$ значение $[2^{i+1}, d]/2^{i-1}$.

5. (Улучшение окончательной оценки.) Присвоить [3k, d] значение
$2^{2k}[k + 1, a] - [k + 1, a]^2 \cdot [k, v]$.
Затем присвоить [k + 1, a] значение
$([3k, d] + 2^{2k-2})/2^{2k-1}$.
6. (Окончательное деление.) Выдать в качестве результата
$([k + 1, a] \cdot [m, u] + 2^{k+n-2})/2^{2+k-1}$.

Как использовать алгоритмы?

Для нахождения π надо будет провести вычисления по одной из формул, выписанных ранее в этом пункте, с использованием ряда для арктангенса.

Фактически для страховки следует использовать две формулы и затем сравнить результаты бит за битом. Значением π будет общая начальная часть этих двух результатов.

Все еще остается открытым вопрос, как с помощью алгоритмов, работающих только с целыми числами, получить очевидно дробные значения членов ряда. Пусть мы хотим вычислить я, скажем, с точностью 1000 битов. Вычислим тогда $2^{1000}\pi$, умножив все числители на 2^{1000}. Эта процедура делает также все делимые много больше делителей (как предполагалось выше) и позволяет прекратить вычисления, когда частные станут нулевыми.

Выберем теперь (не обязательно наилучший) ряд для вычислений, скажем

$\pi = 16 \operatorname{arctg}(1/5) - 4 \operatorname{arctg}(1/239)$.

Мы фактически будем вычислять $2^{1000}\pi$, поэтому хотелось бы вычислить $2^{1000}\cdot 16 \operatorname{arctg}(1/5)$. Первым членом соответствующего ряда будет $2^{1000}\cdot 16/5$; назовем его a_1 (отметим, что a_1 складывается с суммой). Теперь, чтобы получить следующий член a_{i+1} из a_i, поделим a_1 на $5\cdot 5\cdot(2i - 1)$. Если a_i добавлялся к сумме, то вычтем a_{i+1} из суммы, если a_i вычитался, прибавим a_{i+i}. Будем поделим a_1 на $5\cdot 5\cdot(2i - 1)$. Если a_i добавлялся к сумме, то вычисления заканчиваются, когда члены обоих рядов станут нулевыми. В результате получим примерно тысячу битов числа π. Результат, конечно, надо будет перевести в десятичную систему.

Тема. Составьте программы, реализующие описанные выше алгоритмы умножения и деления, и все необходимые им служебные подпрограммы. Используйте их для вычисления я с высокой точностью при помощи одного из выписанных рядов. Проследите, чтобы ваши программы не оказались слишком тесно привязаны к вычислению π; библиотека программ для вычислений с высокой точностью может быть полезна и для других задач. Должна быть предусмотрена возможность увеличения точности счета без изменения программ, а лишь путем расширения памяти для результатов. Выходные данные должны включать статистику по использованию каждой программы, по числу выполнений каждого шага двух центральных алгоритмов и по использованию памяти. Сбор такой информации обойдется очень дешево в сравнении со всей работой.

Указания исполнителю. Эта тема длинная и трудная. Не последнюю роль здесь играет то, что два центральных алгоритма нужно в какой-то степени принимать на веру. Однако, как это часто бывает в реальных задачах, главной проблемой является не кодирование программы, а выбор структур данных. Как представлять длинные числа? Обозначение [m, u] наводит на мысль, что всякое длинное число представляется парой аргументов *длина* и *значение*. Часть *длина* легко реализуется, но *значение* имеет, очевидно, переменную длину, и его трудно будет непосредственно хранить в памяти. Поэтому мы сделаем *значение* указателем на очень длинный вектор битов; тогда каждая пара будет иметь фиксированный размер. Однако имеющийся в нашем распоряжении вектор не настолько длинен, чтобы мы могли позволить себе использовать каждую его часть только по одному разу. Таким образом, нужна программа для **сбора ненужной памяти**. Сейчас мы фактически описали традиционную схему **размещения цепочек**.

Итак, в конечном итоге нам нужны кроме алгоритма умножения и деления следующие вспомогательные подпрограммы:

Выделение памяти. Получая величину *длина* в качестве аргумента, эта подпрограмма возвращает указатель в вектор битов, который может использоваться как *значение*. Начиная с бита, на который указывает *значение*, расположено *длина* битов, которые не будут использоваться ни для каких других целей.

Возврат памяти. Исходными данными для этой подпрограммы служит пара *длина* и *значение*. Связанная с ними память освобождается для повторного использования. Эту подпрограмму необходимо вызывать всякий раз, когда длина какой-либо переменной меняется.

Уплотнение памяти. Эта подпрограмма должна просмотреть всю используемую память и попытаться объединить неиспользуемые отрезки вектора битов в более длинные отрезки. Обычно подпрограмма уплотнения вызывается в тех случаях, когда подпрограмма выделения памяти не смогла найти достаточно длинную цепочку последовательных битов. Поскольку такая возможность может не потребоваться при решении коротких задач, эту подпрограмму можно запрограммировать позднее. Существует много способов хранения информации о неиспользуемой памяти.

Сдвиг. Исходными данными для подпрограммы служат длинное число и величина сдвига; результатом должно быть длинное число, сдвинутое вправо или влево на соответствующую величину. Эта операция отвечает умножению или делению на степень двойки.

Сложение. Исходными данными подпрограммы служат два длинных числа, а результатом должна быть их сумма в виде числа на один бит длиннее более длинного из операндов. Такое сложение можно выполнять так же, как вручную, двигаясь справа налево.

Вычитание. Эта подпрограмма аналогична подпрограмме сложения и выдает разность двух длинных чисел.

Подавление нулей. Исходными данными этой подпрограммы служит длинное число, а результатом должно быть более короткое длинное число, имеющее то же значение, но без старших нулей. Если окажется, что исходное число равно нулю, то результатом должно быть [1, 0].

Короткое умножение. Исходными данными служат два длинных числа длиной точно 32 бита; результатом должно быть их 64-разрядное произведение. Эту операцию можно выполнять справа налево, как в ручном методе.

Умножение длинного на короткое. Исходными данными служат длинное число и обычное число, по величине равное 64 или меньше; результатом должно быть их произведение в виде длинного числа. Эту операцию можно выполнять справа налево, как вручную.

Деление длинного на короткое. Исходными данными служат длинное число и обычное число, не превосходящее 64, а результатом должно быть длинное частное от деления длинного числа на короткое. Эту операцию можно выполнять слева направо, как это делается вручную.

Перевод. Исходными данными для этой подпрограммы является длинное число, а результатом должно быть то же число, записанное в десятичной системе на некотором устройстве вывода. При появлении потребности в более сложном выводе можно разработать более детальные спецификации подпрограммы перевода.

Развитие темы. Как только у нас появляется арифметика высокой точности, сразу же возникает много интересных задач. Одна из них — точное вычисление числа e. Ряд для e особенно прост:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!},$$

где 0! = 1. Любой студент, изучающий математический анализ, может придумать еще очень много рядов и констант.

Тема 21: Оптимальные стратегии для игры с угадыванием

В игре, как и в музыкальном произведении, можно выделить тему и мотивы. Причина успеха самых удачных игр часто состоит в том, что они мастерски соединяют по-новому некоторые из давно известных принципов построения игр. Как и в музыке, старая идея, возрожденная в новом обличье, может выглядеть привлекательней, чем мешанина свежеиспеченных новых веяний. В середине 70-х годов широкую популярность в Англии получила игра *великий комбинатор* (Mastermind), и она, похоже, станет классикой. Вы и ваш компьютер получите большое удовольствие, сыграв в нее.

Правила великого комбинатора крайне просты. Один из игроков, **загадывающий**, записывает секретную комбинацию из любых четырех цифр от 1 до 6 (повторения допускаются), называемую **кодом**. Второй игрок, **отгадывающий**, пытается раскрыть код, высказывая разумные предположения, называемые **пробами**. Каждая проба, как и код, представляет собой произвольную комбинацию из четырех цифр в диапазоне от 1 до 6. Отгадывающий игрок сообщает пробу загадывающему, и тот должен ответить, сколько цифр в пробе совпадает с цифрами кода как по положению, так и по величине и сколько из остальных цифр пробы входят в код, но стоят на другом месте. Так, на пробу 1123 при коде 4221 будет получен ответ: «Одна цифра совпадает и стоит на том же месте, и еще одна совпадает, но стоит на другом месте». Тур игры продолжается до тех пор, пока отгадывающий не назовет пробу, в точности совпадающую с кодом, т. е. пока не отгадает код. После этого игроки меняются ролями и проводят еще один тур. Победителем считается тот из игроков, кто определит код противника за меньшее число проб. Хотя здесь не последнюю роль играет везенье, тем не менее игрок, систематически делающий правильные умозаключения из получаемой информации, должен иметь лучшие результаты по итогам нескольких партий. Практически вы должны пытаться выводить из ответов на ваши пробы отрицательные следствия относительно того, какие коды невозможны; психологические тесты показывают, что для многих людей это оказывается совсем не просто. В табл. 21.1 приведен один полный тур.

Таблица 21.1. Великий комбинатор. Пример партии

Код: 4651

Проба		Кол-во точных попаданий	Кол-во совпадений по значению
1	2345	0	2
2	4516	1	3
3	5461	1	3
4	4165	1	3
5	4615	2	2
6	4651	4	Игра окончена

Написать программу, имитирующую роль загадывающего, не составляет труда. Отгадывание головоломок, заданных машиной, — тоже развлечение, позволяющее отточить ум. Однако гораздо интереснее, если компьютер сможет выступать также и в роли отгадывающего, чтобы можно было сыграть несколько партий и определить победителя. Боб Кули из Lawrence Livermore Laboratory и Д. Кнут разработали довольно близкие стратегии, позволяющие ЭВМ достигнуть высокого класса игры. Центральное место в обеих стратегиях занимает идея **пространства решений**. Начальное пространство решений P_0 состоит из всех возможных кодов (и имеет, следовательно, 64 элементов); после i-й пробы G_i пространство P_i состоит из всех тех членов пространства P_{i-1}, которые не опровергаются ответом R_i. Иными словами, пространство P_i — это множество всех комбинаций, которые все еще могут быть кодом; задача отгадывающего — свести пространство к одному элементу.

Первая стратегия, предложенная Кули, несколько проще. Пробой Gi пусть будет любая случайно выбранная комбинация с одной повторяющейся цифрой, например 4311, 6552 или 1335. Выполните эту пробу и постройте пространство P_i на основе ответа R_i. Новая проба G_{i+1} ищется по пространству P_i, $i \geq 1$, путем поочередного сравнения всех комбинаций С из P_i с пробой G_i. В качестве следующей пробы выбирается *наименее* похожая на G_i комбинация С. Мерой сходства служит число точных совпадений, а в случае равенства — число цифр, совпадающих по значению, но расположенных по-другому. Так, среди трех комбинаций 2641, 2356 и 1345 наиболее похожей на 2345 будет 1345, а 2641 — наименее похожей. Если имеется несколько наименее похожих комбинаций, то можно выбрать любую кандидатуру случайным образом. Тур прекращается, когда будет получен ответ «четыре точных попадания», и, разумеется, в случае пространства из одного элемента в качестве следующей пробы всегда надо брать этот элемент. Как показывают эксперименты, размеры пространства решений сокращаются после каждой пробы примерно в 4 раза и никогда не требуется более шести проб.

Вторая стратегия предложена Дональдом Кнутом. Он утверждает, что она минимизирует наибольшее число проб, необходимых для нахождения кода; никакой код не требует более пяти проб. В основе алгоритма лежит наблюдение, что нам хотелось бы сделать пространство P_i как можно меньше. Следовательно, мы выбираем пробу G_i, минимизирующую $|P_i|$ по всем возможным ответам R_i. Кандидатом в G_i будет любая комбинация С. Попробуйте применить все возможные комбинации С в качестве проб к пространству P_{i-1}; пусть $S_{c,\ <0,0>}$ обозначает число членов P_{i-1}, дающих в ответе нулевое число точных совпадений и нулевое Число совпадений только по цвету $S_{c,\ <0,1>}$ — число членов, дающих соответственно нуль и одно совпадение и т. д. до $S_{c,\ <4,0>}$ для наиболее удачной комбинации с четырьмя точными совпадениями. Введем обозначение

$$S_C = \max_{(i,\ j)} S_{C,\ (i,\ j)}.$$

Теперь в качестве пробы G_i выберите комбинацию С, минимизирующую S_c (при наличии нескольких таких С выберите комбинацию из P_{i-1}, если это возможно; если же нет — делайте случайный выбор). Вы, вероятно, уже заметили, что этот

алгоритм можно использовать для предварительного анализа *великого комбинатора*, так чтобы в процессе игры не был нужен никакой анализ комбинаций. Проделав такой анализ, Кнут показал, что оптимальной первой пробой при использовании его стратегии будет ххуу, где х ≠ у. Для проверки своей программы посмотрите, начинает ли она с пробы ххуу.

Тема. Напишите программу, которая будет разыгрывать партии *великого комбинатора*. Реализуйте стратегию отгадывания, так чтобы машина могла загадывать коды и отгадывать их. Кроме собственно игры ваша программа может накапливать сведения о мастерстве разных игроков. Ваш местный великий комбинатор, возможно, пожелает приехать в Англию на очередной чемпионат. С вашей программой, как и другими игровыми программами, вероятно, будет иметь дело не слишком искушенный пользователь. Поэтому следует позаботиться о том, чтобы ввод данных был простым и естественным, а вывод понятным и красиво оформленным.

Рекомендации исполнителю. Единственная серьезная проблема в этом этюде связана с эффективностью при программировании алгоритма анализа — эффективностью как по памяти, так и по времени. Особенно длинный внутренний цикл требуется для второй стратегии. Заметьте, что комбинации суть не что иное, как числа, записанные по основанию 6 (но вместо цифр от 0 до 5 используются цифры от 1 до 6).

Развитие темы. Наиболее очевидное расширение — это изменение множества цифр, из которых составляется код, или количество цифр в коде. Более развитая версия великого комбинатора допускает коды из пяти цифр от 1 до 8. Слишком большое значение любого из двух параметров может привести к непомерному росту времени работы, однако ни один из алгоритмов не зависит сколько-нибудь существенно от чисел 6 и 4. Программа без всякого труда могла бы читать объем словаря (число различных цифр) и длину кода в качестве исходных данных и соответствующим образом изменять свои алгоритмы анализа.

Тема 22: Математический подход к раскрытию шифров

Представьте себе такую ситуацию. Благодаря выдающимся профессиональным познаниям и незаурядным программистским способностям вас выдвинули на должность руководителя большой группы сотрудников, занимающихся разработкой суперновейшего и пока еще секретного Мини-компилятора. Как-то раз, уходя со службы около часу ночи (руководитель должен подавать хороший пример), вы замечаете торчащий в дверях измятый клочок бумаги (содержание которого воспроизведено на рис. 22.1). Сначала вы решаете, что это запись содержимого памяти машины, и уже собираетесь выбросить бумажку. Но, присмотревшись повнимательнее, замечаете, что буквы собраны в группы по пять. Что бы это могло быть?

```
ЖНФЖП ЕЕЫШВ ЛПЖАТ ГФБЦМ КЖЬЗА ЮЪИВУ ЩЖРСЮ БЬЬКЬ ЫЕСУУ ЦТЮБШ УНЭДДО ЭЭШЮЗ УЬЕКН АУЕЫЩ ШЖРЬЙ
ЛЮПКН ДЙЯГЭ ЪНЫГЖ ОУШИШ УФГБР ШМАГВ ВУВОС ЗХЧИУ ГНЛАЯ ЬЬКИЯ РЦЖРЫ АХЪБИ ЖГЭЯЦ СЪУЫФ ЯРМЭФ
ЧФЬЩС ЬФШВЕ ОМКТИ МБЭВЪ КФХЙЦ ХНЪЮЪ МФЛБИ МРУЛМ ЯЗФЧЪ ЪЧЗНК ЗНИВЯ НШГЛЩ ИЛЗНФ ФУЖКН ДЙЯГЭ
ЕУЮЛЛ ЮЖНЯИ ЕМДЙШ ГЯУГВ ЦФЩВЮ МФАГЯ ВХМЭВ ВФПГФ ФЖККГ ЦМЛЫБ ШМПУЕ ШЖЯЯЮ ЯРЧВЪ ЖУПВМ КЛЫЭС
ЭЧИРЫ ГЫЩЗЗ ЗКЖЛЕ ШВРЪЧ ЪААЖЗ ДХЪФС БРНМЪ КЫБЪФ УНЦЮБ ТЖУНЯ ЕШИМУ КФВГВ ГЧМЗВ ЗРВМЪ ЪЕЕТО
ЯЦБЖГ ВИЖМД КЗЗПА ФЯВНР ЫГЮЩЭ ЯЫДОЪ ЧНГВЫ АХЪВЛ НШАПВ ЧОЬОЙ КЮАШО КЗЛЩУ ШЯРНЗ ГХЛТЮ ЖЫШШГ
ППЬЫШ АЬФМА ФЕЙЗА ЙШЕУЭ ЖЛЗИЗ НЖККР ЦЯДЧК НДЙЯГ ЭБФЬА ВБЭКЗ ФКЫТВ ЛЕЪЭЯ ЛЭЩЗН ФХГЧК ТКЫЮЗ
ЗЪУЖА ПВЧОЬ ОЙКЕС ЛЗАЮЬ ИВУНЫ БКЗВЯ ЪГОСЩ ЛБЬГМ ЯВЗГЬ КШЪГЙ ЕНПСМ ЭВГОГ ЧСОРГ ЩОЦМВ ДГЩКЧ
ЮЗВЗК ЦЧЯРЧ ВЪЖФЫ ЕЛЖАЪ УССХР УОЬЫЕ ЙГЫОТ УЕАГЖ ГЫСЩИ ЯРВТЮ ДЖНЛГ ЦМЗЬЪ ЯИЦТР ЕМИКЦ ЩВЦОР
ЛХМХЖ БРЬПУ ГВЯРЬ ПМЯЖЖ РЧПШЪ ЧУВГЧ СЕЕГЦ ЬПЗДМ ОЬОЧЗ КВУФЯ УПОХЪ ГЪЭЯЖ ВЖФ
```

Рисунок 22.1. Таинственная записка, найденная в вычислительном центре. Случайное вкрапление русских слов, например ШИШ или ОЙ, по-видимому, ничего не означает. Но обратите внимание на повторение сочетаний ЗАЮЪИВУ, ЬЬК, других коротких сочетаний, а особенно на повторяющуюся группу букв КНДЙЯГЭ.

Снова возвращаетесь в свой кабинет, пытаясь решить загадку. Бумага отменная, слегка пахнет мускусом; почерк явно женский и веет от него этаким французским шармом. Теперь, по здравом размышлении, новая сотрудница мисс Хари начинает казаться вам, пожалуй, немножко слишком экзотичной. Ее французский акцент, неизменное черное платье для коктейля, нитка черного жемчуга, подчеркивающая декольте, и этот будоражащий запах мускуса, наполняющий комнату, когда она туда входит... Она говорит, что работала раньше в региональном вычислительном центре Мак-Дональда в Киокаке. Что-то тут не так. Подождите... Неужели мисс Хари шпионит в пользу знаменитой французской фирмы И Бей Эм? А эта записка — шифровка, в которой все секреты вашего новейшего чудо-компилятора? Чтобы уличить мисс Хари, записку нужно расшифровать. Но как? Может, обратимся за помощью к компьютеру?

Основы шифрования

ЭВМ, безусловно, может оказать помощь, иначе Управление национальной безопасности просто пускает на ветер деньги налогоплательщиков, закупая такое количество техники. Для начала необходимо как следует присмотреться к секретному сообщению. Возможно, что найденная записка была зашифрована при помощи **простой подстановки**, т. е. каждая буква первоначального текста была заменена какой-либо другой буквой согласно некоторому **правилу шифрования**. Сообщение, подвергшееся зашифровке, называется исходным текстом, а в

результате получается **шифрованный текст**. Задача состоит в том, чтобы **восстановить** исходный текст и правило шифрования (последнее нужно лишь в том случае, если могут появиться другие сообщения, зашифрованные по тому же правилу). Будем предполагать, что исходный текст написан по-русски. Разбиение шифрованного текста на группы по пять букв скрывает, по-видимому, исходную структуру текста, разбитого на слова, которая была бы весьма ценной подсказкой, облегчающей расшифровку.

В простейшем общем классе подстановочных шифров для построения правила шифрования используется некоторый **смешанный алфавит**, например перестановка обычного алфавита. На рис. 22.2 показан полный исходный алфавит, смешанный алфавит и шифрование короткого сообщения, в котором каждая буква заменяется соответствующей буквой смешанного алфавита. Всякий, кто увлекается головоломками из воскресных газет, знает, что зашифрованные такой подстановкой тексты расшифровываются до смешного просто: сообщения из 30 или 40 букв зачастую оказывается для этого вполне достаточно. Тем не менее слегка усовершенствовав эту систему, можно сделать ее значительно более надежной.

```
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
ЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕР
ПОПРОБУЙТЕ ПРОЧИТАТЬ КРИПТОГРАММУ ТОЧКА
ЪПЪЫПУОГЮЯ ЪЫПТКЮЗЮХ ЛЫКЪЮПВЫЗММО ЮПТЛЗ
```

Рисунок 22.2. Простая подстановка по смешанному алфавиту. Обратите внимание, что точка заменена словом ТОЧКА.

На рис. 22.3 изображен **квадрат Виженера**, построенный на основе смешанного алфавита, приведенного на рис. 22.2. Сверху и по левому краю квадрата выписан исходный алфавит. В первой строке квадрата представлен смешанный алфавит. Во второй строке тот же алфавит циклически сдвинут на одну позицию, при этом первая буква переместилась в правый конец строки. Квадрат состоит из 32 смешанных алфавитов, полученных из одного смешанного алфавита, каждому из них соответствует та буква исходного алфавита, которая записана слева от него. На рис. 22.4 показано шифрование фразы при помощи **ключевого слова** ЛИСП и данного квадрата. Ключевое слово многократно записывается под исходным текстом, и каждая буква исходного текста шифруется при помощи смешанного алфавита, соответствующего той букве ключевого слова, которая стоит под данной буквой исходного текста. Эта схема шифрования уже не поддается раскрытию при помощи простого подсчета частот букв, поскольку одна и та же буква исходного текста шифруется по-разному в зависимости от выпавшей на нее буквы ключевого слова. Кроме того, выбрав заранее список ключевых слов и порядок их смены, отправитель и получатель могут повысить секретность переписки, поскольку разным сообщениям будут соответствовать разные ключевые слова, благодаря чему затрудняется анализ, основанный на частотах букв. Тем не менее не так уж все это безнадежно.

```
  АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
А ЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕР
Б УШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗ
В ШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУ
Г ВЬЯЖЩКГЛФМДПЪЫШООСИЙТЧБАЭХЦЕРЗУШ
Д ЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВ
Е ЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬ
Ж ЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯ
З ЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖ
И КГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩ
Й ГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩК
К ЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГ
Л ФМДПЪЬНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛ
М МДПЪЬНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФ
Н ДПЪЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМ
О ПЪЫЮООСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМД
П ЪЫНЮОСЙЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДП
Р ЫНЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪ
С НЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫ
Т ЮОСИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫН
У ОСИЙТЧБАЭХЦВРЗУШВЬЯЖЩКГЛФМДПЪЫНЮ
Ф СИЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮО
Х ЦЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОС
Ц ЙТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИ
4 ТЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙ
Ш ЧБАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТ
Щ БАЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧ
Ъ АЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБ
Ы ЭХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБА
Ь ХЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭ
Э ЦЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХ
Ю ЕРЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦ
Я РЗУШВЬЯЖЩКГЛФМДПЪЫНЮОСИЙТЧБАЭХЦЕ
```

Рисунок 22.3. Квадрат Виженера, построенный на основе смешанного алфавита, приведенного на рис. 22.2.

```
ПОПРОБУЙТЕ ПРОЧИТАТЬ КРИПТОГРАММУ ТОЧКА
ЛИСПЛИСПЛИ СПЛИСПЛИС ПЛИСПЛИСПЛИС ПЛИСП
АЙЗРЕГЬЧЦЦ ЗРЕРБУФАД БЭЫЗУБФУЪТСЬ УБРЭЪ
или
АЙЗРБ ГЬЧЦД ЗРБРБ УФАДБ ЭЫЗУБ ФУЪТС ЬУБРЭ Ъ
```

Рисунок 22.4. Шифрование при помощи квадрата Виженера. Обратите внимание на повторение сочетания РБ на расстоянии 8. Второе повторение этого сочетания на расстоянии 2 — ложное. Статистика языка проявляется даже на коротких примерах.

Как раскрыть шифр

Будем предполагать, что криптограмма мисс Хари получена при помощи квадрата Виженера, хотя бы по той причине, что он — ее соотечественник. Если наше предположение неверно, методы решения позволят обнаружить это. Если бы сообщение было зашифровано при помощи простой подстановки, то расшифровать его можно было бы, подсчитав количество появлений каждой буквы в шифрованном тексте, поделив это количество на длину сообщения и сравнив полученные величины с частотами букв русского алфавита, приведенными на рис. 22.5. Для сообщений такой длины, как наше, распределения частот, если выписать их в убывающем порядке, почти полностью совпадут, и, таким образом, для

каждой буквы исходного текста откроется ее двойник в шифрованном тексте. Но для квадрата Виженера такой простой метод уже не сработает. Необходимо определить не только смешанный алфавит, но и ключевое слово; поскольку каждый из этих элементов искажен другим, то трудно даже догадаться, с какого конца начать.

```
О .0940
А .0896
Е .0856
И .0739
Н .0662
Т .0611
Р .0561
С .0554
П .0421
М .0417
В .0400
Л .0358
К .0322
Л .0280
Я .0243
Ы .0225
Б .0197
З .0193
У .0179
Г .0153
Ь .0125
Ч .0118
Й .0094
Х .0093
Ц .0087
Ж .0064
Ю .0063
Щ .0048
Ф .0034
Э .0033
Ш .0032
Ъ .0002
```

Рисунок 22.5. Таблица частот букв русского алфавита. Получена по текстам нескольких препринтов, издававшихся в ИПМ АН СССР им. М. В. Келдыша.

Правильной отправной точкой будет нахождение длины ключевого слова. Обратите внимание, что в примере на рис. 22.4 первая, пятая, девятая, ... буквы исходного текста зашифрованы при помощи одного и того же смешанного алфавита Л. Если рассматривать лишь каждую четвертую букву шифрованного текста, то получим распределение частот, подобное распределению для букв русского алфавита, поскольку буквы в этих позициях зашифрованы при помощи одного и того же смешанного алфавита, т. е. при помощи простой подстановки. Аналогично если взять каждую четвертую букву шифрованного текста, начиная со второй, третьей или четвертой позиции, то снова получим распределение частот как для букв русского алфавита. Существует способ измерить, насколько данное распределение частот подобно распределению букв алфавита. Рассмотрим **индекс совпадения**

$$\text{ИС} = \sum_{i=1}^{32} \frac{f_i (f_i - 1)}{N (N - 1)},$$

где f_i — количество появлений i-й буквы, а N — общее число рассматриваемых букв. Если все буквы рассматриваемого подмножества текста зашифрованы при помощи одного алфавита, то этот индекс совпадения должен иметь значение больше 0.045 и, вероятно, меньше 0.065 (теоретическое значение равно 0.055). Исходя из этого, алгоритм определения длины ключевого слова будет таким.

Шаг 1. Для i от 1 до 20 предположить, что длина ключевого слова равна i, и выполнить шаги 2, 3, 4. Мы выбрали верхнюю границу равной 20 лишь для удобства. Разумеется, ключевое слово может быть и длиннее.

Шаг 2. Для j от 1 до i выполнить шаг 3. В этих двух шагах будут вычислены i различных значений НС.

Шаг 3. Построить распределение числа появления букв в позициях j, i + j, 2i + j, ..., т. е. в каждой i-й цозиции, начиная с j-й позиции. По формуле, приведенной выше, вычислить $ИС_j$ для полученного распределения. В качестве N в этой формуле нужно использовать число букв в данном подмножестве текста, а не длину всего текста.

Шаг 4. Если все значения $ИС_1$, $ИС_2$, ..., $ИС_i$ больше 0.045, то, вероятно, i кратно длине ключевого слова. Если только один из ИС меньше 0.045, то i также может быть кратно длине ключевого слова.

Проверить длину ключевого слова можно и другим способом. Найдите два места в шифрованном тексте, где две одинаковые буквы идут в том же порядке, например ЦМ в позициях 19 и 54 на рис. 22.1. Такое повторение могло произойти по двум разным причинам. Возможно, в соответствующих местах исходного текста были различные сочетания букв, которым отвечали разные части ключевого слова, и они случайно отобразились в одинаковые сочетания букв, либо в исходном тексте были повторения, которые попали на одинаковые части ключевого слова, и, таким образом, оказались зашифрованными дважды одним и тем же способом. Во втором случае расстояние между началами повторяющихся сочетаний букв должно быть кратно длине ключевого слова. К сожалению, невозможно определить, по какой из двух причин произошло повторение данного сочетания букв: случайное повторение пар букв в шифрованном тексте довольно частое явление. Но если в шифрованном тексте повторяются сочетания из трех или более букв, то вероятность того, что это повторение произошло случайно, а не в результате повторения ключа, очень мала (для сочетаний из четырех и более букв она практически нулевая). Таким образом, другой способ выявления длины ключевого слова — отыскать в шифрованном тексте все пары повторяющихся групп из трех и более букв и измерить расстояния между ними. Число, которое делит 90% или более из этих расстояний, — прекрасный претендент на роль длины ключевого слова. Данная проверка вместе с вычислением значений ИС однозначно определяет длину ключевого слова.

Предположим, нам удалось выяснить, что длина ключевого слова равна k. Тогда первоначальный шифрованный текст можно разбить на k групп G_1, G_2, ..., G_k, где каждая группа начинается с позиции i, $1 \leq i \leq k$, и содержит каждую k-ю букву

текста, начиная с i-й буквы. Каждая из этих к групп была зашифрована при помощи только одного алфавита, т. е. при помощи простой подстановки. Остается в каждой группе для каждой шифрованной буквы определить ее эквивалент в исходном тексте. Но здесь у нас имеется хорошее подспорье. Если бы был известен алфавит, по которому была зашифрована какая-нибудь из групп, то алфавит, по которому была зашифрована любая другая группа, можно было бы найти путем циклического сдвига уже известного алфавита на некоторое число букв. С другой стороны, определить исходные эквиваленты букв было бы проще, если бы удалось распределения числа появлений букв для различных групп скомбинировать в одно обобщенное распределение, поскольку, чем больше данных было использовано для построения какого-либо распределения, тем достовернее будут сделанные на его основе статистические выводы. Для построения такой комбинации необходимо знать относительные сдвиги между алфавитами, использованными для шифрования различных групп.

Относительные сдвиги находятся при помощи некой модификации индекса совпадения. Построим для каждой группы Gi распределение числа появлений букв и запишем его в алфавитном порядке шифрованных букв. В табл. 22.1 показаны распределения для сообщения, приведенного на рис. 22.1, в предположении, что k = 7. Пусть $f_{i,\alpha}$ — количество появлений буквы α алфавита i; определим функцию

$$R_{i,\ j,\ r} = \sum_{\beta=1}^{32} f_{i,\ \beta} f_{j,\ \beta+r}.$$

Считается, что если β + r больше 32, то происходит циклический возврат к началу алфавита. Чем больше значение $R_{i,\ j,\ r}$, тем больше вероятность того, что алфавит для группы j в квадрате Виженера находится на r позиций *ниже* алфавита для группы i. Вычислим все значения $R_{i,\ j,\ r}$ (для $j \leq i$ их можно не вычислять благодаря свойству симметрии) и выберем i и j, которые дают максимальное значение $R_{i,\ j,\ r}$. Вероятно, группа j сдвинута на r позиций относительно группы i.

Из групп G_i и G_j построим новую супергруппу G_{ij}, положив величину $f_{ij,\ \alpha}$ равной $f_{i,\ \alpha} + f_{i,\ \alpha+r}$. Отбросим из рассмотрения группы G_i и G_j, заменив их группой G_{ij}, и повторим описанный в последних двух абзацах процесс. После k - 1 повторений станут известны относительные сдвиги для всех k алфавитов. Кроме того, будет найдено обобщенное распределение частот. Для того чтобы найти исходные эквиваленты букв шифрованного текста, переупорядочим последние согласно их частотам. В результате буквы шифрованного текста должны расположиться в том же порядке, что и буквы русского алфавита (см. рис. 22.5). Теперь нетрудно восстановить весь квадрат Виженера и расшифровать текст. Ключевое слово можно найти, перебрав 32 набора из к букв, относительные расстояния между которыми соответствуют найденным сдвигам алфавитов. Возможно, что некоторые редко встречающиеся буквы окажутся не на своих местах. Эту ситуацию можно поправить при помощи визуального исследования полученного текста. Следует восстановить и смешанный алфавит, и ключевое слово, поскольку они оба могут иметь некоторую психологическую связь с содержанием сообщения и их выявление поможет дополнительно убедиться в правильности решения. Между прочим, что же написала мисс Хари?

Таблица 22.1. Распределения для сообщения с рис. 22.1 при k = 7

G1		G2		G3		G4		G5		G6		G7	
А	5	А	0	А	3	А	2	А	8	А	0	А	3
Б	1	Б	7	Б	0	Б	1	Б	0	Б	1	Б	3
В	19	В	5	В	6	В	4	В	1	В	1	В	8
Г	0	Г	13	Г	2	Г	10	Г	5	Г	2	Г	8
Д	1	Д	0	Д	0	Д	4	Д	0	Д	0	Д	5
Е	4	Е	3	Е	1	Е	0	Е	2	Е	2	Е	11
Ж	10	Ж	8	Ж	3	Ж	7	Ж	3	Ж	2	Ж	1
З	3	З	7	З	9	З	2	З	2	З	5	З	4
И	2	И	3	И	4	И	4	И	2	И	0	И	3
Й	4	Й	0	Й	0	Й	1	Й	6	Й	1	Й	0
К	2	К	9	К	4	К	9	К	1	К	4	К	3
Л	3	Л	6	Л	4	Л	2	Л	1	Л	7	Л	3
М	5	М	1	М	0	М	5	М	4	М	14	М	0
Н	1	Н	6	Н	9	Н	3	Н	2	Н	3	Н	1
О	0	О	4	О	2	О	8	О	1	О	1	О	4
Л	1	Л	0	Л	1	Л	4	Л	9	Л	4	Л	0
Р	1	Р	2	Р	12	Р	2	Р	2	Р	0	Р	5
С	5	С	0	С	0	С	0	С	2	С	6	С	1
Т	1	Т	0	Т	5	Т	0	Т	3	Т	1	Т	1
У	2	У	7	У	6	У	1	У	9	У	4	У	2
Ф	0	Ф	1	Ф	9	Ф	4	Ф	5	Ф	5	Ф	5
Х	4	Х	0	Х	0	Х	0	Х	4	Х	3	Х	2
Ц	3	Ц	0	Ц	1	Ц	3	Ц	8	Ц	1	Ц	3
Ч	10	Ч	0	Ч	0	Ч	2	Ч	8	Ч	0	Ч	1
Ш	5	Ш	8	Ш	2	Ш	4	Ш	2	Ш	1	Ш	2
Щ	0	Щ	0	Щ	4	Щ	0	Щ	0	Щ	1	Щ	8
Ъ	0	Ъ	6	Ъ	4	Ъ	4	Ъ	0	Ъ	9	Ъ	5
Ы	3	Ы	0	Ы	1	Ы	8	Ы	2	Ы	8	Ы	0
Ь	1	Ь	5	Ь	9	Ь	5	Ь	4	Ь	2	Ь	0
Э	8	Э	0	Э	0	Э	0	Э	4	Э	0	Э	6
Ю	1	Ю	0	Ю	1	Ю	3	Ю	4	Ю	5	Ю	4
Я	1	Я	5	Я	4	Я	3	Я	1	Я	12	Я	3

Значение $R_{1, 0, 2}$ равно 333, а значение $R_{3, 6, 12}$ равно 335. Значение $R_{3, 6, 12}$ получается перемножением чисел появлений букв от А до У для G_3 на числа появлений букв от М до Я для G_6 и чисел появлений букв от Ф до Я для G_3 на числа появлений букв от А до Л для G_6 и сложением всех этих произведений.

Тема. Напишите программу, которая в качестве входных данных воспринимает шифрованное сообщение и, в предположении, что оно зашифровано по схеме Виженера, печатает расшифрованный текст. Программа должна также печатать квадрат Виженера и ключевое слово, которые она вычисляет в процессе решения задачи. Специальные входные параметры должны управлять выводом промежуточных результатов, таких, как, например, все возможные длины

ключевого слова, распределения частот букв для отдельных алфавитов, значения ИС и т. д., которые нужны для контроля. Эти результаты могут быть полезны при отладке, а также в тех, к сожалению, вполне реальных ситуациях, когда предложенное машиной решение оказалось не совсем точным. Четкость оформления выводных данных имеет большое значение: бестолковые распечатки лишь затрудняют работу интуиции специалиста по расшифровке сообщений.

Указания исполнителю. Описанные здесь алгоритмы вполне понятны и легко реализуются, но обладают одним неприятным свойством — они не дают однозначного результата. Длина ключевого слова, например, будет лишь «вероятной», так что необходимо еще сделать обоснованный выбор одной из возможных длин. Аналогично алгоритмическое определение исходных эквивалентов для редко встречающихся букв шифрованного текста следует проверить, убедившись, что при расшифровке получаются правильные русские слова. Увеличивая статистическую информацию, доступную программе, мы получим более надежное основание для алгоритмических решений, но все равно эти решения должен проверить человек. Помимо указанных алгоритмов в вашей программе должны быть реализованы средства, позволяющие подтвердить обоснованность выводов, которые делает программа. Один хороший способ обеспечить такую оценочную функцию — написать программу в рамках какой-либо диалоговой системы, чтобы программа и пользователь смогли совместно обсудить качество каждого решения до того, как оно будет окончательно принято. «Обсуждение» обычно состоит в том, что программа сообщает человеку факты, говорящие в пользу того или иного возможного решения, а человек либо принимает его, либо отвергает, после чего вычисление может быть продолжено.

Несмотря на то что алгоритмы неоднозначны и такая расплывчатость обычно порождает у программиста чувство неуверенности, эту программу легко проверить. Первой частью работы, по-видимому, должна быть программа шифровки, которая воспринимает в качестве исходных данных русский текст и, выбрав некоторым случайным образом смешанный алфавит и ключевое слово, выдает квадрат Виженера и печатает зашифрованный текст в стандартном пятибуквенном формате. Пробелы и пунктуация должны убираться из текста автоматически. Эта программа должна уметь также воспринимать в качестве возможных параметров квадрат Виженера и ключевое слово, чтобы можно было повторно проверять отдельные особенности работы программы расшифровки. Помните о том, что для хорошего статистического поведения алгоритмов необходимо, чтобы сообщение было в 30–40 раз длиннее ключевого слова.