

Третий том известной монографии одного из крупнейших американских специалистов по программированию Д. Кнута (первый том вышел в издательстве "Мир" в 1976 г., второй—в 1977 г.) состоит из двух частей: "Сортировка" и "Поиск". В них подробно исследуются различные алгоритмы внутренней и внешней сортировки, изучаются методы поиска информации в таблицах на основе сравнение или преобразования ключей, даются оценки эффективности предлагаемых алгоритмов. Книга снабжена большим количеством задач и примеров разной степени трудности, существенно дополняющих основной текст.

От других руководств по программированию книга выгодно отличается строгостью изложения и широким применением математического аппарата. Вместе с тем она доступна студентам первого курса. Знакомство с двумя первыми томами желательно, но не обязательно. Каждый, кто хочет научиться квалифицированно программировать, найдет в ней много полезного.

Рассчитана на широкий круг программистов.

ПРЕДИСЛОВИЕ РЕДАКТОРОВ ПЕРЕВОДА

Д. Э. Кнут хорошо знаком советскому читателю по переводам двух первых томов его обширной монографии "Искусство программирования для ЭВМ" и не нуждается в аттестации. Настоящая книга представляет собой третий том и посвящена алгоритмам сортировки и поиска информации.

Исторически зарождение методов машинной сортировки можно отнести еще к прошлому столетию, и за столь длительное время многие специалисты успели испробовать свои силы в этой области. Написано немало отчетов, статей, монографий. И даже в этих условиях книга Д. Кнута стала событием. По существу это энциклопедия, в которой можно найти любую справку, касающуюся алгоритмов, методов их оценок, истории вопроса и нерешенных проблем.

Нет нужды говорить о важности самой области. Практически сортировка и поиск в той или иной мере присутствуют во всех приложениях; в частности, при обработке больших объемов данных эффективность именно этих операций определяет эффективность, а иногда и работоспособность всей системы. Поэтому, как справедливо отмечает автор, книга адресована не только системным программистам, занимающимся разработкой программ сортировки и поиска информации. Можно сказать, что достаточно четкие представления об этой области нужны при решении любой задачи на ЭВМ как обязательные элементы искусства программирования.

Кроме теоретической и практической ценности, книга имеет большое методическое значение. Многие авторы и преподаватели смогут извлечь из нее новые и полезные сведения не только по существу рассматриваемых вопросов, но и по способу их изложения. Автору мастерски удается "рас-слоить" весь материал таким образом, что книгу можно использовать практически на любом уровне знакомства с предметом и при различной общей математической подготовленности читателя.

Перевод выполнен по изданию 1973 г. (первая редакция) с внесением многих (около 700) исправлений и добавлений, любезно предоставленных автором. Разделы с 5.1 по 5.3.2 переведены. Н. И. Вьюковой; разделы с 5.3.3 по 5.5 и предисловие — А. Б. Ходулевым; главу 6 перевел В. А. Галатенко.

Ю. М. Баяковский
В. С. Штаркман

ПРЕДИСЛОВИЕ

Кулинария стала искусством, высокой наукой;
повара теперь — благородные люди.
Тит Ливий, *Ab Urbe Condita*, XXXIX.vi
(Роберт Бертон, *Anatomy of Melancholy*, 1.2.2.2)

Материал этой книги логически продолжает материал по информационным структурам, изложенный в гл. 2, поскольку здесь к уже рассмотренным концепциям структур добавляется понятие линейно упорядоченных данных. Подзаголовок "Сортировка и поиск" может привести к мысли, что эта книга предназначена лишь для системных программистов, занимающихся построением универсальных программ сортировки или связанных с выборкой информации. Однако в действительности предмет сортировки и поиска дает нам прекрасную основу для обсуждения широкого класса важных общих вопросов:

Как находить хорошие алгоритмы?

Как улучшать данные алгоритмы и программы?

Как исследовать эффективность алгоритмов математически?

Как разумно выбрать один из нескольких алгоритмов для решения конкретной задачи?

В каком смысле можно доказать, что некоторые алгоритмы являются "наилучшими из возможных"?

Как теория вычислений согласуется с практическими соображениями?

Как эффективно использовать различные виды внешней памяти — ленты, барабаны, диски — для больших баз данных? Я думаю, что на самом деле в контексте сортировки и поиска встречается практически *любой* важный аспект программирования.

Настоящий том состоит из гл. 5 и 6 монографии. В гл. 5 рассматривается сортировка (упорядочение); это очень большая тема, она разбита на две главные части — внутреннюю и внешнюю сортировку. В эту главу входят также дополнительные разделы, развивающие вспомогательную теорию перестановок (§ 5.1) и теорию оптимальных алгоритмов сортировки (§ 5.3). В гл. 6 мы имеем дело с поиском определенного элемента в таблице или файле; содержимое этой главы подразделяется на

методы последовательного поиска, методы поиска со сравнением ключей, поиска с использованием свойств цифр, поиска с помощью "хеширования"; затем рассматривается более сложная задача выборки по вторичным ключам. Обе главы поразительно тесно переплетаются между собой, между их предметами имеются близкие аналогии. В дополнение к гл. 2 рассматриваются два важных вида информационных структур, а именно приоритетные очереди (п. 5.2.3) и линейные списки, представляемые посредством сбалансированных деревьев (п. 6.2.3).

Читателю, не знакомому с первым томом этой монографии, рекомендуется обращаться к указателю обозначений (приложение В), так как некоторые из встречающихся в книге обозначений не являются общепринятыми.

Эта книга без большей части математического материала была использована мной в качестве учебника по второму курсу лекций "Структуры данных" для студентов младших и средних курсов. Математические части этой книги, особенно § 5.1, п. 5.2.2, § 6.3 и 6.4, могли бы составить учебник по анализу алгоритмов для студентов средних и старших курсов. Кроме того, на основе п. 4.3.3, 4.6.3, 4.6.4, § 5.3 и п. 5.4.4 можно построить курс лекций "Сложность вычислений" для старшекурсников.

Быстрое развитие информатики и вычислительных наук задержало выход в свет этой книги почти на три года, поскольку очень многие аспекты сортировки и поиска подвергались детальной разработке. Я очень благодарен Национальному научному фонду, Отделению военно-морских исследований, Институту обороны, фирмам IBM и Norges Almementenskapelige Forskningsrad за постоянную поддержку моих исследований.

В подготовке этого тома к печати мне оказали помощь многие лица, особенно Эдвард А. Бендер, Кларк Э. Крэйн, Дэвид Э. Фергюсон, Роберт У. Флloyd, Рональд Л. Грэхем, Леонидас Гюйба, Джон Хопкрофт, Ричард М. Карп, Гэри Д. Кнотт, Рудольф А. Крутар, Шень Линь, Воган Р. Пратт, Стефан О. Райе, Ричард П; Стэнли, Я. А. ван дер Пул и Джон У. Ренч мл., а также студенты Стэнфорда и Беркли, которым пришлось искать ошибки в рукописи.

Д. Э. Кнут сентябрь 1972

Писатель пользуется известными привилегиями, в благодетельности которых, надеюсь, нет никаких оснований сомневаться. Так, встретив у меня непонятное место, читатель должен предположить, что под ним кроется нечто весьма полезное и глубокомысленное.¹

(Джонатан Свифт, Сказка бочки, предисловие, 1704)

ЗАМЕЧАНИЯ ОБ УПРАЖНЕНИЯХ

Упражнения, помещенные в книгах настоящей серии, предназначены как для самостоятельной проработки, так и для семинарских занятий. Трудно, если не невозможно изучить предмет, только читая теорию и не применяя полученную информацию для решения специальных задач и тем самым не заставляя себя обдумывать то, что было прочитано. Кроме того, мы лучше всего заучиваем то, что сами открываем для себя. Поэтому упражнения образуют важную часть данной работы; были предприняты определенные попытки, чтобы отобрать упражнения, в которых бы содержалось как можно больше информации и которые было бы интересно решать.

Во многих книгах легкие упражнения даются вперемешку с исключительно трудными. Зачастую это очень неудобно, так как перед тем, как приступить к решению задачи, читатель обязательно должен представлять себе, сколько времени уйдет у него на это решение (иначе он может разве только просмотреть все задачи). Классическим примером здесь является книга Ричарда Беллмана "Динамическое программирование"; это важная пионерская работа, в которой в конце каждой главы под рубрикой "Упражнения и исследовательские проблемы" дается целый ряд задач, где наряду с глубокими еще нерешенными проблемами встречаются исключительно тривиальные вопросы. Говорят, что однажды кто-то спросил д-ра Беллмана, как отличить упражнения от исследовательских проблем, и тот ответил: "Если вы можете решить задачу, это — упражнение; в противном случае это — проблема".

Можно привести много доводов в пользу того, что в книге типа этой должны быть как исследовательские проблемы, так и очень простые упражнения, и для того чтобы читателю не приходилось ломать голову над тем, какая задача легкая, а какая трудная, мы ввели "оценки", которые указывают степень трудности каждого упражнения. Эти оценки имеют следующее значение:

¹ Перевод А. А. Франковского. — Прим. Перев.

Оценка Объяснение

- 00 Чрезвычайно легкое упражнение, на которое можно ответить сразу же, если понят материал текста, и которое почти всегда можно решить "в уме".
- 10 Простая задача, которая заставляет задуматься над прочитанным материалом, но не представляет никаких особых трудностей. На решение такой задачи требуется не больше одной минуты; в процессе решения могут понадобиться карандаш и бумага.
- 20 Задача средней трудности, позволяющая проверить, насколько хорошо понят текст. На то чтобы дать исчерпывающий ответ, требуется примерно 15–20 минут.
- 30 Задача умеренной трудности и/или сложности, для удовлетворительного решения которой требуется больше двух часов.
- 40 Очень трудная или трудоемкая задача, которую, вероятно, следует включить в план практических занятий. Предполагается, что студент может решить такую задачу, но для этого ему потребуется значительный отрезок времени; задача решается нетривиальным образом.
- 50 Исследовательская проблема, которая (насколько это было известно автору в момент написания) еще не получила удовлетворительного решения. Если читатель найдет решение этой задачи, его настоятельно просят опубликовать его; кроме того, автор данной книги будет очень признателен, если ему сообщат решение как можно быстрее (при условии, что оно правильно).

Интерполируя по этой "логарифмической" шкале, можно прикинуть, что означает любая промежуточная оценка. Например, оценка 17 говорит о том, что данное упражнение чуть легче, чем упражнение средней трудности. Задача с оценкой 50, если она будет решена каким-либо читателем, в следующих изданиях данной книги может иметь уже оценку 45.

Автор честно старался давать объективные оценки, но тому, кто составляет задачи, трудно предвидеть, насколько трудными эти задачи окажутся для кого-то другого; к тому же у каждого человека существует определенный тип задач, которые он решает быстрее. Надеюсь, что выставленные мной оценки дают правильное представление о степени трудности задач, но в общем их нужно воспринимать как ориентировочные, а не абсолютные.

Эта книга написана для читателей самых разных степеней математической подготовки и искушенности, поэтому некоторые упражнения предназначены только для читателей с математическим уклоном. Если в каком-либо упражнении математические понятия или результаты используются более широко, чем это необходимо для тех, кого в первую очередь интересует программирование алгоритмов, то перед оценкой такого упражнения ставится буква "М". Если для решения упражнения требуется знание высшей математики в большем объеме, чем это дано в настоящей книге, то ставятся буквы "ВМ". Пометка "ВМ" отнюдь не является свидетельством того, что данное упражнение трудное.

Перед некоторыми упражнениями стоит стрелка ">"; это означает, что данное упражнение особенно поучительно и его рекомендуется обязательно выполнить. Само собой разумеется, никто не ожидает, что читатель (или студент) будет решать все задачи, потому-то наиболее полезные из них и выделены. Это совсем не значит, что другие задачи не стоит решать! Каждый читатель должен по крайней мере попытаться решить все задачи с оценкой 10 и ниже; стрелки же помогут выбрать, какие задачи с более высокими оценками следует решить в первую очередь.

К большинству упражнений приведены ответы; они помещены в специальном разделе в конце книги. Пользуйтесь ими мудро; в ответ смотрите только после того, как вы приложили достаточно усилий, чтобы решить задачу самостоятельно, или же если для решения данной задачи у вас нет времени. Если получен собственный ответ, либо если вы действительно пытались решить задачу, только в этом случае ответ, помещенный в книге, будет поучительным и полезным. Как правило, ответы к задачам излагаются очень кратко, схематично, так как предполагается, что читатель уже честно пытался решить задачу собственными силами. Иногда в приведенном решении дается меньше информации, чем спрашивалось, чаще — наоборот. Вполне возможно, что полученный вами ответ окажется лучше ответа, помещенного в книге, или вы найдете ошибку в этом ответе; в таком случае автор был бы очень обязан, если бы вы как можно скорее подробно сообщили ему об этом. В последующих изданиях настоящей книги будет помещено уже исправленное решение вместе с именем его автора.

Сводка условных обозначений

- > Рекомендуется
- М С математическим уклоном
- ВМ Требуется знания "высшей математики"
- 00 Требуется немедленного ответа
- 10 Простое (на одну минуту)
- 20 Средней трудности (на четверть часа)

- 30 Повышенной трудности.
 40 Для "матпрактикума"
 50 Исследовательская проблема

Упражнения

- >1. [00] Что означает пометка "М20"?
 2. [10] Какое значение для читателя имеют упражнения, помещаемые в учебниках?
 3. [М50] Докажите, что если n — целое число, $n > 2$, то уравнение $x^n + y^n = z^n$ неразрешимо в целых положительных числах x, y, z .

5. Сортировка

Нет дела более трудного по замыслу, более сомнительного по успеху, более опасного при осуществлении, чем вводить новые порядки.

Никколо Макьявелли, "Государь" (1513)

"Но мы не успеем, просмотреть все номера автомобилей", — возразил Дрейк. "А нам и не нужно этого делать. Пол. Мы просто расположим их по порядку и поищем одинаковые".

Перри Мейсон * Из "The Case of Angry Mourner" (1951)

Сортировка деревьев с использованием ЭВМ.

При таком новом, "машинном подходе" к изучению природы вы получите возможность быстро распознавать более 260 различных деревьев США, Аляски, Канады, включая пальмы, деревья пустынь и прочую экзотику. Чтобы определить породу дерева, достаточно просто вставить спицу.

Каталог "Edmund Scientific Company" (1964)

В этой главе мы изучим вопрос, который часто возникает в программировании: переразмещение элементов в возрастающем или убывающем порядке. Представьте, насколько трудно было бы пользоваться словарем, если бы слова в нем не располагались в алфавитном порядке. Точно так же от порядка, в котором хранятся элементы в памяти ЭВМ, во многом зависит скорость и простота алгоритмов, предназначенных для их обработки.

Хотя в словарях слово "сортировка" (sorting) определяется как "распределение, отбор по сортам; деление на категории, сорта, разряды", программисты традиционно используют это слово в гораздо более узком смысле, обозначая им сортировку предметов в возрастающем или убывающем порядке. Этот процесс, пожалуй, следовало бы назвать не сортировкой, а *упорядочением* (ordering), но использование этого слова привело бы к путанице из-за перегруженности значениями слова "порядок". Рассмотрим, например, следующее предложение: "Так как только два наших лентопротяжных устройства в порядке, меня призвали к порядку и обязали в срочном порядке заказать еще несколько устройств, чтобы можно было упорядочивать данные разного порядка на несколько порядков быстрее¹. В математической терминологии это слово также изобилует значениями (порядок группы, порядок перестановки, порядок точки ветвления, отношение порядка и т. п.). Итак, слово "порядок" приводит к хаосу.

В качестве обозначения для процесса упорядочения предлагалось также слово "ранжирование"², но оно во многих случаях, по-видимому, не вполне отражает суть дела, особенно если присутствуют равные элементы, и, кроме того, иногда не согласуется с другими терминами. Конечно, слово

* Перри Мейсон — герой серии детективных романов популярного американского писателя Эрла Стенли Гарднера. — *Прим. перев.*

¹ В оригинале "Since only two of our tape drives were in working order I was ordered to order more tape units in short order, in order to order the data several orders of magnitude faster". — *Прим. перев.*

² В оригинале "sequencing". — *Прим. перев.*

"сортировка" и само имеет довольно много значений³, но оно прочно вошло в программистский жаргон. Поэтому мы без дальнейших извинений будем использовать слово "сортировка" в узком смысле "сортировка по порядку".

Вот некоторые из наиболее важных применений сортировки:

- а) Решение задачи "группировки", когда нужно собрать вместе все элементы с одинаковым значением некоторого признака. Допустим, имеется 10000 элементов, расположенных в случайном порядке, причем значения многих из них равны; и предположим, нам нужно переупорядочить файл так, чтобы элементы с равными значениями занимали соседние позиции в файле. Это, по существу, задача "сортировки" в широком смысле слова, и она легко может быть решена путем сортировки файла в узком смысле слова, а именно расположением элементов в неубывающем порядке $v_1 \leq v_2 \leq \dots \leq v_{10000}$. Эффективностью, которая может быть достигнута в этой процедуре, и объясняется изменение первоначального смысла слова "сортировка".
- б) Если два или более файла отсортировать в одном и том же порядке, то можно отыскать в них все общие элементы за один последовательный просмотр всех файлов, без возвратов. Это тот самый принцип, которым воспользовался Перри Мейсон для раскрытия дела об убийстве (см. эпиграфы к этой главе). Оказывается, что, как правило, гораздо экономнее просматривать список последовательно, а не перескакивая с места на место случайным образом, если только список не настолько мал, что он целиком помещается в оперативной памяти. Сортировка позволяет использовать последовательный доступ к большим файлам в качестве приемлемой замены прямой адресации.
- с) Как мы увидим в гл. 6, сортировка помогает и при поиске, с ее помощью можно сделать выдачи ЭВМ более удобными для человеческого восприятия. В самом деле, листинг (напечатанный машиной документ), отсортированный в алфавитном порядке, зачастую выглядит весьма внушительно, даже если соответствующие числовые данные были рассчитаны неверно.

Хотя сортировка традиционно и большей частью использовалась для обработки коммерческих данных, на самом деле она является инструментом, полезным в самых разных ситуациях, и поэтому о нем не следует забывать; В упр. 2.3.2–17 мы обсудили ее применение для упрощения алгебраических формул. Упражнения, приведенные ниже, иллюстрируют разнообразие типичных применений сортировки.

Одной из первых крупных систем программного обеспечения, продемонстрировавших богатые возможности сортировки, был компилятор *Larc Scientific Compiler*, разработанный фирмой *Computer Sciences Corporation* в 1960 г. В этом оптимизирующем компиляторе для расширенного ФОРТРАНа сортировка использовалась весьма интенсивно, так что различные алгоритмы компиляции работали с относящимися к ним частями исходной программы, расположенными в удобной последовательности. При первом просмотре осуществлялся лексический анализ, т. е. выделение в исходной программе лексических единиц (лексем), каждая из которых соответствует либо идентификатору (имени переменной), либо константе, либо оператору и т. д. Каждая лексема получала несколько порядковых номеров. В результате сортировки по именам и соответствующим порядковым номерам все использования данного идентификатора оказывались собранными вместе. "Определяющие вхождения", специфицирующие идентификатор как имя функции, параметр или многомерную переменную, получали меньшие номера, поэтому они оказывались первыми в последовательности лексем, отвечающих этому идентификатору. Тем самым облегчалась проверка правильности употребления идентификаторов, распределение памяти с учетом деклараций эквивалентности и т. д. Собранный таким образом информация о каждом идентификаторе присоединялась к соответствующей лексеме. Поэтому не было необходимости хранить в оперативной памяти "таблицу символов", содержащую сведения об идентификаторах. После такой обработки лексемы снова сортировались по другому порядковому номеру; в результате в программе, по существу, восстанавливался первоначальный порядок, если не считать того, что арифметические выражения оказывались записанными в более удобной, "польской префиксной" форме. Сортировка использовалась и на последующих фазах компиляции — для облегчения оптимизации циклов, включения в листинг сообщений об ошибках и т. д. Короче говоря, компилятор был устроен так, что всю обработку файлов, хранящихся на барабанах, фактически можно было вести последовательно. Поэтому-то данные и снабжались такими порядковыми номерами, которые позволяли упорядочивать эти данные различными удобными способами.

Другое, более очевидное применение сортировки возникает при редактировании файлов, где каждая строка снабжена ключом. Пока пользователь вносит с клавиатуры изменения и добавления,

³ Это в большей степени относится к английскому слову "sorting". Здесь автор приводят пример: "He was sort of out sorts after sorting that sort of data". (Он был как будто не в духе после сортировки такого сорта денных), который в русском переводе не столь выразителен. — *Прим. перев.*

необязательно держать весь файл в оперативной памяти. Все изменяемые строки можно позднее отсортировать (а они и так обычно в основном упорядочены) и слить с исходным файлом. Это дает возможность разумно использовать память в режиме мультипрограммирования. [Ср. с С. С. Foster, *Comp. J.*, 11 (1968), 134–137].

Поставщики вычислительных машин считают, что в среднем более 25% машинного времени систематически тратится на сортировку. Во многих вычислительных системах на нее уходит больше половины машинного времени. Из этой статистики можно заключить, что либо (i) сортировка имеет много важных применений, либо (ii) ею часто пользуются без нужды, либо (iii) применяются в основном неэффективные алгоритмы сортировки. По-видимому, каждое из трех предположений содержит долю истины. Во всяком случае ясно, что сортировка заслуживает серьезного изучения с точки зрения ее практического использования.

Но даже если бы сортировка была почти бесполезна, нашлась бы масса других причин заняться ею! Изобретательные алгоритмы сортировки говорят о том, что она и сама по себе интересна как объект исследования. В этой области существует множество увлекательных нерешенных задач наряду с весьма немногими уже решенными.

Рассматривая вопрос в более широком плане, мы обнаружим, что алгоритмы сортировки представляют собой интересный частный пример того, как следует подходить к решению проблем программирования вообще. Мы познакомимся со многими важными принципами манипулирования структурами данных и проследим за эволюцией различных методов сортировки, причем читателю часто будет предоставляться возможность самому "открывать" те же идеи, как будто бы до него никто с подобными задачами не сталкивался. Обобщение этих частных методов позволит нам в значительной степени овладеть теми способами мышления, которые помогут создавать добротные алгоритмы для решения других проблем, связанных с ЭВМ.

Методы сортировки служат великолепной иллюстрацией идей *анализа алгоритмов*, т. е. идей, позволяющих оценивать рабочие характеристики алгоритмов, а значит, разумно выбирать среди, казалось бы, равноценных методов. Читатели, имеющие склонность к математике, найдут в этой главе немало способов оценки скорости работы алгоритмов и методов решения сложных рекуррентных соотношений. С другой стороны, изложение построено так, что читатели, не имеющие такой склонности, могут безболезненно пропускать выкладки.

Прежде чем двигаться дальше, необходимо более четко определить задачу и ввести соответствующую терминологию. Пусть надо упорядочить N элементов

$$R_1, R_2, \dots, R_N.$$

Назовем их *записями*, а всю совокупность N записей назовем *файлом*. Каждая запись R_j имеет *ключ* K_j , который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную, "сопутствующую информацию", которая не влияет на сортировку, но всегда остается в этой записи.

Отношение порядка " $<$ " на множестве ключей вводится таким образом, чтобы для любых трех значений ключей a, b, c выполнялись следующие условия:

- i) справедливо одно и только одно из соотношений $a < b$, $a = b$, $b < a$ (закон трихотомии);
- ii) если $a < b$ и $b < c$, то $a < c$ (закон транзитивности).

Эти два свойства определяют математическое понятие *линейного упорядочения*, называемого еще *совершенным упорядочением*. Любое множество с отношением " $<$ ", удовлетворяющим свойствам (i) и (ii), поддается сортировке большинством методов, описанных в этой главе, хотя некоторые из них годятся только для числовых и буквенных ключей с обычным отношением порядка.

Задача сортировки — найти такую перестановку записей $p(1) p(2) \dots p(N)$, после которой ключи расположились бы в неубывающем порядке:

$$K_{p(1)} \leq K_{p(2)} \leq \dots \leq K_{p(n)}. \quad (1)$$

Сортировка называется *устойчивой*, если она удовлетворяет дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке, т. е.

$$p(i) < p(j), \quad \text{если } K_{p(i)} = K_{p(j)} \text{ и } i < j. \quad (2)$$

В ряде случаев может потребоваться физически перемещать записи в памяти так, чтобы их ключи, были упорядочены; в других случаях достаточно создать вспомогательную таблицу, которая некоторым образом описывает перестановку и обеспечивает доступ к записям в соответствии с порядком их ключей.

Некоторые методы сортировки предполагают существование величин " ∞ " и " $-\infty$ " или одной из них. Величина " ∞ " считается больше, а величина " $-\infty$ " меньше любого ключа:

$$-\infty < K_j < \infty, \quad 1 \leq j \leq N. \quad (3)$$

Эти величины используются в качестве искусственных ключей, а также как граничные признаки. Равенство в (3), вообще говоря, исключено. Если же оно тем не менее допускается, алгоритмы можно модифицировать так, чтобы они все-таки работали, хотя нередко при этом их изящество и эффективность отчасти утрачиваются.

Обычно сортировку подразделяют на два класса: *внутреннюю*, когда все записи хранятся в быстрой оперативной памяти, и *внешнюю*, когда все они там не помещаются. При внутренней сортировке имеются более гибкие возможности для построения структур данных и доступа к ним, внешняя же показывает, как поступать в условиях сильно ограниченного доступа.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время порядка $N \log N$; при этом требуется около $\log N$ "проходов" по данным. Как мы увидим в п. 5.3.1, это минимальное время. Так, если удвоить число записей, то и время при прочих равных условиях возрастет немногим более чем вдвое. (На самом деле, когда N стремится к ∞ , время "растет" как $N(\log N)^2$, если все ключи различны, так как и размеры ключей увеличиваются с ростом N ; но практически N всегда остается ограниченным.)

Упражнения

(ПЕРВАЯ ЧАСТЬ)

- [M20] Докажите, что из законов трихотомии и транзитивности вытекает *единственность* перестановки $p(1), p(2), \dots, p(N)$, если сортировка устойчива.
- [21] Пусть каждая запись R . некоторого файла имеет *два* ключа: "большой ключ" K_j и "малый ключ" k_j , причем оба множества ключей линейно упорядочены. Тогда можно обычным способом ввести "лексикографический порядок" на множестве пар ключей (K, k) :

$$(K_i, k_i) < (K_j, k_j), \quad \text{если } K_i < K_j \text{ или если } K_i = K_j \text{ и } k_i < k_j.$$

Некто (назовем его мистер А) отсортировал этот файл сначала по большим ключам, получив n групп записей с одинаковыми большими ключами в каждой группе:

$$K_{p(1)} = \dots = K_{p(i_1)} < K_{p(i_1+1)} = \dots = K_{p(i_2)} < \dots < K_{p(i_{n-1}+1)} = \dots = K_{p(i_n)},$$

где $i_n = N$. Затем каждую из n групп $R_{p(i_{j-1}+1)}, \dots, R_{p(i_j)}$ он отсортировал по малым ключам.

Тот же исходный файл взял мистер В и отсортировал его сначала по малым ключам, а потом получившийся файл отсортировал по большим ключам.

Взяв тот же исходный файл, мистер С отсортировал его один раз, пользуясь лексикографическим порядком между парами ключей (K_j, k_j) .

Получилось. ли у всех троих одно и то же?

- [M25] Пусть на множестве K_1, \dots, K_N определено отношение $<$, для которого закон трихотомии выполняется, а закон транзитивности — *нет*. Докажите, что и в этом случае возможна устойчивая сортировка записей, т. е. такая, что выполняются условия (1) и (2); на самом деле существуют по крайней мере три расположения записей, удовлетворяющих этим условиям!
- [15] Мистер Тупица (программист) захотел узнать, находится ли в ячейке А машины MIX число, большее числа из ячейки В, меньшее или же равное ему. Он написал

LDAA

SUBB

а потом проверил, какой результат получился в rA: положительное число, отрицательное или нуль. Какую серьезную ошибку он допустил и как должен был поступить?

- [17] Напишите MIX-подпрограмму для сравнения ключей, занимающих несколько слов, исходя из следующих условий:

Вызов:	JMP COMPARE
Состояние при входе:	rI1 = n ; CONTENTS($A + k$) = a_k , CONTENTS($B + k$) = b_k при
	$1 \leq k \leq n$; предполагается, что $n \geq 1$.
Состояние при выходе:	CI = +1, если $(a_n, \dots, a_1) > (b_n, \dots, b_1)$;
	CI = 0, если $(a_n, \dots, a_1) = (b_n, \dots, b_1)$;
	CI = -1, если $(a_n, \dots, a_1) < (b_n, \dots, b_1)$;
	rX и rI1, возможно, изменились.

Здесь отношение $(a_n, \dots, a_1) < (b_n, \dots, b_1)$ обозначает лексикографическое упорядочение слева направо, т. е. существует индекс j , такой, что $a_k = b_k$ при $n \geq k > j$, но $a_j < b_j$.

6. [30] В ячейках A и B содержатся соответственно числа a и b . Покажите, что можно написать MIX-программу, которая бы вычисляла $\min(a, b)$ и записывала результат в ячейку C, *не пользуясь командами перехода*. (Предостережение: поскольку арифметическое переполнение невозможно обнаружить без команд перехода, разумно так построить программу, чтобы переполнение не могло возникнуть ни при каких значениях a и b)
7. [M27] Какова вероятность того, что после сортировки в неубывающем порядке N независимых равномерно распределенных на отрезке $[0, 1]$ случайных величин r -е от начала число окажется $\leq x$?

Упражнения

УПРАЖНЕНИЯ (ВТОРАЯ ЧАСТЬ) В каждом из этих упражнений поставлена задача, с которой может столкнуться программист. Предложите "хорошее" решение задачи, предполагая, что имеется сравнительно небольшая оперативная память и около полудюжины, лентопротяжных устройств (этого количества достаточно для сортировки).

1. [75] Имеется лента, на которой записан миллион слов данных. Как определить, сколько на этой ленте различных слов?
2. [18] Вообразите себя в роли Управления внутренних доходов Министерства финансов США. Вы получаете миллионы "информационных" карточек от организаций о том, сколько, денег они выплачивали различным лицам, и миллионы "налоговых" карточек от различных лиц об их доходах. Как бы вы стали отыскивать людей, которые сообщили не обо всех своих доходах?
3. [M25] (*Транспонирование матрицы.*) Имеется магнитная лента, содержащая миллион слов, которые представляют собой элементы 1000×1000 -матрицы, записанные по строкам: $a_{1,1} a_{1,2} \dots a_{1,1000} a_{2,1} \dots a_{2,1000} \dots a_{1000,1000}$. Ваша задача — получить ленту, на которой элементы этой матрицы были бы записаны по столбцам: $a_{1,1} a_{2,1} \dots a_{1000,1} a_{1,2} \dots a_{1000,2} \dots a_{1000,1000}$ (Постарайтесь сделать не более десяти просмотров данных.)
4. [M26] В вашем распоряжении довольно большой файл из N слов. Как бы вы его "перетасовали" случайным образом?
- >5. [24] В некоем университете работает около 1000 преподавателей и имеется 500 комитетов. Считается, что каждый преподаватель является членом по крайней мере двух комитетов. Вам нужно подготовить с помощью машины удобочитаемые списки членов всех комитетов. Вы располагаете колодой из приблизительно 1500 перфокарт, сложенных произвольным образом и содержащих следующую информацию: *Членские карточки*: колонка 1 — пробел; колонки 2–18 — фамилия с последующими пробелами; колонки 19–20 — инициалы; колонки 21–23 — номер первого Комитета; колонки 24–26 — номер второго комитета; ... ; колонки 78–80 — номер двадцатого комитета (если нужно) или пробелы.

Комитетские карточки: колонка 1 — "*"; колонки 2–77 — название комитета; колонки 78–80 — номер комитета. Как вы должны действовать? (Опишите свой метод достаточно подробно.)

6. [20] Вы работаете с двумя вычислительными системами, в которых по-разному упорядочены литеры (буквы и цифры). Как заставить первую ЭВМ сортировать файлы с буквенно-цифровой информацией, предназначенные для использования на второй ЭВМ?
7. [18] Имеется довольно большой список людей, родившихся в США, с указанием штата, в котором они родились. Как подсчитать число людей, родившихся в каждом штате? (Предполагается, что ни один человек не указан в списке более одного раза.)
8. [20] Чтобы облегчить внесение изменений в большие программы, написанные на ФОРТРАНе, вы хотите написать программу, выпечатающую таблицу "перекрестных ссылок". Входными данными для нее служит программа на ФОРТРАНе, а в результате получается листинг исходной программы, снабженный указателем всех случаев употребления каждого идентификатора (т. е. имени) в программе. Как написать такую программу?
9. [33] (Сортировка каталожных карточек.) Способы составления алфавитных каталогов в разных библиотеках несколько отличаются друг от друга. В следующем "алфавитном" списке содержатся рекомендации, взятые из правил регистрации и хранения каталожных карточек Американ-

ской библиотечной ассоциации (Чикаго, 1942):

Текст карточки

R. Accademia nazionale dei Lincei, Rome
1812; ein historischer roman.
Bibliothèque d'histoire révolutionnaire.
Bibliothèque des curiosités.
Brown, Mrs. J. Crosby
Brown, John
Brown, John, mathematician
Brown, John, of Boston
Brown, John, 1715–1766
BROWN, JOHN, 1715–1766
Brown, John, d. 1811
Brown, Dr. John, 1810–1882
Brown-Williams, Reginald Makepeace
Brown America.
Brown & Dallison's Nevada directory.
Brownjohn, Alan
Den', Vladimir Eduardovich, 1867—
The den.
Den lieben süßen mädeln.
Dix, Morgan, 1827–1908
1812 ouverture.
Le XIXe siècle français.
The 1847 issue of U. S. stamps.
1812 overture.
I am a mathematician,
IBM journal of research and development.
ha-I ha-eḥad.
Ia; a love story.
International Business Machines Corporation
al-Khuwārizmī, Muḥammad ibn Mūsā, *fl.* 813–846
Labour; a magazine for all workers.
Labor research association
Labour, *see* Labor
McCall's cookbook
McCarthy, John, 1927—
Machine-independent computer programming.
MacMahon, Maj. Percy Alexander, 1854–1929
Mrs, Dalloway.
Mistress of mistresses.
Royal society of London
St. Petersburger Zeitung.
Saint-Saëns, Camille, 1835–1921
Ste. Anne des Monts, Quebec
Seminumerical algorithms.
Uncle Tom's cabin.
U.S. Bureau of the census.
Vandermonde, Alexander Théophile, 1735–1796
Van Valkenburg, Mac Elwyn, 1921 —
Von Neumann, John, 1903–1957
The whole art of legerdemain.
Who's afraid of Virginia Woolf?
Wijngaarden, Adriaan van, 1916—

Замечания

В названиях иностранных (кроме британских) учреждений
Achtzehnhundert zwölf
Во французском тексте апостроф рассматривается как пробел
Надстрочные знаки игнорируются
Указание положения (Mrs.) игнорируется
Фамилии с датами следуют за фамилиями без дат . . .
. . . которые упорядочиваются по
описательным словам
Одинаковые фамилии упорядочиваются по датам рождения
Работы "о нем" идут после его работ
Иногда год рождения определяется приблизительно
Указание положения (Dr.) игнорируется
Дефис рассматривается как пробел
Названия книг идут после составных фамилий
& в английском тексте превращается в "and"

Апостроф в именах игнорируется
Артикль в начале текста игнорируется
. . . если существительное стоит в именительном падеже
Фамилии идут раньше других слов
Dix-huit cent douze
Dix-neuvième
Eighteen forty-seven
Eighteen twelve
(by Norbert Weiner)
Аббревиатуры рассматриваются как ряд однобуквенных слов
Артикль в начале текста игнорируется
Знаки препинания в названиях игнорируются

Начальное "al-" в арабских именах игнорируется
Заменяется на "Labor"

Ссылка на другую карточку в картотеке
Апостроф в английском тексте игнорируется
Mc = Mac
Дефис рассматривается как пробел
Указание положения (Maj) игнорируется
"Mrs." = "Mistress"

"St." = "Saint" даже в немецком тексте
Дефис рассматривается как пробел
Sainte

"U.S." = "United States"

Пробел после префикса в фамилиях игнорируется

Апостроф в английском тексте игнорируется
Фамилия никогда не начинается с малой буквы

(У большинства из этих правил есть исключения; кроме того, существует много других правил, которые здесь не упомянуты.) Предположим, вам пришлось сортировать большое количество таких карточек с помощью вычислительной машины и впоследствии обслуживать огромную картотеку, причем у вас нет возможности изменить уже сложившиеся порядки заполнения карточек.

Как бы вы организовали информацию, чтобы упростить операции включения новых карточек и сортировки?

10. [M21](*Дискретные логарифмы.*) Пусть известно, что p — (довольно большое) простое число, а a — первообразный корень по модулю p . Следовательно, для любого b в диапазоне $1 \leq b < p$ существует единственное n , такое, что $a^n \bmod p = b$, $1 \leq n < p$. Как по заданному b найти n менее чем за $O(n)$ шагов? [Указание. Пусть $m = \lceil \sqrt{p} \rceil$. Попытайтесь решить уравнение $a^{mn_1} \equiv ba^{-n_2} \pmod{p}$ при $0 \leq n_1, n_2 < m$.]
11. [M25] (Э. Т. Паркер.) Эйлер выдвинул предположение, что уравнение

$$u^6 + v^6 + w^6 + x^6 + y^6 = z^6$$

не имеет решений (за исключением тривиальных) среди целых неотрицательных чисел u, v, w, x, y, z , когда по крайней мере четыре переменные равны нулю. Помимо этого, он предполагал, что уравнение

$$x_1^n + \dots + x_{n-1}^n = x_n^n$$

не имеет нетривиальных решений при $n \geq 3$, но это предположение было опровергнуто: с помощью вычислительной машины найдено тождество $27^5 + 84^5 + 110^5 + 133^5 = 144^5$; см. Л. Дж. Лэндер, Т. Р. Паркин и Дж. Л. Селфридж, *Math. Comp.*, **21** (1967), 446–459. Придумайте, как можно было бы использовать сортировку для поиска примеров, опровергающих предположение Эйлера при $n = 6$.

- >12. [24] Файл содержит большое количество 30-разрядных двоичных слов: x_1, \dots, x_N . Придумайте хороший способ нахождения в нем всех *дополнительных* пар (x_i, x_j) . (Два слова называются дополнительными, если второе содержит 0 во всех разрядах, в которых были 1 в первом слове, и наоборот; таким образом, они дополнительные тогда и только тогда, когда их сумма равна $(11 \dots 1)_2$, если они рассматриваются как двоичные числа.)
- >13. [25] Имеется файл, содержащий 1000 30-разрядных двоичных слов x_1, \dots, x_{1000} . Как бы вы стали составлять список всех пар (x_i, x_j) , таких, что x_i отличается от x_j не более чем в двух разрядах?
14. [22] Как бы вы поступили при отыскании всех пятибуквенных анаграмм, таких, как CARET, CARTE, CATER, CRATE, REACT, TRACE; CRUEL, LUCRE, ULCER; DOWRY, ROWDY, WORDY? [Если бы вы, скажем, захотели узнать, существуют ли в английском языке наборы из десяти или более анаграмм, кроме замечательной серии:

APERS, ASPER, PARES, PARSE, PEARS, PRASE, RAPES, REAPS, SPARE, SPEAR,

к которой можно добавить еще французское слово APRES.]

15. [M28] Пусть даны описания весьма большого числа направленных графов. Каким путем можно сгруппировать *изоморфные* графы? (Два направленных графа называются изоморфными, если существует взаимно однозначное соответствие между их вершинами и взаимно однозначное соответствие между их дугами, причем эти соответствия сохраняют инцидентность вершин и дуг.)
16. [30] В некоторой группе из 4096 человек каждый имеет около 100 знакомых. Был составлен список всех пар людей, знакомых между собой. (Это отношение симметрично, т. е. если x знаком с y , то и y знаком с x . Поэтому список содержит примерно 200000 пар.) Придумайте алгоритм, который по заданному k выдавал бы все клики из k человек. (Клика — это группа людей, в которой все между собой знакомы.) Предполагается, что слишком больших клик не бывает.
17. [30] Три миллиона человек с различными именами были уложены рядом непрерывной цепочкой от Нью-Йорка до Калифорнии. Каждому из них дали листок бумаги, на котором он написал свое имя и имя своего ближайшего западного соседа. Человек, находившийся в самой западной точке цепи, не понял, что ему делать, и выкинул свой листок. Остальные 2999999 листков собрали в большую корзину и отправили в Национальный архив, в Вашингтон, округ Колумбия. Там содержимое корзины тщательно перетасовали и записали на магнитные ленты. Специалист по теории информации определил, что имеется достаточно информации для восстановления списка людей в исходном порядке. Специалист по программированию нашел способ сделать это менее чем за 1000 просмотров лент с данными, используя лишь последовательный доступ к файлам на лентах и небольшое количество памяти с произвольным доступом. Как ему это удалось?

[Другими словами, как, имея расположенные произвольным образом пары (x_i, x_{i+1}) , $1 \leq i < N$, где все x_i различны, получить последовательность x_1, x_2, \dots, x_N , применяя лишь методы последовательной обработки данных, пригодные для работы с магнитными лентами? Это задача сортировки в случае, когда трудно определить, какой из двух ключей предшествует другому; мы уже поднимали этот вопрос в упр. 2.2.3–25.]

5.1. * КОМБИНАТОРНЫЕ СВОЙСТВА ПЕРЕСТАНОВОК

Перестановкой конечного множества называется некоторое расположение его элементов в ряд. Перестановки особенно важны при изучении алгоритмов сортировки, так как они служат для представления неупорядоченных исходных данных. Чтобы исследовать эффективность различных методов сортировки, нужно уметь подсчитывать число перестановок, которые вынуждают повторять некоторый шаг алгоритма определенное число раз.

Мы, конечно, уже не раз встречались с перестановками в гл. 1, 2 и 3. Например, в п. 1.2.5 обсуждались два основных теоретических метода построения $n!$ перестановок из n объектов; в п. 1.3.3 проанализированы некоторые алгоритмы, связанные с циклической структурой и мультипликативными свойствами перестановок; в п. 3.3.2 изучены их отрезки монотонности. Цель настоящего параграфа — изучить некоторые другие свойства перестановок и рассмотреть общий случай, когда допускается наличие одинаковых элементов. Попутно мы узнаем многое о комбинаторной математике.

Свойства перестановок настолько красивы, что представляют и самостоятельный интерес. Удобно будет дать их систематическое изложение в одном месте, а не разбрасывать материал по всей главе. Читателям, не имеющим склонности к математике, а также тем, кто жаждет поскорее добраться до самих методов сортировки, рекомендуется перейти сразу к § 5.2, потому что настоящий параграф непосредственного отношения к сортировке почти не имеет.

5.1.1. *Инверсии

Пусть $a_1 a_2 \dots a_n$ — перестановка множества $\{1, 2, \dots, n\}$. Если $i < j$, а $a_i > a_j$, то пара (a_i, a_j) называется инверсией перестановки; например, перестановка 3 1 4 2 имеет три инверсии: (3, 1), (3, 2) и (4, 2). Каждая инверсия — это пара элементов, "нарушающих порядок"; следовательно, единственная перестановка, не содержащая инверсий, — это отсортированная перестановка 1 2 ... n . Такая связь с сортировкой и есть главная причина нашего интереса к инверсиям, хотя это понятие уже использовалось нами при анализе алгоритма динамического распределения памяти (см. упр. 2.2.2–9).

Понятие инверсии ввел Г. Крамер в 1750 г. [Intr. à l'Analyse des Lignes Courbes algébriques (Geneva, 1750), 657–659; ср. с Томас Мюир, Theory of Determinants, 1 (1906), 11–14] в связи со своим замечательным правилом решения линейных уравнений. В сущности, он определил детерминант $n \times n$ -матрицы следующим образом:

$$\det \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} = \sum (-1)^{I(a_1 a_2 \dots a_n)} x_{1a_1} x_{2a_2} \dots x_{na_n},$$

где сумма берется по всем перестановкам $a_1 a_2 \dots a_n$, а $I(a_1 a_2 \dots a_n)$ — число инверсий в перестановке.

Таблицей инверсий перестановки $a_1 a_2 \dots a_n$ называется последовательность чисел $b_1 b_2 \dots b_n$, где b_j — число элементов, больших j и расположенных левее j . (Другими словами, b_j — число инверсий, у которых второй элемент равен j .) Например, таблицей инверсий перестановки

$$5 \ 9 \ 1 \ 8 \ 2 \ 6 \ 4 \ 7 \ 3 \tag{1}$$

будет

$$2 \ 3 \ 6 \ 4 \ 0 \ 2 \ 2 \ 1 \ 0, \tag{2}$$

поскольку 5 и 9 расположены левее 1; 5, 9, 8 — левее 2 и т.д., всего 20 инверсий. По определению

$$0 \leq b_1 \leq n-1, 0 \leq b_2 \leq n-2, \dots, 0 \leq b_{n-1} \leq 1, b_n = 0. \tag{3}$$

Пожалуй, наиболее важный факт, касающийся перестановок, и установленный Маршаллом Холлом, это то, что таблица инверсий единственным образом определяет соответствующую перестановку. [См. Proc. Symp. Applied Math., 6 (American Math. Society, 1956), 203.] Из любой таблицы инверсий $b_1 b_2 \dots b_n$, удовлетворяющей условиям (3), можно однозначно восстановить перестановку, которая порождает данную таблицу, путем последовательного определения относительного расположения элементов $n, n-1, \dots, 1$ (в этом порядке). Например, перестановку, соответствующую (2), можно построить следующим образом: выпишем число 9; так как $b_8 = 1$, то 8 стоит правее 9. Поскольку $b_7 = 2$, то 7 стоит правее 8 и 9. Так как $b_6 = 2$, то 6 стоит правее двух уже выписанных нами чисел; таким образом, имеем

$$9 \ 8 \ 6 \ 7.$$

Припишем теперь 5 слева, потому что $b_5 = 0$; помещаем 4 вслед за четырьмя из уже записанных чисел, 3 — после шести выписанных чисел (т. е. в правый конец) и получаем

$$5 \ 9 \ 8 \ 6 \ 4 \ 7 \ 3.$$

Вставив аналогичным образом 2 и 1, придем к (1).

Такое соответствие важно, потому что часто можно заменить задачу, сформулированную в терминах перестановок, эквивалентной ей задачей, сформулированной в терминах таблиц инверсий, которая, возможно, решается проще. Рассмотрим, например, самый простой вопрос: сколько существует перестановок множества $\{1, 2, \dots, n\}$? Ответ должен быть равен числу всевозможных таблиц инверсий, а их легко пересчитать, так как b_1 можно выбрать n различными способами, b_2 можно независимо от b_1 выбрать $n-1$ способами, \dots , b_n —одним способом; итого $n(n-1) \dots 1 = n!$ различных таблиц инверсий. Таблицы инверсий пересчитать легче, потому что b независимы, в то время как a должны быть все различны.

В п. 1:2.10 мы исследовали задачу о числе локальных максимумов перестановки, если читать ее справа налево; иными словами, требовалось подсчитать количество элементов, каждый из которых больше любого из следующих после него. (Например, правосторонние максимумы в (1)—это 9, 8, 7 и 3.) Оно равно количеству индексов j , таких, что $b_j = n - j$. Так как b_1 принимает значение $n-1$ с вероятностью $1/n$, b_2 (независимо) принимает значение $n-2$ с вероятностью $1/(n-1)$ и т.д., то из рассмотрения инверсий ясно, что среднее число правосторонних максимумов равно

$$\frac{1}{n} + \frac{1}{n-1} + \dots + 1 = H_n.$$

Аналогичным способом легко получить и соответствующую производящую функцию.

Другие применения таблиц инверсий встретятся далее в этой главе в связи с конкретными алгоритмами сортировки.

Ясно, что если поменять местами *соседние* элементы перестановки, то общее число инверсий увеличится или уменьшится на единицу. На рис. 1 показаны 24 перестановки множества $\{1, 2, 3, 4\}$; линиями соединены перестановки, отличающиеся друг от друга положением двух соседних элементов; двигаясь вдоль линии вниз, мы увеличиваем число инверсий на единицу. Следовательно, число инверсий в перестановке n равно длине нисходящего пути из 1 2 3 4 в n на рис. 1; все такие пути должны иметь одинаковую длину.

Заметим, что эту диаграмму можно рассматривать как трехмерное твердое тело—"усеченный октаэдр", имеющий 8 шестиугольных и 6 квадратных граней. Это один из равномерных многогранников, которые обсуждал Архимед (см. упр., 10).

Picture: Рис. 1. Усеченный октаэдр, на котором показано изменение числа инверсий, когда меняются местами два соседних элемента перестановки;

Не следует путать "инверсии" перестановок с обратными перестановками. Вспомним, что перестановку можно записывать в виде двух строк

$$\begin{pmatrix} 1 & 2 & 3 & \dots & n \\ a_1 & a_2 & a_3 & \dots & a_n \end{pmatrix}; \quad (4)$$

обратной к этой перестановке называется перестановка a'_1, a'_2, \dots, a'_n , которая получается, если в (4) поменять местами строки, а затем упорядочить столбцы в возрастающем порядке по верхним элементам:

$$\begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_n \\ 1 & 2 & 3 & \dots & n \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & \dots & n \\ a'_1 & a'_2 & a'_3 & \dots & a'_n \end{pmatrix}; \quad (5)$$

Например, обратной к перестановке 5 9 1 8 2 6 4 7 3 будет перестановка 3 5 9 7 1 6 8 4 2, так как

$$\begin{pmatrix} 5 & 9 & 1 & 8 & 2 & 6 & 4 & 7 & 3 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 5 & 9 & 7 & 1 & 6 & 8 & 4 & 2 \end{pmatrix}.$$

Можно дать другое определение обратной перестановки: $a'_j = k$ тогда и только тогда, когда $a_k = j$.

Обратную перестановку впервые ввёл Х. А. Роте [в К.Ф. Hindenburg(ed.), *Sammlung combinatorisch-analytischer Abhandlungen*, 2 (Leipzig, 1800), 263–305]; он заметил интересную связь между обратными перестановками и инверсиями: *обратная перестановка содержит ровно столько же инверсий, сколько исходная*. Роте дал не самое простое доказательство этого факта, но оно поучительно и при этом довольно красиво. Построим таблицу размера $n \times n$ и поставим точки в j -й клетке i -й строки, если $a_i = j$. После этого расставим крестики во всех клетках, снизу от которых (в том же столбце) и справа (в той же строке) есть точки. Например, для 5 9 1 8 2 6 4 7 3 диаграмма будет такой:

Picture: Диаграмма

Количество крестиков равно числу инверсий, так как нетрудно видеть, что b_j равно числу крестиков в j -м столбце. Если мы теперь транспонируем диаграмму (поменяв ролями строки и столбцы), то получим диаграмму для обратной по отношению к исходной перестановки; значит, число крестиков (число инверсий) одинаково в обоих случаях. Роте использовал этот факт для доказательства того, что детерминант матрицы не меняется при транспонировании.

Для анализа некоторых алгоритмов сортировки необходимо знать число перестановок n элементов, содержащих ровно k инверсий. Обозначим это число через $I_n(k)$; в табл. 1 приведены первые несколько значений этой функции.

Таблица 1

Перестановки с k инверсиями												
n	$I_n(0)$	$I_n(1)$	$I_n(2)$	$I_n(3)$	$I_n(4)$	$I_n(5)$	$I_n(6)$	$I_n(7)$	$I_n(8)$	$I_n(9)$	$I_n(10)$	$I_n(11)$
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0	0
3	1	2	2	1	0	0	0	0	0	0	0	0
4	1	3	5	6	5	3	1	0	0	0	0	0
5	1	4	9	15	20	22	20	15	9	4	1	0
6	1	5	14	29	49	71	90	101	101	90	71	49

Из рассмотрения таблицы инверсий $b_1 b_2 \dots b_n$ ясно, что $I_k(0) = 1$, $I_n(1) = n - 1$ и что выполняется свойство симметрии:

$$I_n\left(\binom{n}{2} - k\right) = I_n(k) \quad (6)$$

Далее, так как значения b можно выбирать независимо друг от друга, то нетрудно видеть, что производящая функция

$$G_n(z) = I_n(0) + I_n(1)z + I_n(2)z^2 + \dots \quad (7)$$

удовлетворяет соотношению $G_n(z) = (1 + z + \dots + z^{n-1})G_{n-1}(z)$; следовательно, она имеет довольно, простой вид

$$(1 + z + \dots + z^{n-1}) \dots (1 + z)(1) = (1 - z^n) \dots (1 - z^2)(1 - z)/(1 - z)^n. \quad (8)$$

С помощью этой производящей функции можно легко продолжить табл. 1 и убедиться, что числа, расположенные под ступенчатой линией в таблице, удовлетворяют соотношению

$$I_n(k) = I_n(k-1) + I_{n-1}(k) \quad \text{при } k < n. \quad (9)$$

(Для чисел *над* ступенчатой линией это соотношение *не* выполняется.) Более сложные рассуждения (см. упр. 14) показывают, что на самом деле имеют место формулы

$$\begin{aligned} I_n(2) &= \binom{n}{2} - 1, n \geq 2; \\ I_n(3) &= \binom{n+1}{3} - \binom{n}{1}, n \geq 3; \\ I_n(4) &= \binom{n+2}{4} - \binom{n+1}{2}, n \geq 4; \\ I_n(5) &= \binom{n+3}{5} - \binom{n+2}{3} + 1, n \geq 5. \end{aligned}$$

Общая формула для $I_n(k)$ содержит около $1.6\sqrt{k}$ слагаемых:

$$\begin{aligned} I_n(k) &= \binom{n+k-2}{k} - \binom{n+k-3}{k-2} + \binom{n+k-6}{k-5} + \binom{n+k-8}{k-7} - \dots \\ &\quad + (-1)^j \left(\binom{n+k-u_j-1}{k-u_j} + \binom{n+k-u_j-j-1}{k-u_j-j} \right) + \dots, \quad n \geq k, \end{aligned} \quad (10)$$

где $u_j = (3j^2 - j)/2$ — так называемое "пятиугольное число".

Разделив $G_n(z)$ на $n!$, получим производящую функцию $g_n(z)$ распределения вероятностей числа инверсий в случайной перестановке n элементов. Она равна произведению

$$g_n(z) = h_1(z)h_2(z) \dots h_n(z), \quad (11)$$

где $h_k(z) = (1 + z + z^2 + \dots + z^{k-1})/k$ — производящая функция равномерного распределения случайной величины, принимающей целые неотрицательные значения, меньшие k . Отсюда

$$\begin{aligned} \text{mean}(g_n) &= \text{mean}(h_1) + \text{mean}(h_2) + \dots + \text{mean}(h_n) = \\ &= 0 + \frac{1}{2} + \dots + \frac{n-1}{2} = \frac{n(n-1)}{4}; \end{aligned} \quad (12)$$

$$\begin{aligned} \text{var}(g_n) &= \text{var}(h_1) + \text{var}(h_2) + \dots + \text{var}(h_n) = \\ &= 0 + \frac{1}{4} + \dots + \frac{n^2-1}{12} = \frac{n(2n+5)(n-1)}{72} \end{aligned} \quad (13)$$

Таким образом, среднее число инверсий довольно велико — около $\frac{1}{4}n^2$; стандартное отклонение также весьма велико — около $\frac{1}{6}n^{3/2}$.

В качестве интересного завершения изучения инверсий рассмотрим одно замечательное открытие, принадлежащее П. А. Мак-Магону [*Amer. J. Math.*, **35** (1913), 281–322]. Определим индекс перестановки $a_1 a_2 \dots a_n$ как сумму всех j , таких, что $a_j > a_{j+1}$, $1 \leq j < n$. Например, индекс перестановки 5 9 1 8 2 6 4 7 3 равен $2 + 4 + 6 + 8 = 20$. Индекс случайно совпал с числом инверсий. Если составить список всех 24 перестановок множества $\{1, 2, 3, 4\}$, а именно

Перестановка	Индекс	Инверсии	Перестановка	Индекс	Инверсии
1 2 3 4	0	0	3 1 2 4	1	2
1 2 4 3	3	1	3 1 4 2	4	3
1 3 2 4	2	1	3 2 1 4	3	3
1 3 4 2	3	2	3 2 4 1	4	4
1 4 2 3	2	2	3 4 1 2	2	4
1 4 3 2	5	3	3 4 2 1	5	5
2 1 3 4	1	1	4 1 2 3	1	3
2 1 4 3	4	2	4 1 3 2	4	4
2 3 1 4	2	2	4 2 1 3	3	4
2 3 4 1	3	3	4 2 3 1	4	5
2 4 1 3	2	8	4 3 1 2	3	5
2 4 3 1	5	4	4 3 2 1	6	6

то видно, что число перестановок, имеющих данный индекс k , равно числу перестановок, имеющих k инверсий.

На первый взгляд этот факт может показаться почти очевидным, однако после некоторых размышлений он начинает казаться чуть ли не мистическим, и не видно никакого простого прямого его доказательства. Мак-Магон нашел следующее остроумное косвенное доказательство: пусть $J(a_1 a_2 \dots a_n)$ — индекс перестановки $a_1 a_2 \dots a_n$, и соответствующая производящая функция есть

$$H_n(z) = \sum z^{J(a_1 a_2 \dots a_n)}, \quad (14)$$

где сумма берется по всем перестановкам множества $\{1, 2, \dots, n\}$. Мы хотели бы доказать, что $H_n(z) = G_n(z)$. Для этого определим взаимно однозначное соответствие между n -ками (q_1, q_2, \dots, q_n) неотрицательных целых чисел, с одной стороны, и упорядоченными парами n -ок

$$((a_1, a_2, \dots, a_n), (p_1, p_2, \dots, p_n)),$$

с другой стороны; здесь $a_1 a_2 \dots a_n$ — перестановка множества $\{1, 2, \dots, n\}$ и $p_1 \geq p_2 \geq \dots \geq p_n \geq 0$. Это соответствие будет удовлетворять условию

$$q_1 + q_2 + \dots + q_n = J(a_1 a_2 \dots a_n) + (p_1 + p_2 + \dots + p_n). \quad (15)$$

Производящая функция $\sum z^{q_1+q_2+\dots+q_n}$, где сумма берется по всем n -кам неотрицательных целых чисел (q_1, q_2, \dots, q_n) , равна $Q_n(z) = 1/(1-z)^z$; а производящая функция $\sum z^{p_1+p_2+\dots+p_n}$, где сумма берется по всем n -кам целых чисел (p_1, p_2, \dots, p_n) , таких, что $p_1 \geq p_2 \geq \dots \geq p_n \geq 0$, равна, как показано в упр. 15,

$$P_n(z) = 1/(1-z)(1-z^2)\dots(1-z^n). \quad (16)$$

Существование взаимно однозначного соответствия, которое удовлетворяет условию (15) и которое мы собираемся установить, доказывает равенство $Q_n(z) = H_n(z)P_n(z)$, т.е.

$$H_n(z) = Q_n(z)/P_n(z) = G_n(z).$$

Требуемое соответствие определяется с помощью алгоритма "сортировки". Начав с пустого списка, при $k = 1, 2, \dots, n$ (в таком порядке) вставляем в этот список числа q_k следующим образом: пусть после $k - 1$ шагов в списке содержатся элементы p_1, p_2, \dots, p_{k-1} , где $p_1 \geq p_2 \geq \dots \geq p_{k-1}$, и определена перестановка $a_1 a_2 \dots a_n$ множества $\{n, n-1, \dots, n-k+2\}$. Пусть j — единственное целое число, такое, что $p_j > q_k \geq p_{j+1}$; если $q_k \geq p_1$, то полагаем $j = 0$, а если $p_{k-1} > q_k$, то полагаем $j = k - 1$. Вставим теперь q_k в список между p_j и p_{j+1} , а целое число $(n-k+1)$ — в перестановку между a_j и a_{j+1} . Прделаав это для всех k , получим перестановку $a_1 a_2 \dots a_n$ множества $\{1, 2, \dots, n\}$ и n -ку чисел (p_1, p_2, \dots, p_n) , таких, что $p_1 \geq p_2 \geq \dots \geq p_n \geq 0$ и

$$p_j > p_{j+1}, \quad \text{если } a_j > a_{j+1}.$$

Наконец, для $1 \leq j < n$ вычтем единицу из всех чисел p_1, \dots, p_j при всех j , таких, что $a_j > a_{j+1}$. Полученная пара $((a_1, a_2, \dots, a_n), (p_1, p_2, \dots, p_n))$ удовлетворяет условию (15).

Пусть, например, $n = 6$ и $(q_1, \dots, q_6) = (3, 1, 4, 0, 0, 1)$. Построение происходит следующим образом:

k	$p_1 \dots p_k$	$a_1 \dots a_k$
1	3	6
2	3 1	6 5
3	4 3 1	4 6 5
4	4 3 1 0	4 6 5 3
5	4 3 1 0 0	4 6 5 2 3
6	4 3 1 1 0 0	4 6 1 5 2 3

После заключительной корректировки получаем $(p_1, \dots, p_6) = (2, 1, 0, 0, 0, 0)$.

Нетрудно проверить, что этот процесс обратим; таким образом, требуемое соответствие установлено и теорема Мак-Магона доказана. Аналогичное взаимно однозначное соответствие встретится нам в п. 5.1.4.

Упражнения

- [10] Какова таблица инверсий для перестановки 2 7 1 8 4 5 9 3 6? Какой перестановке соответствует таблица инверсий 5 0 1 2 1 2 0 0?
- [M15] Решением задачи Иосифа, сформулированной в упр. 1.3.2–22, является перестановка множества $\{1, 2, \dots, n\}$; решение для приведенного там примера ($n = 8, m = 4$) — перестановка 5 4 6 1 3 8 7 2. Соответствующая этой перестановке таблица инверсий — 3 6 3 1 0 0 1 0. Найдите простое рекуррентное соотношение для элементов $b_1 b_2 \dots b_n$ таблицы инверсий в общей задаче Иосифа для n человек, если казнят каждого m -го человека.
- [18] Пусть перестановке $a_1 a_2 \dots a_n$ соответствует таблица инверсий $b_1 b_2 \dots b_n$; какой перестановке $\bar{a}_1 \bar{a}_2 \dots \bar{a}_n$ соответствует таблица инверсий

$$(n-1-b_1)(n-2-b_2) \dots (0-b_n)?$$

- [20] Придумайте алгоритм, годный для реализации на ЭВМ, который по данной таблице инверсий $b_1 b_2 \dots b_n$, удовлетворяющей условиям (3), строил бы соответствующую перестановку $a_1 a_2 \dots a_n$. [Указание: вспомните методы работы со связанной памятью.]
- [35] Для выполнения на типичной ЭВМ алгоритм из упр. 4 требует времени, приблизительно пропорционального n^2 ; можно ли создать алгоритм, время работы которого было бы существенно меньше n^2 ?
- [26] Придумайте алгоритм вычисления таблицы инверсий $b_1 b_2 \dots b_n$, соответствующей данной перестановке $a_1 a_2 \dots a_n$ множества $\{1, 2, \dots, n\}$, время работы которого на типичной ЭВМ было бы порядка $n \log n$.
- [20] Помимо таблицы $b_1 b_2 \dots b_n$, определенной в этом пункте, можно определить некоторые другие типы таблиц инверсий, соответствующих данной перестановке $a_1 a_2 \dots a_n$ множества $\{1, 2, \dots, n\}$. В этом упражнении мы рассмотрим три других типа таблиц инверсий, которые возникают в приложениях. Пусть c_j — число инверсий, первая компонента которых равна j , т. е. число элементов, меньших j и расположенных правее j . [Перестановке (1) соответствует таблица 0 0 0 1 4 2 1 5 7; ясно, что $0 \leq c_j < j$.] Пусть $B_j = b_{a_j}$ и $C_j = c_{a_j}$.

Покажите, что при $1 \leq j \leq n$ справедливы неравенства $0 \leq B_j < j$ и $0 \leq C_j \leq n - j$; покажите также, что перестановку $a_1 a_2 \dots a_n$ можно однозначно определить, если задана или таблица $c_1 c_2 \dots c_n$, или $B_1 B_2 \dots B_n$, или $C_1 C_2 \dots C_n$.

8. [M24] Сохраним обозначения упр. 7; пусть $a'_1 a'_2 \dots a'_n$ — перестановка, обратная к $a_1 a_2 \dots a_n$ и пусть соответствующие ей таблицы инверсий — $b'_1 b'_2 \dots b'_n, c'_1 c'_2 \dots c'_n, B'_1 B'_2 \dots B'_n$ и $C'_1 C'_2 \dots C'_n$. Найдите как можно больше интересных соотношений между $a_j, b_j, c_j, B_j, C_j, a'_j, b'_j, c'_j, B'_j, C'_j$.
- >9. [M21] Докажите, что в обозначениях упр. 7 перестановка $a_1 a_2 \dots a_n$ обратна самой себе тогда и только тогда, когда $b_j = C_j$ при $1 \leq j \leq n$.
10. [BM20] Рассмотрите рис. 1 как многогранник в трехмерном пространстве. Чему равен диаметр усеченного октаэдра (расстояние между вершинами 1234 и 4321), если все ребра имеют единичную длину?
11. [M25] (а) Пусть $\pi = a_1 a_2 \dots a_n$ — перестановка множества $\{1, 2, \dots, n\}$, $E(x) = \{(a_i, a_j) | i < j, a_i > a_j\}$ — множество ее инверсий, а

$$\overline{E}(\pi) = \{(a_i, a_j) | i > j, a_i > a_j\}$$

— множество ее "неинверсий". Докажите, что $E(\pi)$ и $\overline{E}(\pi)$ транзитивны. [Множество S упорядоченных пар называется *транзитивным*, если для любых (a, b) и (b, c) , принадлежащих S , пара (a, c) также принадлежит S .] (б) Обратно, пусть E — любое транзитивное подмножество множества $T = \{(x, y) | 1 \leq y < x \leq n\}$, дополнение которого $T \setminus E$ транзитивно. Докажите, что существует перестановка π , такая, что $E(\pi) = E$.

12. [M28] Используя обозначения предыдущего упражнения, докажите, что если π_1 и π_2 — перестановки, а E — минимальное транзитивное множество, содержащее $E(\pi_1) \cup E(\pi_2)$, то \overline{E} — тоже транзитивное множество. [Следовательно, если мы будем говорить, что π_1 находится "над" π_2 , когда $E(\pi_1) \subseteq E(\pi_2)$, то определена решетка перестановок; существует единственная "самая низкая" перестановка, находящаяся "над" двумя данными перестановками. Диаграмма решетки при $n = 4$ представлена на рис. 1.]

Упражнения

Существует ли комбинаторное доказательство тождества Якоби, аналогичное доказательству Франклина для частного случая упр. 14? (Таким образом, нужно рассмотреть "комплексные разбиения")

$$m + ni = (p_1 + q_1i) + (p_2 + q_2i) + \dots + (p_k + q_ki),$$

где $p_j + q_ji$ — различные ненулевые комплексные числа; p_j, q_j — неотрицательные целые числа, причем $|p_j - q_j| \leq 1$. Согласно тождеству Якоби, число таких представлений с четными k равно числу представлений с нечетными k , если только m и n не являются соседними треугольными числами! Какими еще замечательными свойствами обладают комплексные разбиения?

- >1. [M25] (Г. Д. Кнотт.) Покажите, что перестановку $a_1 \dots a_n$ можно получить с помощью стека в смысле упр. 2.2.1–5 или 2.3.1–6 тогда и только тогда, когда $C_j \leq C_{j+1} + 1$ при $1 \leq j < n$ (см. обозначения в упр. 7).
2. [M28] (К. Мейер.) Мы знаем, что если m и n — взаимно простые числа, то последовательность $(m \bmod n) (2m \bmod n) \dots ((n-1)m \bmod n)$ представляет собой перестановку множества $\{1, 2, \dots, n-1\}$. Покажите, что число инверсий в этой перестановке можно выразить через суммы Дедекинда (ср. с п. 3.3.3).

5.1.2. *Перестановки мультимножества

До сих пор мы рассматривали перестановки *множества* элементов; это частный случай перестановок *мультимножества*. (Мультимножество — это то же самое, что и множество, но в нем могут содержаться одинаковые элементы. Некоторые основные свойства мультимножеств обсуждались в п. 4.6.3.)

Рассмотрим, например, мультимножество

$$M = \{a, a, a, b, b, c, d, d, d, d\}, \quad (1)$$

в котором содержится 3 элемента a , 2 элемента b , 1 элемент c и 4 элемента d . Повторения элементов можно указать и другим способом:

$$M = \{3 \cdot a, 2 \cdot b, c, 4 \cdot d\}. \quad (2)$$

Перестановка мультимножества — это некоторое расположение его элементов в ряд, например

$$c a b d d a b d a d.$$

С другой стороны, такую последовательность можно назвать цепочкой букв, содержащей 3 буквы a , 2 буквы b , 1 букву c и 4 буквы d .

Сколько существует перестановок мультимножества M ? Если бы мы рассматривали все элементы M как различные, обозначив их $a_1, a_2, a_3, b_1, b_2, c_1, d_1, d_2, d_3, d_4$, то получили бы $10! = 3\,628\,800$ перестановок, но после отбрасывания индексов многие из них оказались бы одинаковыми. Фактически каждая перестановка M встретилась бы ровно $3!2!1!4! = 288$ раз, поскольку в любой перестановке M индексы при буквах a можно расставить 3! способами, при b (независимо) — 2! способами, при c — одним способом, а при d — соответственно 4! способами. Поэтому число перестановок M равно

$$\frac{10!}{3!2!1!4!} = 12\,600.$$

В применении к общему случаю те же рассуждения доказывают, что число перестановок любого мультимножества равно мультиномиальному коэффициенту

$$\binom{n}{n_1, n_2, \dots} = \frac{n!}{n_1! n_2! \dots},$$

где n_1 — число элементов первого типа, n_2 — число элементов второго типа и т. д., а $n = n_1 + n_2 + \dots$ — общее число элементов.

Количество перестановок множества было известно еще в древние времена. В древнееврейской Книге Творения (около 100 г. н. э.)⁴, наиболее раннем литературном произведении иудейского философского мистицизма, даны верные значения первых семи факториалов, после чего говорится: "Продолжай и получишь числа, которые уста не могут произнести, а ухо не может воспринять." [Sefer Yezirah, ed. by R. Mordecai Atia (Jerusalem: Sh. Monson, 1962), стих 52 (стр. 107–108); ср.

⁴ Книга Творения (Йоцира) — одна из основополагающих книг каббалистики. — Прим. перев.

также с Solomon Gandz, *Studies in Hebrew Astronomy and Mathematics* (New York: Ktav, 1970), 494–496. Книга Творения была основана на считавшихся важными отношениях между семью планетами, семью согласными звуками с двойным произношением, семью отверстиями в голове человека и семью днями сотворения мира.] Это первый известный в истории подсчет числа перестановок. Второй встречается в индийском классическом произведении Ануйюгадвара-сутра (около 500 г. н. э.), правило 97, где приводится формула числа перестановок шести элементов, которые не расположены ни в возрастающем, ни в убывающем порядке:

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 - 2.$$

[См. G. Chakravarti, *Bull. Calcutta Math. Soc.*, 24 (1932), 79–88. Ануйюгадвара-сутра—одна из книг канонов джайнизма, религиозной секты, распространенной в Индии.]

Соответствующее правило для мультимножеств впервые, по-видимому, встречается в книге Лилавати, написанной Бхаскарой Ахарьей (ок. 1150 г.), разд. 270–271. У Бхаскары это правило сформулировано весьма сжато и проиллюстрировано лишь двумя простыми примерами $\{2, 2, 1, 1\}$ и $\{4, 8, 5, 5, 5\}$. В результате в английском переводе это правило не сформулировано корректно, впрочем, имеются некоторые сомнения относительно того, понимал ли сам Бхаскара, о чем он говорил. Вслед за этим правилом Бхаскара приводит интересную формулу

$$\frac{(4 + 8 + 5 + 5 + 5) \times 120 \times 11111}{5 \times 6}$$

для суммы 20 чисел $48\,555 + 45855 + \dots$.

Верное правило для нахождения числа перестановок в случае, когда только один элемент может повторяться, найдено независимо немецким ученым иезуитом Атанасиусом Кирхером в его много-томном труде о музыке *Musurgia Universalis* (Rome, 1650), том 2, стр. 5–7. Кирхера интересовал вопрос о количестве мелодий, которые можно создать из данного набора нот; для этого он придумал то, что называл "музарифметикой". На стр. 18–21 своего труда он дает верное значение числа перестановок мультимножества $\{m \cdot C, n \cdot D\}$ при нескольких значениях m и n , хотя описал он свой метод вычислений лишь для случая $n = 1$.

Общее правило (3) появилось позже в книге Жана Престэ *Eléments de Mathématiques* (Paris, 1675), стр. 351–352, которая содержит одно из первых изложений комбинаторной математики, написанных в западной Европе. Престэ верно сформулировал правило для случая произвольного мультимножества, но проиллюстрировал его лишь простым примером $\{a, a, b, b, c, c\}$. Он особо отметил, что деление на сумму факториалов, которое он считал естественным обобщением правила Кирхера, было бы ошибкой. Несколько лет спустя Джон Валлис в своей книге *Treatise of Algebra* (Oxford, 1685), том 2, стр. 117–118, обсудил это правило несколько более подробно.

В 1965 г. Доминик Фоата ввел одно интересное понятие, так называемое "соединительное произведение"⁵, которое позволило распространить многие известные результаты, касающиеся обычных перестановок, на общий случай перестановок мультимножества. [См. *Publ. Inst. Statistique, Univ. Paris*, 14 (1965), 81–241. а также *Lecture Notes in Math.*, 85 (Springer, 1969).] Предполагая, что элементы мультимножества каким-то способом линейно упорядочены, можно рассмотреть *двустрочное обозначение*, например

$$\begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix}.$$

Здесь верхняя строка содержит элементы M в неубывающем порядке, и нижняя—это сама перестановка. *Соединительное произведение* $\alpha \top \beta$ двух перестановок мультимножеств α и β —это перестановка, которая получается, если (а) взять двустрочные обозначения для α и β , (б) записать соответствующие строки в одну и (с) отсортировать столбцы так, чтобы элементы верхней строки расположились в неубывающем порядке. Сортировка должна быть устойчивой в том смысле, что взаимное расположение элементов нижней строки сохраняется, если соответствующие элементы верхней строки равны. Например, $c a d a b \top b d d a d = c a b d d a b d a d$, так как

$$\begin{pmatrix} a & a & b & c & d \\ c & a & d & a & b \end{pmatrix} \top \begin{pmatrix} a & b & d & d & d \\ b & d & d & a & d \end{pmatrix} = \begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix}. \quad (5)$$

Нетрудно видеть, что операция соединительного произведения ассоциативна, т. е.

$$\alpha \top \beta \top \gamma = \alpha \top (\beta \top \gamma), \quad (6)$$

⁵ В оригинале—"intercalation product".—Прим. перев.

и что она подчиняется законам сокращения

$$\begin{aligned} \text{если } \pi \top \alpha = \pi \top \beta, \text{ то } \alpha = \beta, \\ \text{если } \alpha \top \pi = \beta \top \pi, \text{ то } \alpha = \beta. \end{aligned}$$

Существует "единичный элемент"

$$\alpha \top \varepsilon = \varepsilon \top \alpha = \alpha, \quad (8)$$

где ε — пустая перестановка, "расположение в ряд" элементов пустого множества. Закон коммутативности, вообще говоря, не выполняется (см. упр. 2), тем не менее

$$\alpha \top \beta = \beta \top \alpha, \quad \text{если } \alpha \text{ и } \beta \text{ не содержат общих букв.} \quad (9)$$

Аналогичным способом и понятие *цикла* можно распространить на случай, когда элементы могут повторяться. Будем записывать в виде

$$(x_1 \ x_2 \ \dots \ x_n) \quad (10)$$

перестановку, двустрочное представление которой получается путем устойчивой сортировки столбцов

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \\ x_2 & x_3 & \dots & x_1 \end{pmatrix} \quad (11)$$

по верхним элементам. Например,

$$(d \ b \ d \ d \ a \ c \ a \ a \ b \ d) = \begin{pmatrix} d & b & d & d & a & c & a & a & b & d \\ b & d & d & a & c & a & a & b & d & d \end{pmatrix} = \begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix}$$

так что перестановка (4) фактически является циклом. Мы могли бы описать этот цикл словесно, сказав что-нибудь вроде " d переходит в b , переходит в d , переходит в d , переходит в \dots переходит в d и возвращается обратно". Заметим, что эти обобщенные циклы не обладают всеми свойствами обычных циклов; $(x_1 \ x_2 \ \dots \ x_n)$ не обязательно то же самое, что и $(x_2 \ \dots \ x_n \ x_1)$.

В п. 1.3.3 мы выяснили, что каждую перестановку множества можно единственным с точностью до порядка сомножителей образом представить в виде произведения непересекающихся циклов, где произведение перестановок определяется законом композиции. Легко видеть, что *произведение непересекающихся циклов — то же самое, что их соединительное произведение*; это наводит на мысль о том, что можно будет обобщить полученные ранее результаты, если найти единственное (в каком-то смысле) представление и для произвольной перестановки мультимножества в виде соединительного произведения циклов. В действительности существуют по крайней мере два естественных способа сделать это, и каждый из них имеет важные приложения.

Равенство (5) дает один способ представления $c \ a \ b \ d \ d \ a \ b \ d \ a \ d$ в виде соединительного произведения более коротких перестановок; рассмотрим общую задачу о нахождении всех разложений $\pi = \alpha \top \beta$ данной перестановки π . Для исследования этого вопроса полезно рассмотреть конкретную перестановку, скажем

$$\pi = \begin{pmatrix} a & a & b & b & b & b & b & c & c & c & d & d & d & d & d \\ d & b & c & b & c & a & c & d & a & d & d & b & b & b & d \end{pmatrix}, \quad (12)$$

Если можно записать π в виде $\alpha \top \beta$, где α содержит по крайней мере одну букву a , то самое левое a в верхней строке двустрочного представления α должно оказаться над d , значит, перестановка α должна содержать по крайней мере одну букву d . Если взглянуть теперь на самое левое d в верхней строке α , то увидим точно так же, что оно должно оказаться над d , значит, в α должны содержаться по меньшей мере *две* буквы d . Посмотрев на второе d , видим, что α содержит также b . Одно-единственное предположение о том, что α есть левый сомножитель π , содержащий букву a , приводит к такому промежуточному результату:

$$\begin{pmatrix} a & & b & & d & d & & & & & & & & & \\ & \dots & & \dots & & & \dots & & & & & & & & \\ d & & & & & & & d & b & & & & & & \end{pmatrix}. \quad (13)$$

Продолжая рассуждать точно так же и далее, обнаружим, что буква b в верхней строке (13) должна оказаться над c и т. д. В конце концов этот процесс вновь приведет нас к букве a , и мы сможем, если захотим, отождествить ее с первой буквой a . Только что проведенное рассуждение, по существу, доказывает,

что любой левый сомножитель в разложении перестановки (12), содержащий a , имеет вид $(ddbcdbbca) \top \alpha'$, где α' — некоторая перестановка. (Удобно записывать a не в начале, а в конце цикла; это допустимо, поскольку буква a только одна.) Аналогично, если бы мы предположили, что α содержит букву b , то вывели бы, что $\alpha = (cd db) \top \alpha''$, где α'' — некоторая перестановка.

В общем случае эти рассуждения показывают, что если есть какое-нибудь разложение $\alpha \top \beta = \pi$, где α содержит данную букву y , то существует единственный цикл вида

$$(x_1 \dots x_n y), \quad n \geq 0, x_1, \dots, x_n \neq y, \quad (14)$$

который является левым сомножителем в разложении перестановки α . Такой цикл легко отыскать, зная π и y ; это самый короткий левый сомножитель в разложении перестановки π , содержащий букву y . Одно из следствий этого наблюдения дает

Теорема А. Пусть элементы мультимножества M линейно упорядочены отношением " $<$ ". Каждая перестановка π мультимножества M имеет единственное представление в виде соединительного произведения

$$\pi = (x_{11} \dots x_{1n_1} y_1) \top (x_{21} \dots x_{2n_2} y_2) \top \dots (x_{t1} \dots x_{tn_t} y_t), \quad t \geq 0, \quad (15)$$

удовлетворяющее следующим двум условиям:

$$\begin{aligned} y_1 &\leq y_2 \leq \dots \leq y_t; \\ y_i &< x_{ij} \text{ при } 1 \leq j \leq n_i, 1 \leq i \leq t. \end{aligned} \quad (16)$$

(Иными словами, в каждом цикле последний элемент меньше любого другого, и последние элементы циклов образуют убывающую последовательность.)

Доказательство. При $\pi = \varepsilon$ получим требуемое разложение, положив $t = 0$. В противном случае пусть y_1 — минимальный элемент π ; определим $(x_{11} \dots x_{1n_1} y_1)$ — самый короткий левый сомножитель разложения π , содержащий y_1 , как в рассмотренном примере. Теперь $\pi = (x_{11} \dots x_{1n_1} y_1) \top \rho$, где ρ — некоторая перестановка; применив индукцию по длине перестановки, можем написать

$$\rho = (x_{21} \dots x_{2n_2} y_2) \top \dots \top (x_{t1} \dots x_{tn_t} y_t), \quad t \geq 1,$$

где условия (16) выполнены. Тем самым доказано существование такого разложения.

Докажем единственность разложения (15), удовлетворяющего условиям (16). Ясно, что $t = 0$ тогда и только тогда, когда π — пустая перестановка ε . При $t > 0$ из (16) следует, что y_1 — минимальный элемент перестановки π и что $(x_{11} \dots x_{1n_1} y_1)$ — самый короткий левый сомножитель, содержащий y_1 . Поэтому $(x_{11} \dots x_{1n_1} y_1)$ определяется однозначно; доказательство единственности такого представления завершается применением индукции и законов сокращения (7). ■

Например, "каноническое" разложение перестановки (12), удовлетворяющее данным условиям, таково:

$$(ddbcdbbca) \top (ba) \top (cdb) \top (d), \quad (17)$$

если $a < b < c < d$.

Важно отметить, что на самом деле в этом определении можно отбросить скобки и знаки операции \top , и это не приведет к неоднозначности! Каждый цикл заканчивается появлением наименьшего из оставшихся элементов. Таким образом, наше построение связывает с исходной перестановкой

$$\pi' = d d b c d b b c a b a c d b d$$

перестановку

$$\pi = d b c b c a c d a d d b b b d.$$

Если в двустрочном представлении π содержится столбец вида $\begin{smallmatrix} y \\ x \end{smallmatrix}$, где $x < y$, то в связанной с π перестановке присутствует соответствующая пара соседних элементов $\dots y x \dots$. Так, в нашем примере π содержит три столбца вида $\begin{smallmatrix} d \\ b \end{smallmatrix}$, а в π' трижды встречается пара $d b$. Вообще из этого построения вытекает замечательная

Теорема В. Пусть M — мультимножество. Существует взаимно однозначное соответствие между перестановками M , такое, что если π соответствует π' , то выполняются следующие условия:

- а) крайний левый элемент π' равен крайнему левому элементу π ;
- б) для всех пар участвующих в перестановке элементов (x, y) ,

таких, что $x < y$, число вхождений столбца $\frac{y}{x}$ в двустрочное представление перестановки π равно числу случаев, когда в перестановке π' элемент x следует непосредственно за y .

Если M — множество, то это, по существу, "нестандартное соответствие", обсуждавшееся в конце п. 1.3.3, с незначительными изменениями. Более общий результат теоремы В полезен при подсчете числа перестановок специальных типов, поскольку часто проще решить задачу с ограничениями, наложенными на двустрочное представление, чем эквивалентную задачу с ограничениями на пары соседних элементов.

П. А. Мак-Магон рассмотрел задачи этого типа в своей выдающейся книге *Combinatory Analysis* (Cambridge Univ. Press, 1915), том 1, стр. 168–186. Он дал конструктивное доказательство теоремы В в частном случае, когда M содержит элементы лишь двух различных типов, скажем a и b , его построение для этого случая, по существу, совпадает с приведенным здесь, но представлено в совершенно ином виде. Для случая трех различных элементов a, b, c Мак-Магон дал сложное неконструктивное доказательство теоремы В; общий случай впервые доказал Фоата в 1965 г.

В качестве нетривиального примера применения теоремы В найдем число цепочек букв a, b, c , содержащих ровно

$$\begin{aligned} &A \text{ вхождений буквы } a; \\ &B \text{ вхождений буквы } b; \\ &C \text{ вхождений буквы } c; \\ &k \text{ вхождений пары стоящих рядом букв } c a; \\ &l \text{ вхождений пары стоящих рядом букв } c b; \\ &m \text{ вхождений пары стоящих рядом букв } b a; \end{aligned} \quad (18)$$

Из теоремы следует, что это то же самое, что найти число двустрочных массивов вида

$$\begin{array}{ccc} \overbrace{\begin{pmatrix} a & \dots & a \\ \square & \dots & \square \end{pmatrix}}^A & \overbrace{\begin{pmatrix} b & \dots & b \\ \square & \dots & \square \end{pmatrix}}^B & \overbrace{\begin{pmatrix} c & \dots & c \\ \square & \dots & \square \end{pmatrix}}^C \\ \underbrace{A-k-m \text{ букв } a \quad m \text{ букв } a}_{B-l \text{ букв } b} & \underbrace{k \text{ букв } a}_{l \text{ букв } b} & \\ \underbrace{\hspace{10em}}_{C \text{ букв } c} & & \end{array} \quad (19)$$

Буквы a можно расположить во второй строке

$$\binom{A}{A-k-m} \binom{B}{m} \binom{C}{k} \quad \text{способами;}$$

после этого буквы b можно разместить в оставшихся позициях

$$\binom{B+k}{B-l} \binom{C-k}{l} \quad \text{способами.}$$

Остальные свободные места нужно заполнить буквами c ; следовательно, искомое число равно

$$\binom{A}{A-k-m} \binom{B}{m} \binom{C}{k} \binom{B+k}{B-l} \binom{C-k}{l}. \quad (20)$$

Вернемся к вопросу о нахождении всех разложений данной перестановки. Существует ли такой объект, как "простая" перестановка, которая не разлагается на множители, отличные от нее самой и ε ? Обсуждение, предшествующее теореме А, немедленно приводит к выводу о том, что перестановка будет простой тогда и только тогда, когда она есть цикл без повторяющихся элементов, так как, если перестановка является таким циклом, наше рассуждение доказывает, что не существует левых множителей, кроме ε и самого цикла. Если же перестановка содержит повторяющийся элемент y , то всегда можно выделить нетривиальный цикл в качестве левого сомножителя, в котором элемент y встречается всего однажды.

Если перестановка не простая, то ее можно разлагать на все меньшие и меньшие части, пока не будет получено произведение простых перестановок. Можно даже показать, что такое разложение единственно с точностью до порядка записи коммутирующих сомножителей.

Теорема С. Каждую перестановку мультимножества можно записать в виде произведения

$$\sigma_1 \top \sigma_2 \top \dots \top \sigma_t, \quad t \geq 0, \quad (21)$$

где σ_j — циклы, не содержащие повторяющихся элементов. Это представление единственно в том смысле, что любые два таких представления одной и той же перестановки можно преобразовать одно в другое, последовательно меняя местами соседние непересекающиеся циклы.

Термин "непересекающиеся циклы" относится к циклам, не имеющим общих элементов. В качестве примера можно проверить, что перестановка

$$\begin{pmatrix} a & a & b & b & c & c & d \\ b & a & a & c & d & b & c \end{pmatrix}$$

разлагается на множители ровно пятью способами:

$$\begin{aligned} (a b) \top (a) \top (c d) \top (b c) &= (a b) \top (c d) \top (a) \top (b c) = \\ &= (a b) \top (c d) \top (b c) \top (a) = \\ &= (c d) \top (a b) \top (a) \top (b c) = \\ &= (c d) \top (a b) \top (b c) \top (a). \end{aligned} \quad (22)$$

Доказательство. Нужно установить, что выполняется сформулированное в теореме свойство единственности. Применим индукцию по длине перестановки; тогда достаточно доказать, что если ρ и σ — два различных цикла, не содержащие повторяющихся элементов, и

$$\rho \top \alpha = \sigma \top \beta,$$

то ρ и σ — непересекающиеся циклы, и

$$\alpha = \sigma \top \theta, \quad \beta = \rho \top \theta,$$

где θ — некоторая перестановка.

Пусть y — произвольный элемент цикла ρ , тогда у любого левого сомножителя в разложении $\sigma \top \beta$, содержащего этот элемент y , будет левый сомножитель ρ . Значит, если ρ и σ имеют общий элемент, то цикл σ должен быть кратен ρ ; следовательно, $\sigma = \rho$ (так как они простые), что противоречит нашему предположению. Следовательно, цикл, содержащий y и не имеющий общих элементов с σ , должен быть левым сомножителем в разложении β . Применив законы сокращения (7), завершим доказательство. ■

В качестве иллюстрации теоремы С рассмотрим перестановки мультимножества $M = \{A \cdot a, B \cdot b, C \cdot c\}$, состоящего из A элементов a , B элементов b и C элементов c . Пусть $N(A, B, C, m)$ — число перестановок мультимножества M , двустрочное представление которых не содержит столбцов вида $\begin{smallmatrix} a & b \\ a & b \end{smallmatrix}$ и содержит ровно m столбцов вида $\begin{smallmatrix} a \\ a \end{smallmatrix}$. Отсюда следует, что имеется ровно $A - m$ столбцов вида $\begin{smallmatrix} a \\ a \end{smallmatrix}$, $B - m$ столбцов вида $\begin{smallmatrix} b \\ b \end{smallmatrix}$, $C - B + m$ столбцов вида $\begin{smallmatrix} c \\ a \end{smallmatrix}$, $C - A + m$ столбцов вида $\begin{smallmatrix} b \\ c \end{smallmatrix}$ и $A + B - C - m$ столбцов вида $\begin{smallmatrix} b \\ a \end{smallmatrix}$; следовательно,

$$N(A, B, C, m) = \binom{A}{m} \binom{B}{C - A + m} \binom{C}{B - m}. \quad (23)$$

Теорема С предлагает другой способ для подсчета этих перестановок: коль скоро столбцы $\begin{smallmatrix} a \\ a \end{smallmatrix}$, $\begin{smallmatrix} b \\ b \end{smallmatrix}$, $\begin{smallmatrix} c \\ c \end{smallmatrix}$ исключены, то в разложении перестановки единственно возможны такие простые множители:

$$(a b), (a c), (b c), (a b c), (a c b). \quad (24)$$

Каждая пара этих циклов имеет хотя бы одну общую букву, значит, разложение единственно. Если цикл $(a b c)$ встречается в разложении k раз, то из нашего предыдущего предположения следует, что $(a b)$ встречается $m - k$ раз, $(b c)$ встречается $C - A + m - k$ раз, $(a c)$ встречается $C - B + m - k$ раз и $(a c b)$ встречается $A + B - C - 2m + k$ раз. Следовательно, $N(A, B, C, m)$

В заключение этого пункта рассмотрим свойства отрезков в случае, когда в перестановке допускаются одинаковые элементы. Бесчисленные пасьянсы, которым посвящал свои досуги знаменитый американский астроном 19-го века Саймон Ньюкомб, имеют непосредственное отношение к интересующему нас вопросу. Он брал колоду карт и складывал их в одну стопку до тех пор, пока они шли в неубывающем порядке по старшинству; как только следующая карта оказывалась младше предыдущей, он начинал новую стопку. Он хотел найти вероятность того, что в результате вся колода окажется разложенной в заданное количество стопок.

Задача Саймона Ньюкомба состоит, следовательно, в нахождении распределения вероятностей для отрезков случайной перестановки мультимножества. В общем случае ответ довольно сложен (см. упр. 12), хотя мы уже видели, как решать задачу в частном случае, когда все карты различны по старшинству. Мы удовлетворимся здесь выводом формулы для *среднего* числа стопок в этом пасьянсе.

Пусть имеется m различных типов карт и каждая встречается ровно p раз. Например, в обычной колоде для бриджа $m = 13$, а $p = 4$, если пренебрегать различием масти. Замечательную симметрию обнаружил в этом случае П. А. Мак-Магон [Combinatory Analysis (Cambridge, 1915), том 1, стр. 212–213]: число перестановок с $k + 1$ отрезками равно числу перестановок с $mp - p - k + 1$ отрезками. Это соотношение легко проверить при $p = 1$ (формула (7)), однако при $p > 1$ оно кажется довольно неожиданным.

Можно доказать это свойство симметрии, установив взаимно однозначное соответствие между перестановками, такое, что каждой перестановке с $k + 1$ отрезками соответствует другая, с $mp - p - k + 1$ отрезками. Мы настойчиво рекомендуем читателю самому попробовать найти такое соответствие, прежде чем двигаться дальше.

Какого-нибудь очень простого соответствия на ум не приходит; доказательство Мак-Магона основано на производящих функциях, а не на комбинаторном построении. Однако установленное Фоатой соответствие (теорема 5.1.2В) позволяет упростить задачу, так как там утверждается существование взаимно однозначного соответствия между перестановками с $k + 1$ отрезками и перестановками, в двустрочном представлении которых содержится ровно k столбцов $\begin{smallmatrix} y \\ x \end{smallmatrix}$, таких, что $x < y$.

Пусть дано мультимножество $\{p \cdot 1, p \cdot 2, \dots, p \cdot m\}$; рассмотрим перестановку с двустрочным обозначением

$$\left(\begin{array}{cccccccccc} 1 & \dots & 1 & 2 & \dots & 2 & \dots & m & \dots & m \\ x_{11} & \dots & x_{1p} & x_{21} & \dots & x_{2p} & \dots & x_{m1} & \dots & x_{mp} \end{array} \right). \quad (32)$$

5.2. Внутренняя сортировка

Начнем обсуждение хорошего "сортирования" с маленького эксперимента: как бы вы решили следующую задачу программирования?

"Ячейки памяти $R + 1$, $R + 2$, $R + 3$, $R + 4$ и $R + 5$ содержат пять чисел. Напишите программу, которая переразмещает, если нужно, эти числа так, чтобы они расположились в возрастающем порядке."

(Если вы уже знакомы с какими-либо методами сортировки, постарайтесь, пожалуйста, на минуту забыть о них; вообразите, что вы решаете такую задачу впервые, не имея никаких предварительных знаний о том, как к ней подступиться.)

Прежде чем читать дальше, настоятельно просим вас найти хоть какое-нибудь решение этой задачи.

.....

Время, затраченное на решение приведенной выше задачи, окупится с лихвой, когда вы продолжите чтение этой главы. Возможно, ваше решение окажется одним из следующих:

А. Сортировка вставками. Элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов. (Именно таким способом игроки в бридж упорядочивают свои карты, беря по одной.)

В. Обменная сортировка. Если два элемента расположены не по порядку, то они меняются местами. Этот процесс повторяется до тех пор, пока элементы не будут упорядочены. **С. Сортировка посредством выбора.** Сначала выделяется наименьший (или, быть может, наибольший) элемент и каким-либо образом отделяется от остальных, затем выбирается наименьший (наибольший) из оставшихся и т. д.

Д. Сортировка подсчетом. Каждый элемент сравнивается со всеми остальными; окончательное положение элемента определяется подсчетом числа меньших ключей.

Е. Специальная сортировка, которая хороша для пяти элементов, указанных в задаче, но не поддается простому обобщению на случай большего числа элементов. **Ф. Ленивое решение.** Вы не откликнулись на наше предложение и отказались решать задачу. Жаль, но теперь, когда вы прочли так много, ваш шанс упущен.

Г. Новая суперсортировка, которая представляет собой определенное усовершенствование известных методов. (Пожалуйста, немедленно сообщите об этом автору.)

Если бы эта задача была сформулирована, скажем, для 1000 элементов, а не для 5, то вы могли бы открыть и некоторые более тонкие методы, о которых будет упомянуто ниже. В любом случае, приступая к новой задаче, разумно найти какую-нибудь очевидную процедуру, а затем попытаться улучшить ее. Случаи А, В и С приводят к важным классам методов сортировки, которые представляют собой усовершенствования сформулированных выше простых идей.

Было изобретено множество различных алгоритмов сортировки, и в этой книге рассматривается около 25 из них. Это пугающее количество методов на самом деле лишь малая толика всех алгоритмов, придуманных до сих пор; многие, уже устаревшие методы мы вовсе не будем рассматривать или упомянем лишь вскользь. Почему же так много методов сортировки? Для программирования это частный случай вопроса: "Почему существует так много методов x ", где x пробегает множество всех задач. Ответ состоит в том, что каждый метод имеет свои преимущества и недостатки, поэтому он оказывается эффективнее других при некоторых конфигурациях данных и аппаратуры. К сожалению, неизвестен "наилучший" способ сортировки; существует *много* наилучших методов в зависимости от того, что сортируется, на какой машине, для какой цели. Говоря словами Редьярда Киплинга, "существует 9 и еще 60 способов сложить песню племени, и каждый из них в отдельности хорош".

Полезно изучить характеристики каждого метода сортировки, чтобы можно было производить разумный выбор для конкретных приложений. К счастью, задача изучения этих алгоритмов не столь уж громоздка, поскольку между ними существуют интересные взаимосвязи.

В начале этой главы мы ввели основную терминологию и обозначения, которые и будем использовать при изучении сортировки. Записи

$$R_1, R_2, \dots, R_N \quad (1)$$

должны быть отсортированы в неубывающем порядке по своим ключам K_1, K_2, \dots, K_N , по существу, путем нахождения перестановки $p(1) p(2) \dots p(N)$, такой, что

$$K_{p(1)} \leq K_{p(2)} \leq \dots \leq K_{p(N)}. \quad (2)$$

В этом параграфе рассматривается *внутренняя сортировка*, когда число записей, подлежащих сортировке, достаточно мало, так что весь процесс можно провести в оперативной памяти ЭВМ.

В одних случаях может понадобиться физически переразместить записи в памяти так, чтобы их ключи были упорядочены; в других можно обойтись вспомогательной таблицей некоторого

Picture: Рис. 6. Сортировка таблицы адресов.

вида, которая определяет перестановку. Если записи и/или ключи занимают несколько слов памяти, то часто лучше составить новую таблицу адресов (ссылок), которые указывают на записи, и работать с этими адресами, не перемещая громоздкие записи. Такой метод называется *сортировкой таблицы адресов*

Picture: Рис. 7. Сортировка списка.

(рис. 6.). Если ключи короткие, а сопутствующая информация в записях велика, то для повышения скорости ключи можно вынести в таблицу адресов; это называется *сортировкой ключей*. Другие схемы сортировки используют вспомогательное поле связи, которое включается в каждую запись. Связи обрабатываются таким образом, что в результате все записи оказываются связанными в линейный список, в котором каждая связь указывает на следующую по порядку запись. Это называется *сортировкой списка* (рис. 7).

После сортировки таблицы адресов или сортировки списка можно по желанию расположить записи в неубывающем порядке. Для этого имеется несколько способов, требующих дополнительной памяти для хранения всего одной записи (см. упр. с 10 по 12); или же можно просто переместить записи в новую область памяти, если она может вместить все эти записи. Последний способ обычно вдвое быстрее первого, но требует почти в два раза больше памяти. Во многих приложениях вовсе не обязательно перемещать записи, так как поля связи, как правило, вполне приемлемы для операций с последовательной адресацией.

Все методы сортировки, которые мы исследуем "досконально", будут проиллюстрированы четырьмя способами: посредством

- a) словесного описания алгоритма,
- b) блок-схемы,
- c) программы для машины MIX,
- d) примера применения этого метода сортировки к заданному множеству чисел.

[В тех примерах, где это уместно, будет обрабатываться множество из 16 чисел, которые автор, пользуясь набором десятичных игральные костей, выбрал случайным образом 19 марта 1963 г.; ср. с упр. 3.1–1 (с).]

Из соображений удобства программы для машины MIX будут, как правило, написаны в предположении, что ключ числовой и что он помещается в одном слове; иногда мы даже будем ограничивать значения ключей так, чтобы они занимали лишь часть слова. Отношением порядка $<$ будет обычное арифметическое отношение порядка, а записи будут состоять из одного ключа, без сопутствующей информации. В этих предположениях программы получаются короче, проще для понимания, и не представляет труда распространить их на общий случай (например, применяя сортировку таблиц адресов). Вместе с MIX-программами приводится анализ времени выполнения соответствующего алгоритма сортировки.

Сортировка подсчетом. Чтобы проиллюстрировать способ, которым мы будем изучать методы внутренней сортировки, рассмотрим идею "подсчета", упомянутую в начале этого параграфа. Этот простой метод основан на том, что j -й ключ в окончательно упорядоченной последовательности превышает ровно $(j - 1)$ из остальных ключей. Иначе говоря, если известно, что некоторый ключ превышает ровно 27 других, то после сортировки соответствующая запись должна занять 28-е место. Таким образом, идея состоит в том, чтобы сравнить попарно все ключи и подсчитать, сколько из них меньше каждого отдельного ключа.

Очевидный способ выполнить сравнения —

$$((\text{сравнить } K_j \text{ с } K_i) \text{ при } 1 \leq j \leq N) \text{ при } 1 \leq i \leq N,$$

но легко видеть, что более половины этих действий излишни, поскольку не нужно сравнивать ключ сам с собой и после сравнения K_a с K_b уже не надо сравнивать K_b с K_a . Поэтому достаточно

$$((\text{сравнить } K_j \text{ с } K_i) \text{ при } 1 \leq j \leq i) \text{ при } 1 < i \leq N.$$

Итак, приходим к следующему алгоритму.

Алгоритм С. (Сравнение и подсчет.) Этот алгоритм сортирует записи R_1, \dots, R_N по ключам K_1, \dots, K_N , используя для подсчета числа ключей, меньших данного, вспомогательную таблицу $\text{COUNT}[1], \dots, \text{COUNT}[N]$. После завершения алгоритма величина $\text{COUNT}[j] + 1$ определяет окончательное положение записи R_j .

C1 [Сбросить счетчики.] Установить $\text{COUNT}[1], \dots, \text{COUNT}[N]$ равными нулю.

C2 [Цикл по i .] Выполнить шаг **C3** при $i = N, N - 1, \dots, 2$; затем завершить работу алгоритма.

C3 [Цикл по j .] Выполнить шаг **C4** при $j = i - 1, i - 2, \dots, 1$.

C4 [Сравнить K_i, K_j .] Если $K_i < K_j$, то увеличить $\text{COUNT}[j]$ на 1; в противном случае увеличить $\text{COUNT}[i]$ на 1. ■

Заметим, что в этом алгоритме записи не перемещаются. Он аналогичен сортировке таблицы адресов, поскольку таблица COUNT определяет конечное расположение записей; но эти методы несколько различаются, потому что $\text{COUNT}[j]$ указывает то место, куда нужно переслать запись R_j , а не ту запись, которую надо переслать на место R_j . (Таким образом, таблица COUNT определяет перестановку, обратную $p(1) \dots p(n)$; см. п. 5.1.1.)

В рассуждении, предшествующем этому алгоритму, мы не учитывали, что ключи могут быть равными. Это, вообще говоря, серьезное упущение, потому что если бы равным ключам соответствовали равные счетчики, то заключительное перемещение записей было бы довольно сложным. К счастью, как показано в упр. 2, алгоритм С дает верный результат независимо от числа равных ключей.

Picture: Рис. 8. Алгоритм С: сравнение и подсчет.

Таблица 1

Сортировка подсчетом (алгоритм С)																
ключи:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
COUNT(нач.):	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COUNT($i = N$):	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	12
COUNT($i = N - 1$):	0	0	0	0	2	0	2	0	0	0	0	0	0	0	13	12
COUNT($i = N - 2$):	0	0	0	0	3	0	3	0	0	0	0	0	0	11	13	12
COUNT($i = N - 3$):	0	0	0	0	4	0	4	0	1	0	0	0	9	11	13	12
COUNT($i = N - 4$):	0	0	1	0	5	0	5	0	2	0	0	7	9	11	13	12
COUNT($i = N - 5$):	1	0	2	0	6	1	6	1	3	1	2	7	9	11	13	12
...																
COUNT($i = 2$):	6	1	8	0	15	3	14	4	10	5	2	7	9	11	13	12

Программа С. (Сравнение и подсчет.) В следующей реализации алгоритма С для машины MIX предполагается, что запись R_j находится в ячейке $\text{INPUT} + j$, а $\text{COUNT}[j]$ — в ячейке $\text{COUNT} + j$, где $1 \leq j \leq N$; $\text{rI1} \equiv i$; $\text{rI2} \equiv j$; $\text{rA} \equiv K_i \equiv R_i$; $\text{rX} \equiv \text{COUNT}[i]$.

```

START ENT1 N          1    C1 . Сбросить счетчики.
      STZ  COUNT,1      N    COUNT[i] ← 0.
      DEC1 1            N
      J1P  *-2          N    N ≥ i > 0.
      ENT1 N            1    C2. Цикл по i.
      JMP  1F           1
2H    LDA  INPUT,1      N-1
      LDX  COUNT,1      N-1
3H    CMPA INPUT,2      A    C4. Сравнить Ki, Kj.
      JGE  4F           A    Переход, если Ki ≥ Kj.
      LD3  COUNT,2      B    COUNT[j]
      INC3 1            B    +1

```

	ST3	COUNT, 2	$B \rightarrow \text{COUNT}[j]$
	JMP	5F	B
4H	INCX	1	$A - B \text{ COUNT}[i] + 1 \rightarrow \text{COUNT}[i].$
5H	DEC2	1	A СЗ.Цикл по j .
	J2P	3B	A
	STX	COUNT, 1	$N - 1$
	DEC1	1	$N - 1$
1H	ENT2	-1, 1	$N \quad N \geq i > j > 0.$
	J2P	2B	N

■

Время работы этой программы равно $13N + 6A + 5B - 4$ единиц, где N —число записей, A —число способов выбрать 2 предмета из N , т. е. $\binom{N}{2} = (N^2 - N)/2$, а B —число пар индексов, таких, что $j < i$, и $K_j > K_i$. Таким образом, B —число инверсий перестановки K_1, \dots, K_N ; эта величина подробно анализировалась в п. 5.1.1, и было найдено [формулы (5.1.1–12,13)], что для неравных ключей, расположенных в случайном порядке,

$$B = \left(\min 0, \text{ave } \frac{(N^2 - N)}{4}, \max \frac{(N^2 - N)}{2}, \text{dev } \frac{\sqrt{N(N-1)(N+2.5)}}{6} \right).$$

Следовательно, выполнение программы С занимает от $3N^2 + 10N - 4$ до $5.5N^2 + 7.5N - 4$ единиц времени, а среднее время работы находится посередине между этими двумя границами. Например, для данных табл. 1 имеем $N = 16$, $A = 120$, $B = 41$, значит, программа С отсортирует их за время 1129*u*. Модификацию программы С, обладающую несколько иными временными характеристиками, см. в упр. 5.

Множитель N^2 , которым определяется время работы, свидетельствует о том, что алгоритм С не дает эффективного способа сортировки, когда N велико; при удвоении числа записей время увеличивается в четыре раза. Поскольку этот метод требует сравнения всех пар ключей (K_i, K_j) , то нет очевидного способа исключить зависимость от N^2 , тем не менее мы увидим дальше в этой главе, что, пользуясь "разделением и обменом", можно снизить порядок среднего времени работы до $N \log N$. Алгоритм С интересен для нас не эффективностью, а главным образом своей простотой; его описание служит примером того стиля, в котором будут описаны более сложные (и более эффективные) методы.

Существует другая разновидность сортировки посредством подсчета, которая действительно весьма важна с точки зрения эффективности; она применима в основном в том случае, когда имеется много равных ключей и все они попадают в диапазон $u \leq K_j \leq v$, где значение $(v - u)$ невелико. Эти предположения представляются весьма ограничительными, но на самом деле мы увидим немало приложений этой идеи; например, если применить этот метод лишь к старшим цифрам ключей, а не ко всем ключам целиком, то файл окажется частично отсортированным, и будет уже сравнительно просто довести дело до конца.

Чтобы понять принцип, предположим, что все ключи лежат между 1 и 100. При первом просмотре файла можно подсчитать, сколько имеется ключей, равных 1, 2, ..., 100, а при втором просмотре можно уже располагать записи в соответствующих местах области вывода. В следующем алгоритме все это описано более подробно.

Picture: Рис. 9. Алгоритм D: распределяющий подсчет.

Алгоритм D. (Распределяющий подсчет.) Этот алгоритм в предположении, что все ключи—целые числа в диапазоне $u \leq K_j \leq v$ при $1 \leq j \leq N$, сортирует записи R_1, \dots, R_N , используя вспомогательную таблицу $\text{COUNT}[u], \dots, \text{COUNT}[v]$. В конце работы алгоритма все записи в требуемом порядке переносятся в область вывода S_1, \dots, S_N .

- D1 [Сбросить счетчики.] Установить $\text{COUNT}[u], \dots, \text{COUNT}[v]$ равными нулю.
D2 [Цикл по j .] Выполнить шаг D3 при $1 \leq j \leq N$, затем перейти к шагу D4.
D3 [Увеличить $\text{COUNT}[K_j]$.] Увеличить значение $\text{COUNT}[K_j]$ на 1.
D4 [Суммирование.] (К этому моменту значение $\text{COUNT}[i]$ есть число ключей, равных i .) Установить $\text{COUNT}[i] \leftarrow \text{COUNT}[i] + \text{COUNT}[i - l]$ при $i = u + l, u + 2, \dots, v$.
D5 [Цикл по j .] (К этому моменту значение $\text{COUNT}[i]$ есть число ключей, меньших или равных i , в частности $\text{COUNT}[v] = N$.) Выполнить шаг D6 при $j = N, N - 1, \dots, 1$ и завершить работу алгоритма.

D[Вывод R_j .] Установить $i \leftarrow \text{COUNT}[K_j]$, $S_i \leftarrow R_j$, и $\text{COUNT}[K_j] \leftarrow i - 1$. ■

Пример применения этого алгоритма приведен в упр. 6; программу для машины MIX можно найти в упр. 9. При сформулированных выше условиях это очень быстрая процедура сортировки.

Сортировка посредством сравнения и подсчета, как, в алгоритме C, впервые упоминается в работе Э. Х. Фрэнда [JACM, 3 (1965), 152], хотя он и не заявил о ней как о своем собственном изобретении. Распределяющий подсчет, как в алгоритме D, впервые разработан Х. Сьювордом в 1954 г. и использовался при поразрядной сортировке, которую мы обсудим позже (см. п. 5.2.5); этот метод также был опубликован под названием "Mathsort" в работе W. Feurzig, CACM, 3 (1960), 601.

Упражнения

1. [15] Будет ли работать алгоритм C, если в шаге C2 значение C будет изменяться от 2 до N, а не от N до 2? Что произойдет, если в шаге C3 значение j будет изменяться от 1 до i - 1?
2. [21] Покажите, что алгоритм C работает правильно и при наличии одинаковых ключей. Если $K_j = K_i$ и $j < i$, то где окажется в конце концов R_j — до или после R_i ?
- >3. [21] Будет ли алгоритм C работать правильно, если в шаге C4 заменить проверку " $K_i < K_j$ " на " $K_i \leq K_j$ "?
4. [16] Напишите MIX-программу, которая завершит сортировку, начатую программой C; ваша программа должна поместить ключи в ячейки OUTPUT + 1, ..., OUTPUT + N в возрастающем порядке. Сколько времени затратит на это ваша программа?
5. [22] Улучшится ли программа C в результате следующих изменений?

Ввести новую строку 8a: INCX 0, 2
 Изменить строку 10: JGE 5F
 Изменить строку 14: DECX 1
 Исключить строку 15.

6. [18] Промоделируйте ручную работу алгоритма D, показывая промежуточные результаты, получающиеся при сортировке 16 записей

5T, 0C, 5U, 00, 9, 1N, 8S, 2R, 6A, 4A, 1G, 5L, 6T, 61, 70, 7N.

Здесь цифры — это ключи, а буквы — сопутствующая информация в записях.

7. [13] Является ли алгоритм D алгоритмом "устойчивой" сортировки?
7. [15] Будет ли алгоритм D работать правильно, если в шаге D5 значение j будет изменяться от 1 до N, а не от N до 1?
8. [23] Напишите MIX-программу для алгоритма D, аналогичную программе C и упр. 4. Выразите время работы вашей программы в виде функции от N и $(v - u)$.
9. [25] Предложите эффективный алгоритм, который бы заменял N величин (R_1, \dots, R_N) соответственно на $(R_{p(1)}, \dots, R_{p(N)})$, если даны значения R_1, \dots, R_N и перестановка $p(1) \dots p(N)$ множества $\{1, \dots, N\}$. Постарайтесь не использовать излишнего пространства памяти. (Эта задача возникает, когда требуется переразместить в памяти записи после сортировки таблицы адресов, не расходуя память на $2N$ записей.)
10. [M27] Напишите MIX-программу для алгоритма из упр. 10 и проанализируйте ее эффективность.
- >11. [25] Предложите эффективный алгоритм переразмещения в памяти записей R_1, \dots, R_N в отсортированном порядке после завершения сортировки списка (рис. 7). Постарайтесь не использовать излишнего пространства памяти.
- >12. [27] Алгоритму D требуется пространство для $2N$ записей R_1, \dots, R_N и S_1, \dots, S_N . Покажите, что можно обойтись пространством для N записей R_1, \dots, R_N , если вместо шагов D5 и D6 подставить новую упорядочивающую процедуру. (Таким образом, задача состоит в том, чтобы разработать алгоритм, который бы переразмещал записи R_1, \dots, R_N основываясь на значениях

COUNT[u], ..., COUNT[v]

после выполнения шага D4, не используя дополнительной памяти; это, по существу, обобщение задачи, рассмотренной в упр. 10.)

5.2.1. Сортировка вставками

Одно из важных семейств-методов сортировки основано на упомянутом в начале § 5.2 способе, которым пользуются игроки в бридж; предполагается, что перед рассмотрением записи R_j предыдущие записи R_1, \dots, R_{j-1} уже упорядочены, и R_j вставляется в соответствующее место. На основе этой схемы возможны несколько любопытных вариаций.

Простые вставки. Простейшая сортировка вставками относится к наиболее очевидным. Пусть $1 < j \leq N$ и записи R_1, \dots, R_{j-1} уже размещены так, что

$$K_1 \leq K_2 \leq \dots \leq K_{j-1}.$$

(Напомним, что в этой главе через K_j обозначается ключ записи R_j .) Будем сравнивать по очереди K_j с K_{j-1}, K_{j-2}, \dots до тех пор, пока не обнаружим, что запись R_j следует вставить между R_i и R_{i+1} ; тогда подвинем записи R_{i+1}, \dots, R_{j-1} на одно место вверх и поместим новую запись в позицию $i+1$. Удобно совмещать операции сравнения и перемещения, перемежая их друг с другом, как показано в следующем алгоритме; поскольку запись R_j как бы "проникает на положенный ей уровень", этот способ часто называют *просеиванием*, или *погружением*.

Алгоритм S. (*Сортировка простыми вставками.*) Записи R_1, \dots, R_N переразмещаются на том же месте; после завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$.

S1 [Цикл. по j .] Выполнить шаги от S2 до S5 при $j = 2, 3, \dots, N$ и после этого завершить алгоритм.

Picture: Рис. 10. Алгоритм S: простые вставки.

S2 [Установить i, K, R .] Установить $i \leftarrow j - 1, K \leftarrow K_j, R \leftarrow R_j$. (В последующих шагах мы попытаемся вставить запись R в нужное место, сравнивая K с K_i при убывающих значениях i .)

S3 [Сравнить K с K_i .] Если $K \geq K_i$, то перейти к шагу S5. (Мы нашли искомое место для записи R .)

S4 [Переместить R_i , уменьшить i .] Установить $R_{i+1} \leftarrow R_i, i \leftarrow i - 1$. Если $i > 0$, то вернуться к шагу S3. (Если $i = 0$, то K — наименьший из рассмотренных до сих пор ключей, а, значит, запись R должна занять первую позицию.)

S5 [R на место R_{i+1} .] Установить $R_{i+1} \leftarrow R$. ■

В табл. 1 показано применение алгоритма S к шестнадцати числам, взятым нами для примеров. Этот метод чрезвычайно просто реализуется на вычислительной машине; фактически следующая MIX-программа — самая короткая из всех приемлемых программ сортировки в этой книге.

Программа S. (*Сортировка простыми вставками*). Записи, которые надо отсортировать, находятся в ячейках $\text{INPUT} + 1, \dots, \text{INPUT} + N$; они сортируются в той же области по ключу, занимающему одно слово целиком. Здесь $rI1 \equiv j - N$; $rI2 \equiv i$, $rA \equiv R \equiv K$; предполагается, что $N \geq 2$.

START	ENT1	2-N	1	S1. Цикл по j . $j \leftarrow 2$.
2H	LDA	INPUT+N, 1	$N - 1$	S2. Установить i, K, R
	ENT2	N-1, 1	$N - 1$	$i \leftarrow j - 1$.
3H	CMPA	INPUT, 2	$B + N - 1 - A$	S3. Сравнить K, K_i
	JGE	5F	$B + N - 1 - A$	K S5, если $K \geq K_i$.
4H	LDX	INPUT, 2	B	S4. Переместить R_i , уменьшить i
	STX	INPUT+1, 2	B	$R_{i+1} \leftarrow R_i$.
	DEC2	1	B	$i \leftarrow i - 1$.
	J2P	3B	B	K S3, если $i > 0$.
5H	STA	INPUT+1, 2	$N - 1$	S5. R на место $R_i + 1$.
	INC1	1	$N - 1$	
	J1NP	2B	$N - 1$	$2 \leq j \leq N$.

■

Время работы этой программы равно $9B + 10N - 3A - 9$ единиц, где N — число сортируемых записей, A — число случаев, когда в шаге S4 значение i убывает до 0, а B — число перемещений. Ясно, что A равно числу случаев, когда $K_j < (K_1, \dots, K_{j-1})$ при $1 < j \leq N$, т. е. это число левосторонних минимумов — величина, которая была тщательно исследована в п. 1.2.10. Немного подумав, нетрудно понять, что B — тоже известная величина: число перемещений при фиксированном j равно числу инверсий ключа K_j , так что B равно числу инверсий перестановки K_1, K_2, \dots, K_N . Следовательно, согласно формулам (1.2.10–16) и (5.1.1–12, 13),

$$A = (\min 0, \text{ave } H_N - 1, \max N - 1, \text{dev } \sqrt{H_n - H_n^{(2)}});$$

$$B = (\min 0, \text{ave}(N^2 - N)/4, \max(N^2 - N)/2, \text{dev } \sqrt{N(N-1)(N+2.5)/6}),$$

а среднее время работы программы S в предположении, что ключи различны и расположены в случайном порядке, равно $(2.25N^2 + 7.75N - 3H_N - 6)u$. В упр. 33 показано, как можно чуть-чуть уменьшить это время.

Приведенные в качестве примера в табл. 1 данные содержат 16 элементов; имеются два левосторонних минимума, 087 и 061, и, как мы видели в предыдущем пункте, 41 инверсия. Следовательно, $N = 16$, $A = 2$, $B = 41$, а общее время сортировки равно $514u$.

Бинарные вставки и двухпутевые вставки. Когда при сортировке простыми вставками обрабатывается j -я запись, ее ключ сравнивается в среднем примерно с $j/2$ ранее отсортированными ключами;

Таблица 1

Пример применения простых вставок

```

~503 :087
087 503~:512
~087 503 512 :061
061 087 503 512~:908
061 087~ 503 512 908 :170
061 087 170 503 512~ 908 :897
. . . . .
061 087 154 170 275 426 503 509 512 612 653 677~ 765 897 908 :703
061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

```

поэтому общее число сравнений равно приблизительно $(1 + 2 + \dots + N)/2 \approx N^2/4$, а это очень много, даже если N умеренно велико. В п. 6.2.1 мы изучим методы "бинарного поиска", которые указывают, куда вставлять j -й элемент после приблизительно $\log_2 j$ соответствующим образом выбранных сравнений. Например, если вставляется 64-я запись, можно сначала сравнить ключ K_{64} с K_{32} ; затем, если он меньше, сравниваем его с K_{16} , если больше — с K_{48} и т. д., так что место для R_{64} будет найдено после всего лишь шести сравнений. Общее число сравнений для N вставляемых элементов равно приблизительно $N \log_2 N$, что существенно лучше, чем $N^2/4$; в п. 6.2.1 показано, что соответствующая программа не обязательно намного сложнее, чем программа для простых вставок. Этот метод называется *бинарными вставками*. Он упоминался Джоном Мочли еще в 1946 г., в первой публикации по машинной сортировке.

Неприятность состоит в том, что бинарные вставки решают задачу только наполовину: после того, как мы нашли, куда вставлять запись R_j , все равно нужно подвинуть примерно $j/2$ ранее отсортированных записей, чтобы освободить место для R_j так что общее время работы остается, по существу, пропорциональным N_2 . Некоторые вычислительные машины (например, IBM 705) имеют встроенные инструкции "переброски", выполняющие операции перемещения с большой скоростью, но с ростом N зависимость от N^2 в конце концов начинает преобладать. Например, анализ, проведенный Х. Нэгдером [CASM, 3 (1960), 618–620], показывает, что не следует рекомендовать бинарные вставки при сортировке более $N = 128$ записей на машине IBM 705, если каждая запись состоит из 80 литер; аналогичный анализ применим и к другим машинам.

Таблица 2

Двухпутевые вставки

				~503			
			087	503~			
		~087	503	512			
	061	087	503	512~			
	061	087~	503	512	908		
	061	087	170	503	512	908	
	061	087	170~	503	512	897	908
061	087	170	276	503	512	897	908

Разумеется, изобретательный программист может придумать какие-нибудь способы, позволяющие сократить число необходимых перемещений; первый такой прием, предложенный в начале 50-х годов, проиллюстрирован в табл. 2. Здесь первый - элемент помещается в середину области вывода, и место для последующих элементов освобождается при помощи сдвигов влево или вправо, туда, куда удобнее. Таким образом удастся сэкономить примерно половину времени работы по сравнению с простыми вставками за счет некоторого усложнения программы. Можно применять этот метод, используя не больше памяти, чем требуется для N записей (см. упр. 6), но мы не станем дольше задерживаться на таких "двухпутевых" вставках, так как были разработаны гораздо более интересные методы.

Метод Шелла. Для алгоритма сортировки, который каждый раз перемещает запись только на одну позицию, среднее время работы будет в лучшем случае пропорционально N^2 , потому что в процессе сортировки каждая запись должна пройти в среднем через $N/3$ позиций (см. упр. 7). Поэтому, если мы хотим получить метод, существенно превосходящий по скорости простые вставки, то необходим некоторый механизм, с помощью которого записи могли бы перемещаться большими скачками, а не короткими шажками.

Такой метод предложен в 1959 г. Дональдом Л. Шеллом [CASM, 2 (July, 1959), 30–32]; мы будем называть его *сортировкой с убывающим шагом*. В табл. 3 проиллюстрирована общая идея, лежащая в основе этого метода. Сначала делим 16 записей на 8 групп по две записи в каждой группе: (R_1, R_9) , (R_2, R_{10}) , \dots , (R_8, R_{16}) . В результате сортировки каждой группы записей по отдельности приходим ко второй строке табл. 3,

Picture: Таблица 3

это называется "первым просмотром". Заметим, что элементы 154 и 512 поменялись местами, а 908 и 897 переместились вправо. Разделим теперь записи на четыре группы по четыре в каждой: (R_1, R_5, R_9, R_{13}) , \dots , $(R_4, R_8, R_{12}, R_{16})$ —и опять отсортируем каждую группу в отдельности; этот второй просмотр приводит к третьей строке таблицы. При третьем просмотре сортируются две группы по восемь записей; процесс завершается четвертым просмотром, во время которого сортируются все 16 записей. В каждом из этих промежуточных процессов сортировки участвуют либо сравнительно короткие файлы, либо уже сравнительно хорошо упорядоченные файлы, поэтому на каждом этапе можно пользоваться простыми вставками; записи довольно быстро достигают своего конечного положения.

Последовательность шагов 8, 4, 2, 1 не следует считать незыблемой, можно пользоваться *любой* последовательностью h_t, h_{t-1}, \dots, h_1 , в которой последний шаг h_1 равен 1. Например, в табл. 4 будет показана сортировка тех же данных с шагами 7, 5, 3, 1. Одни последовательности оказываются гораздо лучше других; мы обсудим выбор последовательностей шагов позже.

Алгоритм D. (Сортировка с убывающим шагом.) Записи R_1, \dots, R_N переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$. Для управления процессом сортировки используется вспомогательная последовательность шагов h_t, h_{t-1}, \dots, h_1 , где $h_1 = 1$; правильно выбрав эти приращения, можно значительно сократить время сортировки. При $t = 1$ этот алгоритм сводится к алгоритму S.

- D1 [Цикл по s .] Выполнить шаг D2 при $s = t, t-1, \dots, 1$, после чего завершить работу алгоритма.
D2 [Цикл по j .] Установить $h \leftarrow h_s$ и выполнить шаги D3, ..., D6 при $h < j \leq N$. (Для сортировки элементов, отстоящих друг от друга на h позиций, воспользуемся простыми вставками и в результате получим $K_i \leq K_{i+h}$ при $1 \leq i \leq N-h$. Шаги D3, ..., D6, по существу, такие же, как соответственно S2, ..., S5 в алгоритме S.)
D3 [Установить i, K, R .] Установить $i \leftarrow j-h$, $K \leftarrow K_j$, $R \leftarrow R_j$.
D4 [Сравнить K, K_i .] Если $K \geq K_i$, то перейти к шагу D6.
D5 [Переместить R_i , уменьшить i .] Установить $R_{i+h} \leftarrow R_i$, затем $i \leftarrow i-h$. Если $i > 0$, то возвратиться к шагу D4.
D6 [R на место R_{i+h} .] Установить $R_{i+h} \leftarrow R$. ■

Соответствующая MIX-программа не намного длиннее, чем наша программа для простых вставок. Строки 08–19 этой программы перенесены из программы S в более общий контекст алгоритма D.

Программа D. (Сортировка с убывающим шагом.) Предполагается, что шаги сортировки хранятся во вспомогательной таблице и h_s находится в ячейке $H + s$; все шаги сортировки меньше N . Содержимое регистров: $rI1 \equiv j - N$; $rI2 \equiv i$; $rA \equiv R \equiv K$; $rI3 \equiv s$; $rI4 \equiv h$. Заметим, что эта программа сама себя изменяет. Это сделано для того, чтобы добиться более эффективного выполнения внутреннего цикла.

START	ENT3	T	1	D1. Цикл по s . $s \leftarrow t$.
1H	LD4	H, 3	T	D2. Цикл по j . $h \leftarrow h_s$.
	ENT1	INPUT, 4	T	Изменить адреса в трех
	ST1	6F(0:2)	T	инструкциях в
	ST1	7F(0:2)	T	основном цикле.
	ENN1	-N, 4	T	$rI1 \leftarrow N - h$.
	ST1	4F(0:2)	T	
	ENT1	1-N, 4	T	$j \leftarrow h + 1$.

2H	LDA	INPUT+N, 1	$NT - S$	D3. Установить i, K, R .
4H	ENT2	N-H, 1	$NT - S$	$i \leftarrow j - h$. (Изменяемая инструкция)
5H	CMPA	INPUT, 2	$B + NT - S - A$	D4. Сравнить K, K_i .
	JOE	7F	$B + NT - S - A$	K шагу D6, если $K \geq K_i$.
	LDX	INPUT, 2	B	D5. Переместить R_i , уменьшить i .
6H	STX	INPUT+N, 2	B	$R_{i+h} \leftarrow R_i$. (Изменяемая инструкция)
	DEC2	0, 4	B	$i \leftarrow i - h$.
	J2P	5B	B	K шагу D4, если $i > 0$.
7H	STA	INPUT+N, 2	$NT - S$	D6. R на место R_{i+h} . (Изменяемая инструкция)
8H	INC1	1	$NT - S$	$j \leftarrow j + 1$.
	J1NP	2B	$NT - S$	K D3, если $j \leq N$.
	DEC3	1	T	
	J3P	1B	T	$t \geq s \geq 1$.

■

***Анализ метода Шелла.** Чтобы выбрать хорошую последовательность шагов сортировки для алгоритма D, нужно проанализировать время работы как функцию от этих шагов. Это приводит к очень красивым, но еще не до конца решенным математическим задачам; никому до сих пор не удалось найти наилучшую возможную последовательность шагов для больших N . Тем не менее известно довольно много интересных фактов о поведении сортировки методом Шелла с убывающим шагом, и мы здесь их кратко изложим; подробности можно найти в приведенных ниже упражнениях. [Читателям, не имеющим склонности к математике, лучше пропустить следующие несколько страниц, до формул (8).]

Счетчики частот выполнения в программе D показывают, что на время выполнения влияют пять факторов: размер файла N , число просмотров (т.е. число шагов) $T = t$, сумма шагов

$$S = h_1 + \dots + h_t,$$

число сравнений $B + NT - S - A$ и число перемещений B . Как и при анализе программы S, здесь A равно числу левосторонних минимумов, встречающихся при промежуточных операциях сортировки, а B равно числу инверсий в подфайлах. Основным фактором, от которого зависит время работы, является величина B , поэтому на нее мы и обратим главным образом свое внимание. При анализе будет предполагаться, что ключи различны и первоначально расположены в случайном порядке.

Назовем операцию шага D2 " h -сортировкой". Тогда сортировка методом Шелла состоит из h_t -сортировки, за которой следует h_{t-1} -сортировка, ..., за которой следует h_1 -сортировка. Файл, в котором $K_i \leq K_{i+h}$ при $1 \leq i \leq N - h$, будем называть h -упорядоченным.

Рассмотрим сначала простейшее обобщение простых вставок, когда имеются всего два шага $h_2 = 2$ и $h_1 = 1$. Во время второго просмотра имеем 2-упорядоченную последовательность ключей $K_1 K_2 \dots K_N$. Легко видеть, что число перестановок $a_1 a_2 \dots a_n$ множества $\{1, 2, \dots, n\}$, таких, что $a_i \leq a_{i+2}$ при $l \leq i \leq n - 2$, равно

$$\binom{n}{\lfloor n/2 \rfloor},$$

так как существует всего одна 2-упорядоченная перестановка для каждого выбора $\lfloor n/2 \rfloor$ элементов, расположенных в четных позициях $a_2 a_4, \dots$, тогда остальные $\lceil n/2 \rceil$ элементов попадают в позиции с нечетными номерами. После 2-сортировки случайного файла с одинаковой вероятностью может получиться любая 2-упорядоченная перестановка. Каково среднее число инверсий во всех таких перестановках?

Picture: Рис. 11. Соответствие между 2-упорядочением и путями на решетке. Курсивом набраны веса, соответствующие числу инверсий в 2-упорядоченной перестановке.

Пусть A_n — суммарное число инверсий во всех 2-упорядоченных перестановках множества $\{1, 2, \dots, n\}$. Ясно, что $A_1 = 0$, $A_2 = 1$, $A_3 = 2$, а из рассмотрения шести случаев

$$1324 \quad 1234 \quad 1243 \quad 2134 \quad 2143 \quad 3142$$

находим, что $A_4 = 1 + 0 + 1 + 1 + 2 + 3 = 8$. Чтобы исследовать A_n в общем случае, рассмотрим решетчатую диаграмму на рис. 11 для $n = 15$. В такой диаграмме 2-упорядоченную перестановку можно представить в виде пути из верхней левой угловой точки $(0, 0)$ в нижнюю правую угловую точку $(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$, если делать k -й шаг пути вправо или вниз в соответствии с тем, находится ли k в

четной или нечетной позиции перестановки. Этим правилом определяется взаимно однозначное соответствие между 2-упорядоченными перестановками и n -шаговыми путями из одного угла решетчатой диаграммы в другой. Например, изображенный на рисунке путь соответствует перестановке

$$2\ 1\ 3\ 4\ 6\ 5\ 7\ 10\ 8\ 11\ 9\ 12\ 14\ 13\ 15. \quad (1)$$

Далее, вертикальным отрезкам пути можно приписать "веса", как показано на диаграмме; отрезку, ведущему из точки (i, j) в точку $(i + 1, j)$ приписывается вес $|i - j|$. Немного подумав, читатель убедится в том, что сумма этих весов вдоль каждого пути равна числу инверсий в соответствующей перестановке (см. упр. 12). Так, например, перестановка (1) содержит $1 + 0 + 1 + 0 + 1 + 2 + 1 = 6$ инверсий.

Если $a \leq a'$ и $b \leq b'$, то число допустимых путей из (a, b) в (a', b') равно числу способов перемешать $a' - a$ вертикальных отрезков с $b' - b$ горизонтальными, а именно

$$\binom{a' - a + b' - b}{a' - a}.$$

Следовательно, число перестановок, для которых соответствующие пути проходят через вертикальный отрезок из (i, j) в $(i + 1, j)$, равно

$$\binom{i + j}{i} \binom{n - i - j - 1}{\lfloor n/2 \rfloor - j}.$$

Умножая это значение на вес данного отрезка и суммируя по всем отрезкам, получаем

$$\begin{aligned} A_{2n} &= \sum_{\substack{0 \leq i \leq n \\ 0 \leq j \leq n}} |i - j| \binom{i + j}{i} \binom{2n - i - j - 1}{n - j}; \\ A_{2n+1} &= \sum_{\substack{0 \leq i \leq n \\ 0 \leq j \leq n}} |i - j| \binom{i + j}{i} \binom{2n - i - j}{n - j}; \end{aligned} \quad (2)$$

Знаки абсолютной величины в этих суммах несколько усложняют вычисления, но в упр. 14 показано, что величина A_n имеет удивительно простой вид: $\lfloor n/2 \rfloor 2^{n-2}$. Следовательно, среднее число инверсий в случайной 2-упорядоченной перестановке равно

$$\lfloor n/2 \rfloor 2^{n-2} / \binom{n}{\lfloor n/2 \rfloor}.$$

По формуле Стирлинга эта величина асимптотически приближается к $\sqrt{\pi/128} n^{3/2} \approx 0.15 n^{3/2}$. Как легко видеть, максимальное число инверсий равно

$$\binom{\lfloor n/2 \rfloor + 1}{2} \approx \frac{1}{8} n^2.$$

Полезно исследовать распределение числа инверсий более тщательно, рассмотрев производящие функции

$$\begin{aligned} h_1(z) &= 1, & h_2(z) &= 1 + z, \\ h_3(z) &= 1 + 2z, & h_4(z) &= 1 + 3z + z^2 + z^3, \dots, \end{aligned} \quad (3)$$

как в упр. 15. Таким образом, найдем, что стандартное отклонение тоже пропорционально $n^{3/2}$, так что число инверсий не слишком устойчиво распределено около своего среднего значения. Рассмотрим теперь общий двухпроходный случай алгоритма D, когда шаги сортировки равны h и 1.

Теорема Н. Среднее число инверсий в h -упорядоченной перестановке множества $\{1, 2, \dots, n\}$ равно

$$f(n, h) = \frac{2^{2q-1} q! q!}{(2q+1)!} \left(\binom{h}{2} q(q+1) + \binom{r}{2} (q+1) - \frac{1}{2} \binom{h-r}{2} q \right), \quad (4)$$

где $q = \lfloor n/h \rfloor$, $r = n \bmod h$. Эта теорема принадлежит Дугласу Ханту [Bachelor's thesis, Princeton University (April, 1967)]. Заметим, что формула справедлива и при $h \geq n$: $f(n, h) = \frac{1}{2} \binom{n}{2}$.

Доказательство. В h -упорядоченной перестановке содержится r упорядоченных подпоследовательностей длины $q + 1$ и $h - r$ подпоследовательностей длины q . Каждая инверсия образуется из элементов двух различных подпоследовательностей, а каждая пара различных упорядоченных подпоследовательностей в случайной h -упорядоченной перестановке определяет случайную 2-упорядоченную перестановку. Поэтому среднее число инверсий равно сумме средних значений числа инверсий во всех парах различных подпоследовательностей, а именно

$$\binom{r}{2} \frac{A_{2q+2}}{\binom{2q+2}{q+1}} + r(h-r) \frac{A_{2q+1}}{\binom{2q+1}{q}} + \binom{h-r}{2} \frac{A_{2q}}{\binom{2q}{q}} = f(n, h). \blacksquare$$

Следствие. Если последовательность приращений удовлетворяет условию

$$h_{s+1} \bmod h_s = 0 \quad \text{при } t > s \geq 1, \quad (5)$$

то среднее число операций перемещения в алгоритме D равно

$$\sum_{t \geq s \geq 1} (r_s f(q_s + 1, h_{s+1}/h_s) + (h_s - r_s) f(q_s, h_{s+1}/h_s)), \quad (6)$$

где $r_s = N \bmod h_s$, $q_s = \lfloor N/h_s \rfloor$, $h_{t+1} = Nh_t$, а функция f определяется формулой (4).

Доказательство. Процесс h_s -сортировки состоит из сортировки простыми вставками $r_s(h_{s+1}/h_s)$ -упорядоченных подфайлов длины $q_s + 1$ и $(h_s - r_s)$ таких подфайлов длины q_s . Поскольку мы предполагаем, что исходная перестановка случайна и все ее элементы различны, то из условий делимости следует, что каждый из подфайлов — "случайная" (h_{s+1}/h_s) -упорядоченная перестановка в том смысле, что все (h_{s+1}/h_s) -упорядоченные перестановки равновероятны. \blacksquare

Условие (5) этого следствия всегда выполняется для *двухпроходной* сортировки методом Шелла, когда шаги равны соответственно h и 1. Пусть $q = \lfloor N/h \rfloor$, а $r = N \bmod h$, тогда среднее значение величины B в программе D равно

$$rf(q + 1, N) + (h - r)f(q, N) + f(N, h) = \frac{r}{2} \binom{q+1}{2} + \frac{h-r}{2} \binom{q}{2} + f(N, h).$$

В первом приближении функция $f(n, h)$ равна $(\sqrt{\pi}/8)n^{3/2}h^{1/2}$; ср. с гладкой кривой на рис. 12. Следовательно, время работы

Picture: Рис. 12. Среднее число инверсий $f(n, h)$ в h -упорядоченном файле из n элементов для случая $n = 64$.

двухпроходной программы D примерно пропорционально $2N^2/h + \sqrt{\pi N^3 h}$. Поэтому наилучшее значение h равно приблизительно $\sqrt[3]{16N/\pi} \approx 1.72\sqrt[3]{N}$; при таком выборе h среднее время работы пропорционально $N^{5/3}$.

Таким образом, даже применяя метод Шелла с всего двумя шагами, можно существенно сократить время по сравнению с простыми вставками, с $O(N^2)$ до $O(N^{1.667})$. Ясно, что можно добиться лучших результатов, если использовать больше шагов. В упр. 18 обсуждается оптимальный выбор h_t, \dots, h_1 при фиксированном t в случае, когда значения h ограничены условием делимости; время работы при больших N сокращается до $O(N^{1.5+\varepsilon/2})$ где $\varepsilon = 1/(2^t - 1)$. Если мы пользуемся приведенными выше формулами, барьер $N^{1.5}$ преодолеть невозможно, потому что при последнем просмотре в сумму инверсий всегда вносится вклад

$$f(N, h_2) \approx (\sqrt{\pi}/8)N^{3/2}h_2^{1/2}.$$

Но интуиция подсказывает, что, если шаги h_t, \dots, h_1 не будут удовлетворять условию делимости (5), можно достичь большего. Например, при последовательном выполнении 8-, 4- и 2-сортировок невозможно взаимодействие между ключами в четных и нечетных позициях; поэтому на долю заключительной 1-сортировки останется $O(N^{3/2})$ инверсий. В то же время при последовательном выполнении 7-, 5- и 3-сортировок файл перетряхивается так, что при заключительной 1-сортировке не может встретиться более $2N$ инверсий! (См. упр. 26.) На самом деле наблюдается удивительное явление.

Теорема К. После h -сортировки k -упорядоченный файл остается k -упорядоченным.

Таким образом, файл, который был сначала 7-отсортирован, а потом 5-отсортирован, становится одновременно 7- и 5-упорядоченным. А если мы подвергнем его 3-сортировке, то полученный файл будет 7-, 5- и 3-упорядочен. Примеры проявления этого замечательного свойства можно найти в табл. 4.

Picture: Таблица 4 Сортировка с убывающим шагом (7, 5, 3, 1)

Доказательство. В упр. 20 показано, что эта теорема вытекает из следующей леммы:

Лемма L. Пусть m, n, r — неотрицательные целые числа, а x_1, \dots, x_{m+r} и y_1, \dots, y_{n+r} — произвольные последовательности чисел, такие, что

$$y_1 \leq x_{m+1}, y_2 \leq x_{m+2}, \dots, y_r \leq x_{m+r}. \quad (7)$$

Если последовательности x и y отсортировать независимо, так что $x_1 \leq \dots \leq x_{m+r}$ и $y_1 \leq \dots \leq y_{n+r}$, то соотношения (7) останутся в силе.

Доказательство. Известно, что все, кроме, быть может, m элементов последовательности x , превосходят (т. е. больше или равны) некоторые элементы последовательности y , причем различные элементы x превосходят различные элементы y . Пусть $1 \leq j \leq r$. Так как после сортировки элемент x_{m+j} превосходит $m+j$ элементов x , то он превосходит по крайней мере j элементов y , а значит, он превосходит j наименьших элементов y . Следовательно, после сортировки имеем $x_{m+j} \geq y_j$. ■ ■

Из теоремы К видно, что при сортировке желательно пользоваться взаимно простыми значениями шагов, однако непосредственно из нее не следуют точные оценки числа перемещений, выполняемых алгоритмом D. Так как число перестановок множества $\{1, 2, \dots, n\}$, одновременно h - и k -упорядоченных, не всегда является делителем $n!$, то понятно, что теорема К объясняет далеко не все; в результате k - и h -сортировок некоторые k - и h -упорядоченные файлы получаются чаще других. Более того, не существует очевидного способа отыскать "наихудший случай" для алгоритма D при произвольной последовательности шагов сортировки h_t, \dots, h_1 . Поэтому до сих пор все попытки анализа этого алгоритма в общем случае были тщетны; по существу, все, что нам известно, — это приближенное асимптотическое поведение максимального времени работы в некоторых случаях.

Теорема Р. Если $h_s = 2^s - 1$ при $1 \leq s \leq t = \lfloor \log_2 N \rfloor$, то время работы алгоритма D есть $O(N^{3/2})$

Доказательство. Достаточно найти оценку B_s числа перемещений при s -м просмотре, такую, чтобы $B_t + \dots + B_1 = O(N^{3/2})$. Для первых $t/2$ просмотров при $t \leq s \leq t/2$ можно воспользоваться очевидной оценкой $B_s = O(h_s(N/h_s)^2)$, а для последующих просмотров можно применить результат упр. 23: $B_s = O(Nh_{s+2}h_{s+1}/h_s)$. Следовательно, $B_t + \dots + B_1 = O(N(2 + 2^2 + \dots + 2^{t/2} + 2^{t/2} + \dots + 2)) = O(N^{3/2})$. ■

Эта теорема принадлежит А. А. Папернову и Г. В. Стасевичу [Проблемы передачи информации, 1,3 (1965), 81–98]. Она дает верхнюю оценку максимального времени работы алгоритма, а не

Таблица 5

Анализ алгоритма D при $N = 8$							
Шаги		A_{ave}	B_{ave}	S	T	время машины MIX	
	1	1.718	14.000	1	1	204.85 <i>u</i>	
	2	1	2.667	9.657	3	2	235.91 <i>u</i>
	3	1	2.917	9.100	4	2	220.16 <i>u</i>
	4	1	3.083	10.000	5	2	217.75 <i>u</i>
	5	1	2.601	10.000	6	2	210.00 <i>u</i>
	6	1	2.135	10.667	7	3	206.60 <i>u</i>
	7	1	1.718	12.000	8	2	208.85 <i>u</i>
4	2	1	3.500	8.324	7	3	272.32 <i>u</i>
5	3	1	3.301	8.167	9	3	251.60 <i>u</i>
3	2	1	3.320	7.829	6	3	278.50 <i>u</i>

просто оценку среднего времени работы. Этот результат нетривиален, поскольку максимальное время работы в случае, когда приращения h удовлетворяют условию делимости (5), имеет порядок N^2 , а в упр. 24 доказано, что показатель $3/2$ уменьшить нельзя.

Интересное улучшение по сравнению с теоремой Р обнаружил в 1969 г. Воган Пратт. Если все шаги сортировки выбираются из множества чисел вида $2^p 3^q$, меньших N , то время работы алгоритма D будет порядка $N(\log N)^2$. В этом случае также можно внести в алгоритм несколько существенных упрощений. К сожалению, метод Пратта требует сравнительно большого числа просмотров, так что это не лучший способ выбора шагов, если только N не очень велико; см. упр. 30 и 31.

Рассмотрим общее время работы программы D , именно $(9B + 10NT + 13T - 10S - 3A + 1)u$. В табл. 5 показано среднее время работы для различных последовательностей шагов при $N = 8$. Каждый элемент таблицы можно вычислить с помощью формул, приведенных выше или в упр. 19, за исключением случаев, когда шаги равны 5 3 1 и 3 2 1; для этих двух случаев было проведено тщательное исследование всех $8!$ перестановок. Заметим, что при таком малом значении N в общем времени работы преобладают вспомогательные операции, поэтому наилучшие результаты получаются при $t = 1$; следовательно, при $N = 8$ лучше всего пользоваться простыми вставками. (Среднее время работы программы S при $N = 8$ равно всего $191.85u$.) Любопытно, что наилучший результат в двухпроходном алгоритме достигается при $h_2 = 6$, поскольку большая величина 5 оказывается важнее малой величины B . Аналогично три шага 3 2 1 минимизируют среднее число перемещений, но это не самая лучшая последовательность для трех просмотров. Быть может, интересно привести некоторые "наихудшие" перестановки, максимизирующие число перемещений, так как общий способ построения таких перестановок до сих пор не известен:

$$\begin{array}{ll} h_3 = 5, & h_2 = 3, & h_1 = 1: & 8 & 5 & 2 & 6 & 3 & 7 & 4 & 1 & (19 \text{ перемещений}) \\ h_3 = 3, & h_2 = 2, & h_1 = 1: & 8 & 3 & 5 & 7 & 2 & 4 & 6 & 1 & (17 \text{ перемещений}) \end{array}$$

С ростом N наблюдается несколько иная картина. В табл. 6 показаны приближенные значения числа перемещений для различных последовательностей шагов при $N = 1000$. Первые несколько последовательностей удовлетворяют условию делимости (5), так что можно воспользоваться формулой (6); для получения средних значений при других последовательностях шагов применялись эмпирические тесты. Были сгенерированы пять файлов по 1000 случайных элементов, и каждый файл сортировался с каждой последовательностью шагов.

Эти данные позволяют выявить некоторые характеристики, но поведение алгоритма D все еще остается неясным. Шелл первоначально предполагал использовать шаги $\lfloor N/2 \rfloor$, $\lfloor N/4 \rfloor$, $\lfloor N/8 \rfloor$, ... но это нежелательно, если двоичное представление числа N содержит длинные цепочки нулей. Лазарус и Фрэнк [CASM, 3 (1960), 20–22] предложили использовать, по существу, ту же последовательность, но добавляя 1 там, где это необходимо, чтобы сделать все шаги нечетными. Хиббард [CASM, 6 (1963), 206–213] предложил шаги вида $2^k - 1$; Папернов и Стасевич предложили последовательность $2^k + 1$. Среди других естественных последовательностей, использованных для получения табл. 6, — последовательности $(2^k - (-1)^k)/3$, $(3^k - 1)/2$ и числа Фибоначчи.

Минимальное число перемещений 6600 наблюдается для шагов вида $2^k + 1$, но важно понимать, что надо учитывать не только число перемещений, хотя именно оно асимптотически доминирует в общем времени работы. Так как время работы программы D равно $9B + 10NT + \dots$ единиц, ясно, что экономия одного просмотра примерно эквивалентна сокращению числа перемещений на $\frac{10}{9}N$; мы готовы добавить 1100 перемещений, если за счет этого удастся сэкономить один просмотр. Поэтому представляется неразумным начинать с h_t , большего, чем, скажем, $N/3$, поскольку большой шаг не убавит числа последующих перемещений настолько, чтобы оправдать первый просмотр.

Большое число экспериментов с алгоритмом D провели Джеймс Петерсон и Дэвид Л. Рассел в Стэнфордском университете в 1971 г. Они обнаружили, что для среднего числа перемещений B хорошим

Picture: Таблица 6.

приближением при $100 \leq N \leq 60\,000$ служат следующие формулы:

1. $0.09N^{1.27}$ или $.30N(\ln N)^2 - 1.35N \ln N$ для последовательности шагов $2^k + 1, \dots, 9, 5, 3, 1$;
 1. $2.2N^{1.26}$ или $.29N(\ln N)^2 - 1.26N \ln N$ для последовательности шагов $2^k - 1, \dots, 15, 7, 3, 1$;
 1. $1.12N^{1.28}$ или $.36N(\ln N)^2 - 1.73N \ln N$ для последовательности шагов $(2^k \pm 1)/3, \dots, 11, 5, 3, 1$;
 1. $6.6N^{1.25}$ или $.33N(\ln N)^2 - 1.26N \ln N$ для последовательности шагов $(3^k - 1)/2, \dots, 40, 13, 4, 1$.

Например, при $N = 20\,000$ для всех этих типов шагов получаем соответственно $B \approx 31\,000, 33\,000, 35\,000, 39\,000$. Табл. 7 дает

Таблица 7

Количества перемещений по просмотрам (примеры для $N = 20000$)

h_s	B_s	h_s	B_s	h_s	B_s
4095	19460	4097	19550	3280	25210
2047	15115	2049	14944	1093	28324
1023	15869	1025	15731	364	35477
511	18891	513	18548	121	47158
255	22306	257	21827	40	62110
127	27400	129	27814	13	88524
63	35053	65	33751	4	74599
31	34677	33	34303	1	34666
15	51054	17	46044		
7	40382	9	35817		
3	24044	5	19961		
1	16789	3	9628		
		1	13277		

типичную картину того, как распределяются перемещения по просмотрам в трех из этих экспериментов. Любопытно, что *оба* вида функций $\alpha N(\ln N)^2 + \beta N \ln N$ и βN^α , кажется, довольно хорошо согласуются с наблюдаемыми данными, хотя степенная функция была существенно лучше при меньших значениях N . Дальнейшие эксперименты были выполнены для последовательности шагов $2^k - 1$ со значением N , достигающим 250 000; сорок пять испытаний с $N = 250\,000$ дали значение $B \approx 7\,900\,000$ при наблюдаемом стандартном отклонении 50 000. "Наиболее подходящими" формулами для диапазона $100 \leq N \leq 250\,000$ оказались соответственно $1.21N^{1.26}$ и $.39N(\ln N)^2 - 2.33N \ln N$. Так как коэффициенты в представлении степенной функцией остались почти такими же, в то время как коэффициенты в логарифмическом представлении довольно резко изменились, то разумно предположить, что именно степенная функция описывает истинное асимптотическое поведение метода Шелла.

Эти эмпирические данные ни коим образом не исчерпывают всех возможностей, и мы не получили оснований для решительных заключений о том, какие же последовательности шагов являются наилучшими для алгоритма D. Поскольку приращения вида $(3^k - 1)/2$ не увеличивают существенно числа перемещений и поскольку для них требуется лишь примерно 5/8 числа просмотров, необходимых для шагов других типов, то, очевидно, *разумно выбирать последовательность шагов следующим образом:*

$$\text{Принять } h_t = 1, h_{s+1} = 3h_s + 1 \text{ и остановиться на } h_t, \text{ когда } h_{t+2} \geq N. \quad (8)$$

Вставки в список. Оставим теперь метод Шелла и рассмотрим другие пути усовершенствования простых вставок. Среди общих способов улучшения алгоритма один из самых важных основывается на тщательном анализе структур данных, поскольку реорганизация структур данных, позволяющая избежать ненужных операций, часто дает существенный эффект. Дальнейшее обсуждение этой общей идеи можно найти в § 2.4, в котором изучается довольно сложный алгоритм. Посмотрим, как она применяется к такому нехитрому алгоритму, как простые вставки. Какова наиболее подходящая структура данных для алгоритма S?

Сортировка простыми вставками состоит из двух основных операций:

- i) просмотра упорядоченного файла для нахождения наибольшего ключа, меньшего или равного данному ключу;
- ii) вставки новой записи в определенное место упорядоченного файла.

Файл—это, очевидно, линейный список, и алгоритм S обрабатывает его, используя последовательное распределение (п. 2.2.2); поэтому для выполнения каждой операции вставки необходимо переместить примерно половину записей. С другой стороны, нам известно, что для вставок идеально подходит связанное распределение (п. 2.2.3), так как при этом требуется изменить лишь несколько связей; другая операция—последовательный просмотр—при связанном распределении почти так же проста, как и при последовательном. Поскольку списки всегда просматриваются в одном и том же направлении, то достаточно иметь списки с одной связью. Таким образом, приходим к выводу, что "правильная" структура данных для метода простых вставок—линейные списки с одной связью. Удобно также изменить алгоритм S, чтобы список просматривался в возрастающем порядке.

$$L_0 = p(1); L_{p(i)} = p(i+1) \text{ при } 1 \leq i < N; L_{p(N)} = 0. \quad (9)$$

(9)

- L5** [Вставить в список.] Установить $L_q \leftarrow j, L_j \leftarrow p$. ■

Пример применения алгоритма вставок в список

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K_i	—	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
L_j	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0
L_j	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0	15
L_j	14	—	—	—	—	—	—	—	—	—	—	—	—	—	16	0	15

рый выполняет около $\frac{1}{4}N^2$ сравнений и около $\frac{1}{4}N^2$ перемещений. Мы усовершенствовали его, рассмотрев бинарные вставки, при которых выполняется около $N \log_2 N$ сравнений и $\frac{1}{4}N^2$ перемещений. Несколько изменив структуру данных, применив "двухпутевые вставки", сумели сократить число перемещений до $\frac{1}{8}N^2$. При сортировке методом Шелла "с убывающим шагом" число сравнений и перемещений снижается примерно до $N^{1.3}$ для тех значений N , которые встречаются на практике; при $N \rightarrow \infty$ это число можно сократить до порядка $N(\log_2 N)^2$. Дальнейшее стремление улучшить алгоритм—путем применения связанной структуры данных—привело нас к вставкам в список, при которых выполняется около $\frac{1}{4}N^2$ сравнений, 0 перемещений и $2N$ изменений связей.

Можно ли соединить лучшие свойства этих методов, сократив число сравнений до порядка $N \log N$, как при бинарных вставках, и исключив при этом перемещения данных, как при вставках

Picture: Рис. 13. Пример схемы Уилера для вставок в дерево.

в список? Ответ утвердительный: это достигается переходом к древовидной структуре. Такая возможность была впервые исследована около 1957 г. Д. Дж. Уилером, который предложил использовать двухпутевые вставки до тех пор, пока не появится необходимость перемещать данные. Тогда вместо того, чтобы их перемещать, вставляется указатель на новую область памяти, и тот же самый метод рекуррентно применяется ко всем элементам, которые нужно вставить в эту новую область памяти. Оригинальный метод Уилера [см. A. S. Douglas, *Comp. J.*, 2 (1959), 5] представляет собой сложную комбинацию последовательной и связанной памяти с узлами переменного размера; для наших шестнадцати чисел было бы сформировано дерево, показанное на рис. 13. Аналогичную, но более простую схему вставки в дерево с использованием бинарных деревьев независимо разработал около 1958 г. К.М. Бернес-Ли [см. *Comp. J.*, 3 (1960), 174, 184]. Сам этот метод и его модернизации весьма важны как для сортировки, так и для поиска, поэтому подробно они обсуждаются в гл. 6.

Еще один путь улучшить простые вставки—попытаться вставлять несколько элементов одновременно. Если, например, имеется файл из 1000 элементов и 998 из них уже отсортированы, то алгоритм S выполнит еще два просмотра файла (вставив сначала R_{999} , а потом R_{1000}). Очевидно, можно сэкономить время, если сначала сравнить ключи K_{999} с K_{1000} чтобы выяснить, который из них больше, а потом вставить их *оба* за один просмотр файла. Комбинированная операция такого рода требует около $(2/3)N$ сравнений и перемещений (ср. с упр. 3.4.2–5) вместо двух просмотров, примерно по $N/2$ сравнений и перемещений каждый.

Иначе говоря, обычно бывает полезно "группировать" операции, которые требуют длительного поиска, чтобы можно было выполнить несколько операций вместе. Если довести эту идею до ее естественного завершения, то мы заново откроем для себя сортировку посредством слияния, настолько важную, что ей посвящен отдельный пункт.

Сортировка с вычислением адреса. Теперь уж мы, несомненно, исчерпали все возможные способы усовершенствовать метод простых вставок, но давайте подумаем еще! Представьте себе, что вам дали чистый лист бумаги и собираются диктовать какие-то слова. Ваша задача—записать их в алфавитном порядке и вернуть листок с отсортированным списком слов. Услышав слово на букву А, вы будете стремиться записать его ближе к верхнему краю страницы, тогда как слово на букву Я будет, по-видимому, помещено ближе к нижнему краю страницы и т. д. Аналогичный способ применяется при расстановке книг на полке по фамилиям авторов, если книги берутся с пола в случайном порядке: ставя книгу на полку, вы оцениваете ее конечное положение, сокращая таким образом число необходимых сравнений и перемещений. (Эффективность процесса повышается, если на полке имеется немного больше места, чем это абсолютно необходимо.) Такой метод машинной сортировки впервые предложили Исаак и Синглтон, [*JACM*, 3 (1956), 169–174]; он получил дальнейшее развитие в работе Кронмэла и Тартара [*Proc. ACM Nat'l Conf.*, 21 (1966), 331–337].

Сортировка с вычислением адреса обычно требует дополнительного пространства памяти либо для того, чтобы оставить достаточно свободного места и не делать много лишних перемещений, либо для хранения вспомогательных таблиц, которые бы позволяли учитывать неравномерность распределения ключей. (См. сортировку распределяющим подсчетом (алгоритм 5.2D),

Упражнения

1. [M25](*Беспорядок в библиотеке.*) Небрежные читатели часто ставят книги на полки в библиотеке не на свое место. Один из способов измерить степень беспорядка в библиотеке—посмотреть, какое минимальное, количество раз пришлось бы брать книгу с одного места и вставлять ее в другое место до тех пор, пока книги не окажутся расположенными в правильном порядке. Пусть $\pi = a_1 a_2 \dots a_n$ — перестановка множества $\{1, 2, \dots, n\}$. "Операция удаления-вставки" заменяет π на

$$a_1 a_2 \dots a_{i-1} \dots a_j a_i a_{j+1} \dots a_n$$

или на

$$a_1 \dots a_j a_i a_{j+1} \dots a_{i-1} a_{i+1} \dots a_n$$

при некоторых i и j . Пусть $\text{dis}(\pi)$ —минимальное число операций удаления-вставки, необходимое для упорядочения перестановки π . Можно ли выразить $\text{dis}(\pi)$ через более простые характеристики π ?

2. [40] Проведите эксперименты со следующей модификацией сортировки с убывающим шагом, имеющей целью ускорение "внутреннего цикла": для $s = t, t - 1, \dots, 2$ и для $j = h_s + 1, h_s + 2, \dots, N$ поменять местами $R_{j-h_s} \leftrightarrow R_j$, если $K_{j-h_s} > K_j$. (Таким образом, результат обменов не распространяется; не производится сравнений $K_{j-h_s} : K_{j-2h_s}$. Записи не обязательно будут h_s -отсортированы, но эти предварительные просмотры способствуют сокращению числа инверсий.) Сортировка завершается применением простых вставок.

5.2.2. Обменная сортировка

Мы подошли теперь ко второму из семейств алгоритмов сортировки, упомянутых в самом начале § 5.2,—к методам "обменов" или "транспозиций", предусматривающих систематический обмен местами между элементами пар, в которых нарушается упорядоченность, до тех пор, пока таких пар не останется. Процесс простых вставок (алгоритм 5.2.1S) можно рассматривать как обменную сортировку: мы берем новую запись R_j и, по существу, меняем местами с соседями слева до тех пор, пока она не займет нужного места. Таким образом, классификация методов сортировки по таким семействам, как "вставки", "обмены", "выбор" и т. д., не всегда четко определена. В этом пункте мы рассмотрим четыре типа методов сортировки, для которых обмены являются основной характеристикой: *обменную сортировку с выбором* ("метод пузырька"), *обменную сортировку со слиянием* (параллельную сортировку Бэтчера), *обменную сортировку с разделением* ("быструю сортировку" Хоара), *поразрядную обменную сортировку*.

Метод пузырька. Пожалуй, наиболее очевидней способ обменной сортировки это сравнивать K_1 с K_2 , меняя местами R_1 и R_2 , если их ключи не упорядочены, затем проделать то же самое с R_2 и R_3 , R_3 и R_4 и т. д. При выполнении этой последовательности операций записи с большими ключами будут продвигаться вправо, и на самом деле запись с наибольшим ключом займет положение R_N . При многократном выполнении этого процесса соответствующие записи попадут в позиции R_{N-1} , R_{N-2} и т. д., так что в конце концов все записи будут упорядочены.

На рис. 14 показано действие этого метода сортировки на шестнадцати ключах 503 087 512... 703. Файл чисел удобно

Picture: Рис. 14. Сортировка методом пузырька в действии.

представлять не горизонтально, а вертикально, чтобы запись R_N была сверху, а R_1 —снизу. Метод назван "методом пузырька", потому что большие элементы, подобно пузырькам, "всплывают" на соответствующую позицию в противоположность "методу погружения" (т. е. методу простых вставок), в котором элементы погружаются на соответствующий уровень. Метод пузырька известен также под более прозаическими именами, такими, как "обменная сортировка с выбором" или метод "распространения". Нетрудно видеть, что после каждого просмотра файла все записи, начиная с самой последней, которая участвовала в обмене, и выше, должны занять свой окончательные позиции, так что их не нужно проверять при последующих просмотрах. На рис. 14 такого рода продвижения алгоритма отмечены черточками. Заметим, например, что после четвертого просмотра стало известно, что еще пять записей заняли свои окончательные позиции. При последнем, просмотре вообще не было произведено обменов. Теперь, сделав эти наблюдения, мы готовы сформулировать алгоритм.

Алгоритм В. (Метод пузырька.) Записи R_1, \dots, R_N переразмещаются на том же месте; после завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$.

B1 [Начальная установка BOUND.] Установить $\text{BOUND} \leftarrow N$. (BOUND—индекс самого верхнего элемента, о котором еще не известно, занял ли он уже свою окончательную позицию; таким образом, мы отмечаем, что к этому моменту еще ничего не известно.)

- B2** [Цикл по j .] Установить $t \leftarrow 0$. Выполнить шаг **B3** при $j = 1, 2, \dots, \text{BOUND} - 1$. Затем перейти к шагу **B4**. (Если $\text{BOUND} = 1$, то сразу перейти к **B4**.)
- B3** [Сравнение/обмен $R_j : R_{j+1}$]⁶. Если $K_j > K_{j+1}$, то поменять местами $R_j \leftrightarrow R_{j+1}$ и установить $t \leftarrow j$.
- B4** [Были ли обмены?] Если $t = 0$, то завершить работу алгоритма. В противном случае установить $\text{BOUND} \leftarrow t$ и возвратиться к шагу **B2**. ■

Picture: Рис. 15. Блок-схема сортировки методом пузырька.

Программа В. (Метод пузырька.) Как и в предыдущих MIX-программах этой главы, мы предполагаем, что сортируемые элементы находятся в ячейках $\text{INPUT} + 1, \dots, \text{INPUT} + N$; регистры: $\text{rI1} \equiv t$; $\text{rI2} \equiv j$.

START	ENT1	N	1	B1. Начальная установка BOUND . $t \leftarrow N$.
1H	ST1	BOUND (1:2)	A	$\text{BOUND} \leftarrow t$.
	ENT2	1	A	B2. Цикл. по j .
	ENT1	0	A	$t \leftarrow 0$.
	JMP	BOUND	A	Выход, если $j \geq \text{BOUND}$.
3H	LDA	INPUT, 2	C	B3. Сравнение/обмен $R_j : R_{j+1}$.
	CMPA	INPUT+1,2	C	
	JLE	2F	C	Если $K_j \leq K_{j+1}$, то без обмена.
	LDX	INPUT+1,2	B	R_{j+1}
	STX	INPUT, 2	B	$\rightarrow R_j$.
	STA	INPUT+1,2	B	(прежнее R_j) $\rightarrow R_{j+1}$
	ENT1	0,2	B	$t \leftarrow j$.
9H	INC2	1	C	$j \leftarrow j + 1$.
BOUND	ENTX	*,2	A + C	$\text{rX} \leftarrow j - \text{BOUND}$. (Изменяемая инструкция)
	JXN	3B	A + C	$1 \leq j < \text{BOUND}$.
4H	J1P	1B	A	B4. Были ли обмены? К B2 , если $t > 0$.

■

Анализ метода пузырька. Очень полезно исследовать время работы алгоритма В. Оно определяется тремя величинами: числом просмотров A , числом обменов B и числом сравнений C . Если исходные ключи различны и расположены в случайном порядке, то можно предположить, что они образуют случайную перестановку множества $\{1, 2, \dots, n\}$. Понятие таблицы инверсий (п. 5.1.1) приводит к простому способу описания действия каждого просмотра при сортировке методом пузырька.

Теорема I. Пусть $a_1 a_2 \dots a_n$ — перестановка множества $\{1, 2, \dots, n\}$, $b_1 b_2 \dots b_n$ — соответствующая таблица инверсий. Если в результате очередного просмотра при сортировке методом пузырька (алгоритм В) перестановка $a_1 a_2 \dots a_n$ преобразуется в $a'_1 a'_2 \dots a'_n$, то соответствующая таблица инверсий $b'_1 b'_2 \dots b'_n$ получается из $b_1 b_2 \dots b_n$ уменьшением на единицу каждого нулевого элемента.

Доказательство. Если перед a_i имеется больший элемент, то a_i поменяется местами с наибольшим из предшествующих элементов, так что b_i уменьшится на единицу. С другой стороны, если перед a_i нет элемента, большего a_i , то a_i никогда не поменяется местами с большим элементом, так что b_{a_i} останется 0. ■

Итак, можно разобраться в том, что происходит в процессе сортировки методом пузырька, изучая последовательность таблиц инверсий между просмотрами. Вот как выглядят, например,

⁶ Здесь, как и ранее, двоеточие используется для обозначения оператора сравнения. —Прим. ред.

ритма Бэтчера завершается; этот алгоритм можно было бы назвать сортировкой посредством перегибов!

МХ-программа для алгоритма М приведена в упр. 12. К сожалению, количество вспомогательных операций, необходимых для управления последовательностью сравнений, весьма велико, так что программа менее эффективна, чем другие методы, которые мы разбирали. Однако алгоритм обладает одним важным компенсирующим качеством: все сравнения/обмены, определяемые данной итерацией шага МЗ, можно выполнять *одновременно* на ЭВМ или логических схемах, которые допускают параллельные

Picture: Рис. 18. Геометрическая интерпретация метода Бэтчера, $N = 16$.

вычисления. С такими параллельными операциями сортировка выполняется за $\frac{1}{2}[\log_2 N]([\log_2 N] + 1)$ шагов, и это один из самых быстрых известных общих методов. Например, *1024 элемента можно отсортировать методом Бэтчера всего за 55 параллельных шагов*. Его ближайшим соперником является метод Пратта (см. упр. 5.2.1–30), который затрачивает 40 или 73 шага в зависимости от того, как считать: если мы готовы допустить перекрытие сравнений до тех пор, пока не потребуются выполнять перекрывающиеся обмены, то для сортировки 1024 элементов методом Пратта требуется всего 40 циклов сравнения/обмена. Дальнейшие пояснения см. в п. 5.3.4.

”Быстрая сортировка”. В методе Бэтчера последовательность сравнений предопределена: каждый раз сравниваются одни и те же пары ключей независимо от того, что мы могли узнать о файле из предыдущих сравнений. Это утверждение в большой мере справедливо и применительно к методу пузырька, хотя алгоритм В и использует в ограниченной степени полученные сведения, с тем чтобы сократить количество работы в правом конце файла. Обратимся теперь к совсем иной стратегии, при которой используется результат каждого сравнения, чтобы определить, какие ключи сравнивать следующими. Такая стратегия не годится для параллельных вычислений, но она может оказаться плодотворной для вычислительных машин, работающих последовательно.

Итак, рассмотрим следующую схему сравнений/обменов. Имеются два указателя i и j , причем вначале $i = l$, а $j = N$. Сравним $K_i : K_j$, и если обмен не требуется, то уменьшим j на единицу и повторим этот процесс. После первого обмена увеличим i на единицу и будем продолжать сравнения, увеличивая i , пока не произойдет еще один обмен. Тогда опять уменьшим j и т. д.; будем “сжигать свечку с обоих концов”, пока не станет $i = j$. Посмотрим, например, что произойдет с нашим файлом из шестнадцати чисел:

Дано:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703	
																	уменьшить j
1-й обмен	154	087	512	061	908	170	897	275	653	426	503	509	612	677	765	703	
																	увеличить i
2-й обмен	154	087	503	061	908	170	897	275	653	426	512	509	612	677	765	703	
																	уменьшить j
3-й обмен	154	087	426	061	908	170	897	275	653	503	512	509	612	677	765	703	
																	увеличить i
4-й обмен	154	087	426	061	503	170	897	275	653	908	512	509	612	677	765	703	
																	уменьшить j
5-й обмен	154	087	426	061	275	170	897	503	653	908	512	509	612	677	765	703	
																	увеличить i
6-й обмен	154	087	426	061	275	170	503	897	653	808	512	509	612	677	765	703	
																	уменьшить j

(Чтобы выявить состояние i и j , ключи K_i и K_j напечатаны жирным шрифтом.) Заметим, что в каждом сравнении этого примера участвует ключ 503; вообще в каждом сравнении будет участвовать исходный ключ K_1 , потому что он будет продолжать обмениваться местами с другими ключами каждый раз, когда мы переключаем направление. К моменту, когда $i = j$, исходная запись R_1 займет свою окончательную позицию, потому что, как нетрудно видеть, слева от нее не будет больших ключей, а справа — меньших. Исходный файл окажется разделен таким образом, что первоначальная задача сведется к двум более простым: сортировке файла $R_1 \dots R_{i-1}$ и (независимо) сортировке файла $R_{i+1} \dots R_N$. К каждому из этих подфайлов можно применить тот же самый метод.

В табл. 2 показано, как выбранный нами для примеров файл полностью сортируется при помощи этого метода за 11 стадий. В скобки заключены подфайлы, которые еще предстоит отсортировать; в машине эти подфайлы можно представлять двумя переменными l и r (границы рассматриваемого в данный момент файла) и стеком дополнительных пар (l_k, r_k) . Каждый раз, когда файл подразделяется,

мы помещаем в стек *большой* из подфайлов и начинаем обрабатывать оставшийся, и так до тех пор, пока не придем к тривиально коротким файлам; как показано в упр. 20, такая процедура гарантирует, что в стеке никогда не будет находиться более, чем примерно $\log_2 N$ элементов.

для сортировки) быстрая сортировка превращается в отнюдь не быструю. Время ее работы становится пропорциональным N^2 , а не $N \log N$ (см. упр. 25). В отличие от других алгоритмов сортировки, которые нам встречались, алгоритм Q предпочитает неупорядоченные файлы!

В упомянутой статье Хоара предложены два способа поправить ситуацию, основывающихся на выборе лучшего значения проверяемого ключа K , который управляет разделением. Одна из его рекомендаций состоит в том, чтобы в последней части шага Q2 выбирать *случайное* целое число q между l и r ; в этом шаге можно заменить инструкции " $K \leftarrow K_l, R \leftarrow R_l$ " на

$$K \leftarrow K_q, \quad R \leftarrow R_q, \quad R_q \leftarrow R_l. \quad (27)$$

Согласно формулам (25), такие случайные целые числа придется вычислять в среднем лишь $2(N+1)/(M+2)-1$ раз, так что дополнительное время работы несущественно, а случайный выбор — хорошая защита от опасности оказаться в наихудшей ситуации.

Второе предложение Хоара состоит в том, чтобы просмотреть небольшой участок файла и найти медиану для этой совокупности данных. Такому подходу последовал Р. К. Синглтон [CASM, 12(1969), 185–187], который предложил в качестве K_q брать медиану трех значений

$$K_l, \quad K_{\lfloor (l+r)/2 \rfloor}, \quad K_r. \quad (28)$$

Процедура Синглтона сокращает число сравнений с $2N \ln N$ примерно до $\frac{12}{7} N \ln N$ (см. упр. 29). Можно показать, что, в этом случае B_N асимптотически приближается к $C_N/5$, а не к $C_N/6$, так что метод медианы несколько увеличивает время, затрачиваемое на пересылку данных, поэтому общее время работы сокращается примерно, на 8%. (Подробный анализ см. в упр. 56.) Время работы в наихудшем случае все еще порядка N^2 , однако с таким медленным поведением алгоритма вряд ли когда-либо придется встретиться на практике.

У. Д. Фрэйзер и А. Ч. Мак-Келлар [JACM, 17 (1970), 496–507] предложили рассматривать совокупность гораздо большего объема из $2^k - 1$ записей, где k выбирается так, чтобы $2^k \approx N/\ln N$. Эту совокупность можно отсортировать обычным методом быстрой сортировки, после чего элементы вставляются среди остальных записей за k просмотров всего файла (в результате файл будет разделен на 2^k подфайлов, ограниченных элементами первоначальной совокупности). На заключительном этапе сортируются полученные подфайлы. Среднее число, сравнений, выполняемых такой процедурой "сортировки совокупности", примерно такое же, как и для метода медианы Синглтона, когда N находится в практическом диапазоне значений, но при $N \rightarrow \infty$ оно асимптотически приближается к $N \log_2 N$.

Обменная поразрядная сортировка. Мы подходим теперь к методу, совершенно отличному от всех схем сортировки, которые рассматривались прежде; в нем используется *двоичное представление* ключей, и потому он предназначен исключительно для двоичных машин. Вместо того чтобы сравнивать между собой два ключа, в этом методе проверяется, равны ли 0 или 1 отдельные биты ключа. В других отношениях он обладает характеристиками обменной сортировки и на самом деле очень напоминает быструю сортировку. Так как он зависит от разрядов ключа, представленного в двоичной системе счисления, мы называем его "обменной поразрядной сортировкой". В общих чертах этот алгоритм можно описать следующим образом:

- 1) Последовательность сортируется *по старшему значащему двоичному биту* так, чтобы все ключи, начинающиеся с 0, оказались перед всеми ключами, начинающимися с 1. Для этого надо найти самый левый ключ K_i , начинающийся с 1, и самый правый ключ K_j , начинающийся с 0, после чего R_i и R_j меняются местами, и процесс повторяется, пока не станет $i > j$.
- 2) Пусть F_0 — множество элементов, начинающихся с 0, а F_1 — все остальные. Применим к F_0 поразрядную сортировку (начав теперь со *второго* бита слева, а не со старшего) до тех пор, пока множество F_0 не будет полностью отсортировано; затем сделаем то же с F_1 .

Например, в табл. 3 показано, как действует обменная поразрядная сортировка на наши 16 случайных чисел, записанных теперь в восьмеричной системе счисления. На стадии 1 показан исходный файл; после обменов по первому биту приходим ко второй стадии. На второй стадии сортируется первая группа по второму, биту, на третьей — по третьему биту. (Читатель должен мысленно преобразовать восьмеричные числа в 10-разрядные двоичные.) Когда мы после сортировки по четвертому биту достигаем пятой стадии, то обнаруживаем, что каждая из оставшихся групп содержит всего по одному элементу, — так что эту часть файла можно больше не рассматривать. Запись "⁴[0232 0252]" означает, что подфайл 0232 0252 еще предстоит сортировать по четвертому биту слева. В этом конкретном случае сортировка по четвертому биту не дает ничего нового; чтобы разделить элементы, необходимо добраться до пятого бита.

Весь процесс сортировки, показанный в табл. 3, выполняется за 22 стадии; это несколько больше соответствующего числа в быстрой сортировке (табл. 2). Число проверок битов 82 также велико; но мы увидим, что число проверок битов при больших 151

Picture: Таблица 3

N в действительности меньше, чем число сравнений в быстрой сортировке, в предположении о равномерном распределении ключей. Общее число обменов в табл. 3 равно 17, т. е. весьма умеренно. Заметим, что, хотя сортируются 10-битовые числа, в данном примере при проверке битов никогда не приходится идти дальше седьмого бита.

Как и при быстрой сортировке, для хранения "информации о границах" подфайлов, ожидающих сортировки, можно воспользоваться стеком. Вместо того чтобы сортировать в первую очередь наименьший из подфайлов, удобно просто продвигаться слева направо, так как размер стека в этом случае никогда не превзойдет числа битов в сортируемых ключах. В алгоритме, приведенном ниже, элемент стека (r, b) указывает на то, что подфайл с правой границей r ожидает сортировки по биту b ; левую границу можно не запоминать в стеке: она всегда задана неявно, поскольку в этой процедуре файл всегда обрабатывается слева направо.

Алгоритм R. (*Обменная поразрядная сортировка.*) Записи R_1, \dots, R_N переразмещаются на том же месте; после завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$. Предполагается, что все ключи — m -разрядные двоичные числа $(a_1 a_2 \dots a_m)_2$; i -й по старшинству бит a_i называется "бит i " ключа. Требуется вспомогательный стек, вмещающий до $m - 1$ элементов. Этот алгоритм, по существу, следует процедуре обменной поразрядной сортировки с разделениями, описанной выше; возможны некоторые усовершенствования с целью повышения эффективности (они описаны далее в тексте и в упражнениях).

- R1** [Начальная установка.] Опустошить стек и установить $l \leftarrow 1, r \leftarrow N, b \leftarrow 1$.
- R2** [Начать новую стадию.] (Мы хотели бы теперь отсортировать подфайл $R_l \leq \dots \leq R_r$ по биту b ; по смыслу алгоритма $l \leq r$.) Если $l = r$, то перейти к шагу **R10** (так как файл, состоящий из одного слова, уже отсортирован). В противном случае установить $i \leftarrow l, j \leftarrow r$.
- R3** [Проверить K_i на 1.] Проверить бит b ключа K_i . Если он равен 1, то перейти к шагу **R6**.
- R4** [Увеличить i .] Увеличить i на 1. Если $i \leq j$, то возвратиться к шагу **R3**; в противном случае перейти к шагу **R8**.
- R5** [Проверить K_{j+1} на 0.] Проверить бит b ключа K_{j+1} . Если он равен 0, то перейти к шагу **R7**.
- R6** [Уменьшить j .] Уменьшить j на 1. Если $i \leq j$, то перейти к шагу **R5**; в противном случае перейти к шагу **R8**.
- R7** [Поменять местами R_i, R_{j+1} .] Поменять местами $R_i \leftrightarrow R_{j+1}$, затем перейти к шагу **R4**.
- R8** [Проверить особые случаи.] (К этому моменту стадия разделения завершена, $i = j + 1$, бит b ключей K_l, \dots, K_j равен 0, а бит b ключей K_i, \dots, K_r равен 1.) Увеличить b на 1. Если $b > m$, где m — общее число битов в ключах, то перейти к шагу **R10**. (Это означает, что подфайл $R_l \dots R_r$ отсортирован. Если в файле не может быть равных ключей, то такую проверку можно не делать.) В противном случае, если $j < l$ или $j = r$, возвратиться к шагу **R2** (все просмотренные биты оказались равными соответственно 1 или 0). Если же $j = l$, то увеличить l на 1 и перейти к шагу **R2** (встретился только один бит, равный 0).
- R9** [Поместить в стек.] Поместить в стек элемент (r, b) , затем установить $r \leftarrow j$ и перейти к шагу **R2**.
- R10** [Взять из стека.] Если стек пуст, то сортировка закончена. В противном случае установить $l \leftarrow r + 1$, взять из стека элемент (r', b') , установить $r \leftarrow r', b \leftarrow b'$ и возвратиться к шагу **R2**. ■

Программа R. (*Обменная поразрядная сортировка.*) В следующей программе для машины MIX используются, по существу, те же соглашения, что и в программе Q. Значения регистров: $rI1 \equiv l - r$, $rI2 \equiv r$, $rI3 \equiv i$, $rI4 \equiv j$, $rI5 \equiv m - b$, $rI6$ = размер стека, если не считать того, что в некоторых командах (отмеченных ниже) удобно оставить $rI3 \equiv i - j$ или $rI4 \equiv j - i$. Двоичной природой поразрядной сортировки объясняется использование в этой программе команд SRB (двоичный сдвиг содержимого AX вправо), JAE (переход, если содержимое A четно) и JAO (переход, если содержимое A нечетно), определенных в п. 4.5.2. Предполагается, что $N \geq 2$.

START	ENT6	0	1	R1. Начальная установка. Опустошить стек.
	ENT1	1-N	1	$l \leftarrow 1$.
	ENT2	N	1	$r \leftarrow N$.
	ENT5	M-1	1	$b \leftarrow 1$.
	JMP	1F	1	K R2 (опустить проверку $l = r$).

9H	INC6	1	S	R9. Поместить в стек. $[rI4 = l - j]$
	ST2	STACK, 6 (A)	S	
	ST5	STACK, 6 (B)	S	$(r, b) \Rightarrow$ стек.
	ENN1	0, 4	S	$rI1 \leftarrow l - j$.
	ENT2	-1, 3	S	$r \leftarrow j$.
1H	ENT3	0, 1	A	R2. Начать новую стадию. $[rI3 = i - j]$
	ENT4	0, 2	A	$i \leftarrow l, j \leftarrow r$. $[rI3 = i - j]$
3H	INC3	0, 4	C'	R3. Проверить K_i на 1.
	LDA	INPUT, 3	C'	
	SRB	0, 5	C'	младший бит $rA \leftarrow$ бит b ключа K_i .
	JAE	4F	C'	К R4, если он равен 0.
6H	DEC4	1, 3	$C'' + X$	R6. Уменьшить $j \leftarrow j - l$. $[rI4 = j - i]$.
	J4N	8F	$C'' + X$	К R8, если $j < i$. $[rI4 = j - i]$
5H	INC4	0, 3	C''	R5. Проверить K_{j+1} на 0.
	LDA	INPUT+1, 4	C''	
	SRB	0, 5	C''	младший бит $rA \leftarrow$ бит b ключа K_{j+1} .
	JAO	6B	C''	К R6, если он равен 1.
7H	LDA	INPUT+1, 4	B	R7. Поменять местами R_i, R_{j+1} .
	LDX	INPUT, 3	B	
	STX	INPUT+1, 4	B	
	STA	INPUT, 3	B	
4H	DEC3	-1, 4	$C' - X$	R4. Увеличить i . $i \leftarrow i + 1$. $[rI3 = i - j]$
	J3NP	3B	$C' - X$	К R3, если $i \leq j$. $[rI3 = i - j]$
	INC3	0, 4	$A - X$	$rI3 \leftarrow i$.
8H	J5Z	0F	A	R8. Проверить особые случаи. $[rI4 = \text{неизвестен}]$
	DEC5	1	$A - G$	К R10, если $b = m$, иначе $b \leftarrow b - 1$.
	ENT4	-1, 3	$A - G$	$rI4 \leftarrow j$.
	DEC4	0, 2	$A - G$	$rI4 \leftarrow j - r$.
	J4Z	1B	$A - G$	К R2, если $j = r$.
	DEC4	0, 1	$A - G - R$	$rI4 \leftarrow j - l$.
	J4N	1B	$A - G - R$	К R2, если $j < l$.
	J4NZ	9B	$A - G - L - R$	К R9, если $j \neq l$.
	INC1	1	K	$l \leftarrow l + 1$.
2H	J1NZ	1B	$K + S$	Переход, если $l \neq r$.
0H	ENT1	1, 2	$S + 1$	R10. Взять из стека.
	LD2	STACK, 6 (A)	$S + 1$	
	DEC1	0, 2	$S + 1$	
	LD5	STACK, 6 (B)	$S + 1$	Стек $\Rightarrow (r, b)$
	DEC6	1	$S + 1$	
	J6NN	2B	$S + 1$	К R2, если стек не был пуст.

Время работы программы обменной поразрядной сортировки зависит от

$$\begin{aligned}
 A &= \text{число стадий, в которых } l < r; \\
 B &= \text{число замещений}; \\
 C &= C' + C'' = \text{число проверок битов}; \\
 G &= \text{число случаев, когда в шаге R8 } b > m; \\
 K &= \text{число случаев, когда в шаге R8 } b \leq m, j = l; \\
 L &= \text{число случаев, когда в шаге R8 } b \leq m, j < l; \\
 R &= \text{число случаев, когда в шаге R8 } b \leq m, j = r; \\
 S &= \text{число случаев, когда что-либо помещается в стек}; \\
 X &= \text{число случаев, когда в шаге R6 } j < i.
 \end{aligned} \tag{29}$$

По закону Кирхгофа $S = A - G - K - L - R$, так что общее время выполнения равно $27A + 8B + 8C - 23G - 14K - 17L - 19R - X + 13$ единиц. За счет усложнения программы можно несколько ускорить циклы проверки битов (см. упр. 34). Обменную поразрядную сортировку можно также ускорить, если при достаточно малых значениях разности $r - l$ применять простые вставки, как это сделано в алгоритме Q, но мы не будем задерживаться на таких усовершенствованиях.

Для анализа времени выполнения обменной поразрядной сортировки напрашиваются два типа исходных данных:

- i) Пусть $N = 2^m$ и ключи, которые надо отсортировать, — просто числа $0, 1, 2, \dots, 2^m - 1$, расположенные в случайном порядке.
- ii) Пусть $m = \infty$ (неограниченная точность) и ключи, которые надо отсортировать, — независимые, равномерно распределенные в промежутке $[0, 1)$ действительные числа.

Анализ случая (i) относительно прост, поэтому он оставлен читателю в качестве упражнения (см. упр. 35). Случай (ii) сравнительно сложен, поэтому он *также* оставлен для упражнений. В следующей таблице приведены грубые оценки результатов того и другого анализов:

Величина	Случай (i)	Случай (ii)
A	N	αN
B	$\frac{1}{4}N \log_2 N$	$\frac{1}{4}N \log_2 N$
C	$N \log_2 N$	$N \log_2 N$
G	$\frac{1}{2}N$	0
K	0	$\frac{1}{2}N$
L	0	$\frac{1}{2}(\alpha - 1)N$
R	0	$\frac{1}{2}(\alpha - 1)N$
S	$\frac{1}{2}N$	$\frac{1}{2}N$
X	$\frac{1}{2}N$	$\frac{1}{4}(\alpha - 1)N$

(30)

Здесь $\alpha = 1/(\ln 2) = 1.4427$. Заметим, что среднее число обменов, проверок битов и обращений к стеку, по существу, одинаково в обоих случаях, несмотря даже на то, что в случае (ii) число стадий на 44% больше. На сортировку N элементов в случае (ii) наша MIX-программа затратит в среднем приблизительно $14.4N \ln N$ единиц времени. Это число можно было бы сократить примерно до $11.5N \ln N$, если воспользоваться предложением из упр. 34. Соответствующая величина для программы Q равна $12.7N \ln N$, но и ее можно уменьшить до $11.7N \ln N$, если воспользоваться предложением Синглтона и выбирать медиану из трех ключей.

В случае равномерного распределения данных обменная поразрядная сортировка занимает в среднем примерно столько же времени, сколько и быстрая сортировка; фактически для машины MIX она немного быстрее, чем быстрая сортировка. В упр. 53 показано, в какой мере замедляется этот процесс при неравномерном распределении. Важно отметить, что весь наш анализ основан на предположении о том, что все ключи различны. *Обменная поразрядная сортировка в том виде, как она описана выше, не очень эффективна, если имеются одинаковые ключи*, так как она проходит через несколько стадий, требующих времени, пытаясь разделить множества одинаковых ключей, пока b не станет $> m$. Один приемлемый способ исправить этот недостаток предложен в ответе к упр. 40.

Как обменная поразрядная сортировка, так и быстрая сортировка основаны на идее разделения. Записи меняются местами до тех пор, пока файл не будет разбит на две части: левый подфайл, в котором все ключи $\leq K$ при некотором K , и правый подфайл, в котором все ключи $\geq K$. В быстрой сортировке в качестве K выбирается реальный ключ из файла, в то время как в обменной поразрядной сортировке, по существу, выбирается некоторый искусственный ключ на основе двоичных представлений. Что касается исторической стороны дела, то обменную поразрядную сортировку открыли П. Хильдебрандт, Г. Исбитц, Х. Райзинг и Ж. Шварц [JACM, 6 (1959), 156–163] примерно за год до изобретения быстрой сортировки. Возможны также и другие схемы разделения; например, Джон Маккарти предложил выбирать $K \approx \frac{1}{2}(u + v)$, если известно, что все ключи лежат в диапазоне между u и v .

Еще одну стратегию разделения предложил М. Х. ван Эмден [CACM, 13 (1970), 563–567]: вместо того чтобы выбирать K заранее, мы "узнаем", каково может быть хорошее значение K , следя в процессе разделения за изменением величин $K' = \max(K_l, \dots, K_i)$ и $K'' = \min(K_j, \dots, K_r)$. Можно увеличивать i до тех пор, пока не встретится ключ, больший K' ; затем начать уменьшать j , пока не встретится ключ, меньший K'' , после чего поменять их местами и/или уточнить значения K' и K'' . Эмпирические тесты для этой интервальной сортировки показывают, что она требует около $1.64N \ln N = 1.14N \log_2 N$ сравнений. Это единственный метод, обсуждаемый в этой книге, для поведения которого еще не найдено адекватного теоретического объяснения.

Обобщение обменной поразрядной сортировки на случай системы счисления с основанием, большим 2, обсуждается в п. 5.2.5.

* **Асимптотические методы.** Анализ алгоритмов обменной сортировки приводит к некоторым особенно поучительным математическим задачам, которые позволяют больше узнать о способах определения асимптотического поведения функции. Например, при анализе метода пузырька [формула

(9)] мы столкнулись с функцией

$$W_n = \frac{1}{n!} \sum_{0 \leq r < s \leq n} s! r^{n-s}. \quad (31)$$

Каково ее асимптотическое значение?

Можно действовать так же, как при исследовании числа инволюций [формула (5.1.4–41)]. Поэтому, прежде чем двигаться дальше, полезно еще раз просмотреть обсуждение в конце п. 5.1.4.

Исследование формулы (31) показывает, что вклад при $s = n$ больше, чем вклад при $s = n - 1$, и т. д.; это наводит на мысль о замене s на $n - s$. Мы скоро увидим, что на самом деле удобнее всего применить подстановки $t = n - s + 1$, $m = n + 1$, в результате которых формула (31) примет вид

$$\frac{1}{m} W_{m-1} = \frac{1}{m!} \sum_{1 \leq t < m} (m-t)! \sum_{0 \leq r < m-t} r^{t-1}. \quad (32)$$

Для внутренней суммы существует хорошо известный асимптотический ряд, полученный из формулы суммирования Эйлера:

$$\begin{aligned} \sum_{0 \leq r < N} r^{t-1} &= \frac{N^t}{t} - \frac{1}{2}(N^{t-1} - \delta_{t1}) + \frac{B_2}{2!}(t-1)(N^{t-2} - \delta_{t2}) + \dots = \\ &= \frac{1}{t} \sum_{0 \leq j \leq k} \binom{t}{j} B_j (N^{t-j} - \delta_{tj}) + O(N^{t-k}) \end{aligned} \quad (33)$$

(см. упр. 1.2.11.2–4). Следовательно, наша задача сводится к изучению сумм вида

$$\frac{1}{m!} \sum_{1 \leq t < m} (m-t)!(m-t)^t t^k, \quad k \geq -1. \quad (34)$$

Как и в п. 5.1.4, можно показать, что при значениях t , больших $m^{1/2+\varepsilon}$, члены суммы пренебрежимо малы: $O(\exp(-n^\delta))$. Значит, можно положить $t = O(m^{1/2+\varepsilon})$ и заменить факториалы по формуле Стирлинга

$$\frac{(m-t)!(m-t)^t}{m!} = \sqrt{1 - \frac{t}{m}} \exp \left(\frac{t}{12m^2} - \left(\frac{t^2}{2m} + \frac{t^3}{3m^2} + \frac{t^4}{4m^3} + \frac{t^5}{5m^4} \right) + O(m^{-2+6\varepsilon}) \right).$$

Таким образом, нас интересует асимптотическое значение суммы

$$r_k(m) = \sum_{1 \leq t < m} e^{-t^2/2m} t^k, \quad k \geq -1. \quad (35)$$

(Суммирование здесь также можно было бы распространить на весь диапазон $1 \leq t < \infty$, не изменив при этом асимптотического значения суммы, поскольку, как указано выше, входящие в сумму значения при $t > m^{1/2+\varepsilon}$ пренебрежимо малы.)

Пусть $g_k(x) = x^k e^{-x^2}$ и $f_k(x) = g_k(x/\sqrt{2m})$. По формуле суммирования Эйлера при $k \geq 0$

$$\begin{aligned} \sum_{0 \leq t < m} f_k(t) &= \int_0^m f_k(x) dx + \sum_{1 \leq j \leq p} \frac{B_j}{j!} (f_k^{(j-1)}(m) - f_k^{(j-1)}(0)) + R_p, \\ R_p &= \frac{(-1)^{p+1}}{p!} \int_0^m B_p(\{x\}) f_k^{(p)}(x) dx = \\ &= \left(\frac{1}{\sqrt{2m}} \right)^p O \left(\int_0^\infty |g_k^{(p)}(y)| dy \right) = O(m^{-p/2}); \end{aligned} \quad (36)$$

следовательно, при помощи, по существу, тех же приемов, что и раньше, можно получить асимптотический ряд для $r_k(m)$, если только $k \geq 0$. Но при $k = -1$ этот метод не годится, так как значение $f_{-1}(0)$ не определено; нельзя также просто просуммировать от 1 до m , так как остаточные члены не дают убывающих степеней m , если нижний предел равен 1. (Именно в этом суть дела, и, прежде чем двигаться дальше, читатель должен убедиться в том, что он хорошо понял задачу.)

Чтобы разрешить эту дилемму, можно положить по определению $g_{-1}(x) = (e^{-x^2} - 1)/x$, $f_{-1}(x) = g_{-1}(x/\sqrt{2m})$; тогда $f_{-1}(0) = 0$ и $r_{-1}(m)$ нетрудно получить из $\sum_{0 \leq t < m} f_{-1}(t)$. Уравнение (36) справедливо теперь и при $k = -1$, а оставшийся интеграл нам "хорошо знаком" (см. упр. 43):

$$\begin{aligned} \frac{2}{\sqrt{2m}} \int_0^m f_{-1}(x) dx &= 2 \int_0^m \frac{e^{-x^2/2m} - 1}{x} dx = \int_0^{m/2} \frac{e^{-y} - 1}{y} dy = \\ &= \int_0^1 \frac{e^{-y} - 1}{y} dy + \int_1^{m/2} \frac{e^{-y}}{y} dy - \ln(m/2) = \\ &= -\gamma - \ln(m/2) + O(e^{-m/2}). \end{aligned}$$

Теперь у нас достаточно формул и фактов для того, чтобы вывести наконец ответ, который равен, как показано в ответе к упр 44,

$$W_n = \frac{1}{2} m \ln m + \frac{1}{2}(\gamma + \ln 2)m - \frac{2}{3}\sqrt{2\pi m} + \frac{31}{36} + O(n^{-1/2}), \quad m = n + 1. \quad (37)$$

Этим завершается анализ метода пузырька.

Чтобы проанализировать обменную поразрядную сортировку, необходимо знать асимптотическое поведение при $n \rightarrow \infty$ конечной суммы

$$U_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{1}{2^{k-1} - 1}. \quad (38)$$

Этот вопрос оказывается гораздо сложнее других задач об асимптотическом поведении, с которыми мы сталкивались до сих пор; элементарные методы разложения в степенные ряды, формула суммирования Эйлера и т. д. здесь бессильны. Следующий вывод был предложен Н. Г. де Брейном.

Чтобы избавиться в формуле (38) от подавляющего влияния больших множителей $\binom{n}{k}(-1)^k$, мы начнем с того, что перепишем сумму в виде бесконечного ряда:

$$U_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \sum_{j \geq 1} \left(\frac{1}{2^k - 1} \right)^j = \sum_{j \geq 1} (2^j (1 - 2^{-j})^n - 2^j + n). \quad (39)$$

Если положить $x = n/2^j$, то член ряда запишется в виде

$$2^j (1 - 2^{-j})^n - 2^j + n = \frac{n}{x} \left(\left(1 - \frac{x}{n} \right)^n - 1 + x \right).$$

При $x \leq n^\varepsilon$ имеем

$$\left(1 - \frac{x}{n} \right)^n = \exp \left(n \ln \left(1 - \frac{x}{n} \right) \right) = \exp(-x + x^2 O(n^{-1})), \quad (40)$$

а это наводит на мысль о том, чтобы приблизить (39) рядом

$$T_n = \sum_{j \geq 1} (2^j e^{-n/2^j} - 2^j + n). \quad (41)$$

Чтобы оправдать это приближение, рассмотрим разность $U_n - T_n = X_n + Y_n$, где

$$\begin{aligned} X_n &= \sum_{\substack{j \geq 1 \\ 2^j < n^{1-\varepsilon}}} (2^j (1 - 2^{-j})^n - 2^j e^{-n/2^j}) = \quad [\text{т.е. слагаемые при } x > n^\varepsilon] \\ &= \sum_{\substack{j \geq 1 \\ 2^j < n^{1-\varepsilon}}} O(ne^{-n/2^j}) = \quad [\text{так как } 0 < 1 - 2^{-j} < e^{-2^{-j}}] \\ &= O(n \log ne^{-n^\varepsilon}) \quad [\text{так как имеется } O(\log n) \text{ слагаемых}]; \\ Y_n &= \sum_{\substack{j \geq 1 \\ 2^j \geq n^{1-\varepsilon}}} (2^j (1 - 2^{-j})^n - 2^j e^{-n/2^j}) = \quad [\text{слагаемые при } x \leq n^\varepsilon] \\ &= \sum_{\substack{j \geq 1 \\ 2^j \geq n^{1-\varepsilon}}} \left(e^{-n/2^j} \left(\frac{n}{2^j} \right) O(1) \right) \quad [\text{по формуле (40)}] \end{aligned}$$

Приведенное ниже рассуждение покажет, что эта последняя сумма есть $O(1)$; следовательно, $U_n - T_n = O(1)$. (См. упр. 47.)

До сих пор мы еще не использовали никаких методов, которые бы действительно отличались от применявшихся ранее, но

Picture: Рис. 20. Контур интегрирования для тождеств с гамма-функциями.

для изучения ряда T_n потребуются новая идея, основанная на простых принципах теории функций комплексного переменного. Если x — произвольное положительное число, то

$$e^{-x} = \frac{1}{2\pi i} \int_{1/2-i\infty}^{1/2+i\infty} \Gamma(z) x^{-z} dz = \frac{1}{2\pi} \int_{-\infty}^{\infty} \Gamma\left(\frac{1}{2} + it\right) x^{-(1/2+it)} dt. \quad (42)$$

Для доказательства этого тождества рассмотрим путь интегрирования, показанный на рис. 20(a), где N , N' и M велики. Значение интеграла вдоль этого контура равно сумме вычетов внутри контура, а именно

$$\sum_{0 \leq k < M} x^{-k} \lim_{z \rightarrow -k} (z+k) \Gamma(z) = \sum_{0 \leq k < M} x^{-k} \frac{(-1)^k}{k!}.$$

Интеграл по верхней части контура есть $O\left(\int_{-\infty}^{1/2} |\Gamma(t+iN)| x^{-t} dt\right)$, и у нас имеется известная оценка

$$\Gamma(t+iN) = O(N^{t-1/2} e^{-\pi N/2}) \quad \text{при } N \rightarrow \infty.$$

[Свойства гамма-функций см., например, в книге А. Erdélyi et al., Higher Transcendental Functions, 1 (New York: McGraw-Hill, 1953), гл. 1.] Поэтому интеграл по верхнему отрезку бесконечно мал: $O\left(e^{-\pi N/2} \int_{-\infty}^{1/2} (N/x)^t dt\right)$. Интеграл по нижнему отрезку ведет себя столь же безобидно. Для вычисления интеграла по левому отрезку контура воспользуемся тем фактом, что

$$\begin{aligned} \Gamma\left(\frac{1}{2} + it - M\right) &= \Gamma\left(\frac{1}{2} + it\right) / \left(-M + \frac{1}{2} + it\right) \dots \left(-1 + \frac{1}{2} + it\right) = \\ &= \Gamma\left(\frac{1}{2} + it\right) O(1/(M-1)!). \end{aligned}$$

Следовательно, интеграл по левой стороне есть $O(x^{M-1/2}/(M-1)! \times \int_{-\infty}^{\infty} |\Gamma(\frac{1}{2} + it)| dt)$. Поэтому при $M, N, N' \rightarrow \infty$ уцелеет лишь интеграл по правой стороне; тем самым доказано тождество (42). В действительности тождество (42) остается в силе и в том случае, если заменить $\frac{1}{2}$ любым положительным числом.

Тем же самым рассуждением можно воспользоваться для вывода других полезных соотношений, содержащих гамма-функции. Величину x^{-z} можно заменить другими функциями от z ; можно также заменить другой величиной константу $\frac{1}{2}$. Например,

$$\frac{1}{2\pi i} \int_{-3/2-i\infty}^{-3/2+i\infty} \Gamma(z) x^{-z} dz = e^{-x} - 1 + x, \quad (43)$$

а это — критическая величина в формуле (41) для T_n :

$$T_n = n \sum_{j \geq 1} \frac{1}{2\pi i} \int_{-3/2-i\infty}^{-3/2+i\infty} \Gamma(z) (n/2^j)^{-1-z} dz. \quad (44)$$

Суммирование можно внести под знак интеграла, так как сходимость здесь достаточно хорошая; имеем

$$\sum_{j \leq 1} (n/2^j)^w = n^w \sum_{j \leq 1} (1/2^w)^j = n^w / (2^w - 1), \quad \text{если } \Re(w) > 0,$$

так как $|2^w| = 2^{\Re(w)} > 1$. Поэтому

$$T_n = \frac{n}{2\pi i} \int_{-3/2-i\infty}^{-3/2+i\infty} \frac{\Gamma(z) n^{-1-z}}{2^{-1-z} - 1} dz, \quad (45)$$

и остается оценить последний интеграл.

На этот раз интегрирование производится по контуру, который больше вытянут вправо, как изображено на рис. 20(b). Интеграл по верхнему отрезку есть $O\left(n^{1/2}e^{-\pi M/2} \int_{-3/2}^M N^t dt\right)$, если $2^{iN} \neq 1$, а интеграл по нижнему отрезку также пренебрежимо мал. Интеграл по правому отрезку равен $O\left(n^{-1-M} \int_{-\infty}^{\infty} |\Gamma(M+it)| dt\right)$. Фиксируя M и устремляя N, N' к ∞ , можно показать, что $-T_n/n$ есть $O(n^{-1-M})$ плюс сумма вычетов в области $-3/2 < \Re(z) < M$. Пусть $M \rightarrow \infty$, $\Gamma(z)$ имеет простые полюсы при $z = -1$ и $z = 0$, n^{-1-z} не имеет полюсов, а $1/(2^{-1-z}-1)$ имеет простые полюсы при $z = -1 + 2\pi ik/\ln 2$.

Наибольшую трудность представляет двойной полюс в точке $z = -1$. Если $w = z + 1$ мало, то можно воспользоваться известным соотношением

$$\Gamma(z+1) = \exp(-\gamma z + \zeta(2)z^2/2 - \zeta(3)z^3/3 + \zeta(4)z^4/4 - \dots),$$

где $\zeta(s) = 1^{-s} + 2^{-s} + 3^{-s} + \dots = H_{\infty}^{(s)}$, для вывода следующих разложений:

$$\begin{aligned}\Gamma(z) &= \frac{\Gamma(w+1)}{w(w-1)} = -w^{-1} + (\gamma-1) + O(w); \\ n^{-1-z} &= 1 - (\ln n)w + O(w^2); \\ 1/(2^{-1-z}-1) &= -w^{-1}/\ln 2 - \frac{1}{2} + O(w).\end{aligned}$$

Вычет в точке $z = -1$ равен коэффициенту при w^{-1} в произведении этих трех формул, а именно $\frac{1}{2} - (\ln n + \gamma - 1)/\ln 2$. Прибавляя остальные вычеты, получаем формулу

$$T_n/n = \frac{\ln n + \gamma - 1}{\ln 2} - \frac{1}{2} + f(n) + \frac{2}{n}, \quad (46)$$

где $f(n)$ —функция довольно необычного вида:

$$f(n) = \frac{2}{\ln 2} \sum_{k \geq 1} \Re(\Gamma(-1 - 2\pi ik/\ln 2) \exp(2\pi ik \log_2 n)). \quad (47)$$

Заметим, что $f(n) = f(2n)$. Среднее значение $f(n)$ равно 0, так как среднее значение каждого слагаемого равно 0. (Можно считать, что величина $(\log_2 n) \bmod 1$ распределена равномерно, принимая во внимание результаты о числах с плавающей точкой, полученные в п. 4.2.4.) Кроме того, поскольку $|\Gamma(-1+it)| = |\pi/(t(1+t^2) \sinh \pi t)|^{1/2}$, нетрудно показать, что

$$f(n) < 0.0000001725;$$

таким образом, в практических приложениях $f(n)$ можно спокойно отбросить. Что касается теории, то без $f(n)$ мы не можем получить асимптотического ряда для U_n ; именно поэтому величина U_n сравнительно трудна для анализа. Итак, мы доказали, что

$$U_n = n \log_2 n + n \left(\frac{\gamma-1}{\ln 2} - \frac{1}{2} + f(n) \right) + O(1). \quad (48)$$

Другие примеры применения этого метода гамма-функций можно найти в упр. 51–53 и в § 6.3.

Упражнения

- [M20] Пусть $a_1 \dots a_n$ —перестановка множества $\{1, \dots, n\}$, и пусть i и j таковы, что $i < j$ и $a_i > a_j$. Пусть $a'_1 \dots a'_n$ —перестановка, которая получается из $a_1 \dots a_n$, если поменять местами a_i и a_j . Может ли в $a'_1 \dots a'_n$ быть больше инверсий, чем в $a_1 \dots a_n$?
- [M25] (a) Каково минимальное число обменов, необходимое для того, чтобы отсортировать перестановку 3 7 6 9 8 1 4 5? (b) Вообще пусть дана перестановка $\pi = a_1 \dots a_n$ множества $\{1, \dots, n\}$, и пусть $\text{xch}(\pi)$ —минимальное число обменов, в результате которых перестановка π будет отсортирована в возрастающем порядке. Выразите $\text{xch}(\pi)$, через "более простые" характеристики перестановки π . (Ср. с упр. 5 2.1–39.)
- [10] Является ли устойчивой сортировка методом пузырька (алгоритм В)?
- [M23] Если в шаге В4 получится $t = 1$, то на самом деле работу алгоритма В можно сразу же заканчивать, потому что в следующем шаге В2 не выполнится никаких полезных действий. Какова вероятность того, что при сортировке случайной перестановки в шаге В4 окажется $t = 1$?

5. [M25] Пусть $b_1 b_2 \dots b_n$ —таблица инверсий перестановки $a_1 a_2 \dots a_n$. Покажите, что после r просмотров сортировки методом пузырька значение переменной BOUND равно $\max\{b_i + i \mid b_i \geq r\} - r$, где $0 \leq r \leq \max(b_1, \dots, b_n)$.
6. [M22] Пусть $a_1 \dots a_n$ —перестановка множества $\{1, \dots, n\}$, и пусть $a'_1 \dots a'_n$ —обратная к ней перестановка. Покажите, что число просмотров, необходимых для того, чтобы отсортировать $a_1 \dots a_n$ методом пузырька, равно $1 + \max(a'_1 - 1, a'_2 - 2, \dots, a'_n - n)$.
7. [M28] Вычислите стандартное отклонение числа просмотров сортировки методом пузырька и выразите его через n и функцию $P(n)$. [Ср. с формулами (6) и (7).]
8. [M24] Выведите формулу (8).
9. [M48] Проанализируйте число просмотров и число сравнений в алгоритме шейкер-сортировки. (Замечание: частичная информация содержится в упр. 5.4.8–9.)
10. [M26] Пусть $a_1 a_2 \dots a_n$ —2-упорядоченная перестановка множества $\{1, 2, \dots, n\}$. (а) Каковы координаты конечных точек a_i -го шага соответствующего решеточного пути (ср. с рис. 11)? (б) Докажите, что сравнение/обмен элементов $a_1: a_2, a_3, a_4, \dots$ соответствует перегибанию пути относительно диагонали, как на рис. 18(б). (с) Докажите, что сравнение/обмен элементов $a_2: a_{2+d}, a_4: a_{4+d}, \dots$ соответствует перегибанию пути относительно линии, расположенной на m единиц ниже диагонали, как на рис. 18(с), (д) и (е), если $d = 2m - 1$.
- >11. [M25] На какой перестановке множества $\{1, 2, \dots, 16\}$ достигается максимум числа обменов в алгоритме Бэтчера?
12. [24] Напишите MIX-программу для алгоритма M, предполагая, что MIX—двоичная вычислительная машина с операциями AND и SRB. Сколько времени потребуется вашей программе, чтобы отсортировать шестнадцать записей из табл. 1?
13. [10] Устойчива ли сортировка Бэтчера?
14. [M21] Пусть $c(N)$ —число сравнений ключей, производимых при сортировке N элементов методом Бэтчера; это равно числу выполнению шага M4. (а) Покажите, что при $t \geq 1$ $c(2^t) = 2c(2^{t-1}) + (t - 1)2^{t-1} + 1$. (б) Найдите простое выражение для $c(2^t)$ как функцию от t . (Указание: рассмотрите последовательность $x_t = c(2^t)/2^t$).
15. [M38] Содержание этого упражнения—анализ функции $c(N)$ из упр. 14 и нахождение формулы для $c(N)$, если $N = 2^{e_1} + 2^{e_2} + \dots + 2^{e_r}$, $e_1 > e_2 > \dots > e_r \geq 0$. (а) Пусть $a(N) = c(N + 1) - c(N)$. Докажите, что $a(2n) = a(n) + \lfloor \log_2(2n) \rfloor$, $a(2n + 1) = a(n) + 1$; откуда

$$a(N) = \binom{e_1 + 1}{2} - r(e_1 - 1) + (e_1 + e_2 + \dots + e_r).$$
 (б) Пусть $x(n) = a(n) - a(\lfloor n/2 \rfloor)$, и тогда $a(n) = x(n) + x(\lfloor n/2 \rfloor) + x(\lfloor n/4 \rfloor) + \dots$. Пусть $y(n) = x(1) + x(2) + \dots + x(n)$, и пусть $z(2n) = y(2n) - a(n)$, $z(2n + 1) = y(2n + 1)$. Докажите, что $c(N + 1) = z(N) + 2z(\lfloor N/2 \rfloor) + 4z(\lfloor N/4 \rfloor) + \dots$. (с) Докажите, что $y(N) = N + (\lfloor N/2 \rfloor + 1) \times (e_1 - 1) - 2^{e_1} + 2$. (д) Теперь соберите все вместе и найдите выражение $c(N)$ через показатели e_j при фиксированном значении r .
16. [BM46] Найдите асимптотическое значение среднего числа обменов в случае, когда алгоритм Бэтчера применяется к $N = 2^t$ различным элементам, расположенным в случайном порядке.
- >17. [20] Где в алгоритме Q используется то, что K_0 и K_{N+1} обладают значениями, постулированными неравенствами (13)?
- >18. [20] Объясните, как работает алгоритм Q в случае, когда все ключи в исходном файле равны. Что произойдет, если в шагах Q3 и Q5 заменить знаки "<" на "<="?
19. [15] Будет ли алгоритм Q по-прежнему работать правильно, если вместо стека (последним включается—первым исключается) воспользоваться очередью (первым включается—первым исключается)?
20. [M20] Выразите наибольшее число элементов, которые могут одновременно оказаться в стеке во время работы алгоритма Q, в виде функции от M и N.
21. [20] Объясните, почему фаза разделения в алгоритме Q требует как раз такого числа сравнений, пересылок и т. д., как это описано в (17).
22. [M25] Пусть $p_k N$ —вероятность того, что величина A в (14) будет равна k, если алгоритм Q применяется к случайной перестановке множества $\{1, 2, \dots, N\}$, и пусть $A_N(z) = \sum_k p_k N z^k$ —соответствующая производящая функция. Докажите, что $A_N(z) = 1$ при $N \leq M$, и $A_N(z) = z(\sum_{1 \leq s \leq N} A_{s-1}(z) A_{N-s}(z))/N$ при $N > M$. Найдите аналогичные рекуррентные соотношения, определяющие другие распределения вероятностей $B_N(z)$, $C_N(z)$, $D_N(z)$, $E_N(z)$, $L_N(z)$, $X_N(z)$.
23. [M24] Пусть A_N , B_N , D_N , E_N , L_N , X_N —средние значения соответствующих величин в (14) при сортировке случайной перестановки-множества $\{1, 2, \dots, N\}$. Найдите для этих величин рекуррентные соотношения, аналогичные (18), затем разрешите эти соотношения и получите формулы (25).

24. [M21] Очевидно, в алгоритме Q производится несколько больше сравнений, чем это необходимо, потому что в шагах $Q3$ и $Q5$ может оказаться, что $i = j$ или даже $i > j$. Сколько сравнений G_N производилось бы в среднем, если бы исключались все сравнения при $i \geq j$?
25. [M20] Чему равны точные значения величин A, B, C, D, E, L, X для программы Q в случае, когда исходные ключи представляют собой упорядоченный набор чисел $1, 2, \dots, N$ в предположении, что $N > M$?
- >26. [M2I] Постройте исходный файл, при котором программа Q работала бы еще медленнее, чем в упр. 25. (Попытайтесь найти действительно плохой случай.)

5.2.3. Сортировка посредством выбора

Еще одно важное семейство методов сортировки основано на идее многократного выбора. Вероятно, простейшая сортировка посредством выбора сводится к следующему:

- i) Найти наименьший ключ; переслать соответствующую запись в область вывода и заменить ключ значением " ∞ " (которое по предположению больше любого реального ключа).
- ii) Повторить шаг (i). На этот раз будет выбран ключ, наименьший из оставшихся, так как ранее наименьший ключ был заменен на ∞ .
- iii) Повторять шаг (i) до тех пор, пока не будут выбраны N записей.

Заметим, что этот метод требует наличия всех исходных элементов до начала сортировки, а элементы вывода он порождает последовательно, один за другим. Картина, по существу, противоположна методу вставок, в котором исходные элементы должны поступать последовательно, но вплоть до завершения сортировки ничего не известно об окончательном выводе.

Ряд вычислительных машин (например, машины с циклической барабанной памятью) имеет встроенную команду "найти наименьший элемент", которая выполняется с большой скоростью. На таких машинах сортировка указанным методом особенно привлекательна, если только N не слишком велико.

Описанный метод требует $N - 1$ сравнений каждый раз, когда выбирается очередная запись; он также требует отдельной области вывода в памяти. Имеется очевидный способ несколько поправить ситуацию, избежав при этом использования ∞ : выбранное значение можно записывать в соответствующую позицию, а запись, которая ее занимала, переносить на место выбранной. Тогда эту позицию не нужно рассматривать вновь при последующих выборах. На этой идее основан наш первый алгоритм сортировки посредством выбора.

Алгоритм S. (*Сортировка посредством простого выбора.*) Записи R_1, \dots, R_N переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$. Сортировка основана на описанном выше методе, если не считать того, что более, удобно, оказывается, выбирать сначала *наибольший* элемент, затем второй по величине и т. д.

S1 [Цикл по j .] Выполнить шаги S2 и S3 при $j = N, N - 1, \dots, 2$. ■

Доказательство. Если произведено менее $n - 1$ сравнений, то найдутся по крайней мере два элемента, для которых не было обнаружено ни одного элемента, превосходящего их по величине. Следовательно, мы так и не узнаем, который из этих двух элементов больше, и, значит, не сможем определить максимум. ■

Таким образом, процесс выбора, в котором отыскивается наибольший элемент, должен состоять не менее чем из $n - 1$ шагов. Означает ли это, что для всех методов сортировки, основанных на n повторных выборах, число шагов неизбежно будет порядка n^2 ? К счастью, лемма М применима только к *первому* шагу выбора; при последующих выборах можно использовать извлеченную ранее информацию. Например, в упр. 8 показано, что сравнительно простое изменение алгоритма S наполовину сокращает число сравнений.

Рассмотрим 16 чисел, представленных в 1-й строке в табл. 1. Один из способов сэкономить время при последующих выборах—разбить все числа на четыре группы по четыре числа. Начать можно с определения наибольшего элемента каждой группы, а именно соответственно с ключей

512, 908, 653, 765;

тогда наибольший из этих четырех элементов 908 и будет наибольшим во всем файле. Чтобы получить второй по величине элемент, достаточно просмотреть сначала остальные три элемента группы, содержащей 908; наибольший из {170, 897, 275} равен 897, и тогда наибольший среди

512, 897, 653, 765

это 897. Аналогично, для того чтобы получить третий по величине элемент, определяем наибольший из {170, 275}, а затем наибольший из

512, 275, 653, 765

и т. д. Каждый выбор, кроме первого, требует не более 6 дополнительных сравнений. Вообще, если N —точный квадрат, то можно разделить файл на \sqrt{N} групп по \sqrt{N} элементов в каждой; любой выбор, кроме первого, требует не более чем $\sqrt{N} - 1$ сравнений внутри группы ранее выбранного элемента плюс $\sqrt{N} - 1$ сравнений среди "лидеров групп". Этот метод получил название "квадратичный выбор"; общее время работы для него есть $O(N\sqrt{N})$, что существенно лучше, чем $O(N^2)$.

Метод квадратичного выбора впервые опубликован Э. Х. Фрэндом [JACM, 3 (1956), 152–154]; он указал, что эту идею можно обобщить, получив в результате метод кубического выбора, выбора четвертой степени и т. д. Например, метод кубического выбора состоит в том, чтобы разделить файл на $\sqrt[3]{N}$ больших групп, каждая из которых содержит по $\sqrt[3]{N}$ малых групп по $\sqrt[3]{N}$ записей; время работы пропорционально $N\sqrt[3]{N}$. Если развить эту идею до ее полного завершения, то мы придем к тому, что Фрэнд назвал "выбор n -й степени", основанный на структуре бинарного дерева. Время работы этого метода пропорционально $N \log N$; мы будем называть его *выбором из дерева*.

Выбор из дерева. Принципы сортировки посредством выбора из дерева будет легче воспринять, если воспользоваться аналогией с типичным "турниром с выбыванием". Рассмотрим, например, результаты соревнования по настольному теннису, показанные на рис. 22. Джим побеждает Дона, а Джо побеждает Джека; затем в следующем туре Джо выигрывает у Джима и т. д.

На рис. 22 показано, что Джо—чемпион среди восьми спортсменов, а для того, чтобы определить это, потребовалось $8 - 1 = 7$ матчей (т. е. сравнений). Дик вовсе не обязательно будет вторым по силе игроком: любой из спортсменов, у которых выиграл Джо, включая даже проигравшего в первом туре Джека, мог бы оказаться вторым по силе игроком. Второго игрока можно определить, заставив Джека сыграть с Джимом, а победителя этого матча—с Диком; всего два дополнительных матча требуется для определения второго по силе игрока, исходя из того соотношения сил, которое мы запомнили из предыдущих игр.

Вообще можно "вывести" игрока, находящегося в корне дерева, и заменить его чрезвычайно слабым игроком. Подстановка этого слабого игрока означает, что первоначально второй по силе спортсмен станет теперь наилучшим, и именно он окажется в корне, если вновь вычислить победителей в верхних уровнях дерева. Для этого нужно изменить лишь один путь, в дереве, так что для выбора следующего по силе игрока необходимо менее $\lceil \log_2 N \rceil$ дополнительных сравнений. Суммарное

Picture: Рис. 23. Пример сортировки посредством выбора из дерева...

время выполнения такой сортировки посредством выбора примерно пропорционально $N \log N$.

На рис. 23 сортировке посредством выбора из дерева подвергаются наши 16 чисел. Заметим, что для того, чтобы знать, куда вставлять следующий элемент " $-\infty$ ", необходимо помнить, откуда

пришел ключ, находящийся в корне. Поэтому узлы разветвления в действительности содержат указатели или индексы, описывающие позицию ключа, а не сам ключ. Отсюда следует, что необходима память для N исходных записей, $N - 1$ слов-указателей и N выводимых записей. (Разумеется, если вывод

Picture: Рис. 24. Применение корпоративной системы выдвиганий к сортировке. Каждый поднимается на свой уровень некомпетентности в иерархии.

идет на ленту или на диск, то не нужно сохранять выводимые записи в оперативной памяти.)

Чтобы оценить те замечательные усовершенствования, которые мы собираемся обсудить, в этом месте следует прервать чтение до тех пор, пока вы не освоитесь с выбором из дерева хотя бы настолько, что решение упр. 10 не составит для вас никакого труда.

Одна из модификаций выбора из дерева, введенная, по существу, К. Э. Айверсоном [A Programming Language (Wiley, 1962), 223–227], устраняет необходимость указателей, следующим образом осуществляя "заглядывание вперед": когда победитель матча на нижнем уровне поднимается вверх, то на нижнем уровне его сразу же можно заменить на " $-\infty$ "; когда же победитель перемещается вверх с одного разветвления на другое, то его можно заменить игроком, который в конце концов все равно должен подняться, на его прежнее место (а именно наибольшим из двух ключей, расположенных под ним). Выполнив эту операцию столько раз, сколько возможно, придем от рис. 23(а) к рис. 24.

Коль скоро дерево построено таким образом, можно продолжать сортировку не "восходящим" методом, показанным на рис. 23, а "нисходящим": выводится элемент, находящийся в корне, перемещается вверх наибольший из его потомков, перемещается вверх наибольший из потомков последнего и т. д. Процесс начинает походить не столько на турнир по настольному теннису, сколько на корпоративную систему выдвиганий.

Читатель должен уяснить, что у нисходящего метода есть важное достоинство—он позволяет избежать лишних сравнений $-\infty$ с $-\infty$. (Пользуясь восходящим методом, мы на более поздних стадиях сортировки всюду натываемся на $-\infty$, а при нисходящем методе можно на каждой стадии заканчивать преобразование дерева сразу же после занесения $-\infty$.)

Picture: Рис. 25. Последовательное распределение памяти для полного бинарного дерева.

На рис. 23 и 24 изображены *полные бинарные деревья* с 16 концевыми узлами (ср. с п. 2.3.4.5); такие деревья удобно хранить в последовательных ячейках памяти, как показано на рис. 25. Заметим, что отцом узла номер k является узел $\lfloor k/2 \rfloor$, а его потомками являются узлы $2k$ и $2k + 1$. Отсюда вытекает еще одно преимущество нисходящего метода, поскольку зачастую значительно проще продвигаться вниз от узла k к узлам $2k$ и $2k + 1$, чем вверх от узла k к узлам $k \oplus 1$ и $\lfloor k/2 \rfloor$. (Здесь через $k \oplus 1$ обозначено число $k + 1$ или $k - 1$ в зависимости от того, является ли k четным или нечетным.)

До сих пор в наших примерах выбора из дерева в той или иной мере предполагалось, что N есть степень 2; в действительности можно работать с произвольным значением N , так как полное бинарное дерево с N концевыми узлами нетрудно построить для любого N .

Мы подошли теперь к основному вопросу: нельзя ли в нисходящем методе обойтись совсем без " $-\infty$ "? Не правда ли, было бы чудесно, если бы всю существенную информацию, имеющуюся на рис. 24, удалось расположить в ячейках 1–16 полного бинарного дерева без всяких бесполезных "дыр", содержащих $-\infty$? Поразмыслив, можно прийти к выводу, что эта цель в действительности достижима, причем не только исключается $-\infty$, но и появляется возможность сортировать N записей на том же месте без вспомогательной области вывода. Это приводит к еще одному важному алгоритму сортировки. Его автор Дж. У. Дж. Уильямс [CASM, 7 (1964), 347–348] окрестил свой алгоритм "пирамидальной-сортировкой" ("heapsort").

Пирамидальная сортировка. Будем называть файл ключей K_1, K_2, \dots, K_N "пирамидой", если

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad \text{при } 1 \leq \lfloor j/2 \rfloor < j \leq N. \quad (3)$$

В этом случае $K_1 \geq K_2, K_1 \geq K_3, K_2 \geq K_4$ и т.д.; именно это условие выполняется на рис. 24. Из него следует, в частности, что наибольший ключ оказывается "на вершине пирамиды":

$$K_1 = \max(K_1, K_2, \dots, K_N). \quad (4)$$

Если бы мы сумели как-нибудь преобразовать произвольный исходный файл в пирамиду, то для получения эффективного алгоритма сортировки можно было бы воспользоваться "нисходящей" процедурой выбора, подобной той, которая описана выше.

Эффективный подход к задаче построения пирамиды предложил Р. У. Флойд [САСМ, 7 (1964), 701]. Пусть нам удалось расположить файл таким образом, что

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad \text{при } l < \lfloor j/2 \rfloor < j \leq N, \quad (5)$$

где l — некоторое число ≥ 1 . (В исходном файле это условие выполняется "автоматически" для $l = \lfloor N/2 \rfloor$, поскольку ни один индекс j не удовлетворяет условию $\lfloor N/2 \rfloor < \lfloor j/2 \rfloor < j \leq N$.) Нетрудно понять, как, изменяя лишь поддерево с корнем в узле l , преобразовать файл, чтобы распространить неравенства (5) и на случай, когда $\lfloor j/2 \rfloor = l$. Следовательно, можно уменьшать l на единицу, пока в конце концов не будет достигнуто условие (3). Эти идеи Уильямса и Флойда приводят к изящному алгоритму, который заслуживает пристального изучения.

Алгоритм Н. (*Пирамидальная сортировка.*) Записи R_1, \dots, R_N переразмещаются на том же месте; после завершения сортировки их ключи будут упорядочены: $K_1 \leq \dots \leq K_N$. Сначала файл перестраивается в пирамиду, после чего вершина пирамиды многократно исключается и записывается на свое окончательное место. Предполагается, что $N \geq 2$.

Н1 [Начальная установка.] Установить $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$.

Н2 [Уменьшить l или r .] Если $l > 1$, то установить $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$. (Если $l > 1$, это означает, что происходит процесс преобразования исходного файла в пирамиду; если же $l = 1$, то это значит, что ключи K_1, K_2, \dots, K_r уже образуют пирамиду.) В противном случае установить $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$ и $r \leftarrow r - 1$; если в результате оказалось, что $r = 1$, то установить $R_1 \leftarrow R$ и завершить работу алгоритма.

Н3 [Приготовиться к "протаскиванию".] Установить $j \leftarrow l$. (К этому моменту

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad \text{при } l < \lfloor j/2 \rfloor < j \leq r, \quad (6)$$

а записи R_k , $r < k \leq N$, занимают свои окончательные места. Шаги **Н3–Н8** называются алгоритмом "протаскивания"; их

Picture: Рис. 26. Пирамидальная сортировка; пунктиром обведен алгоритм "протаскивания".

действие эквивалентно установке $R_l \leftarrow R$ с последующим перемещением записей R_l, \dots, R_r таким образом, чтобы условие (6) выполнялось и при $\lfloor j/2 \rfloor = l$.)

Н4 [Продвинуться вниз.] Установить $i \leftarrow j$ и $j \leftarrow 2j$. (В последующих шагах $i = \lfloor j/2 \rfloor$.) Если $j < r$, то перейти к шагу **Н5**; если $j = r$, то перейти к шагу **Н6**; если же $j > r$, то перейти к шагу **Н8**.

Н5 [Найти "большого" сына.] Если $K_j < K_{j+1}$, то установить $j \leftarrow j + 1$.

Н6 [Больше K ?] Если $K \geq K_j$, то перейти к шагу **Н8**.

Н7 [Поднять его вверх.] Установить $R_i \leftarrow R_j$ и возвратиться к шагу **Н4**.

Н8 [Занести R .] Установить $R_i \leftarrow R$. (На этом алгоритм "протаскивания", начатый в шаге **Н3**, заканчивается.) Возвратиться к шагу **Н2**. ■

$7A + 14B + 4C + 20N - 2D + 15\lfloor N/2 \rfloor - 28$ равно, таким образом, в среднем примерно $16N \log_2 N + 0.2N$ единиц.

Глядя на табл. 2, трудно поверить в то, что пирамидальная сортировка так уж эффективна: большие ключи перемещаются влево прежде, чем мы успеваем отложить их вправо! Это и в самом деле странный способ сортировки при малых N . Время сортировки 16 ключей из табл. 2 равно $1068u$, тогда как обычный метод простых вставок (программа 5.2.1S) требует всего $514u$. При сортировке простым выбором (программа S) требуется $853u$.

При больших N программа H более эффективна. Напрашивается сравнение с сортировкой методом Шелла с убывающим шагом (программа 5.2.1D) и быстрой сортировкой Хоара (программа 5.2.2Q), так как во всех трех программах сортировка производится путем сравнения ключей, причем вспомогательной памяти используется мало или она не используется вовсе. При $N = 1000$ средние времена работы равны приблизительно

160000 u для пирамидальной сортировки;
130000 u для сортировки методом Шелла;
80000 u для быстрой сортировки.

(MIX—типичный представитель большинства современных вычислительных машин, но, разумеется, на конкретных машинах получаются несколько иные относительные величины.) С ростом N пирамидальная сортировка превзойдет по скорости метод Шелла, но асимптотическая формула $16N \log_2 N \approx 23.08N \ln N$ никогда не станет лучше выражения для быстрой сортировки $12.67N \ln N$. Модификация пирамидальной сортировки, обсуждаемая в упр. 18, ускорит процесс на многих вычислительных машинах, но даже с этим усовершенствованием пирамидальная сортировка не достигнет скорости быстрой сортировки.

С другой стороны, быстрая сортировка эффективна лишь в среднем; в наихудшем случае ее время работы пропорционально N^2 . Пирамидальная же сортировка обладает тем интересным свойством, что для нее наихудший случай не намного хуже среднего. Всегда выполняются неравенства

$$A \leq 1.5N, \quad B \leq N \lfloor \log_2 N \rfloor, \quad C \leq N \lfloor \log_2 N \rfloor; \quad (8)$$

таким образом, независимо от распределения исходных данных выполнение программы H не займет более $18N \lfloor \log_2 N \rfloor + 38N$ единиц времени. Пирамидальная сортировка—первый из рассмотренных нами до сих пор методов сортировки, время работы которого *заведомо* имеет порядок $N \log N$. Сортировка посредством слияний, которая будет обсуждаться ниже, в п. 5.2.4, тоже обладает этим свойством, но она требует больше памяти.

Наибольший из включенных—первым исключается. В гл. 2 мы видели, что линейные списки часто можно осмысленно расклассифицировать по характеру производимых над ними операций включения и исключения. *Стек* ведет себя по принципу "последним включается—первым исключается" в том смысле, что при каждом исключении удаляется самый молодой элемент списка (элемент, который был вставлен позже всех других элементов, присутствующих в данный момент в списке). Простая *очередь* ведет себя по принципу "первым включается—первым исключается" в том смысле, что при каждом исключении удаляется самый старший из имеющихся элементов. В более сложных ситуациях, таких, как моделирование лифта в п. 2.2.5, требуется список типа "наименьший из включенных—первым исключается", где при каждом исключении удаляется элемент, имеющий наименьший ключ. Такой список можно назвать *приоритетной очередью*, так как ключ каждого элемента отражает его относительную способность быстро покинуть список. Сортировка посредством выбора—частный случай приоритетной очереди, над которой производится сначала N операций вставки, а затем N операций удаления.

Приоритетные очереди возникают в самых разнообразных приложениях. Например, в некоторых численных итеративных схемах повторяется выбор элемента, имеющего наибольшее (или наименьшее) значение некоторого проверочного критерия; параметры выбранного элемента изменяются, и он снова вставляется в список с новым проверочным значением, - соответствующим новым значениям параметров. Приоритетные очереди часто используются в операционных системах при планировании заданий. Другие типичные применения приоритетных очередей упоминаются в упр. 15, 29 и 36; кроме того, много примеров встретится в последующих главах.

Как же реализовать приоритетную очередь? Один из очевидных способов—поддерживать отсортированный список элементов, упорядоченных по ключам. Тогда включение нового элемента, по существу, сводится к задаче, рассмотренной нами при изучении сортировки вставками в п. 5.2.1. При другом, еще более очевидном способе работы с приоритетной очередью элементы в списке хранятся в

произвольном порядке, и тогда для выбора нужного элемента, приходится осуществлять поиск наибольшего (или наименьшего) ключа каждый раз, когда необходимо произвести исключение. В обоих этих очевидных подходах неприятность состоит в том, что требуется порядка N шагов для выполнения либо операции вставки, либо операции удаления, если в списке содержится N элементов, т. е. при больших N эти операции занимают слишком много времени.

В своей статье о пирамидальной сортировке Уильяме указал на то, что пирамиды идеально подходят для приложений с большими приоритетными очередями, так как элемент можно вставить в пирамиду или удалить из нее за $O(\log N)$ шагов; к тому же все элементы пирамиды компактно располагаются в последовательных ячейках памяти. Фаза выбора в алгоритме H—это последовательность шагов удаления в процессе типа *наибольший из включенных—первым исключается*: чтобы исключить наибольший элемент K_1 мы удаляем его и "протаскиваем" элемент K_N в новой пирамиде из $N - 1$ элементов. (Если нужен алгоритм типа *наименьший из включенных—первым исключается*, как при моделировании лифта, то, очевидно, можно изменить определение пирамиды, заменив в (3) знак " \geq " на " \leq "; для удобства мы будем рассматривать здесь лишь случай "наибольший из включенных—первым исключается".) Вообще, если требуется исключить наибольший элемент, а затем вставить новый элемент x , то можно выполнить процедуру протаскивания с $l = 1$, $r = N$ и $K = x$. Если же необходимо вставить элемент без предварительного исключения, то можно воспользоваться "восходящей" процедурой из упр. 16.

Связанное представление приоритетных очередей. Эффективный способ представления приоритетных очередей в виде связанных бинарных деревьев предложил в 1971 г. Кларк Э. Крэйна. Его метод требует наличия в каждой записи двух полей связи и короткого поля счетчика, но по сравнению с пирамидами он обладает следующими преимуществами:

- 1) Если с приоритетной очередью работают как со стеком, то операции включения и исключения более эффективны (они занимают фиксированное время, не зависящее от длины очереди).
- 2) Записи никогда не перемещаются, изменяются только указатели.
- 3) Можно слить две непересекающиеся приоритетные очереди, содержащие в общей сложности N элементов, в одну всего за $O(\log N)$ шагов.

Слегка видоизмененный метод Крэйна проиллюстрирован на рис. 27, на котором показан особый тип структуры бинарного дерева. Каждый узел содержит поле KEY, поле DIST и два поля связи—LEFT и RIGHT. Поле DIST всегда устанавливается равным длине кратчайшего пути от этого узла до конечного узла (т. е. до пустого узла Λ) дерева. Если считать, что $\text{DIST}(\Lambda) = 0$ и $\text{KEY}(\Lambda) = -\infty$, то поля KEY и DIST в этом дереве удовлетворяют следующим соотношениям:

$$\text{KEY}(P) \geq \text{KEY}(\text{LEFT}(P)), \text{KEY}(P) \geq \text{KEY}(\text{RIGHT}(P)); \quad (9)$$

$$\text{DIST}(P) = 1 + \min(\text{DIST}(\text{LEFT}(P)), \text{DIST}(\text{RIGHT}(P))); \quad (10)$$

$$\text{DIST}(\text{LEFT}(P)) \geq \text{DIST}(\text{RIGHT}(P)). \quad (11)$$

Соотношение (9) аналогично условию пирамиды (3) и служит гарантией того, что в корне дерева находится наибольший ключ, а соотношение (10)—это просто определение поля DIST, сформулированное выше. Соотношение (11) представляет собой интересное новшество: из него следует, что кратчайший путь к конечному узлу всегда можно получить, двигаясь вправо. Мы

Picture: Рис. 27. Приоритетная очередь, представленная в виде левостороннего дерева.

будем называть бинарное дерево с этим свойством левосторонним деревом, поскольку оно, как правило, сильно "тянется" влево.

Из этих определений ясно, что равенство $\text{DIST}(P) = n$ подразумевает существование по крайней мере 2^n конечных узлов ниже P ; в противном случае нашелся бы более, короткий путь от P до конечного узла. Таким образом, если в левостороннем дереве имеется N узлов, то путь, ведущий из корня вниз по направлению вправо, содержит не более чем $\lfloor \log_2(N + 1) \rfloor$ узлов. Новый узел можно вставить в приоритетную очередь, пройдя по этому пути (см. упр. 32); следовательно, в худшем случае необходимо всего $O(\log N)$ шагов. Наилучший случай достигается, когда дерево линейно (все связи RIGHT равны Λ), а наихудший случай достигается, когда дерево абсолютно сбалансировано.

Чтобы удалить узел из корня, нужно просто слить два его поддерева. Операция слияния двух непересекающихся левосторонних деревьев, на которые ссылаются указатели P и Q , по своей идее проста: если $\text{KEY}(P) \geq \text{KEY}(Q)$, то берем в качестве корня P и сливаем Q с правым поддеревом P ; при этом изменится $\text{DIST}(P)$, а $\text{LEFT}(P)$ меняется местами с $\text{RIGHT}(P)$, если это необходимо. Нетрудно составить подробное описание этого процесса (см. упр. 32).

Сравнение методов работы с приоритетными очередями. Если число узлов N мало, то для поддержания приоритетной очереди лучше всего применять один из простых методов с использованием линейных списков. Если же N велико, то, очевидно, гораздо более быстрым будет метод, время работы которого порядка $\log N$. Поэтому большие приоритетные очереди обычно представляют в виде пирамид или левосторонних деревьев. В п. 6.2.3 мы обсудим еще один способ представления линейных списков в виде *сбалансированных деревьев*, который приводит к третьему методу, пригодному для представления приоритетных очередей, с временем работы порядка $\log N$. Поэтому уместно сравнить эти три метода.

Мы видели, что операции над левосторонними деревьями в целом несколько быстрее, чем операции над пирамидами, но пирамиды занимают меньше памяти. Сбалансированные деревья занимают примерно столько же памяти, сколько левосторонние деревья (быть может, чуть меньше); операции над ними медленнее, чем над пирамидами, а программирование сложнее, но структура сбалансированных деревьев в некоторых отношениях существенно более гибкая. Работая с пирамидами, не так просто предсказать, что произойдет с элементами, если у них равные ключи; нельзя гарантировать, что элементы с равными ключами будут обрабатываться по принципу "последним включается—первым исключается" или "первым включается—первым исключается", если только ключ не расширен и не содержит дополнительного поля "порядковый номер вставки", и тогда равных ключей просто нет. С другой стороны, если применять сбалансированные деревья, можно легко оговорить твердые условия относительно равных ключей. Можно также выполнять такие действия, как "вставить x непосредственно перед (или после) y ". Сбалансированные деревья симметричны, так что в любой момент можно исключить либо наибольший, либо наименьший элемент, в то время как левосторонние деревья и пирамиды должны быть так или иначе ориентированы. (См. тем не менее упр. 31, в котором

Picture: Рис. 28. Так выглядит пирамида...

показано, как строить *симметричные* пирамиды.) Сбалансированные деревья можно использовать как для поиска, так и для сортировки; и из сбалансированного дерева можно довольно быстро удалять последовательные блоки элементов. Но два сбалансированных дерева нельзя слить менее чем за $O(N)$ шагов, в то время как два левосторонних дерева можно слить всего за $O(\log N)$ шагов.

Итак, пирамиды наиболее экономны с точки зрения памяти; левосторонние деревья хороши тем, что можно быстро слить две непересекающиеся приоритетные очереди; и, если нужно, за умеренное вознаграждение можно получить ту гибкость, какую предоставляют сбалансированные деревья.

*** Анализ пирамидальной сортировки.** Алгоритм Н до сих пор не был полностью проанализирован, но некоторые его свойства можно вывести без особого труда. Поэтому мы завершим этот пункт довольно подробным исследованием, касающимся пирамид.

На рис. 28 показана форма пирамиды из 26 элементов; каждый узел помечен двоичным числом, соответствующим его индексу в пирамиде. Звездочками в этой диаграмме помечены так называемые *особые узлы*, которые лежат на пути от 1 к N .

Одна из наиболее важных характеристик пирамиды—набор размеров ее поддеревьев. Например, на рис. 28 размеры поддеревьев с корнями в узлах 1, 2, ..., 26 равны соответственно

$$26^*, 15, 10^*, 7, 7, 6^*, 3, 3, 3, 3, 3, 3, 2^*, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1^*. \quad (12)$$

Звездочками отмечены *особые поддеревья* с корнями в особых узлах; в упр. 20 показано, что если N имеет двоичное представление

$$N = (b_n b_{n-1} \dots b_1 b_0)_2, \quad n = \lfloor \log_2 N \rfloor, \quad (13)$$

то размеры особых поддеревьев равны

$$(1b_{n-1} \dots b_1 b_0)_2, (1b_{n-2} \dots b_1 b_0)_2, \dots, (1b_1 b_0)_2, (1b_0)_2, (1)_2. \quad (14)$$

(Неособые поддеревья всегда абсолютно сбалансированы, так что их размеры всегда имеют вид $2^k - 1$. В упр. 21 показано, что среди неособых поддеревьев имеется ровно

$$\begin{aligned} \lfloor (N-1)/2 \rfloor \text{ размера } 1, & \quad \lfloor (N-2)/4 \rfloor \text{ размера } 3, \\ \lfloor (N-4)/8 \rfloor \text{ размера } 7, & \quad \dots \quad \lfloor (N-2^{n-1})/2^n \rfloor \text{ размера } (2^n - 1), \end{aligned} \quad (15)$$

Например, на рис. 28 изображено двенадцать неособых поддеревьев размера 1, шесть поддеревьев размера 3, два—размера 7 и одно—размера 15.

Пусть s_l — размер поддерева с корнем l , а M_N — мультимножество $\{s_1, s_2, \dots, s_N\}$ всех этих размеров. Используя (14) и (15), легко вычислить M_N при любом заданном N . В упр. 5.1.4–20 показано, что общее число способов построить пирамиду из целых чисел $\{1, 2, \dots, N\}$ равно

$$N! / s_1 s_2 \dots s_N = N! / \prod_{s \in M_N} s. \quad (16)$$

Например, число способов расположить 26 букв $\{A, B, C, \dots, Z\}$ на рис. 28 так, чтобы по вертикали сохранялся алфавитный порядок, равно

$$26! / (26 \cdot 10 \cdot 6 \cdot 2 \cdot 1 \cdot 1^{12} \cdot 3^6 \cdot 7^2 \cdot 15^1).$$

Теперь мы в состоянии проанализировать фазу построения пирамиды в алгоритме Н, т. е. вычисления, которые завершаются до того, как в шаге Н2 впервые выполнится условие $l = 1$. К счастью, благодаря следующей ниже теореме анализ построения пирамиды можно свести к изучению независимых операций протаскивания.

Теорема Н. Если исходными данными для алгоритма Н служит случайная перестановка множества $\{1, 2, \dots, N\}$, то в фазе построения пирамиды с одинаковой вероятностью может получиться любая из $N! / (\prod_{s \in M_N} s)$ возможных пирамид. Более того, все $\lfloor N/2 \rfloor$ операций протаскивания, выполненные за время этой фазы, "равномерны" в том смысле, что по достижении шага Н8 все s_l возможных значений переменной i равновероятны.

Доказательство. Применим метод, который в численном анализе называется методом "обратной задачи". Пусть в качестве одного из возможных результатов операции протаскивания задана пирамида $K_1 \dots K_N$ с корнем в узле l ; тогда ясно, что имеется всего s_l исходных конфигураций $K'_1 \dots K'_N$ файла, которые после протаскивания дают такой результат. Все эти исходные конфигурации имеют различные значения K'_i , следовательно, рассуждая в обратном направлении, существует ровно $s_l s_{l+1} \dots s_N$ исходных перестановок множества $\{1, 2, \dots, N\}$, которые после завершения операции протаскивания в позицию l дают конфигурацию $K_1 \dots K_N$.

Случай $l = 1$ типичен: пусть $K_1 \dots K_N$ — пирамида, и пусть $K'_1 \dots K'_N$ — файл, который преобразуется в $K_1 \dots K_N$ в результате протаскивания при $l = 1$, $K = K'_1$. Если $K = K_i$, то должны иметь место равенства $K'_i = K_{\lfloor i/2 \rfloor}$, $K'_{\lfloor i/2 \rfloor} = K_{\lfloor i/4 \rfloor}$ и т. д., при этом $K'_j = K_j$ для всех j , не лежащих на пути от 1 к i . Обратно, при любом i в результате такого построения получается файл $K'_1 \dots K'_N$, такой, что (а) операция протаскивания преобразует файл $K'_1 \dots K'_N$ в $K_1 \dots K_N$ и (б) $K_{\lfloor j/2 \rfloor} \geq K_j$ при $2 \leq \lfloor j/2 \rfloor < j \leq N$. Следовательно, возможны ровно N таких файлов $K'_1 \dots K'_N$, и операция протаскивания равномерна. (Пример доказательства этой теоремы см. в упр. 22.) ■

Обращаясь к величинам A, B, C, D в анализе программы Н, можно видеть, что равномерная операция протаскивания, производимая над поддеревом размера s , дает вклад $\lfloor s/2 \rfloor / s$ в среднее значение величины A ; ее вклад в среднее значение величины B равен

$$\frac{1}{s}(0 + 1 + 1 + 2 + \dots + \lfloor \log_2 s \rfloor) = \frac{1}{s} \left(\sum_{1 \leq k \leq s} \lfloor \log_2 k \rfloor \right) = \frac{1}{s} ((s+1) \lfloor \log_2 s \rfloor - 2^{\lfloor \log_2 s \rfloor + 1} + 2)$$

(см. упр. 1.2.4–42); и она дает вклад $2/s$ или 0 в среднее значение величины D в зависимости от того, является ли s четным или нечетным. Несколько сложнее определить соответствующий вклад в среднее значение величины C , так что эта задача предоставляется читателю (см. упр. 26). Производя суммирование по всем операциям протаскивания, находим, что среднее значение величины A за время построения пирамиды равно

$$A'_N = \sum_{s \in M_N} \lfloor s/2 \rfloor / s; \quad (17)$$

аналогичные формулы имеют место и для B, C и D , так что можно без особого труда точно вычислить эти средние значения. В следующей таблице приведены типичные результаты:

N	A'_N	B'_N	C'_N	D'_N
99	19.18	68.35	42.95	0.00
100	19.93	69.39	42.71	1.84
999	196.16	734.66	464.53	0.00
1000	196.94	735.80	464.16	1.92
9999	1966.02	7428.18	4695.54	0.00
10000	1966.82	7429.39	4695.06	1.97
10001	1966.45	7430.07	4695.84	0.00
10002	1967.15	7430.97	4695.95	1.73

Что касается асимптотики, то в M_N можно не обращать внимания на размеры особых поддеревьев, и тогда мы найдем, например, что

$$A'_N = \frac{N}{2} \cdot \frac{0}{1} + \frac{N}{4} \cdot \frac{1}{3} + \frac{N}{8} \cdot \frac{3}{7} + \dots + O(\log N) = \left(1 - \frac{1}{2}\alpha\right) N + O(\log N), \quad (18)$$

где

$$\alpha = \sum_{k \geq 1} \frac{1}{2^k - 1} = 1.60669\ 51524\ 15291\ 76378\ 33015\ 23190\ 92458\ 04805\ldots \quad (19)$$

(Это значение получил Дж. У. Ренч младший, пользуясь преобразованием ряда из упр. 27.) При больших N можно использовать приближенные формулы

$$\begin{aligned} A'_N &\approx 0.1967N + 0.3\ (N \text{ чет.}), \quad 0.1967N - 0.3\ (N \text{ нечет.}); \\ B'_N &\approx 0.74403N - 1.3 \ln N; \\ C'_N &\approx 0.47034N - 0.8 \ln N; \\ D'_N &\approx 1.8 \pm 0.2\ (N \text{ чет.}), \quad 0\ (N \text{ нечет.}). \end{aligned} \quad (20)$$

Нетрудно определить также максимальные и минимальные значения. Для построения пирамиды требуется всего $O(N)$ шагов (ср. с упр. 23).

Этим, по существу, завершается анализ фазы построения пирамиды в алгоритме Н. Анализ фазы выбора—совсем другая задача, которая еще ожидает своего решения! Пусть пирамидальная сортировка применяется к N элементам; обозначим через A''_N, B''_N, C''_N и D''_N средние значения величин A, B, C и D во время фазы выбора. Поведение алгоритма Н подвержено сравнительно малым колебаниям около эмпирически установленных средних значений

$$\begin{aligned} A''_N &\approx 0.152N; \\ B''_N &\approx N \log_2 N - 2.61N; \\ C''_N &\approx \frac{1}{2}N \log_2 N - 1.4N; \\ D''_N &\approx \ln N \pm 2; \end{aligned} \quad (21)$$

тем не менее до сих пор не найдено правильного теоретического объяснения этим константам. Рассмотрев отдельные операции протаскивания, нетрудно вывести верхние оценки, указанные в неравенствах (8), хотя, если рассматривать алгоритм как целое, верхнюю оценку для C , по-видимому, следует уменьшить приблизительно до $\frac{1}{2}N \log_2 N$.

Упражнения

- [10] Является ли сортировка посредством простого выбора (алгоритм S) "устойчивой"?
- [15] Почему в алгоритме S оказывается более удобным находить сначала наибольший элемент, затем наибольший из оставшихся и т. д., вместо того чтобы находить сначала наименьший элемент, затем наименьший из оставшихся и т. д.?
- [M21] (а) Докажите, что если алгоритм S применяется к случайной перестановке множества $\{1, 2, \dots, N\}$, то в результате первого выполнения шагов S2 и S3 получается случайная перестановка множества $\{1, 2, \dots, N-1\}$, за которой следует N . (Иначе говоря, файл $K_1 \dots K_{N-1}$ с одинаковой вероятностью может быть любой перестановкой множества $\{1, 2, \dots, N-1\}$.) (б) Следовательно, если через B_N обозначить среднее значение величины B в программе S, то при условии, что исходный файл упорядочен случайным образом, имеем $B_N = H_N - 1 + B_{N-1}$. [Указание: ср. со статистическими характеристиками 1.2.10–16.]
- [M35] В шаге S3 алгоритма S ничего не делается, если $i = j$. Не лучше ли перед выполнением шага S3 проверить условие $i = j$? Чему равно среднее число случаев выполнения условия $i = j$ в шаге S3, если исходный файл случаен?
- [20] Чему равно значение величины B в анализе программы S для исходного файла $N \dots 3\ 2\ 1$?
- [M29] (а) Пусть $a_1\ a_2 \dots a_N$ —перестановка множества $\{1, 2, \dots, N\}$ с C циклами, I инверсиями и такая, что при ее сортировке с помощью программы S производится B обменов на правосторонний максимум. Докажите, что $2B \leq I + N - C$. [Указание: см. упр. 5.2.2–1.] (б) Покажите, что $I + N - C \leq \lfloor n^2/2 \rfloor$; следовательно, B не превышает $\lfloor n^2/4 \rfloor$.
- [M46] Найдите дисперсию величины B в программе S как функцию от N , считая, что исходный файл случаен.

8. [24] Покажите, что если при поиске $\max(K_1, \dots, K_j)$ в шаге S2 просматривать ключи слева направо: K_1, K_2, \dots, K_j , а не наоборот: K_j, \dots, K_2, K_1 , как в программе S, то за счет этого можно было бы сократить число сравнений при следующих повторениях шага S2. Напишите MIX-программу, основанную на этом наблюдении.
9. [M25] Чему равно среднее число сравнений, выполняемых алгоритмом из упр. 8 для случайного исходного файла?
10. [12] Как будет выглядеть дерево, изображенное на рис. 23, после того как будут выведены 14 из 16 первоначальных элементов?
11. [10] Как будет выглядеть дерево, изображенное на рис. 24, после вывода элемента 908?
12. [M20] Сколько раз будет выполнено сравнение $-\infty$ с ∞ , если применить "восходящий" метод, представленный на рис. 23, для упорядочения файла из 2^n элементов?
13. [20] (Дж. У. Дж. Уильямс.) В шаге H4 алгоритма H различаются три случая: $j < r$, $j = r$ и $j > r$. Покажите, что если $K \geq K_{r+1}$, то можно было бы так упростить шаг H4, чтобы разветвление происходило лишь по двум путям. Как надо изменить шаг H2, чтобы обеспечить в процессе пирамидальной сортировки выполнение условия $K \geq K_{r+1}$?
14. [10] Покажите, что простая очередь—частный случай приоритетной. (Объясните, какие ключи нужно присваивать элементам, чтобы процедура "наибольший из включенных—первым исключается" была эквивалентна процедуре "первым включается—первым исключается".) Является ли стек также частным случаем приоритетной очереди?
- >15. [M22] (В. Э. Чартрс.) Придумайте быстрый алгоритм построения таблицы простых чисел $\leq N$, в котором используется *приоритетная очередь* с целью избежать операций деления. [Указание. Пусть наименьший ключ в приоритетной очереди будет наименьшим нечетным непростым числом, большим, чем самое последнее нечетное число, воспринятое как кандидат в простые числа. Попытайтесь свести к минимуму число элементов в этой очереди.]
16. [20] Постройте эффективный алгоритм, который вставляет новый ключ в данную пирамиду из n элементов, порождая пирамиду из $n + 1$ элементов.
17. [20] Алгоритм из упр. 16 можно использовать для построения пирамиды взамен метода "уменьшения l до 1", применяемого в алгоритме H. Порождают ли оба метода из одного и того же исходного файла одну и ту же пирамиду?
- >18. [21] (Р. У. Флойд) Во время фазы выбора в алгоритме пирамидальной сортировки ключ K , как правило, принимает довольно малые значения, и поэтому почти при всех сравнениях в шаге H6 обнаруживается, что $K < K_j$. Как можно изменить алгоритм, чтобы ключ K не сравнивался с K_j в основном цикле вычислений?
19. [21] Предложите алгоритм исключения данного элемента из пирамиды размера N , порождающий пирамиду размера $N - 1$.
20. [M20] Покажите, что формулы (14) задают размеры особых поддеревьев пирамиды.
21. [M24] Докажите, что формулы (15) задают размеры неособых поддеревьев пирамиды.
- >22. [20] Какие перестановки множества $\{1, 2, 3, 4, 5\}$ фаза построения пирамиды в алгоритме H преобразует в 5 3 4 1 2?
23. [M28] (а) Докажите, что длина пути B в алгоритме протаскивания никогда не превышает $\lfloor \log_2(r/l) \rfloor$. (б) Согласно неравенствам (8), ни при каком конкретном применении алгоритма H величина B не может превзойти $N \lfloor \log_2 N \rfloor$. Найдите максимальное значение B по всевозможным исходным файлам как функцию от N . (Вы должны доказать, что существует исходный файл, на котором B принимает это максимальное значение.)
24. [M24] Выведите точную формулу стандартного отклонения величины B'_N (суммарная длина пути, пройденного по дереву во время фазы построения пирамиды в алгоритме H).
25. [M20] Чему равен средний вклад в значение величины C за время первой операции протаскивания, когда $l = 1$, а $r = N$, если $N = 2^{n+1} - 1$.
26. [M30] Решите упр. 25: (а) для $N = 26$, (б) для произвольного N .
27. [M25] (Дж. У. Ренч мл.) Докажите, что $\sum_{n \geq 1} x^n / (1 - x^n) = \sum_{n \geq 1} x^{n^2} (1 + x^n) / (1 - x^n)$. [Положив $x = \frac{1}{2}$, получите очень быстро сходящийся ряд для вычисления (19).]
28. [35] Продумайте идею *тернарных пирамид*, основанных на полных тернарных, а не бинарных деревьях. Будет ли тернарная пирамидальная сортировка быстрее бинарной?
29. [26] (У. С. Браун.) Постройте алгоритм умножения многочленов или степенных рядов $(a_1 x^{i_1} + a_2 x^{i_2} + \dots)(b_1 x^{j_1} + b_2 x^{j_2} + \dots)$, который бы порождал коэффициенты произведения $c_1 x^{i_1+j_1} + \dots$ по порядку, по мере того как перемножаются коэффициенты исходных многочленов. [Указание: воспользуйтесь подходящей приоритетной очередью.]
30. [M48] Может ли величина C превзойти $\frac{1}{2} N \log_2 N$ при пирамидальной сортировке файла? Чему равно максимальное значение C ? Чему равно минимальное значение?

31. [37] (Дж. У. Дж. Уильямс.) Покажите, что если две пирамиды подходящим образом совместить "основание к основанию", то это даст возможность поддерживать структуру, в которой в любой момент можно за $O(\log n)$ шагов исключить либо наибольший, либо наименьший элемент. (Такую структуру можно назвать *приоритетным деком*.)
32. [21] Разработайте алгоритм слияния двух непересекающихся приоритетных очередей, представленных в виде левосторонних деревьев, в одну. (В частности, если одна из данных очередей содержит всего один элемент, то ваш алгоритм будет вставлять его в другую очередь.)
- >33. [15] Почему в приоритетной очереди, представленной в виде левостороннего дерева, операция удаления корня выполняется путем слияния двух поддеревьев, а не "продвижения" узлов по направлению к вершине, как в пирамиде?
34. [M47] Сколько можно построить левосторонних деревьев из N узлов, если игнорировать значения поля KEY? [Эта последовательность начинается с чисел 1, 1, 2, 4, 8, 17, 38, ...; существует ли какая-нибудь простая асимптотическая формула?]
35. [26] Если в левостороннее дерево с N узлами добавить связи UP (ср. с обсуждением деревьев с тремя связями в п. 6.2.3), то это даст возможность исключать из приоритетной очереди произвольный узел P следующим образом: слить LEFT(P) и RIGHT(P) и поместить полученное поддерево на место P , затем исправлять поля DIST у предков узла P до тех пор, пока не будет достигнут либо корень, либо узел, у которого поле DIST не меняется. Докажите, что при этом никогда не потребуются изменить более чем $O(\log N)$ полей DIST, несмотря даже на то, что дерево может содержать очень длинные восходящие пути.
36. [18] (*Замещение наиболее давно использованной страницы.*) Многие операционные системы используют алгоритм следующего типа: над набором узлов допустимы две операции — "использование" узла и замещение наиболее давно "использованного" узла новым узлом. Какая структура данных облегчает нахождение наиболее давно "использованного" узла?

5.2.4. Сортировка слиянием

Слияние означает объединение двух или более упорядоченных файлов в один упорядоченный файл. Можно, например, слить два подфайла—503 703 765 и 087 512 677, получив 087 503 512 677 703 765. Простой способ сделать это—сравнить два наименьших элемента, вывести наименьший из них и повторить эту процедуру.

Начав с

$$\begin{cases} 503 & 703 & 765 \\ 087 & 512 & 677 \end{cases},$$

получим

$$087 \begin{cases} 503 & 703 & 765 \\ 512 & 677 \end{cases}$$

затем

$$087 \ 503 \begin{cases} 703 & 765 \\ 512 & 677 \end{cases}$$

и т. д. Необходимо позаботиться о действиях на случай, когда исчерпается один из файлов. Весь процесс подробно описан в следующем алгоритме.

Алгоритм М. (Двухпутевое слияние.) Этот алгоритм осуществляет слияние двух упорядоченных файлов $x_1 \leq x_2 \leq \dots \leq x_m$ и $y_1 \leq y_2 \leq \dots \leq y_n$ в один файл $z_1 \leq z_2 \leq \dots \leq z_{m+n}$.

M1 [Начальная установка.] Установить $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$.

M2 [Найти наименьший элемент.] Если $x_i \leq y_j$, то перейти к шагу **M3**; в противном случае перейти к **M5**.

M3 [Вывести x_i .] Установить $z_k \leftarrow x_i, k \leftarrow k + 1, i \leftarrow i + 1$. Если $i \leq m$, то возвратиться к **M2**.

M4 [Вывести y_j, \dots, y_n .] Установить $(z_k, \dots, z_{m+n}) \leftarrow (y_j, \dots, y_n)$ и завершить работу алгоритма.

M5 [Вывести y_j .] Установить $z_k \leftarrow y_j, k \leftarrow k + 1, j \leftarrow j + 1$. Если $j \leq n$, то возвратиться к **M2**.

M6 [Вывести x_i, \dots, x_m .] Установить $(z_k, \dots, z_{m+n}) \leftarrow (x_i, \dots, x_m)$ и завершить работу алгоритма. ■

Picture: Рис. 29. Слияние $x_1 \leq \dots \leq x_m$ с $y_1 \leq \dots \leq y_n$.

В п. 5.3.2 мы увидим, что эта простая процедура, по существу, "наилучший из возможных" способов слияния на традиционной ЭВМ, если $m \approx n$. (Но, если m гораздо меньше n , можно разработать более эффективные алгоритмы сортировки, хотя в общем случае они довольно сложны.) Алгоритм М

без особой потери эффективности можно немного упростить, добавив в конец исходных файлов искусственных "стражей" $x_{m+1} = y_{n+1} = \infty$ и останавливаясь перед выводом ∞ . Анализ алгоритма М см. в упр. 2.

Общий объем работы, выполняемой алгоритмом М, по существу, пропорционален $m + n$, поэтому ясно, что слияние—более простая задача, чем сортировка. Однако задачу сортировки можно свести к слияниям, сливая все более длинные подфайлы до тех пор, пока не будет отсортирован весь файл. Такой подход можно рассматривать как развитие идеи сортировки вставками: вставка нового элемента в упорядоченный файл—частный случай слияния при $n = 1$! Если нужно ускорить процесс вставок, то можно рассмотреть вставку нескольких элементов за раз, "группируя" их, а это естественным образом приводит к общей идее сортировки слиянием. С исторической точки зрения метод слияний—один из самых первых методов, предназначенных для сортировки

Picture: Таблица 1 Сортировка естественным двухпутевым слиянием

на ЭВМ; он был предложен Джоном фон Нейманом еще в 1945 г. (см. § 5.5).

Мы довольно подробно изучим слияния в § 5.4 в связи с алгоритмами внешней сортировки, а в настоящем пункте сосредоточим свое внимание на сортировке в быстрой памяти с произвольным доступом.

Таблица 1 иллюстрирует сортировку слиянием, когда "свечка сжигается с обоих концов", подобно тем процедурам просмотра элементов файла, которые применялись при быстрой сортировке, поразрядной обменной сортировке и т. д. Мы анализируем исходный файл слева и справа, двигаясь к середине. Пропустим пока первую строку и рассмотрим переход от второй строки к третьей. Слева мы видим возрастающий отрезок 503 703 765, а справа, если читать справа налево, имеем отрезок 087 512 677. Слияние этих двух последовательностей дает подфайл 087 503 512 677 703 765, который помещается слева в строке 3. Затем ключи 061 612 908 в строке 2 сливаются с 170 509 897, и результат (061 170 509 612 897 908) записывается *справа* в строке 3. Наконец, 154 275 426 653 сливается с 653 (перекрытие обнаруживается прежде, чем оно может привести к вредным последствиям), и результат записывается слева. Точно так же строка 2 получилась из исходного файла в строке 1.

Вертикальными линиями в табл. 1 отмечены границы между отрезками. Это так называемые "ступеньки вниз", где меньший элемент следует за большим. В середине файла обычно возникает двусмысленная ситуация, когда при движении с обоих концов мы прочитываем один и тот же ключ; это не приведет к осложнениям, если проявить чуточку осторожности, как в следующем алгоритме. Такой метод по традиции называется "естественным" слиянием, потому что он использует отрезки, которые "естественно" образуются в исходном файле.

Алгоритм N. (Сортировка естественным двухпутевым слиянием.) При сортировке записей R_1, \dots, R_N используются две области памяти, каждая из которых может содержать N записей. Для удобства обозначим записи, находящиеся во второй области, через R_{N+1}, \dots, R_{2N} , хотя в действительности запись R_{N+1} может и не примыкать непосредственно к R_N . Начальное содержимое записей R_{N+1}, \dots, R_{2N} не имеет значения. После завершения сортировки ключи будут упорядочены: $K_1 \leq \dots \leq K_N$.

Picture: Рис. 30. Сортировка слиянием.

- N1** [Начальная установка.] Установить $s \leftarrow 0$. (При $s = 0$ мы будем пересылать записи из области (R_1, \dots, R_N) в область (R_{N+1}, \dots, R_{2N}) ; при $s = 1$ области по отношению к пересылкам поменяются ролями.)
- N2** [Подготовка к просмотру.] Если $s = 0$, то установить $i \leftarrow 1, j \leftarrow N, k \leftarrow N + 1, l \leftarrow 2N$; если $s = 1$, то установить $i \leftarrow N + 1, j \leftarrow 2N, k \leftarrow 1, l \leftarrow N$. (Переменные i, j, k, l указывают текущие позиции во входных "файлах", откуда идет чтение, и в выходных "файлах", куда идет запись.) Установить $d \leftarrow 1, f \leftarrow 1$. (Переменная d определяет текущее направление вывода, f устанавливается равной 0, если необходимы дальнейшие просмотры.)
- N3** [Сравнение $K_i : K_j$] Если $K_i > K_j$, перейти к шагу **N8**. Если $i = j$, установить $P_k \leftarrow R_i$ и перейти к шагу **N13**.
- N4** [Пересылка R_i .] (Шаги **N4–N7** аналогичны шагам **M3–M4** алгоритма М.) Установить $R_k \leftarrow R_i, k \leftarrow k + d$.
- N5** [Ступенька вниз?] Увеличить i на 1. Затем, если $K_{i-1} \leq K_i$, возвратиться к шагу **N3**.
- N6** [Пересылка R_j .] Установить $R_k \leftarrow R_j, k \leftarrow k + d$.
- N7** [Ступенька вниз?] Уменьшить j на 1. Затем, если $K_{j+1} \leq K_j$, возвратиться к шагу **N6**; в противном случае перейти к шагу **N12**.
- N8** [Пересылка R_j .] (Шаги **N8–N11** двойственны по отношению к шагам **N4–N7**.) Установить $R_k \leftarrow R_j, k \leftarrow k + d$.

- N9 [Ступенька вниз?] Уменьшить j на 1. Затем, если $K_{j+1} \leq K_j$, возвратиться к шагу N3.
- N10 [Пересылка R_i .] Установить $R_k \leftarrow R_i$, $k \leftarrow k + d$.
- N11 [Ступенька вниз?] Увеличить i на 1. Затем, если $K_{i-1} \leq K_i$, возвратиться к шагу N10.
- N12 [Переключение направления.] Установить $f \leftarrow 0$, $d \leftarrow -d$ и взаимозаменить $k \leftrightarrow l$. Возвратиться к шагу N3.
- N13 [Переключение областей.] Если $f = 0$, то установить $s \leftarrow 1 - s$ и возвратиться к N2. В противном случае сортировка завершена; если $s = 0$, то установить $(R_1, \dots, R_N) \leftarrow (R_{N+1}, \dots, R_{2N})$. (Если результат можно оставить в области (R_{N+1}, \dots, R_{2N}) , то последнее копирование необязательно.)

В этом алгоритме есть одна небольшая тонкость, которая объясняется в упр. 5.

Запрограммировать алгоритм N для машины MIX нетрудно, но основные сведения о его поведении можно получить и без построения всей программы. Если файл случаев, то в нем около $\frac{1}{2}N$ возрастающих отрезков, так как $K_i > K_{i+1}$ с вероятностью $\frac{1}{2}$; подробная информация о числе отрезков при несколько отличных предположениях была получена в п. 5.1.3. При каждом просмотре число отрезков сокращается вдвое (за исключением необычных случаев, таких, как ситуация, описанная в упр. 6). Таким образом, число просмотров, как правило, составляет около $\log_2 N$. При каждом просмотре мы должны переписать все N записей, и, как показано в упр. 2, большая часть времени затрачивается в шагах N3, N4, N5, N8, N9. Если считать, что равные ключи встречаются с малой вероятностью, то время, затрачиваемое во внутреннем цикле, можно охарактеризовать следующим образом:

Шаг	Операции	Время
N3	CMPA, JG, JE	3.5u
Либо {	N4 STA, INC	3u
	N5 INC, LDA, CMPA, JGE	6u
Либо {	N8 STX, INC	3u
	N9 DEC, LDX, CMPX, JGE	6u

Таким образом, при каждом просмотре на каждую запись затрачивается 12.5 единиц времени, и общее время работы асимптотически приближается к $12.5N \log_2 N$ как в среднем, так и в наихудшем случае. Это медленнее быстрой сортировки и не настолько лучше времени работы пирамидальной сортировки, чтобы оправдать вдвое больший расход памяти, так как асимптотическое время работы программы 5.2.3N равно $16N \log_2 N$.

В алгоритме N границы между отрезками полностью определяются "ступеньками вниз". Такой подход обладает тем возможным преимуществом, что исходные файлы с преобладанием возрастающего или убывающего расположения элементов могут обрабатываться очень быстро, но при этом замедляется основной цикл вычислений. Вместо проверок ступенек вниз можно принудительно установить длину отрезков, считая, что все отрезки исходного файла имеют длину 1, после первого просмотра все отрезки (кроме, возможно, последнего) имеют длину 2, ..., после k -го просмотра все отрезки (кроме, возможно, последнего) имеют длину 2^k . В отличие от "естественного" слияния в алгоритме N такой способ называется *простым* двухпутевым слиянием.

Алгоритм простого двухпутевого слияния очень напоминает алгоритм N—он описывается, по существу, той же блок-схемой; тем не менее методы достаточно отличаются друг от друга, и поэтому стоит записать весь алгоритм целиком.

Алгоритм S. (Сортировка простым двухпутевым слиянием.) Как и в алгоритме N, при сортировке записей R_1, \dots, R_N используются две области памяти.

- S1 [Начальная установка.] Установить $s \leftarrow 0$, $p \leftarrow 1$. (Смысл переменных s, i, j, k, l, d см. в алгоритме N. Здесь p —размер возрастающих отрезков, которые будут сливаться во время текущего просмотра; q и r —количества неслитых элементов в отрезках.)
- S2 [Подготовка к просмотру.] Если $s = 0$, то установить $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N$, $l \leftarrow 2N + 1$; если $s = 1$, то установить $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 0$, $l \leftarrow N + 1$. Затем установить $d \leftarrow 1$, $q \leftarrow p$, $r \leftarrow p$.
- S3 [Сравнение $K_i : K_j$.] Если $K_i > K_j$, то перейти к шагу S8.
- S4 [Пересылка R_i .] Установить $k \leftarrow k + d$, $R_k \leftarrow R_i$.
- S5 [Конец отрезка?] Установить $i \leftarrow i + 1$, $q \leftarrow q - 1$. Если $q > 0$, то возвратиться к шагу S3.
- S6 [Пересылка R_j .] Установить $k \leftarrow k + d$. Затем, если $k = l$, перейти к шагу S13; в противном случае установить $R_k \leftarrow R_j$.
- S7 [Конец отрезка?] Установить $j \leftarrow j - 1$, $r \leftarrow r - 1$. Если $r > 0$, возвратиться к шагу S6; в противном случае перейти к шагу S12.
- S8 [Пересылка R_j .] Установить $k \leftarrow k + d$, $R_k \leftarrow R_j$.
- S9 [Конец отрезка?] Установить $j \leftarrow j - 1$, $r \leftarrow r - 1$. Если $r > 0$, то возвратиться к шагу S3.

- S10 [Пересылка R_i .] Установить $k \leftarrow k + d$. Затем, если $k = l$, перейти к шагу S13; в противном случае установить $R_k \leftarrow R_i$.
- S11 [Конец отрезка?] Установить $i \leftarrow i + 1, q \leftarrow q - 1$. Если $q > 0$, то возвратиться к шагу S10.
- S12 [Переключение направления.] Установить $q \leftarrow p, r \leftarrow p, d \leftarrow -d$ и взаимозаменить $k \leftrightarrow l$. Возвратиться к шагу S3.
- S13 [Переключение областей.] Установить $p \leftarrow p + p$. Если $p < N$, то установить $s \leftarrow 1 - s$ и возвратиться к S2. В противном случае сортировка завершена; если $s = 0$, то установить

$$(R_1, \dots, R_n) \leftarrow (R_{N+1}, \dots, R_{2N}).$$

(Независимо от распределения исходного файла последнее копирование будет выполнено тогда и только тогда, когда значение $\lceil \log_2 N \rceil$ нечетно. Так что можно заранее предсказать положение отсортированного файла, и копирование, как правило, не требуется.) ■

Таблица 2

Сортировка простым двухпутевым слиянием															
503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	512	677	509	908	426	897	653	275	170	154	612	061	765	087
087	503	703	765	154	170	509	908	897	653	426	275	677	612	512	061
061	087	503	512	612	677	703	765	908	897	653	509	426	275	170	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Пример работы алгоритма см. в табл. 2. Довольно удивительно, что этот метод справедлив и тогда, когда N не является степенью 2; сливаемые отрезки не все имеют длину 2^k , тем не менее никаких явных мер предосторожности на случай таких исключений не предусмотрено! (См. упр. 8.) Проверки ступенек вниз заменены уменьшением переменных q и r и проверкой на равенство нулю. Благодаря этому время работы на машине MIX асимптотически приближается к $11N \log_2 N$ единицам, что несколько лучше значения, которого нам удалось добиться в алгоритме N.

На практике имеет смысл комбинировать алгоритм S с простыми вставками; вместо первых четырех просмотров алгоритма S можно простыми вставками отсортировать группы, скажем, из 16 элементов, исключив таким образом довольно расточительные вспомогательные операции, связанные со слиянием коротких файлов. Как мы уже видели в случае быстрой сортировки, такое комбинирование методов не влияет/на асимптотическое время работы, но дает тем не менее, немалую выгоду.

Рассмотрим теперь алгоритмы N и S с точки зрения структур данных. Почему нам необходима память под $2N$, а не под N записей? Причина относительно проста: мы работаем с четырьмя списками переменного размера (два "входных списка" и два "выходных списка" в каждом просмотре); при этом для каждой пары последовательно распределенных списков мы пользуемся стандартным понятием "встречного роста", обсуждавшимся в (п. 2.2.2. Но в любой момент времени половина памяти не используется, и после некоторого размышления становится ясно, что в действительности, для наших четырех списков следовало бы воспользоваться *связанным* распределением памяти. Если к каждой из N записей добавить поле связи, то все необходимое можно проделать, пользуясь алгоритмами слияния, которые производят простые манипуляции со связями и совсем не перемещают сами записи. Добавление N полей связи, как правило, выгоднее, чем добавление пространства памяти еще под N записей, а отказавшись от перемещения записей, мы можем также сэкономить и время. Итак, нам нужно рассмотреть алгоритм, подобный следующему.

Алгоритм L. (Сортировка посредством слияния списков.) Предполагается, что записи R_1, \dots, R_N содержат ключи K_1, \dots, K_N и "поля связи" L_1, \dots, L_N , в которых могут храниться числа от $-(N+1)$ до $(N+1)$. В начале и в конце файла имеются искусственные записи R_0 и R_{N+1} с полями связи L_0 и L_{N+1} . Этот алгоритм сортировки списков устанавливает поля связи таким образом, что записи оказываются связанными в возрастающем порядке. После завершения сортировки L_0 указывает на запись с наименьшим ключом; при $1 \leq k \leq N$ связь L_k указывает на запись, следующую за R_k , а если R_k — запись с наибольшим ключом, то $L_k = 0$. (См. формулы (5.2.1–9).)

В процессе выполнения этого алгоритма записи R_0 и R_{N+1} служат "головами" двух линейных списков, подписки которых в данный момент сливаются. Отрицательная связь означает конец подсписка, о котором известно, что он упорядочен; нулевая связь означает конец всего списка. Предполагается, что $N \geq 2$.

Через " $|L_s| \leftarrow p$ " обозначена операция "присвоить L_s значение p или $-p$, сохранив прежний знак L_s ". Такая операция легко реализуется на машине MIX, но, к сожалению, это не так для большинства

ЭВМ. Нетрудно изменить алгоритм, чтобы получить столь же эффективный метод и для большинства других машин.

L1 [Подготовить два списка.] Установить $L_0 \leftarrow 1$, $L_{N+1} \leftarrow 2$, $L_i \leftarrow -(i+2)$ при $l \leq i \leq N-2$ и $L_{N-1} \leftarrow L_N \leftarrow 0$. (Мы создали два списка, содержащие соответственно записи R_1, R_3, R_5, \dots и R_2, R_4, R_6, \dots ; отрицательные связи говорят о том, что каждый упорядоченный "подсписок" состоит всего лишь из одного элемента. Другой способ выполнить этот шаг,

Эта схема не является наиболее эффективной среди всех возможных, но она интересна тем, что показывает, что методы с частичными проходами рассматривались для поразрядной сортировки еще в 1946 г., хотя в литературе по слиянию они появились лишь около 1960 г.

Эффективная конструкция схем распределения с обратным чтением была предложена Э. Байесом [JACM, 11 (1968), 491–493]. Пусть дано $P + 1$ лент и S ключей; разделите ключи на P подфайлов, каждый из которых содержит $\lfloor S/P \rfloor$ или $\lceil S/P \rceil$ ключей, и применяйте эту процедуру рекурсивно к каждому подфайлу. Если $S < 2P$, то один подфайл должен состоять из единственного наименьшего ключа; его и следует записать на выводную ленту. (Общая конструкция с прямым порядком Р. М. Карпа, описанная в конце п. 5.4.4, включает этот метод как частный случай.)

Обратное чтение несколько усложняет слияние, поскольку оно обращает порядок отрезков. Соответствующий эффект имеется и в поразрядной сортировке. Результат оказывается устойчивым или "антиустойчивым" в зависимости от того, какой уровень достигнут в дереве. После поразрядной сортировки с обратным чтением, когда некоторые внешние узлы находятся на четных уровнях, а некоторые — на нечетных, для одних ключей относительный порядок различных записей с одинаковыми ключами будет *совпадать* с первоначальным порядком, но для других он будет *противоположен* исходному. (Ср. с упр. 6.)

Осциллирующая сортировка слиянием также имеет свою пару в этой двойственности. В *осциллирующей поразрядной сортировке* мы продолжаем разделять ключи, пока не достигнем подфайлов, содержащих только один ключ или достаточно малых, чтобы поддаваться внутренней сортировке; такие подфайлы сортируются и записываются на выводную ленту, затем процесс разделения возобновляется. Например, если имеются три рабочие ленты и одна выводная и если ключи являются двоичными числами, мы можем начать с того, что поместим ключи вида '0x' на ленту T1, а ключи '1x' на ленту T2. Если на ленте T1 окажется больше записей, чем емкость памяти, то вновь просматриваем ее и помещаем '00x' на T2 и '01x' на T3. Теперь, если подфайл '00x' достаточно короткий, производим внутреннюю сортировку его и выводим результат, а затем начинаем обработку подфайла '01x'. Подобный метод был назван Э. Х. Фрэндом "каскадной псевдопоразрядной сортировкой" [JACM, 3 (1956), 157–159]; более подробно его разработали Х. Нэглер [JACM, 6 (1959), 459–468], который дал ему красочное имя "метод двуглавого змия", и К. Х. Годетт [IBM Tech. Disclosure Bull., 12 (April, 1970), 1849–1853].

Превосходит ли поразрядная сортировка слияние? Одним важным следствием принципа двойственности является то, что *поразрядная сортировка обычно хуже сортировки слиянием*. Это связано с тем, что метод выбора с замещением дает сортировке слиянием определенное преимущество: нет очевидного пути так построить поразрядную сортировку, чтобы можно было использовать внутренние сортировки, включающие более одной емкости памяти за один раз. На самом деле осциллирующая поразрядная сортировка часто будет порождать подфайлы, несколько меньшие емкости памяти, так что ее схема распределения соответствует дереву со значительно большим числом внешних узлов, чем было бы при использовании слияния и выбора с замещением. Соответственно возрастает длина внешнего пути дерева (т. е. время сортировки). (См. упр. 5.3.1–33.)

Для внешней поразрядной сортировки существует, однако, одно важное применение. Предположим, например, что имеется файл, содержащий фамилии всех сотрудников большого предприятия в алфавитном порядке; предприятие состоит из 10 отделений, и требуется отсортировать этот файл по отделениям, *сохраняя* алфавитный порядок сотрудников в каждом отделении. Если файл длинный, то мы имеем дело именно той ситуацией, где следует применять стабильную поразрядную сортировку, так как число записей, принадлежащих каждому из 10 отделений, будет, вероятно, больше, чем число записей, которое было бы в начальных отрезках, полученных выбором с замещением. Вообще говоря, если диапазон ключей так мал, что набор записей с одинаковыми ключами более чем вдвое превысит оперативную память, то разумно использовать поразрядную сортировку.

Мы видели в п. 5.2.5, что на некоторых высокоскоростных ЭВМ *внутренняя* поразрядная сортировка предпочтительнее слияния, поскольку "внутренний цикл" поразрядной сортировки обходится без сложных переходов. Если внешняя память очень быстрая, то для таких машин может оказаться проблемой проводить слияние данных с такой скоростью, чтобы успеть за оборудованием ввода/вывода. Поэтому в подобной ситуации поразрядная сортировка, возможно, лучше слияния, особенно если известно, что ключи равномерно распределены.

Упражнения

1. [20] Ближе к началу п. 5.4 было определено общее сбалансированное слияние на T лентах с параметром P , $1 \leq P < T$. Покажите, что оно соответствует поразрядной сортировке, использующей систему числения со смешанным основанием.

2. [M28] В тексте схематически представлена многофазная поразрядная сортировка 21 ключа! Обобщите ее на случай F_n ключей; объясните, какие ключи и на какой ленте оказываются в конце каждой фазы. [Указание: рассмотрите систему счисления, использующую числа Фибоначчи; упр. 1.2.8-34.]
3. [M40] Распространите результаты упр. 2 на многофазную поразрядную сортировку с четырьмя или большим количеством лент (ср. с упр. 5.4.2–10).
4. [M20] Докажите, что схема распределения Эшенхерста служит наилучшим способом сортировки 10 ключей на четырех лентах без обратного чтения в том смысле, что соответствующее дерево имеет минимальную длину внешнего пути среди всех "сильных 4-fifo деревьев". (Таким образом, это, по существу, наилучший метод, если не учитывать время перемотки.)
5. [15] Нарисуйте 4-lifo дерево, соответствующее поразрядной сортировке Мочли с обратным чтением 10 ключей.
- >6. [20] Некоторый файл содержит двухразрядные ключи 00, 01, ..., 99. После выполнения поразрядной сортировки Мочли по цифре единиц мы можем повторить ту же схему по цифре десятков, поменяв ролями ленты T_2 и T_4 . В каком порядке в конце концов окажутся ключи на T_2 ?
7. [21] Применим ли принцип двойственности также и к файлам на нескольких бобинах?

5.2.5. Сортировка с двумя лентами

Для того чтобы при выполнении слияния не было чрезмерного движения лент, необходимы три ленты. Интересно подумать о том, как можно разумным образом выполнить внешнюю сортировку с использованием только двух лент.

В 1956 г. Г. Б. Демут предложил некий метод, представляющий собой комбинацию выбора с замещением и сортировки методом пузырька. Предположим, что исходные данные занимают ленту T_1 , и начнем с того, что прочитаем в память $P + 1$ записей. Теперь выведем запись с наименьшим ключом на ленту T_2 и заменим ее следующей исходной записью. Продолжаем выводить записи, ключ которых в текущий момент наименьший в памяти, сохраняя дерево выбора или приоритетную очередь из $P + 1$ элементов. Когда ввод наконец исчерпается, в памяти окажутся наибольшие P ключей файла; выведем их в возрастающем порядке. Теперь перемотаем обе ленты и повторим этот процесс, читая с T_2 и записывая на T_1 ; каждый такой проход помещает еще по крайней мере P записей на свои места. В программу можно встроить простую проверку для определения момента, когда весь файл станет упорядоченным. Потребуется не более $\lceil (N - 1)/P \rceil$ проходов.

Минутное размышление показывает, что каждый проход этой процедуры эквивалентен P последовательным проходам сортировки методом пузырька (алгоритм 5.2.2B)! Если элемент имеет P или более инверсий, то при вводе он окажется меньше всех элементов в дереве и поэтому будет немедленно выведен (потеряв, таким образом, P инверсий). Если элемент имеет менее P инверсий, то он попадает в дерево выбора и будет выведен раньше всех больших ключей (потеряв, таким образом, все свои инверсии). Если $P = 1$, то происходит то же самое, что и в методе пузырька, по теореме 5.2.21.

Общее число проходов будет, следовательно, равно $\lceil I/P \rceil$, где I — максимальное число инверсий любого элемента. По теории, развитой в п. 5.2.2, среднее значение I есть $N - \sqrt{\pi N/2} + (2/3) + O(1/\sqrt{N})$.

Если файл не слишком сильно превосходит размер оперативной памяти или если он первоначально почти упорядочен, то эта сортировка методом пузырька P -го порядка будет довольно быстрой; в действительности ее можно предпочесть даже в том случае, когда имеются дополнительные лентопротяжные устройства, так как весь процесс сортировки может закончиться раньше, чем оператор успеет установить третью ленту! С другой стороны, она будет работать весьма медленно над довольно большими файлами со случайным расположением элементов, так как время ее работы приблизительно пропорционально N^2 .

Посмотрим, как реализуется этот метод для 100000 записей в примере из п. 5.4.6. Нам нужно разумно выбрать P , чтобы учесть межблочные промежутки при совмещении операций чтения и записи с вычислениями. Так как в примере предполагается, что каждая запись имеет длину 100 литер, а 100000 литер заполняют память, то у нас будет место для двух буферов ввода и двух буферов вывода размера B , если выбрать значения P и B , такие, что

$$100(P + 1) + 4B = 100000. \quad (1)$$

Если использовать обозначения п. 5.4.6, то приблизительное время работы каждого прохода выражается как

$$NC\omega\tau(1 + \rho), \quad \omega = (B + \gamma)/B. \quad (2)$$

Поскольку число проходов обратно пропорционально P , мы хотим выбрать такое B , кратное 100, которое минимизирует величину ω/P . Элементарный анализ показывает, что минимум достигается,

когда B равно приблизительно $\sqrt{24975\gamma} + \gamma^2 - \gamma$. Поэтому мы выбираем $B = 3000$, $P = 879$. Положив в приведенных выше формулах $N = 100000$, получаем, что число проходов $\lceil I/P \rceil$ будет около 114, а оценка общего времени решения составляет примерно 8.57 ч (предполагая для удобства, что исходные данные и окончательный вывод также имеют $B = 3000$). Здесь представлен случай, когда данные занимают около 0.44 бобины; полная бобина потребовала бы примерно в пять раз больше времени. Можно произвести некоторые улучшения, предусмотрев в алгоритме периодические прерывания и пересылку записей с наибольшими ключами на вспомогательную ленту, которая затем снимается, несколько эти записи просто копируются туда и обратно после того, как они уже оказались на своих местах.

Применение быстрой сортировки. Еще одним методом внутренней сортировки, который проходит данные почти последовательно, является обменная сортировка с разделением или быстрая сортировка (алгоритм 5.2.2Q) Можно ли ее приспособить к двум лентам? [N. В; Yoash, *CACM*, 8 (1965), 649.]

Нетрудно увидеть как можно сделать это, воспользовавшись обратным чтением. Предположим, что две ленты помечены 0 и 1, и представим, что файл располагается следующим образом:

Picture: Расположение файла на ленте, стр. 419

Каждая лента выступает в качестве стека. Две ленты вместе, используемые как представлено здесь, дают возможность считать файл линейным списком, в котором мы можем перемещать текущую позицию влево или вправо, копируя элементы из одного стека в другой. Следующие рекурсивные подпрограммы определяют соответствующую процедуру сортировки.

- **SORT00** [отсортировать верхний подфайл с ленты 0 и вернуть его на ленту 0]. Если подфайл помещается в оперативную память, то применить к нему внутреннюю сортировку и затем вернуть его на ленту. В противном случае выбрать одну запись R из подфайла; пусть ее ключом будет K . Читая ленту 0 в обратном направлении, копировать все записи, ключи которых $> K$, получая таким образом новый "верхний" подфайл на ленте 1. Теперь, читая ленту 0 в прямом направлении, копировать все записи с ключами, равными K , на ленту 1. Затем, вновь читая ленту 0 в обратном направлении, копировать все записи с ключами $< K$ на ленту 1. Выполнить **SORT10** над ключами $< K$, затем скопировать ключи, равные K , на ленту 0 и, наконец, выполнив **SORT10** над ключами $> K$, завершить сортировку.
- **SORT01** [отсортировать верхний подфайл с ленты 0 и записать его на ленту 1]. Аналогично **SORT00**, но последнее обращение к "**SORT10**" заменено на "**SORT11**", за которым следует копирование ключей $\leq K$ на ленту 1.
- **SORT10** [отсортировать верхний подфайл с ленты 1 и записать его на ленту 0]. Такая же, как **SORT01**, но меняются местами 0 и 1, а также операторы отношений $<$ и $>$.
- **SORT11** [отсортировать верхний подфайл с ленты 1 и вернуть его на ленту 1]. Такая же, как **SORT00**, но меняются местами 0 и 1, а также отношения $<$ и $>$. Можно без труда справиться с рекурсивной природой этих процедур, записывая подходящую управляющую информацию на ленты.

Если считать, что данные находятся в случайном порядке и вероятность равных ключей пренебрежимо мала, то можно оценить время работы этого алгоритма следующим образом. Пусть M — число записей, помещающихся в оперативной памяти. Пусть X_N — среднее число записей, читаемых во время применения **SORT00** или **SORT11** к подфайлу из N записей, где $N > M$, и пусть Y_N — соответствующая величина для **SORT01** и **SORT10**. Тогда имеем:

$$\begin{aligned} X_N &= \begin{cases} 0, & \text{если } N \leq M, \\ 3N + 1 + \frac{1}{N} \sum_{0 \leq k < N} (Y_k + Y_{N-1-k}), & \text{если } N > M, \end{cases} \\ Y_N &= \begin{cases} 0, & \text{если } N \leq M, \\ 3N + 2 + \frac{1}{N} \sum_{0 \leq k < N} (Y_k + X_{N-1-k} + k), & \text{если } N > M. \end{cases} \end{aligned} \quad (3)$$

Решение этих рекуррентных соотношений (см. упр. 2) показывает, что общий объем информации, читаемой с ленты в течение фаз внешнего разделения, в среднем равен $6\frac{2}{3}N \ln N + O(N)$ при $N \rightarrow \infty$. Мы также знаем из формулы (5.2.2–25), что среднее число фаз внутренней сортировки будет равно $2(N+1)/(M+2) - 1$.

Если мы применим этот анализ к примеру 100000 записей, рассмотренному в п. 5.4.6, причем воспользуемся буферами по 25000 литер и будем считать, что время сортировки подфайла из

$n \leq M = 1000$ записей равно $2nC\omega\tau$, то получим среднее время сортировки, приблизительно равное 103 мин (включая, как в схеме А, окончательную перемотку). Итак, метод быстрой сортировки в среднем неплох; но, конечно, в *наихудшем* случае он ужасен и уступает даже методу пузырька, обсуждавшемуся выше.

Поразрядная сортировка. Обменную поразрядную сортировку (алгоритм 5.2.2R) можно аналогичным образом приспособить для сортировки с двумя лентами, так как он очень похож на быструю сортировку. В качестве трюка, который позволил применить оба эти метода, использовалась идея чтения файла более чем один раз—то, чего мы никогда не делали в предыдущих алгоритмах для лент.

С помощью того же трюка можно осуществить обычную поразрядную сортировку на двух лентах "сначала-по-младшей-цифре". Имея исходные данные на $T1$, копируем на $T2$ все записи, ключ которых в двоичной системе оканчивается на 0; затем после перемотки $T1$ читаем ее вновь, копируя записи с ключами, оканчивающимися на 1. Теперь перематываются обе ленты и выполняется аналогичная пара проходов, но с заменой $T1$ на $T2$ и использованием *предпоследней* двоичной цифры. В этот момент $T1$ будет содержать все записи с ключами $(\dots 00)_2$, за которыми следуют записи с ключами $(\dots 01)_2$, затем $(\dots 10)_2$, затем $(\dots 11)_2$. Если ключи имеют размер b битов, нам потребуется, чтобы завершить сортировку, только $2b$ проходов по всему файлу.

Подобную поразрядную сортировку можно применять только к *старшим* b битам ключа для некоторого разумно выбранного числа b ; таким образом, число инверсий уменьшится примерно в 2^b раз, если ключи были равномерно распределены; и тогда несколько проходов P -путевой сортировки методом пузырька позволят завершить работу.

Новый, но несколько более сложный подход к распределяющей сортировке с двумя лентами предложили А. И. Никитин и Л. И. Шолмов [*Кибернетика*, 2, 6 (1966), 79–84]. Имеются счетчики числа ключей по одному на каждую возможную конфигурацию старших битов, и на основе этих счетчиков строятся искусственные ключи $\kappa_1, \kappa_2, \dots, \kappa_M$ так, чтобы для каждого i число действительных ключей, лежащих между κ_i и κ_{i+1} , было между заранее определенными границами P_1 и P_2 . Таким образом, M лежит между $\lceil N/P_1 \rceil$ и $\lceil N/P_2 \rceil$. Если счетчики старших битов не дают достаточной информации для определения таких $\kappa_1, \kappa_2, \dots, \kappa_M$, то делается еще один или несколько проходов для подсчета частоты конфигураций менее значащих битов при некоторых конфигурациях старших битов. После того как таблица искусственных ключей построена, $2\lceil \log_2 M \rceil$ проходов будет достаточно для завершения сортировки. (Этот метод требует пространства памяти, пропорционального N , и поэтому не может использоваться для внешней сортировки при $N \rightarrow \infty$. На практике мы не станем использовать этот метод для файлов на нескольких бобинах, и, следовательно, M будет сравнительно невелико, так что таблица искусственных ключей легко поместится в памяти.)

Имитация дополнительных лент. Ф. К. Хенни и Р. Э. Стирнз изобрели общий метод имитации k лент всего на двух лентах, причем таким образом, что требуемое суммарное перемещение ленты возрастает всего лишь в $O(\log L)$ раз, где L —максимальное расстояние, которое нужно пройти на любой одной ленте [*JACM*, 13 (1966), 533–546]. Их построение в случае сортировки можно слегка упростить, что и сделано в следующем методе, предложенном Р. М. Карпом.

Будем имитировать обычное сбалансированное слияние на четырех лентах, используя две ленты: $T1$ и $T2$. На первой из них (т. е. на $T1$) содержимое имитируемых лент хранится таким способом, как изображено на рис. 86; представим себе, что данные записаны на четырех дорожках по одной для каждой имитируемой ленты. (В действительности лента не имеет таких

Picture: Рис. 86. Разбивка ленты $T1$ в конструкции Хенни и Стирнза

дорожек; мы мыслим блоки 1, 5, 9, 13, ... как дорожку 1, блоки 2, 6, 10, 14, ... как дорожку 2 и т. д.) Другая лента ($T2$) используется только для вспомогательного хранения, чтобы помочь в выполнении перестановок на $T1$.

Блоки на каждой дорожке разделяются на *зоны*, содержащие соответственно 1, 2, 4, 8, ..., 2^k , ... блоков. Зона k на каждой дорожке либо заполнена точно 2^k блоками данных, либо целиком пуста. Например, на рис. 86 на дорожке 1 данные содержатся в зонах 1 и 3; на дорожке 2—в зонах 0, 1 и 2; на дорожке 3—в зонах 0 и 2; на дорожке 4—в зоне 1, а все остальные зоны пусты.

Предположим, что мы сливаем данные с дорожек 1 и 2 на дорожку 3. В оперативной памяти ЭВМ находятся два буфера, используемые двухпутевым слиянием для ввода, а также третий буфер—для вывода. Когда буфер ввода для дорожки 1 станет пустым, можно заполнить его следующим образом: найти первую непустую зону дорожки 1, скажем зону k , и скопировать первый ее блок в буфер ввода, затем скопировать остальные $2^k - 1$ блоков данных на $T2$ и переместить их в зоны 0, 1, ..., $k - 1$ дорожки 1. (Теперь зоны 0, 1, ..., $k - 1$ заполнены, зона k пуста.) Аналогичная процедура

используется для заполнения буфера ввода для дорожки 2, как только он станет пустым. Когда буфер вывода подготовлен для записи на дорожку 3, мы обращаем этот процесс, т. е. просматриваем $T1$ пока не найдется первая *пустая* зона на дорожке 3, скажем зона k , и в то же время копируем данные из зон $0, 1, \dots, k-1$ на $T2$. Данные на $T2$, к которым присоединяется содержимое буфера вывода, используются теперь для заполнения зоны k на дорожке 3.

Для этой процедуры мы должны уметь писать в середину ленты $T1$, не разрушая последующую информацию на этой ленте. Как и в случае осциллирующей сортировки с прямым чтением (п. 5.4.5), можно без опасений выполнять это действие, если принять меры предосторожности.

Поскольку просмотр до зоны k выполняется только один раз за каждые 2^k шагов, то, чтобы переписать $2^i - 1$ блоков с дорожки 1 в память, потребуется переместить ленту на $\sum_{v \leq k < l} 2^{l-1-k} \cdot c \cdot 2^k = cl2^{l-1}$, где c — некоторая константа. Таким образом, каждый проход слияния требует $O(N \log N)$ шагов. Так как в сбалансированном слиянии имеется $O(\log N)$ проходов, то общее время работы будет $O(N(\log N)^2)$, что асимптотически значительно лучше, чем наихудший случай для быстрой сортировки.

Но на самом деле этот метод оказывается почти бесполезным, если применяется для сортировки 100000 записей из примера п. 5.4.6, поскольку информация, которая должна размещаться на ленте $T1$, не уместится на одной бобине ленты. И даже если мы пренебрежем этим фактом и будем исходить из самых оптимистических предположений относительно совмещения чтения/записи/вычислений, относительно длин межблочных промежутков и т. д., то найдем, что для выполнения сортировки потребуется около 37 ч! Итак, этот метод представляет чисто академический интерес: константа в $O(N(\log N)^2)$ слишком велика, чтобы метод был удовлетворителен для практических значений N .

Одноленточная сортировка. Можно ли обойтись всего одной лентой? Нетрудно видеть, что сортировку методом пузырька P -го порядка можно преобразовать в одноленточную сортировку, но результат будет ужасен.

Г. Б. Демут [Ph. D. thesis (Stanford University, 1956), 85] сделал наблюдение, что в вычислительной машине с ограниченной оперативной памятью нельзя, уменьшить число инверсий перестановки больше, чем на ограниченную величину, после просмотра ленты на ограниченное расстояние; следовательно, любой алгоритм сортировки с одной лентой потребует в среднем по крайней мере dN^2 единиц времени (где d — некоторая положительная константа, зависящая от конфигурации ЭВМ).

Р. М. Карп нашел очень интересный подход к исследованию этой темы, обнаружив то, что, по существу, является *оптимальным* способом сортировки с одной лентой. При обсуждении алгоритма Карпа удобно следующим образом переформулировать задачу: *как быстрее всего перевезти людей между этажами, если работает только один лифт?*

Рассмотрим здание с n этажами; помещение каждого этажа рассчитано на c человек. В этом здании нет ни окон, ни дверей, ни лестниц, но все же есть лифт, который может останавливаться на любом этаже. В здании находятся cn человек, и ровно c из них хотят попасть на каждый отдельный этаж. В лифт вмещается самое большее b человек, и он затрачивает одну единицу времени для перемещения с этажа i на этаж $i+1$. Мы хотели бы найти быстрейший способ переместить всех людей на нужные этажи, если требуется, чтобы лифт начал и закончил свое движение на первом этаже.

Нетрудно заметить связь между этой задачей и одноленточной сортировкой: люди — это записи, здание — лента, этажи — отдельные блоки на ленте, а лифт — оперативная память ЭВМ. Действиям программ для ЭВМ свойственна большая гибкость, чем действиям лифтера (она может, например, создавать двойников или, разрезав человека на две части, оставить их на время на разных этажах и т. д.); но в приводимом ниже алгоритме задача решается быстрейшим мыслимым способом без выполнения таких операций. Алгоритм Карпа использует следующие два вспомогательных массива:

$$\begin{aligned} u_k, 1 \leq k \leq n : & \quad \text{число людей на этажах } \leq k, \text{ стремящихся попасть на этажи } > k \\ d_k, 1 \leq k \leq n : & \quad \text{число людей на этажах } \geq k, \text{ стремящихся попасть на этажи } < k \end{aligned} \quad (4)$$

. Когда лифт пуст, мы всегда имеем $u_k = d_{k+1}$ при $1 \leq k < n$, так как на каждом этаже находятся c человек; количество людей, направляющихся с этажей $\{1, \dots, k\}$ на этажи $\{k+1, \dots, n\}$, должно равняться числу людей, стремящихся переправиться в обратном направлении. По определению $u_n = d_1 = 0$.

Ясно, что лифт должен сделать по крайней мере $\lceil u_k/b \rceil$ рейсов с этажа k на этаж $k+1$ при $1 \leq k < n$, так как только b пассажиров могут подняться за один рейс. Аналогично, он должен сделать не менее $\lceil d_k/b \rceil$ рейсов с этажа k на этаж $k-1$. Следовательно, лифту потребуется по крайней мере

$$\sum_{1 \leq k \leq n} (\lceil u_k/n \rceil + \lceil d_k/b \rceil) \quad (5)$$

единиц времени при любом правильном графике работы. Карп обнаружил, что эта нижняя граница действительно достижима, если u_1, \dots, u_{n-1} ненулевые.

Теорема К. Если $u_k > 0$ при $1 \leq k < n$, то существует график работы лифта, при котором все люди доставляются на свои этажи за минимальное время (5).

5.2.6. Диски и барабаны

До сих пор мы рассматривали ленты как единственное средство для внешней сортировки, однако нередко в нашем распоряжении оказываются и другие типы устройств массовой памяти с более гибкими возможностями. Хотя такие запоминающие устройства "большого объема" или "запоминающие устройства с прямым доступом" весьма многообразны, можно выделить следующие общие свойства:

- i) Для доступа к любой определенной части хранимой информации не требуется очень много времени.
- ii) Блоки, содержащие последовательные слова, могут быстро передаваться между внутренней (оперативной) и внешней памятью.

Магнитная лента удовлетворяет (ii), но не (i), поскольку переход ленты от одного конца к другому занимает много времени. Некоторые устройства удовлетворяют (i), но не (ii); примером может служить память большого объема на ферритовых сердечниках,

Picture: Рис. 89. Пакет дисков

в которой время доступа к каждому слову примерно в десять раз превышает время доступа к внутренней памяти.

Каждое внешнее запоминающее устройство имеет свои характерные особенности, которые следует тщательно изучить, прежде чем писать для него большие программы; однако технология меняется так быстро, что здесь не удастся сколько-нибудь подробно обсудить все существующие разновидности оборудования. Поэтому мы рассмотрим лишь некоторые типичные запоминающие устройства и на них проиллюстрируем продуктивные подходы к задаче сортировки.

Одним из наиболее распространенных типов внешних запоминающих устройств, удовлетворяющих (i) и (ii), является дисковый файл или модуль с пакетом дисков (рис. 89). Данные хранятся на нескольких быстро вращающихся круглых дисках, покрытых магнитным материалом; для записи или выборки информации используется держатель головок в виде гребешка, содержащий одну или несколько "головок чтения/записи" для каждой поверхности диска. Каждая поверхность делится на концентрические кольца, называемые *дорожками* или *треками*, так что за время одного оборота диска под головками чтения/записи проходит целая дорожка. Держатель головок может перемещаться в двух направлениях—внутрь или наружу, передвигая головки чтения/записи от дорожки к дорожке, но это движение требует времени. Множество дорожек, которые могут быть прочитаны или записаны без перемещения держателя головок, называется *цилиндром*. Например, на рис. 89 показан дисковый файл, который имеет по одной головке чтения/записи на каждую поверхность; пунктирными линиями обозначен один из цилиндров, состоящий из всех дорожек, просматриваемых в настоящий момент головками.

Чтобы сделать наши рассуждения более конкретными, рассмотрим гипотетическое дисковое устройство MIXTES, для которого

$$\begin{aligned} 1 \text{ дорожка} &= 5000 \text{ литер}, \\ 1 \text{ цилиндр} &= 20 \text{ дорожек}, \\ 1 \text{ дисковое устройство} &= 200 \text{ цилиндров}. \end{aligned}$$

Такое дисковое устройство содержит 20 миллионов литер, т. е. чуть меньше того объема данных, который можно записать на одну магнитную ленту. На некоторых машинах дорожки вблизи центра содержат меньше литер, чем дорожки ближе к краю. От этого программирование значительно усложняется, но MIXTES, к счастью, не создает таких проблем.

Время, необходимое для чтения или записи на дисковый файл, представляет, по существу, сумму трех величин:

- Время поиска (время, затрачиваемое на перемещение держателя головок к нужному цилиндру).
- Время ожидания (задержка, связанная с вращением диска, пока головка чтения/записи не достигнет нужного места).
- Время передачи (задержка, связанная с вращением диска, пока данные проходят под головками).

На устройствах MIXTES время поиска для перехода от цилиндра i к цилиндру j равно $25 + \frac{1}{2}|i - j|$ мс. Если i и j —случайно выбранные целые числа между 1 и 200, то среднее значение

$|i - j|$ равно $2 \binom{201}{3} / 200^2 \approx 66.7$, т. е. среднее время поиска составляет приблизительно 60 мс. Диски MIXTES совершают один оборот за 25 мс, так что время ожидания равно в среднем 12.5 мс. Время передачи n литер есть $(n/5000) \times 25 \text{ мс} = 5n \text{ мкс}$. (Это примерно в $3\frac{1}{3}$ раза быстрее, чем скорость передачи для лент MIXT, использованных в примерах п. 5.4.6.)

Таким образом, основные различия между дисками MIXTES и лентами MIXT, касающиеся сортировки, следующие:

- а) На лентах возможен только последовательный доступ к данным.
- б) Отдельная операция с диском, как правило, сопряжена со значительно большими накладными расходами (время поиска + время ожидания в сравнении со стартовыми временами).
- в) Скорость передачи у диска больше.

Используя для лент разумные схемы слияния, мы могли до некоторой степени скомпенсировать недостаток (а). Теперь у нас иная цель — нам нужно найти такие рациональные алгоритмы сортировки на дисках, в которых компенсируется недостаток (б).

Как сократить время ожидания? Рассмотрим сначала задачу минимизации задержек, вызываемых тем, что в тот момент, когда мы хотим начать команду ввода/вывода, диск не всегда находится в подходящей позиции. Нельзя заставить диск вращаться быстрее, но все-таки можно прибегнуть к разным уловкам, которые уменьшат или даже полностью устранят время ожидания. Несомненно, поможет добавление еще нескольких держателей головок, но это весьма дорогостоящая модификация оборудования. Вот несколько "программистских" идей.

- 1) Если мы читаем или записываем за один раз несколько дорожек одного цилиндра, то тем самым устраняем время ожидания (и время поиска) для всех дорожек, кроме первой. Вообще зачастую можно таким образом синхронизовать вычисления с вращением диска, что при выполнении последовательности команд ввода/вывода не будет задержек из-за ожидания.
- 2) Рассмотрим задачу чтения половины дорожки данных (рис. 90): если команда чтения выдается, когда головка находится в точке A , то задержка на ожидание отсутствует, и общее время чтения равно времени передачи, т.е. $\frac{1}{2} \times 25 \text{ мс}$. Если команда начинается, когда головка находится в точке B , то требуется $\frac{1}{4}$ оборота для ожидания и $\frac{1}{2}$ для передачи; в итоге имеем $\frac{3}{4} \times 25 \text{ мс}$. Наиболее интересен случай, когда головка первоначально находится в точке C : имея соответствующее оборудование и программное обеспечение, нам не придется терять $\frac{3}{4}$ оборота на ожидание. Можно немедленно начать чтение во вторую половину буфера ввода, затем после паузы в $\frac{1}{2} \times 25 \text{ мс}$ можно возобновить чтение в первую половину буфера, так что команда будет завершена, когда мы снова попадем в точку C . Поступая

Picture: Рис. 90. Анализ времени ожидания при чтении половины дорожки

таким образом, можно гарантировать, что общее время на ожидание+передачу никогда не превзойдет времени одного оборота независимо от начального положения диска. Среднее время ожидания уменьшается этой схемой с половины оборота до $\frac{1}{2}(1 - x^2)$ оборота, если читается или записывается доля x дорожки ($0 < x \leq 1$). Если читается или записывается целая дорожка ($x = 1$), то этот метод позволяет полностью устранить ожидание.

Барабаны: случай, когда поиск не нужен. На некоторых устройствах внешней памяти установлено по одной головке чтения/записи для каждой дорожки, и поэтому время поиска полностью устранено. Если на таком устройстве используется метод, продемонстрированный на рис. 90, то как время поиска, так и время ожидания сведены к нулю, при условии что мы всегда читаем или записываем всю дорожку целиком. В этой идеальной ситуации время передачи является единственным ограничительным фактором.

Рассмотрим вновь пример из п. 5.4.6, в котором сортируются 100000 записей по 100 литер каждая с внутренней памятью в 100000 литер. Весь объем сортируемых данных занимает половину диска MIXTES. Обычно невозможно одновременно читать и писать на одном дисковом устройстве; поэтому предположим, что имеются два диска, так что чтение и запись можно совместить. Пока будем считать, что "диски" — это фактически барабаны, содержащие 4000 дорожек по 5000 литер каждая, и никакого времени поиска не требуется.

Какой алгоритм сортировки следует использовать? Естественно выбрать метод слияния; другие методы внутренней сортировки не столь хороши в применении к дискам, за исключением, возможно, поразрядных методов, но рассуждения в п. 5.4.7 показывают, что поразрядная сортировка обычно

хуже слияния, если речь идет не о каких-нибудь частных приложениях. (Нетрудно видеть, что теорема двойственности, сформулированная в этом пункте, применима к дискам точно так же, как и к лентам.)

Чтобы начать сортировку слиянием, можно использовать выбор с замещением с двумя буферами ввода по 5000 литер и двумя буферами вывода по 5000 литер, фактически можно свести их к *трем* буферам по 5000 литер, если замещать записи в текущем буфере ввода на записи, выводимые из дерева выбора. Остается еще 85000 литер (850 записей) для дерева выбора, так что один проход по данным нашего примера позволит сформировать около 60 начальных отрезков (см. формулу (5.4.6–3)). Этот проход занимает лишь около 50 с, если предположить, что скорость внутренней обработки достаточно высока, чтобы успеть за вводом/выводом (каждые 500 мкс в буфер вывода должна передаваться запись). Если же сортируемые исходные данные находятся на ленте MIXT, а не на барабане, то этот проход будет выполняться медленнее, и время будет зависеть от скорости ленты.

Нетрудно видеть, что в случае двух барабанов и при чтении/записи полными дорожками общее время передачи для P -путевого слияния уменьшается с увеличением P . К сожалению, мы не можем выполнить одно только 60-путевое слияние всех начальных отрезков, так как в памяти нет места для 60 буферов. (Если размер буферов будет меньше 5000 литер, то появится нежелательное время ожидания.) Если мы выполняем P -путевое слияние, переписывая все данные с одного барабана на другой таким образом, что чтение и запись совмещены, то число проходов слияния равно $\lceil \log_P 60 \rceil$; следовательно, мы можем закончить работу за два прохода, если $8 \leq P \leq 59$. С уменьшением P уменьшается объем внутренних вычислений, поэтому выберем $P = 8$; если будет образовано 65 начальных отрезков, мы выберем $P = 9$. Если будет образовано 82 или больше начальных отрезков, мы можем взять $P = 10$, но так как имеется место только для 18 буферов ввода и двух буферов вывода, то появится возможность задержек при слиянии (см. алгоритм 5.4.6F); в таком случае, вероятно, лучше выполнить два частичных прохода над небольшой долей данных, сократив число начальных отрезков до 81 или меньше. При наших предположениях каждый проход слияния займет около 50 с, так что вся сортировка в этой идеальной ситуации будет завершена за $2\frac{1}{2}$ мин (плюс несколько секунд на инициализацию и другие вспомогательные действия). Это в шесть раз быстрее, чем наилучшая из сортировок с шестью лентами, рассмотренных в п. 5.4.6. Причинами этого ускорения являются увеличенная скорость передачи данных между внешней и внутренней памятью (она в $3\frac{1}{2}$ раза выше), более высокий порядок слияния (мы не можем осуществить восьмипутевое слияние на лентах не имея девяти или большего числа лент) и то, что выводные данные остаются на диске (нет необходимости в заключительной перемотке и т. д.). Если требуется, чтобы исходные данные и отсортированный результат были на лентах MIXT, а барабаны используются только для слияния, то такая сортировка займет около 8.2 мин.

Если вместо двух барабанов имеется только один, то время ввода/вывода увеличится вдвое, поскольку чтение/запись придется выполнять по отдельности. (На самом деле операции ввода/вывода займут в *три раза* больше времени, поскольку запись будет идти поверх исходных данных; в таких случаях целесообразно за каждой операцией записи выполнять операцию контрольного чтения, чтобы не потерять необратимо каких-нибудь исходных данных, если только оборудование не обеспечивает автоматической проверки записанной информации.) Однако часть этого дополнительного времени можно возратить, поскольку мы можем использовать частичные проходы, при которых одни записи обрабатываются чаще других. В случае двух барабанов требуется, чтобы все данные обрабатывались четное или нечетное число раз, но в случае одного барабана можно использовать более общие схемы слияния.

Мы видели в п. 5.4.4, что схемы слияния можно изображать с помощью деревьев и что время передачи, соответствующее схеме слияния, пропорционально длине внешнего пути дерева. В качестве схем эффективного слияния на лентах можно использовать лишь вполне определенные деревья ("T-lifo" или "сильные T-fifo"), потому что в процессе слияния некоторые отрезки оказываются "спрятанными" в середине ленты. Но *при использовании дисков или барабанов пригодны любые деревья*, если только степени их внутренних узлов не слишком велики (т. е. согласуются с наличным объемом внутренней памяти).

Следовательно, время передачи можно минимизировать, если выбрать дерево с минимальной длиной внешнего пути, такое, как полное P -арное дерево, где P —самое большое, какое возможно. По формуле (5.4.4–9) длина внешнего пути такого дерева с S внешними узлами (листьями) равна

$$qS - \lfloor (P^q - S)/(P - 1) \rfloor, \quad q = \lceil \log_P S \rceil. \quad (1)$$

Особенно просто строится алгоритм, который осуществляет слияние в соответствии со схемой полного P -арного дерева. (См., например, рис. 91, на котором показан случай $P = 3$, $S = 6$.) Сначала

мы добавляем, если необходимо, "фиктивные отрезки", чтобы сделать $S \equiv 1 \pmod{P-1}$; затем объединяем отрезки в соответствии с дисциплиной "первым включается —

Picture: Рис. 92. Полное тернарное дерево с шестью листьями...

первым исключается", сливая на каждом этапе P самых "старых" отрезков в начале очереди в один отрезок, помещаемый в конец.

Полные P -арные деревья дают оптимальную схему, если все отрезки имеют равную длину, но часто результат может быть еще лучше, если некоторые отрезки длиннее других. Оптимальную схему для этой общей ситуации можно без труда построить с помощью метода Хаффмэна (упр. 2.3.4.5–10), который на языке слияния формулируется так: "сначала добавьте $(1-S) \bmod (P-1)$ фиктивных отрезков длины 0, затем многократно сливайте P кратчайших из имеющихся отрезков, пока не останется один отрезок". Если все начальные отрезки имеют одинаковую длину, то этот метод сводится к описанной выше дисциплине.

В нашем примере со 100000, записей мы можем выполнять 9-путевое слияние, так как в памяти поместятся 18 буферов ввода и два буфера вывода, и в алгоритме 5.4.6F будет достигнуто полное совмещение вычислений. Полное 9-арное дерево с 60 листьями соответствует схеме слияния с $1\frac{29}{30}$ прохода, если все начальные отрезки имеют одинаковую длину. Общее время сортировки с одним барабаном и с использованием "контрольного чтения" после каждой записи становится, таким образом, равным 7.4 мин. Увеличивая P , можно немного уменьшить это время, но ситуация весьма запутанная, поскольку не исключается задержка чтения, так как буферы могут оказаться слишком полными или слишком пустыми.

Влияние времени поиска. Предыдущее обсуждение показывает, что для барабанов относительно легко сконструировать "оптимальную" схему слияния, поскольку время поиска и время ожидания можно свести на нет. Но если используются диски, то поиск информации занимает больше времени, чем ее чтение. Поэтому время поиска оказывает значительное влияние на стратегию сортировки. Уменьшение порядка слияния P дает возможность использовать большие по размеру буферы, так что реже требуется поиск; за счет этого часто компенсируется дополнительное время передачи, которое растет с уменьшением P .

Время поиска зависит от расстояния, проходимого держателем головок, и можно попытаться организовать работу таким образом, чтобы это расстояние было минимальным. Быть может, разумно сначала сортировать записи внутри цилиндров. Однако довольно большое слияние требует большого количества переходов между цилиндрами (см., например, упр. 2). Кроме того, режим мультипрограммирования в современных операционных системах означает, что пользователь лишь в редких случаях может по-настоящему контролировать положение держателя головок; блестящие схемы, минимизирующие поиск, обычно работают только по выходным дням! Таким образом, предположение о том, что каждая команда для диска требует "случайного" поиска, часто вполне оправдано.

Наша цель в том и состоит, чтобы найти такое дерево (т. е. схему слияния), которое обеспечивает наилучший баланс между временем поиска и временем передачи; для этой цели нам нужен некоторый способ, позволяющий оценить достоинства любого конкретного дерева по отношению к конкретной конфигурации оборудования. Рассмотрим, например, дерево на рис. 92; мы хотим оценить, сколько времени займет выполнение соответствующего слияния, чтобы можно было сравнить это дерево с другими.

В последующих рассуждениях мы сделаем некоторые простые предположения относительно слияния на дисках, чтобы проиллюстрировать некоторые общие идеи. Предположим, что (1) на чтение или запись n литер требуется $72.5 + 0.005n$ мс; (2) под рабочее пространство отводится 100000 литер внутренней памяти; (3) для пересылки одной литеры из буфера ввода в буфер вывода затрачивается в среднем 0.004 мс на вычисление; (4) нет совмещения чтения, записи и вычислений; (5) размер буфера, используемого для вывода, не обязательно должен быть равен размеру буфера, используемого для чтения данных на следующем проходе. Анализ задачи сортировки при этих простых предположениях будет полезен для понимания более сложных ситуаций.

Если выполняется P -путевое слияние, то мы можем разделить внутреннюю рабочую память на $P+1$ буферных областей: P — для ввода и 1 — для вывода; в каждом буфере по $B = 100000/(P+1)$ литер. Предположим, что предназначенные для слияния файлы содержали в сумме L литер; тогда мы выполним приблизительно L/B операций вывода и примерно столько же операций ввода; следовательно, общее время слияния при таких предположениях будет равно (в миллисекундах) приблизительно

$$2 \left(72.5 \frac{L}{B} + 0.005L \right) + 0.004L = (0.00145P + 0.011545)L. \quad (2)$$

Иными словами, P -путевое слияние L литер занимает примерно $(\alpha P + \beta)L$ единиц времени, где α и β —некоторые константы, зависящие от времени поиска, времени ожидания, времени вычислений и размера памяти. Эта формула приводит к интересному способу построения хороших схем слияния для

Picture: Рис. 92. Дерево с длиной внешнего пути 16 ...

дисков. Рассмотрим, например, рис. 92 и будем считать, что все начальные отрезки (изображенные квадратными "листьями") имеют длину L_0 . Тогда каждое слияние в узлах 9 и 10 занимает $(2\alpha + \beta)(2L_0)$ единиц времени, слияние в узле 11 занимает $(3\alpha + \beta)(4L_0)$ единиц и окончательное слияние в узле 12 занимает $(4\alpha + \beta)(8L_0)$ единиц. Общее время слияния, следовательно, составляет $(52\alpha + 16\beta)L_0$ единиц. Коэффициент "16" нам хорошо известен: это просто длина внешнего пути дерева. Коэффициент "52" при α соответствует новому понятию, которое мы можем назвать *длиной степенного пути дерева*; она равна сумме, взятой по всем листьям, степеней внутренних узлов, лежащих на пути от листа к корню. Например, на рис. 92 длина степенного пути равна $(2 + 4) + (2 + 4) + (3 + 4) + (2 + 3 + 4) + (2 + 3 + 4) + (3 + 4) + (4) + (4) = 52$.

Если \mathcal{T} —любое дерево, то пусть $D(\mathcal{T})$, $E(\mathcal{T})$ обозначают соответственно длину степенного пути и длину внешнего пути этого дерева. Анализ сводится к следующей теореме:

Теорема Н. Если время, требуемое для выполнения P -путевого слияния L литер, имеет вид $(\alpha P + \beta)L$ и если требуется слить S отрезков равной длины, то наилучшая схема слияния соответствует дереву \mathcal{T} , для которого $\alpha D(\mathcal{T}) + \beta E(\mathcal{T})$ минимально среди всех деревьев с S листьями.

(Эта теорема неявно содержалась в неопубликованной статье, которую Джордж А. Хаббард представил на национальную конференцию АСМ в 1963 г.)

Пусть α и β —фиксированные константы; будем говорить, что дерево *оптимально*, если оно имеет минимальное значение $\alpha D(\mathcal{T}) + \beta E(\mathcal{T})$ среди всех деревьев \mathcal{T} с тем же числом листьев. Нетрудно видеть, что *все поддеревья оптимального дерева также оптимальны*. Поэтому мы можем строить оптимальные деревья с n листьями, объединяя оптимальные деревья, у которых меньше чем n листьев.

Теорема К. Пусть последовательность чисел $A_m(n)$ определена при $1 \leq m \leq n$ правилами

$$A_1(1) = 0; \quad (3)$$

$$A_m(n) = \min_{1 \leq k \leq n/m} (A_1(k) + A_{m-1}(n-k)) \quad \text{при } 2 \leq m \leq n; \quad (4)$$

$$A_1(n) = \min_{2 \leq m \leq n} ((\alpha m n + \beta n + A_m(n))) \quad \text{при } n \geq 2. \quad (5)$$

Тогда $A_1(n)$ есть минимальное значение $\alpha D(\mathcal{T}) + \beta E(\mathcal{T})$ среди всех деревьев \mathcal{T} с n листьями.

Доказательство. Из соотношения (4) следует, что $A_m(n)$ есть минимальное значение $A_1(n_1) + \dots + A_1(n_m)$ по всем положительным числам n_1, \dots, n_m , таким, что $n_1 + \dots + n_m = n$. Требуемый результат получается теперь индукцией по n . ■

Рекуррентные соотношения (3), (4), (5) можно использовать также для построения самих оптимальных деревьев. Пусть $k_m(n)$ —значение, для которого достигается минимум в определении $A_m(n)$. Тогда можно построить оптимальное дерево с n листьями, объединяя $m = k_1(n)$ поддеревьев в корне; поддеревья являются оптимальными деревьями с $k_m(n)$, $k_{m-1}(n - k_m(n))$, $k_{m-2}(n - k_m(n) - k_{m-1}(n - k_m(n)))$, ... листьями соответственно.

Эта конструкция при $\alpha = \beta = 1$ проиллюстрирована в качестве примера в табл. 1. Компактные описания соответствующих оптимальных деревьев имеются в правой части таблицы; элемент "4:9:9" для $n = 22$, например, означает, что оптимальное дерево \mathcal{T}_{22} с 22 листьями можно получить в результате объединения \mathcal{T}_4 , \mathcal{T}_9 и \mathcal{T}_9 (рис. 93). Оптимальные деревья не единственны; например, элемент 5:8:9 был бы столь же хорошим, как и 4:9:9.

местах, где на карте отперфорированы отверстия, входят в контакт с ртутью на нижней панели.

В результате замыкания соответствующей цепи показание связанного с ней циферблата изменяется на 1 и, кроме того, одна из 26 крышек сортировального ящика открывается. В этот момент оператор отпускает пресс, кладет карту в открытое отделение и закрывает крышку. По сообщениям, как-то через эту машину пропустили 19071 карту за один 6.5-часовой рабочий день; в среднем примерно 49 карт в минуту! (Средний оператор работал примерно втрое медленней.)

Население продолжало неуклонно расти, и первые табуляторы-сортировщики оказались недостаточно быстрыми, чтобы справиться с обработкой переписи 1900 г., поэтому Холлерит изобрел еще одну машину, чтобы предотвратить еще один кризис в обработке данных. Его новое устройство (запатентованное в 1901 и 1904 гг.) имело автоматическую подачу карт и выглядело, в сущности, почти так же, как современные карточные сортировальные машины. История ранних машин Холлерита с интересными подробностями изложена Леоном Э. Трусделлом в *The Development of Punch Card Tabulation* (Washington: U. S. Bureau of the Census, 1965); см. также сообщения современников Холлерита: *Columbia College School of Mines Quarterly*, 10 (1889), 238–255; *J. Franklin Inst.*, 129 (1890), 300–306; *The Electrical Engineer*, 12 (Nov. 11, 1891), 521–530; *J. Amer. Statistical Assn.*, 2 (1891), 330–341; 4 (1895), 365; *J. Royal Statistical Soc.*, 55 (1892), 326–327; *Alegemeines Statistisches Archiv*, 2 (1892), 78–126; *J. Soc. Statistique de Paris*, 33 (1892), 87–96; U. S. Patents 395781 (1889), 685608 (1901), 777209 (1904). Холлерит и другой бывший служащий Бюро переписи Джеймс Пауэрс в дальнейшем основали конкурирующие компании, которые в конце концов вошли соответственно в корпорации IBM и Remington Rand.

Сортировальная машина Холлерита—это, конечно, основа методов поразрядной сортировки, используемых в цифровых ЭВМ. В его патенте упоминается, что числовые элементы, содержащие два столбца, должны сортироваться "по отдельности для каждого столбца", но он не говорит, какой столбец (единиц или десятков) должен рассматриваться первым. Далеко не очевидная идея сортировки сначала по столбцу единиц была, по-видимому, открыта каким-то неизвестным оператором и передана остальным (см. п. 5.2.5); она имеется в самом раннем сохранившемся руководстве IBM по сортировке (1936 г.). Первым известным упоминанием этого метода "справа налево" является случайное замечание, встретившееся в статье Л. Дж. Комри, *Trans. of the Office Machinery Users' Assoc.* (London, 1930), 25–37. Нечаянно Комри оказался первым, кто сделал важное наблюдение, что табуляторы можно плодотворно применять в научных вычислениях, хотя первоначально они создавались для статистических и бухгалтерских приложений. Его статья особенно интересна, поскольку, содержит подробное описание табуляторов, имевшихся в Англии в 1930 г. Сортировальные машины в то время обрабатывали от 360 до 400 карт в минуту и сдавались в аренду за 9 фунтов стерлингов в месяц.

Идея слияния восходит к другому устройству для обработки карт—*подборочной машине*, которая была изобретена значительно позднее (в 1938 г.). Снабженная двумя подающими механизмами, она могла слить две отсортированные колоды карт в одну всего за один проход; метод выполнения этого слияния хорошо описан в первом руководстве IBM по методам подборки (апрель 1939 г.). [Ср. с James W. Bryce. U. S. Patent 2189024 (1940).]

Затем на сцене появились ЭВМ и разработка методов сортировки тесно переплелась с их развитием. На самом деле имеются свидетельства того, что программа сортировки была первой когда-либо написанной для вычислительных машин с запоминаемой программой. Конструкторы вычислительной машины EDVAC особенно интересовались сортировкой, поскольку она выступала как наиболее характерный представитель потенциальных нечисленных приложений ЭВМ. Они понимали, что удовлетворительная система команд должна годиться не только для составления программы решения разностных уравнений; в ней должна быть достаточная гибкость, чтобы справиться с комбинаторными аспектами "выбора решений" в алгоритмах. Поэтому Джон фон Нейман подготовил в 1945 г. программы для внутренней сортировки слиянием, чтобы убедиться в необходимости некоторых кодов команд, которые он предлагал для машины EDVAC; существовали эффективные сортировальные машины специального назначения, и они служили тем естественным стандартом, в сопоставлении с которым можно было оценить достоинства предлагаемой организации вычислительной машины. Подробно это интересное исследование описано в статье Д. Э. Кнута [*Computing Surveys*, 2 (1970), 247–260]; первую программу сортировки фон Неймана в окончательном, "отполированном" виде см. в его *Collected Works*, 5 (New York, Macmillan, 1963), 196–214.

Из-за ограниченного объема памяти в ранних машинах приходилось думать о внешней сортировке наравне с внутренней, и в докладе "Progress Report on the EDVAC", подготовленном Дж. П. Эккертом и Дж. У. Мочли для школы Мура по электротехнике [Moore school of Electrical Engineering (September 30, 1945)], указывалось, что ЭВМ, оснащенная устройством с магнитной проволокой или лентой, могла бы моделировать действия карточного оборудования, достигая при этом большей ско-

рости сортировки. Этот доклад описывал сбалансированную двухпутевую поразрядную сортировку и сбалансированное двухпутевое слияние с использованием четырех устройств с магнитной проволокой или лентой, читающих или записывающих "не менее 5000 импульсов в секунду".

Джон Мочли выступил с лекцией о "сортировке и слиянии" на специальной сессии по вычислениям, созывавшейся в школе Мура в 1946 г., и в записях его лекции содержится первое опубликованное обсуждение сортировки с помощью вычислительных машин [Theory and techniques for the design of electronic digital computers, ed. by G. W. Patterson, 3 (1946), 22.1–22.20]. Мочли начал свое выступление с интересного замечания: "Требование, чтобы одна машина объединяла возможности вычислений и сортировки, может выглядеть как требование, чтобы один прибор использовался как ключ для консервов и как авторучка". Затем он заметил, что машины, способные выполнять сложные математические процедуры, должны также иметь возможность сортировать и классифицировать данные; он показал, что сортировка может быть полезна даже в связи с численными расчетами. Он описал простые вставки и бинарные вставки, заметив, что в первом методе в среднем требуется около $N^2/4$ сравнений, в то время как в последнем их никогда не требуется более $N \log_2 N$. Однако бинарные вставки требуют весьма сложной структуры данных, и Мочли затем показал, что при двухпутевом слиянии достигается столь же малое число сравнений, но используется только последовательное прохождение списков. Последняя часть записей его лекций посвящена разбору методов поразрядной сортировки с частичными проходами, которые моделируют цифровую карточную сортировку на четырех лентах, затрачивая менее четырех проходов на цифру (ср. с п. 5.4.7).

Вскоре после этого Эккерт и Мочли организовали компанию, которая выпускала некоторые из самых ранних электронных вычислительных машин BINAC (для военных приложений) и UNIVAC (для коммерческих приложений). Вновь Бюро переписи США сыграло роль в этом развитии, приобретя первый UNIVAC. В это время вовсе не было ясно, что ЭВМ станут экономически выгодными: вычислительные машины могли сортировать быстрее, но они дороже стоили. Поэтому программисты UNIVAC под руководством Франсис Э. Гольбертон приложили значительные усилия к созданию программ внешней сортировки, работающих с высокой скоростью, и их первые программы повлияли также на разработку оборудования. По их оценкам, 100 миллионов записей по 10 слов могли быть отсортированы на UNIVAC за 9000 ч (т. е. 375 дней).

UNIVAC I, официально объявленная в июле 1951 г., имела внутреннюю память в 1000 12-литерных (72-битовых) слов. В ней предусматривалось чтение и запись на ленту блоков по 60 слов со скоростью 500 слов в секунду; чтение могло быть прямым или обратным, допускалось одновременное чтение /запись/ вычисления. В 1948 г. миссис Гольбертон придумала интересный способ выполнения двухпутевого слияния с полным совмещением чтения, записи и вычислений с использованием шести буферов ввода. Пусть для каждого вводного файла имеются один "текущий буфер" и два "вспомогательных буфера"; сливать можно таким образом, что всякий раз, когда приходит время вывести один блок, два текущих буфера ввода содержат вместе количество данных, равное одному блоку. Таким образом, за время формирования каждого выводного блока ровно один буфер ввода становится пустым, и мы можем устроить так, чтобы три или четыре вспомогательных буфера были заполнены всякий раз, как мы читаем в оставшийся буфер. Этот метод чуть быстрее метода прогнозирования алгоритма 5.4.6F, так как нет необходимости проверять результат одного ввода перед началом следующего. [Ср. с Collation Methods for the UNIVAC System (Eckert-Mauchly Computer Corp., 1950) vol. 1,2.]

Кульминационной точкой в этой работе стал генератор программ сортировки, который был первой крупной программой, разработанной для автоматического программирования. Пользователь указывал размер записи, позиции ключей (до пяти) в частичных полях каждой записи и "концевые" ключи, отмечающие конец файла, и генератор сортировки порождал требуемую программу сортировки для файлов на одной бобине. Первым проходом этой программы была внутренняя сортировка блоков по 60 слов с использованием метода сравнения и подсчета (алгоритм 5.2С); затем выполнялся ряд сбалансированных двухпутевых проходов слияния с обратным чтением, исключаяющих сцепление лент, как описано выше. [См. Master Generating Routine for 2-way Sorting (Eckert—Mauchly Div. of Remington Rand, 1952). Первый набросок этого доклада был озаглавлен "Основная составляющая программа двухпутевого слияния" (Master Prefabrication Routine for 2-way Collation)! См. также F. E. Holberton, Symposium on Automatic Programming (Office of Naval Research, 1954), 34–39.]

К 1952 г. многие методы внутренней сортировки прочно вошли в программистский фольклор, но теория была развита сравнительно слабо. Даниэль Голденберг [Time analyses of various methods of sorting data, Digital Computer Lab. memo M-1680 (Mass. Inst. of Tech., October 17, 1952)] запрограммировал для машины Whirlwind пять различных методов и провел анализ наилучшего и наихудшего случаев для каждой программы. Он нашел, что для сортировки сотни 15-битовых записей по 8-битовому ключу наилучшие по скорости результаты получаются в том случае, если использу-

ется таблица из 256 слов и каждая запись помещается в единственную соответствующую ее ключу позицию, а затем эта таблица сжимается. Однако этот метод имел очевидный недостаток, ибо он уничтожал запись, если последующая имела тот же ключ. Остальные четыре проанализированных метода были упорядочены следующим образом: прямое двухпутевое слияние лучше поразрядной сортировки с основанием 2, которая лучше простого выбора, который в свою очередь лучше метода пузырька.

Эти результаты получили дальнейшее развитие в диссертации Гарольда Х. Сьюворда в 1954 г. [Information sorting in the application of electronic digital computers to business operations, Digital Computer Lab. report R-232 (Mass. Inst. of Tech., May 24, 1954), 60 pp.]. Сьюворд высказал идеи распределяющего подсчета и выбора с замещением. Он показал, что первый отрезок случайной перестановки имеет среднюю длину $e - 1$, и анализировал наряду с внутренней сортировкой и внешнюю как на различных типах массовой памяти, так и на лентах.

Еще более достойная внимания диссертация—на этот раз докторская— была написана Говардом Б. Демуттом в 1956 г. [Electronic Data Sorting (Stanford University, October, 1956), 92 pp.]. Эта работа помогла заложить основы теории сложности вычислений. В ней рассматривались три абстрактные модели задачи сортировки: с использованием циклической памяти, линейной памяти и памяти с произвольным доступом; для каждой модели были разработаны оптимальные или почти оптимальные методы. (Ср. с упр. 5.3.4–62.) Хотя непосредственно из диссертации Демутта не вытекает никаких практических следствий, в ней содержатся важные идеи о том, как связать теорию с практикой.

Таким образом, история сортировки была тесно связана со многими исхоженными тропами в вычислениях: с первыми машинами для обработки данных, первыми запоминаемыми программами, первым программным обеспечением, первыми методами буферизации, первой работой по анализу алгоритмов и сложности вычислений.

Ни один из документов относительно ЭВМ, упомянутых до сих пор, не появлялся в "открытой литературе". Так уж случилось, что большая часть ранней истории вычислительных машин содержится в сравнительно недоступных докладах, поскольку относительно немногие лица были в то время связаны с ЭВМ. Наконец в 1955–1956 гг. литература о сортировке проникает в печать в виде трех больших обзорных статей. Первая статья была подготовлена Дж. К. Хоскеном [Proc. Eastern Joint. Computer Conference, 8 (1955), 39–55]. Он начинает с тонкого наблюдения: "Чтобы снизить стоимость единицы результата, люди обычно укрупняют операции. Но при этих условиях стоимость единицы сортировки не уменьшается, а возрастает". Хоскен описал все оборудование специального назначения, имевшееся в продаже, а также методы сортировки на ЭВМ. Его библиография из 54 пунктов основана большей частью на брошюрах фирм-изготовителей.

Подробная статья Э. Х. Фрэнда [Sorting on Electronic Computer Systems, *JACM*, 3 (1956), 134–168] явилась важной вехой в развитии сортировки. Хотя за прошедшее с 1956 г. время были разработаны многочисленные методы, эта статья все еще необычно современна во многих отношениях. Фрэнд дал тщательное описание весьма большого числа алгоритмов внутренней и внешней сортировки и обратил особое внимание на методы буферизации и характеристики магнитных лентопротяжных устройств. Он ввел некоторые новые методы (например, выбор из дерева, метод двуглавого змия и прогнозирование) и разработал некоторые математические свойства старых методов.

Третий обзор по сортировке, который появился в то время, был подготовлен Д. У. Дэвайсом [Proc. Inst. Elect. Engineers, 103B, supplement 1 (1956), 87–93]. В последующие годы еще несколько выдающихся обзоров было опубликовано Д. А. Беллом [Comp. J., 1 (1958), 71–77]; А. Ш. Дугласом [Comp. J., 2 (1959), 1–9]; Д. Д. Мак-Кракеном, Г. Вейссом и Ц. Ли [Programming Business Computers (New York: Wiley, 1959), chapter 15, 298–332]; А. Флоресом [*JACM*, 8 (1961) 41–80]; К. Э. Айверсоном [A programming language (New York: Wiley, 1962), chapter 6, 176–245]; К. К. Готлибом [*CACM*, 6 (1963), 194–201]; Т. Н. Хиббардом [*CACM*, 6 (1963), 206–213]; М. А. Готцем [Digital Computer User's Handbook ed. by M. Klerer and G. A. Korn (New York, McGraw-Hill, 1967), chapter 1.10, 1.292–1.320]. В ноябре 1962 г. АСМ организовала симпозиум по сортировке (большая часть работ, представленных на этом симпозиуме, опубликована в мае 1963 г. в выпуске *CACM*). Они дают хорошее представление о состоянии этой области в то время. Обзор К. К. Готлиба о современных генераторах сортировки, обзор Т. Н. Хиббарда о внутренней сортировке с минимальной памятью и раннее исследование Дж. А. Хаббарда о сортировке файлов на дисках—наиболее заслуживающие внимания статьи в этом собрании.

За прошедший период были открыты новые методы сортировки: вычисление адреса (1956), слияние с вставкой (1959), обменная поразрядная сортировка (1959), каскадное слияние (1959), метод Шелла с убывающим шагом (1959), многофазное слияние (1960), вставки в дерево (1960), осциллирующая сортировка (1962), быстрая сортировка Хоара (1962), пирамидальная сортировка Уильямса (1964), обменная сортировка со слиянием Бэтчера (1964). История каждого отдельного алгоритма

прослеживается в тех разделах настоящей главы где этот метод описывается. Конец 60-х годов нашего столетия ознаменовался интенсивным развитием соответствующей теории.

Полная библиография всех работ, изученных автором при написании этой главы, имеется в [Computing Reviews, 13 (1972), 283–289].

Упражнения

1. [05] Подведите итог этой главе; сформулируйте обобщение теоремы 5.4.6А.
2. [20] Скажите, основываясь на табл. 1, какой из методов сортировки списков для шестидесятизначных ключей будет наилучшим для машины MIX.
3. [47](Устойчивая сортировка с минимальной памятью.) Говорят, что алгоритм сортировки требует минимальной памяти, если он использует для своих переменных только $O((\log N)^2)$ битов памяти сверх пространства, требуемого для размещения N записей. Алгоритм должен быть общим в том смысле, что должен работать при любых N , а не только при определенном значении N , если, конечно, предполагается, что при вызове алгоритма для сортировки ему обеспечивается достаточное количество памяти с произвольным доступом. Во многих из изученных нами алгоритмов сортировки это требование минимальной памяти нарушается; в частности, запрещено использование N полей связи. Быстрая сортировка (алгоритм 5.2.2Q) удовлетворяет требованию минимальной памяти, но ее время работы в наихудшем случае пропорционально N^2 . Пирамидальная сортировка (алгоритм 5.2.3H) является единственным среди изученных нами алгоритмов типа $O(N \log N)$, который использует минимальную память, хотя можно сформулировать и еще один подобный алгоритм, если использовать идею из упр. 5.2.4–18. Самым быстрым общим алгоритмом, изученным нами, который устойчиво сортирует ключи, является метод слияния списков (алгоритм 5.2.4L), однако он использует не минимальную память. Фактически единственными устойчивыми алгоритмами сортировки с минимальной памятью, которые мы видели, были методы типа $O(N^2)$ (простые вставки, метод пузырька и варианты простого выбора).

Существует ли устойчивый алгоритм сортировки с минимальной памятью, требующий менее $O(N^2)$ единиц времени в наихудшем случае или в среднем?

Я достиг своей цели., если, рассортировал и привел в логический порядок хотя бы ядро того огромного материала о сортировке, который появился за последние несколько лет.

Дж. К. Хоскен (1955)

6. Поиск

Поискем запись
Эл Смит (1928)

Эту главу можно было назвать иначе: или претенциозно—“Хранение и выборка информации”, или просто—“Поиск по таблицам”. Нас будет интересовать процесс накопления информации в памяти вычислительной машины с последующим возможно более быстрым извлечением этой информации. Иногда мы сталкиваемся со столь большим объемом данных, что реально использовать их все не можем. В этом случае самым мудрым было бы забыть и разрушить большую их часть; но нередко бывает важно сохранить и так организовать имеющиеся факты, чтобы обеспечить наибо́льшую их выборку.

Эта глава посвящена в основном изучению очень простой поисковой задачи: как находить данные, хранящиеся с определенной идентификацией. Например, в вычислительной задаче нам может понадобиться найти $f(x)$, имея x и таблицу значений функции f ; в лингвистической может интересоваться английский эквивалент данного русского слова.

Вообще будем предполагать, что хранится множество из N записей и необходимо определить положение соответствующей записи. Как и в случае сортировки, предположим, что каждая запись содержит специальное поле, которое называется *ключом*, возможно, потому, что многие люди ежедневно тратят массу времени на поиск своих ключей. Мы обычно требуем, чтобы N ключей были различны, так что каждый ключ однозначно определяет свою запись. Совокупность всех записей называется *таблицей* или *файлом*, причем под таблицей, как правило, подразумевается небольшой файл, а файлом обычно называют большую таблицу. Большой файл, или группа файлов, часто называется *базой данных*.

Смит, Альфред Эммануэль (1873–1944) — американский политический деятель.—Прим. перев.

В алгоритмах поиска присутствует так называемый *аргумент поиска* K , и задача состоит в отыскании записи, имеющей K своим ключом. Существуют две возможности окончания поиска: либо поиск оказался *удачным*, т. е. позволил определить положение соответствующей записи, содержащей K , либо он оказался *неудачным*, т. е. показал, что аргумент K не может быть найден ни в одной из записей. После неудачного поиска иногда желательно вставить в таблицу новую запись, содержащую K ; алгоритм, который делает это, называется алгоритмом "поиска с вставкой". Некоторые технические устройства, известные как "ассоциативная память", решают проблему поиска автоматически аналогично тому, как это делает человеческий мозг; мы же будем изучать методы поиска на обычной универсальной вычислительной машине.

Хотя целью поиска является информация, которая содержится в записи, ассоциированной с ключом K , в алгоритмах этой главы обычно игнорируется все, кроме собственно ключей. В самом деле, если положение K определено, соответствующие данные можно найти. Например, если K встретился в ячейке $TABLE + i$, ассоциированные данные (или указатель на них) могут находиться по адресу $TABLE + i + 1$, или $DATA + i$ и т. д. Таким образом, подробности, касающиеся того, что нужно делать, когда найден ключ K , можно спокойно опустить.

Во многих программах поиск требует наибольших временных затрат, так что замена плохого метода поиска на хороший часто ведет к существенному увеличению скорости работы. Действительно, нередко удается так организовать данные или структуру данных, что поиск полностью исключается, т. е. мы всегда знаем заранее, где находится нужная нам информация. Связанная память является общепринятым методом достижения этого; например, в списке с двумя связями нет необходимости искать элемент, следующий за данным или предшествующий ему. Другой способ избежать поиска открывается перед нами, если предоставлена свобода выбора ключей. Сделаем их числами $\{1, 2, \dots, N\}$, и тогда запись, содержащая K , может быть просто помещена в ячейку $TABLE + K$. Оба эти способа использовались для устранения поиска из алгоритма топологической сортировки, обсуждавшегося в п. 2.2.3. Тем не менее во многих случаях поиск необходим (например, когда объектами топологической сортировки являются символические имена, а не числа), так что весьма важно иметь эффективные алгоритмы поиска.

Методы поиска можно классифицировать несколькими способами. Мы могли бы разделить их на внутренний и внешний поиск в соответствии с разделением алгоритмов сортировки в гл. 5 на внутреннюю и внешнюю сортировку. Или мы могли бы различать статический и динамический методы поиска, где "статический" означает, что содержимое таблицы остается неизменным (так что важно минимизировать время поиска, пренебрегая затратами на перестройку таблицы), а "динамический" означает, что таблица является объектом частых вставок (и, может быть, удалений). Третья возможная схема классифицирует методы поиска в соответствии с тем, основаны ли они на сравнении ключей или на цифровых свойствах ключей, аналогично тому, как различаются сортировка с помощью сравнения и сортировка с помощью распределения. Наконец, мы могли бы разделить методы поиска на методы, использующие истинные ключи, и на методы, работающие с преобразованными ключами.

Организация данной главы есть, в сущности, комбинация двух последних способов классификации. В § 6.1 рассматриваются методы последовательного поиска "в лоб", а в § 6.2 обсуждаются улучшения, которые можно получить на основе сравнений между ключами с использованием алфавитного или числового порядка для управления решениями. В § 6.3 рассматривается цифровой поиск, а в § 6.4 обсуждается важный класс методов, называемых хешированием и основанных на арифметических преобразованиях истинных ключей. В каждом из этих параграфов рассматривается как внутренний, так и внешний поиск, и для статического и для динамического случаев; в каждом параграфе отмечаются сравнительные достоинства и недостатки различных алгоритмов.

Между поиском и сортировкой существует определенная взаимосвязь. Например, рассмотрим следующую задачу:

Даны два множества чисел:

$$A = \{a_1, a_2, \dots, a_m\} \text{ и } B = \{b_1, b_2, \dots, b_n\};$$

определить, является ли A подмножеством B , т. е. $A \subseteq B$.

Напрашиваются три решения, а именно:

- 1) Сравнить каждое a_i последовательно со всеми b_j до установления совпадения.
- 2) Свести все b_j в таблицу, затем искать каждое a_i по таблице.
- 3) Упорядочить A и B , затем совершить один последовательный проход по обоим файлам, проверяя соответствующие условия.

Каждое из этих решений имеет свои привлекательные стороны для различных диапазонов значений m и n . Для решения 1 потребуется приблизительно $c_1 mn$ единиц времени, где c_1 — некоторая константа, а решение 3 займет около $c_2(m \log_2 m + n \log_2 n)$ единиц, где c_2 — некоторая (бóльшая) константа. При подходящем методе хеширования решение 2 потребует примерно $c_3 m + c_4 n$ единиц времени, где c_3 и c_4 — некоторые (еще бóльшие) константы. Следовательно, решение 1 хорошо при очень малых m и n , а при возрастании m и n лучшим будет решение 3. Затем, пока n не достигнет размеров внутренней памяти, более предпочтительно решение 2; после этого обычно решение 3 снова становится лучше, пока n не сделается еще гораздо большим. Значит, мы имеем ситуацию, где сортировка иногда хорошо заменяет поиск, а поиск — сортировку.

Зачастую сложные задачи поиска можно свести к более простым случаям, рассматриваемым здесь. Например, предположим, что в роли ключей выступают слова, которые могли быть слегка искажены; мы хотели бы найти правильную запись, несмотря на эту ошибку. Если сделать две копии файла, в одной из которых ключи расположены в обычном алфавитном порядке, а в другой они упорядочены справа налево (как если бы слова были прочитаны наоборот), искаженный аргумент поиска в большинстве случаев совпадает до половины своей длины или дальше с записью одного из этих двух файлов. Методы поиска, содержащиеся в § 6.2 и 6.3, можно, следовательно, приспособить для нахождения ключа, который был, вероятно, искажен.

Похожие задачи привлекли к себе заметное внимание в связи с созданием систем предварительного заказа авиабилетов и в связи с другими приложениями, когда существует значительная вероятность искажения имени человека из-за неразборчивого почерка или плохой слышимости. Нужно было найти преобразование аргумента в некий код, собирающее вместе все варианты данного имени. Метод "Soundex", описываемый ниже в виде, в котором он применяется сейчас, был первоначально развит Маргарет К. Оуделл и Робертом К. Расселом [см. U. S. Patents 1261167 (1918), 1435663 (1922)]; он нашел широкое применение для кодирования фамилий.

- 1) Оставить первую букву; все буквы a, e, h, i, o, u, w, y, стоящие на других местах, вычеркнуть.
- 2) Оставшимся буквам (кроме первой) присвоить следующие значения:

$$\begin{array}{ll} b, f, p, v \rightarrow 1; & l \rightarrow 4; \\ c, g, j, k, q, s, x, z \rightarrow 2; & m, n \rightarrow 5; \\ d, t \rightarrow 3; & r \rightarrow 0. \end{array}$$

- 3) Если в исходном имени (перед шагом 1) рядом стояли несколько букв с одинаковыми кодами, пренебречь всеми, кроме первой из этой группы.
- 4) Дописывая в случае надобности нули или опуская лишние цифры, преобразовать полученное выражение в форму "буква, цифра, цифра, цифра".

Например, фамилии Euler, Gauss, Hilbert, Knuth, Lloyd и Lukasiewicz имеют коды соответственно E460, G200, H416, K530, L300, L222. Разумеется, такая система собирает вместе не только родственные, но и достаточно различные имена. Приведенные выше шесть кодов могли быть получены из фамилий Ellery, Ghosh, Heilbronn, Kant, Ladd и Lissajous. С другой стороны, такие родственные имена, как Rogers и Rodgers, Sinclair и St. Clair или Tchebysheff и Chebyshev, имеют разную кодировку. Но, вообще говоря, система "Soundex" намного увеличивает вероятность обнаружить имя под одной из его масок. [Для дальнейшего ознакомления, см. C. P. Bourne, D. F. Ford, *JACM*, 8 (1961), 538–552; Leon Davidson, *CACM*, 5 (1962), 169–171; Federal Population Censuses, 1790–1890 (Washington, D. C.: National Archives, 1971), 90.]

Когда мы используем системы типа "Soundex", нет необходимости предполагать, что все ключи различны; можно составить списки из записей с совпадающими кодами, рассматривая каждый список как объект поиска.

При использовании больших баз данных выборка информации намного усложняется, так как часто желательно рассматривать различные поля каждой записи как потенциальные ключи. Здесь важно уметь находить записи по неполной ключевой информации. Например, имея большой файл актеров, режиссер мог бы пожелать найти всех незанятых актрис в возрасте 25–30 лет, хорошо танцующих и говорящих с французским акцентом; у спортивного журналиста может возникнуть желание подсчитать с помощью файла бейсбольной статистики общее число очков, набранных подающими-левшами команды "Красноногие из Цинцинати" в течение седьмых периодов вечерних игр за 1964 г. Имея большой файл данных о чем-либо, люди любят задавать вопросы произвольной сложности. На самом деле мы могли бы рассматривать полную библиотеку как базу данных, в которой некто желает найти все публикации о выборке информации. Введение в методы *поиска по многим признакам* помещено в § 6.5.

Прежде чем переходить к детальному изучению поиска, полезно рассмотреть историю данного вопроса. В докомпьютерный период было составлено множество томов логарифмических, тригонометрических и других таблиц, так что математические вычисления могли быть заменены поиском. Потом эти таблицы были перенесены на перфокарты и использовались для научных задач посредством распознающих, сортировальных и дублирующих перфораторных машин. Однако после появления ЭВМ с запоминаемой программой стало очевидно, что дешевле каждый раз вычислять $\log x$ и $\cos x$, нежели искать ответ по таблице.

Несмотря на то, что задача сортировки привлекла пристальное внимание уже на заре развития ЭВМ, для разработки алгоритмов поиска было сделано сравнительно мало. Располагая небольшой внутренней памятью и только последовательными устройствами типа лент для хранения больших файлов, организовать поиск было либо совершенно тривиально, либо почти невозможно.

Но развитие все большей и большей памяти со случайным доступом в течение 50-х годов привело к пониманию того, что поиск как таковой является интересной задачей. После периода жалоб на ограниченные ресурсы пространства в ранних машинах программисты вдруг столкнулись с таким объемом памяти, который они не умели эффективно использовать.

Первые обзоры задач поиска были опубликованы А. И. Думи [*Computers & Automation*, 5, 12 (December 1956), 6–9], У. У. Петерсоном [*IBM J. Research & Development*, 1 (1957), 130–146], Э. Д. Бутом [*Information and Control*, 1 (1958), 159–164], А. Ш. Дугласом [*Comp. J.*, 2 (1959), 1–9]. Более подробный обзор сделан позднее К. Э. Айверсоном [*A Programming Language* (New York: Wiley, (1962)), 133–158] и В. Буххольцем [*IBM Systems J.*, 2 (1963), 86–111].

В начале 60-х годов было разработано несколько новых алгоритмов поиска, основанных на использовании древовидных структур; с ними мы познакомимся ниже. И в наше время ведутся активные исследования по проблемам поиска.

6.1. Последовательный поиск

"Начни с начала и продвигайся, пока не найдешь нужный ключ; тогда остановись". Такая последовательная процедура является очевидным способом поиска; удобно начать наши рассуждения с нее, так как на ней основаны многие более сложные алгоритмы. Несмотря на свою простоту, последовательный поиск содержит ряд очень интересных идей.

Сформулируем алгоритм более точно.

Алгоритм S. (Последовательный поиск.) Имеется таблица записей R_1, R_2, \dots, R_N , снабженных соответственно ключами K_1, K_2, \dots, K_N . Алгоритм предназначен для поиска записи с данным ключом K . Предполагается, что $N \geq 1$.

S1 [Начальная установка.] Установить $i \leftarrow 1$.

S2 [Сравнение.] Если $K = K_i$, алгоритм оканчивается удачно.

S3 [Продвижение.] Увеличить i на 1.

S4 [Конец файла?] Если $i \leq N$, то вернуться к шагу S2. В противном случае алгоритм оканчивается неудачно. ■

Заметим, что у этого алгоритма может быть два разных исхода: *удачный* (когда найдено положение нужного ключа) и

Picture: Последовательный поиск

неудачный (когда, установлено, что искомого аргумента нет в таблице). Это справедливо для большинства алгоритмов данной главы.

Реализация в виде программы для машины MIX очевидна.

Программа S. (Последовательный поиск.) Предположим, что K_i хранится по адресу $KEY + i$, а оставшаяся часть записи R_i — по адресу $INFO + i$. Приводимая ниже программа использует $rA \equiv K$, $rI1 \equiv i - N$.

START	LDA	K	1	S1. Начальная установка.
	ENT1		1	$i \leftarrow 1$.
2H	CMPA	KEY+N, 1	C	S2. Сравнение.
	JE	SUCCESS	C	Выход, если $K = K_i$.
	INC1	1	$C - S$	S3. Продвижение.
	J1NP	2B	$C - S$	S4. Конец файла?
FAILURE	EQU	*	$1 - S$	Выход, если нет в таблице.

По адресу SUCCESS расположена команда "LDA INFO+N, 1"; она помещает нужную информацию в rA. ■

Анализ данной программы не представляет труда; видно, что время работы алгоритма S зависит от двух параметров:

$$\begin{aligned} C &= (\text{количество сравнений ключей}); \\ S &= 1 \text{ при удаче и } 0 \text{ при неудаче.} \end{aligned} \quad (1)$$

Программа S требует $5C - 2S + 3$ единиц времени. Если мы нашли $K = K_i$, то $C = i$, $S = 1$; значит, полное время равно $(5i + 1)u$. Если же поиск оказался неудачным, то $C = N$, $S = 0$, а работала программа ровно $(5N + 3)u$. Если все ключи поступают на вход с равной вероятностью, то среднее значение C при удачном поиске составляет

$$\frac{1 + 2 + \dots + N}{N} = \frac{N + 1}{2}; \quad (2)$$

среднеквадратичное отклонение C , разумеется, довольно большое—примерно $0.289N$ (см. упр. 1).

Приведенный, алгоритм, несомненно, знаком всем программистам. Но мало кто знает, что этот способ реализации последовательного поиска не всегда самый лучший! Очевидное изменение убывает работу алгоритма (если только записей не слишком мало):

Алгоритм Q. (*Быстрый последовательный поиск*). В отличие от алгоритма S здесь еще предполагается, что в конце файла стоит фиктивная запись R_{N+1} .

Q1 [Начальная установка.] Установить $i \leftarrow 1$ и $K_{N+1} \leftarrow K$.

Q2 [Сравнение.] Если $K = K_i$, то перейти на Q4.

Q3 [Продвижение.] Увеличить i на 1 и вернуться к шагу Q2.

Q4 [Конец файла?] Если $i \leq N$, алгоритм оканчивается удачно; в противном случае—неудачно ($i = N + 1$). ■

Программа Q. (*Быстрый последовательный поиск.*) Значения регистров: rA $\equiv K$, rI1 $\equiv i - N$.

```
BEGIN   LDA    K           1      Q1. Начальная установка
        STA    KEY+N+1     1       $K_{N+1} \leftarrow K$ .
        ENT1   -N          1       $i \leftarrow 0$ .
        INC1   1            $C + 1 - S$  Q3. Продвижение.
        CMPA   KEY+N, 1     $C + 1 - S$  Q2. Сравнение.
        JNE    *-2          $C + 1 - S$  На Q3, если  $K_i \neq K$ .
        J1NP   SUCCESS     1      Q4. Конец файла?
FAILURE  EQU    *            $1 - S$  Выход, если нет в таблице.
```

Используя параметры C и S , введенные при анализе программы S, можно заключить, что время работы программы уменьшилось до $(4C - 4S + 10)u$; это дает улучшение при $C \geq 5$ (для удачного поиска) и при $N \geq 8$ (для неудачного поиска). При переходе от алгоритма S к алгоритму Q использован важный ускоряющий принцип: если во внутреннем цикле программы проверяются два или более условия, нужно постараться оставить там только одно сравнение. Существует способ сделать программу Q еще быстрее.

Программа Q'. (*Сверхбыстрый последовательный поиск.*) Значения регистров: rA $\equiv K$, rI1 $\equiv i - N$.

```
BEGIN   LDA    K           1      Q1. Начальная установка.
        STA    KEY + N + 1   1       $K_{N+1} \leftarrow K$ .
        ENT1   -1-N         1       $i \leftarrow -1$ .
3H       INC1   2             $\lfloor (C - S + 2)/2 \rfloor$  Q3. Продвижение. (Двойное.)
        CMPA   KEY+N, 1      $\lfloor (C - S + 2)/2 \rfloor$  Q2. Сравнение.
        JE     4F             $\lfloor (C - S + 2)/2 \rfloor$  На Q4, если  $K = K_i$ .
        CMPA   KEY+N+1, 1    $\lfloor (C - S + 1)/2 \rfloor$  Q2. Сравнение. (Следующее.)
        JNE    3B            $\lfloor (C - S + 1)/2 \rfloor$  На Q3, если  $K \neq K_{i+1}$ .
        INC1   1             $(C - 5) \bmod 2$  Продвинуть  $i$ .
4H       J1NP   SUCCESS     1      Q4. Конец файла?
FAILURE  EQU    *            $1 - S$  Выход, если нет в таблице.
```


Инструкции внутреннего цикла выписаны дважды; это исключает примерно половину операторов " $i \leftarrow i + 1$ ". Таким образом, время выполнения программы уменьшилось до

$$3.5C - 3.5S + 10 \frac{(C - S) \bmod 2}{2}$$

единиц. При поиске по большим таблицам программа Q' на 30% быстрее программы S; подобным образом могут быть улучшены многие существующие программы. Если известно, что ключи расположены в возрастающем порядке, полезно несколько изменить алгоритм.

Алгоритм Т. (Последовательный поиск в упорядоченной таблице.) Имеется таблица записей R_1, R_2, \dots, R_N , причем ключи удовлетворяют неравенствам $K_1 < K_2 < \dots < K_N$. Алгоритм предназначен для поиска данного ключа K . В целях удобства и увеличения скорости работы предполагается, что в конце таблицы расположена фиктивная запись R_{N+1} , ключ которой $K_{N+1} = \infty > K$.

T1 [Начальная установка.] Установить $i \leftarrow 1$.

T2 [Сравнение.] Если $K \leq K_i$, то перейти на T4.

T3 [Продвижение.] Увеличить i на 1 и вернуться к шагу T2.

T4 [Равенство?] Если $K = K_i$, то алгоритм оканчивается удачно. В противном случае—неудачно, нужной записи в таблице нет. ■

Если величина K с равной вероятностью принимает все возможные значения, то в случае удачного поиска алгоритм Т, по существу, не лучше алгоритма Q. Однако отсутствие нужной записи алгоритм Т позволяет обнаружить примерно в два раза быстрее.

Приведенные выше алгоритмы в целях удобства использовали индексные обозначения для элементов таблицы; аналогичные процедуры применимы и к таблицам в связанном представлении, так как в них данные тоже расположены последовательно. (См. упр. 2, 3 и 4.)

Частота обращений. До сих пор предполагалось, что с равной вероятностью может потребоваться поиск любого аргумента, однако часто такое предположение не правомерно; вообще говоря, ключ K_j будет разыскиваться с вероятностью p_i , где $p_1 + p_2 + \dots + p_N = 1$. Время удачного поиска при больших N пропорционально числу сравнений C , среднее значение которого равно

$$\overline{C}_N = p_1 + 2p_2 + \dots + Np_N. \quad (3)$$

Пусть есть возможность помещать записи в таблицу в любом порядке; тогда величина \overline{C}_N минимальна при

$$p_1 \geq p_2 \geq \dots \geq p_N, \quad (4)$$

т. е. когда наиболее часто используемые записи расположены в начале таблицы.

Посмотрим, что дает нам такое "оптимальное" расположение при различных распределениях вероятностей. Если $p_1 = p_2 = \dots = p_N = 1/N$, то формула (3) сводится к $\overline{C}_N = (N + 1)/2$, что уже было получено нами в (2). Предположим теперь, что

$$p_1 = \frac{1}{2}, p_2 = \frac{1}{4}, \dots, p_{N-1} = \frac{1}{2^{N-1}}, p_N = \frac{1}{2^{N-1}} \quad (5)$$

Согласно упр. 7, $\overline{C}_N = 2 - 2^{1-N}$; среднее число сравнений меньше двух, если записи расположены в надлежащем порядке. Другим напрашивающимся распределением является

$$p_1 = Nc, p_2 = (N - 1)c, \dots, p_N = c,$$

где

$$c = 2/N(N + 1). \quad (6)$$

Это "клиновидное" распределение дает более привычный результат

$$\overline{C}_N = c \sum_{1 \leq k \leq N} k \cdot (N + 1 - k) = \frac{N + 2}{2}; \quad (7)$$

оптимальное расположение экономит около трети поискового времени, требующегося для записей в случайном порядке.

Разумеется, распределения (5) и (6) довольно искусственны, и их нельзя считать хорошим приближением к действительности. Более типичную картину дает "закон Зипфа":

$$p_1 = c/1, p_2 = c/2, \dots, p_N = c/N, \quad \text{где } c = 1/H_N. \quad (8)$$

Это распределение получило известность благодаря Дж. К Зипфу, который заметил, что n -е наиболее употребительное в тексте на естественном языке слово встречается с частотой, приблизительно обратно пропорциональной n . [The Psycho-Biology of Language (Boston, Mass.: Houghton Miffling, 1935); Human Behavior and the Principle of Least Effort (Reading, Mass.: Addison-Wesley, 1949).] Аналогичное явление обнаружено им в таблицах переписи; там столичные районы расположены в порядке убывания населения. В случае, если частота ключей в таблице подчиняется закону Зипфа, имеем

$$\overline{C}_N = N/H_N; \quad (9)$$

поиск по такому файлу примерно в $\frac{1}{2} \ln N$ раз быстрее, чем по неупорядоченному. [Ср. с A. D. Booth et al. Mechanical Resolution of Linguistic Problems (New York: Academic Press, 1958), 79.]

Другое распределение, близкое к реальному, дает правило "80–20", которое часто встречается в коммерческих приложениях [ср. с W. P. Heising, *IBM Systems J.*, 2 (1963), 114–115]. Это правило гласит, что 80% работы ведется над наиболее активной частью файла величиной 20%; оно применимо и к этим 20%, так что 64% работы ведется с наиболее активными 4% файла, и т. д. Иными словами,

$$\frac{p_1 + p_2 + \dots + p_{.20n}}{p_1 + p_2 + p_3 + \dots + p_n} \approx 0.80 \quad \text{для всех } n. \quad (10)$$

Вот одно из распределений, точно удовлетворяющих приведенному правилу при n , кратных 5:

$$p_1 = c, p_2 = (2^\theta - 1)c, p_3 = (3^\theta - 2^\theta)c, \dots, p_N = (N^\theta - (N-1)^\theta)c, \quad (11)$$

где

$$c = 1/N^\theta; \theta = \frac{\log 0.80}{\log 0.20} = 0.1386. \quad (12)$$

Действительно, $p_1 + p_2 + \dots + p_n = cn^\theta$ при любом n . С вероятностями (11) не так просто работать; имеем, однако, $n^\theta - (n-1)^\theta = \theta n^{\theta-1}(1 + O(1/n))$, т.е. существует более простое распределение, приближенно удовлетворяющее правилу "80–20":

$$p_1 = c/1^{1-\theta}, p_2 = c/2^{1-\theta}, \dots, p_N = c/N^{1-\theta}, \quad \text{где } c = 1/H_N^{1-\theta}. \quad (13)$$

Здесь, как и раньше, $\theta = \log 0.80 / \log 0.20$, а $H_N^{(s)}$ есть N -е гармоническое число порядка s , т. е. $1^{-s} + 2^{-s} + \dots + N^{-s}$. Заметим, что это распределение очень напоминает распределение Зипфа (8); когда θ изменяется от 1 до 0, вероятности меняются от равномерно распределенных к зипфовским. (В самом деле, Зипф нашел, что $\theta \approx \frac{1}{2}$ в распределении личных доходов.) Применяя (3) к (13), получаем для правила "80–20" среднее число сравнений

$$\overline{C}_N = H_N^{(-\theta)} / H_N^{(1-\theta)} = \frac{\theta N}{\theta + 1} + O(N^{(1-\theta)}) \approx 0.122N \quad (14)$$

(см. упр. 8).

Ю. С. Шварц, изучавший частоты появления слов [см. интересный график на стр. 422 в *JACM*, 10 (1963)], предложил более подходящее выражение для закона Зипфа:

$$p_1 = c/1^{1+\theta}, p_2 = c/2^{1+\theta}, \dots, p_N = c/N^{1+\theta}, \quad \text{где } c = 1/H_N^{1+\theta}, \quad (15)$$

при малых положительных Q . [По сравнению с (13) здесь изменен знак θ .] В этом случае

$$\overline{C}_N = H_N^\theta / H_N^{(1+\theta)} = N^{1-\theta} / (1-\theta)\zeta(1+\theta) + O(N^{1-2\theta}), \quad (16)$$

что значительно меньше, чем (9) при $N \rightarrow \infty$.

"Самоорганизующийся" файл. Предыдущие вычисления хороши, но в большинстве случаев распределение вероятностей нам не известно. Мы могли бы в каждой записи завести счетчик числа обращений, чтобы на основании показаний счетчиков переупорядочить записи. Выведенные формулы показывают, что такая процедура может привести к заметной экономии. Но не всегда желательно

отводить много памяти под счетчики, так как ее можно использовать более рационально (например, применяя другие методы поиска, излагаемые ниже в данной главе).

Простая схема, происхождение которой не известно, используется уже многие годы. Она позволяет довольно хорошо упорядочить записи без вспомогательных полей для счетчиков: когда мы находим нужную запись, мы помещаем ее в начало таблицы. Подобный метод легко реализовать, когда таблица представлена в виде связанного линейного списка: ведь часто нам бывает нужно значительно изменить найденную запись.

Идея "самоорганизующегося" метода состоит в том, что часто используемые элементы будут расположены довольно близко к началу таблицы. Пусть N ключей разыскиваются с вероятностями $\{p_1, p_2, \dots, p_N\}$ соответственно, причем каждый поиск совершается абсолютно *независимо* от предыдущих. Можно показать, что среднее число сравнений при нахождении записи в таком самоорганизующемся файле стремится к предельному значению

$$\bar{C}_N = 1 + 2 \sum_{1 \leq i < j \leq N} \frac{p_i p_j}{p_i + p_j} = \frac{1}{2} + \sum_{i,j} \frac{p_i p_j}{p_i + p_j}. \quad (17)$$

(См. упр. 11.) Например, если $p_i = 1/N$ при $1 \leq i \leq N$, самоорганизующаяся таблица всегда находится в случайном порядке, а (17) сводится к знакомому выражению $(N+1)/2$, полученному ранее.

Посмотрим, как самоорганизующаяся процедура работает при распределении вероятностей по закону Зипфа (8). Имеем

$$\begin{aligned} \bar{C}_N &= \frac{1}{2} + \sum_{1 \leq i, j \leq N} \frac{(c/i)(c/j)}{c/i + c/j} = \frac{1}{2} + c \sum_{1 \leq i, j \leq N} \frac{1}{i+j} = \\ &= \frac{1}{2} + c \sum_{1 \leq i \leq N} (H_{N+i} - H_i) = \frac{1}{2} + c \sum_{1 \leq i \leq 2N} H_i - 2c \sum_{1 \leq i \leq N} H_i = \\ &= \frac{1}{2} + c((2N+1)H_{2N} - 2N - 2(N+1)H_N + 2N) = \\ &= \frac{1}{2} + c(N \ln 4 - \ln N + O(1)) \approx \\ &\approx 2N/\log_2 N. \end{aligned}$$

(см. формулы (1.2.7–8,3)). Это гораздо лучше, чем у N при достаточно больших N , и лишь в $\ln 4 \approx 1.386$ раз хуже, чем число сравнений при оптимальном расположении записей (ср. с (9)).

Эксперименты, приводившиеся с таблицами символов в компиляторах, показали, что самоорганизующийся метод работает даже лучше, чем предсказывает (18), так как удачные поиски не независимы (небольшие группы ключей часто появляются вместе).

Схему, подобную самоорганизующейся, изучили Г. Шай и Ф. В. Дауэр [*SIAM J. Appl. Math.*, **15** (1967), 874–888.]

Поиск на ленте среди записей различной длины. Рассмотрим теперь нашу задачу в ином ракурсе. Пусть таблица, по которой производится поиск, хранится на ленте и записи имеют различные длины. Лента с системной библиотекой в старых операционных системах служит примером такого файла. Стандартные программы системы, такие, как компиляторы, компоновщики, загрузчики, генераторы отчетов и т. п., являлись "записями" на ленте, а большинство пользовательских работ должно было начинаться с поиска нужной программы. Такая постановка задачи делает неприменимым предыдущий анализ алгоритма S: теперь шаг S3 выполняется за различные промежутки времени. Значит, нас будет интересовать не только число сравнений.

Обозначим через L_i длину записи R_i , а через p_i , вероятность того, что эта запись будет отыскиваться. Время поиска теперь примерно пропорционально величине

$$p_1 L_1 + p_2 (L_1 + L_2) + \dots + p_N (L_1 + L_2 + \dots + L_N). \quad (19)$$

При $L_1 = L_2 = \dots = L_N = 1$ это, очевидно, сводится к изученному случаю (3).

Кажется логичным поместить наиболее нужные записи в начало ленты, но здесь здравый смысл подводит нас. Действительно, пусть на ленте записаны ровно две программы— A и B . Программа A требуется в два раза чаще B , но длиннее B в четыре раза, т. е. $N = 2$, $p_A = \frac{2}{3}$, $L_A = 4$, $p_B = \frac{1}{3}$, $L_B = 1$. Если мы, следуя "логике", поставим A на первое место, то среднее время поиска составит $\frac{2}{3} \cdot 4 + \frac{1}{3} \cdot 5 = \frac{13}{3}$; но если поступить "нелогично", расположив в начале B , то получится $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 5 = \frac{11}{3}$. Следующая теорема позволяет определить оптимальное расположение библиотечных программ на ленте.

Теорема S. Пусть L_i и p_i , определены, как и раньше. Порядок записей на ленте оптимален тогда и только тогда, когда

$$p_1/L_1 \geq p_2/L_2 \geq \dots \geq p_N/L_N. \quad (20)$$

Иными словами, минимум выражения

$$p_{a_1}L_{a_1} + p_{a_2}(L_{a_1} + L_{a_2}) + \dots + p_{a_N}(L_{a_1} + \dots + L_{a_N})$$

по всем перестановкам $a_1 a_2 \dots a_N$ чисел $\{1, 2, \dots, N\}$ равен (19) тогда и только тогда, когда выполняется (20).

Доказательство. Предположим, что R_i и R_{i+1} поменялись местами; величина (19), ранее равная

$$\dots + p_i(L_1 + \dots + L_{i-1} + L_i) + p_{i+1}(L_1 + \dots + L_{i+1}) + \dots,$$

теперь заменится на

$$\dots + p_{i+1}(L_1 + \dots + L_{i-1} + L_{i+1}) + p_i(L_1 + \dots + L_i) + \dots.$$

Изменение равно $p_i L_{i+1} - p_{i+1} L_i$. Так как расположение (19) оптимально, то $p_i L_{i+1} - p_{i+1} L_i \geq 0$. Значит, $p_i/L_i \geq p_{i+1}/L_{i+1}$, т. е. (20) выполняется.

Докажем теперь достаточность условия (20). Разумеется, "локальная" оптимальность расположения (19) видна сразу: если мы поменяем местами две записи, время поиска изменится на $p_i L_{i+1} - p_{i+1} L_i \geq 0$. Однако "глобальная" оптимальность требует обоснования. Мы рассмотрим два доказательства: одно из них использует дискретную математику, а другое предполагает некоторую математическую изворотливость.

Доказательство. 1. Пусть (20) выполняется. Мы знаем, что любую перестановку записей можно "отсортировать", т. е. привести к расположению R_1, R_2, \dots, R_N , последовательно меняя местами лишь соседние элементы. Каждое такое изменение переводит $\dots R_j R_i \dots$ в $\dots R_i R_j \dots$ для некоторых $i < j$, т. е. уменьшает время поиска на неотрицательную величину $p_i L_j - p_j L_i$. Значит, порядок $R_1 R_2 \dots R_N$ оптимален.

Доказательство. 2. Заменим вероятности p_i на

$$p_i(\varepsilon) = p_i + \varepsilon^i - (\varepsilon^1 + \varepsilon^2 + \dots + \varepsilon^N)/N,$$

где ε — малая положительная величина. Равенство $x_1 p_1(\varepsilon) + \dots + x_N p_N(\varepsilon) = y_1 p_1(\varepsilon) + \dots + y_N p_N(\varepsilon)$ выполняется для достаточно малых ε лишь при $x_1 = y_1, \dots, x_N = y_N$; в частности, в (20) станут невозможными равенства $\frac{p_i}{L_i} = \frac{p_j}{L_j}$. Рассмотрим $N!$ перестановок записей. Среди них есть по крайней мере одна оптимальная; в силу первой части теоремы она удовлетворяет (20), но в силу отсутствия равенств условию (20) удовлетворяет лишь одна перестановка. Таким образом, (20) однозначно определяет оптимальное расположение записей в таблице для вероятностей $p_i(\varepsilon)$ при достаточно малых ε . По непрерывности этот же порядок оптимален и при ε . (Такой метод доказательств часто используется в комбинаторной оптимизации.) ■

Теорема S принадлежит У. Э. Смит [Naval Research Logistics Quarterly, 3 (1956), 59–66]. Упражнения содержат дополнительные результаты по оптимальной организации таблиц.

Уплотнение файлов. Последовательный поиск на ленте и других внешних запоминающих устройствах протекает быстрее; если данные занимают меньше места, поэтому полезно рассмотреть несколько различных способов представления данных в таблице. Не всегда нужно запоминать ключи в явном виде.

Пусть, например, требуется таблица простых чисел до миллиона для разложения на множители 12-значных чисел. (Ср. с п. 4.5.4.) Таких чисел имеется 78498; используя 20 битов для каждого из них, мы получим файл длины 1 569 960 битов. Это явное расточительство, так как можно иметь миллион-битовую таблицу, в которой разряды, соответствующие простым числам, равны 1. Поскольку все простые числа (кроме двойки) нечетны, достаточно иметь таблицу в 500000 битов.

Другим способом уменьшения длины файла является хранение не самих простых чисел, а интервалов между ними. В соответствии с табл. 1 величина $(p_{k+1} - p_k)/2$ меньше 64 для всех простых чисел в пределах 1 357 201, т. е. нам достаточно запомнить 78 496 шестизначных чисел (размер

интервала)/2. Такая таблица имеет длину примерно 471 000 битов. Дальнейшего уплотнения можно добиться, используя для представления интервалов двоичные коды переменной длины (ср. с п 6.2.2).

Таблица 1

Таблица интервалов между последовательными простыми числами

Интервал ($p_{k+1} - p_k$)	p_k	Интервал ($p_{k+1} - p_k$)	p_k
1	2	52	19609
2	3	72	31397
4	7	86	155921
6	23	96	360653
8	89	112	370261
14	113	114	492113
18	523	118	1349533
20	887	132	1357201
23	1129	148	2010733
34	1327	154	4652353
36	9551	180	17051707
44	15683	310	20831323

В таблице помещены интервалы $p_{k+1} - p_k$, превышающие $p_{j+1} - p_j$ для всех $j < k$. Более подробно см. F. Gruenberger, G. Armerding, "Statistics on the first six million prime numbers", RAND Corp. report P-2460 (October, 1961).

6.2.3. Сбалансированные деревья

Только что изученный нами алгоритм вставки в дерево порождает хорошие деревья поиска при случайных исходных данных, но все же существует досадная вероятность получить вырожденное дерево. Возможно, мы могли бы изобрести алгоритм, который в любом случае дает оптимальное дерево, но, к сожалению, это далеко не просто. Другой подход состоит в хранении полной длины пути и реорганизации дерева всякий раз, когда длина его пути превышает, скажем, $5N \log_2 N$. Но тогда в процессе построения дерева потребовалось бы около $\sqrt{N/2}$ реорганизаций.

Очень остроумное решение проблемы поддержания хорошего дерева поиска было найдено в 1962 г. двумя советскими математиками—Г. М. Адельсоном-Вельским и Е. М. Ландисом [ДАН СССР, 146 (1962), 263–266]. Их метод требует лишь двух дополнительных битов на узел и никогда не использует более $O(\log N)$ операций для поиска по дереву или для вставки элемента. В дальнейшем мы увидим, что этот подход также приводит к общему методу представления произвольных линейных списков длины N , причем каждая из следующих операций требует лишь $O(\log N)$ единиц времени:

- i) Найти элемент по данному ключу.
- ii) При данном k найти k -й элемент.
- iii) Вставить в определенном месте элемент.
- iv) Удалить определенный элемент.

Если для линейных списков принято последовательное расположение, то операции (i) и (ii) будут эффективными, но операции (iii) и (iv) займут порядка N шагов; с другой стороны, при использовании связанного расположения эффективны операции (iii) и (iv), а (i) и (ii) потребуют порядка N шагов. Представление линейных списков в виде дерева позволяет сделать все четыре операции за $O(\log N)$ шагов. Можно также сравнительно эффективно производить другие стандартные операции; например, возможна конкатенация (сцепление) списка из M элементов со списком из N элементов за $O(\log(M + N))$ шагов.

Метод, дающий все эти преимущества использует так называемые "сбалансированные деревья". Предыдущий абзац служит рекламой сбалансированных деревьев—этакой панацеей от всех бед; по сравнению с ними все другие способы представления данных кажутся устаревшими. Но необходимо сбалансировать наше отношение к сбалансированным деревьям! Если требуются не все, четыре рассмотренные операции, то нас может удовлетворить значительно менее универсальный, но проще программируемый метод. Более того, сбалансированные деревья хороши лишь при достаточно больших N ; так, если есть эффективный алгоритм, требующий $20 \log_2 N$ единиц времени, и неэффективный алгоритм, требующий $2N$ единиц времени, то при $N < 1024$ следует использовать неэффективный метод. С другой стороны, N не должно быть слишком велико; сбалансированные деревья подходят главным образом для хранения данных во *внутренней* памяти, а в п. 6.2.4 мы изучим лучшие методы для внешних файлов с прямым доступом. Так как со временем размеры внутренней памяти становятся все больше и больше, сбалансированные деревья становятся все более важными.

Высота дерева определяется как его наибольший уровень, как максимальная, длина пути от корня до внешнего узла.

Picture: 20. Сбалансированное бинарное дерево

Бинарное дерево называется *сбалансированным*; если высота левого поддерева каждого узла отличается от высоты правого поддерева не более чем на ± 1 . На рис. 20 показано сбалансированное дерево с 17 внутренними узлами и высотой 5; *показатель сбалансированности* представлен внутри каждого узла знаками $+$, \cdot или $-$, что отвечает разности высот правого и левого поддеревьев, равной $+1$, 0 или -1 соответственно. Фибоначчиево дерево на рис. 8 (п. 6.2.1) является другим сбалансированным бинарным деревом высоты 5, имеющим только 12 внутренних узлов; большинство показателей сбалансированности равно -1 . "Зодиакальное дерево" на рис. 10 (п. 6.2.2) не сбалансировано, так как поддерева узлов AQUARIUS и GEMINI не удовлетворяют принятым ограничениям.

Это определение сбалансированности представляет собой компромисс между *оптимальными* бинарными деревьями (все внешние узлы которых расположены на двух смежных уровнях) и *произвольными* бинарными деревьями. Поэтому уместно спросить, как далеко может отклониться от оптимальности сбалансированное дерево? Оказывается, что длина его поискового пути никогда не превысит оптимум более чем на 45%.

Теорема А. (Г. М. Адельсон-Вельский и Е. М. Ландис). *Высота сбалансированного дерева с N внутренними узлами заключена между $\log_2(N + 1)$ и $1.4404 \log_2(N + 2) - 0.328$.*

Доказательство. Бинарное дерево высоты h , очевидно, не может содержать более чем 2^h внешних узлов; поэтому $N + 1 \leq 2^h$, т.е. $h \geq \lceil \log_2(N + 1) \rceil$.

Чтобы найти максимальное значение h , поставим вопрос по-другому: каково минимальное число узлов в сбалансированном дереве высоты h ? Пусть T_h — такое дерево с наименьшим возможным количеством узлов; тогда одно поддереву корня, например левое, имеет высоту $h - 1$, а другое — или $h - 1$, или $h - 2$. В силу определения T_h можно считать, что левое поддерево корня есть T_{h-1} , а правое — T_{h-2} . Таким образом, среди всех сбалансированных деревьев высоты h наименьшее количество узлов имеет *фибоначчиево дерево* порядка $h + 1$. (См. определение деревьев Фибоначчи в п. 6.2.1.) Итак,

$$N \geq F_{h+2} - 1 > \phi^{h+2} / \sqrt{5} - 2,$$

и требуемый результат получается так же, как следствие из теоремы 4.5.3F. ■

Мы видим, что поиск в сбалансированном дереве потребует более 25 сравнений, только если дерево состоит из по крайней мере $F_{27} - 1 = 196417$ узлов.

Рассмотрим теперь, что происходит, когда новый узел вставляется в сбалансированное дерево посредством алгоритма 6.2.2T. Дерево на рис. 20 остается сбалансированным, если новый узел займет место одного из узлов [4], [5], [6], [7], [10] или [13], но в других случаях потребуются некоторая корректировка. Трудности возникают, если имеется узел с показателем сбалансированности $+1$, правое поддерево которого после вставки становится выше, или если показатель сбалансированности равен -1 и выше. становится левое поддерево. Легко понять, что, в сущности, нас беспокоят лишь два случая:

Picture: Случай вставки в AVL-дерево

(Другие "плохие" случаи можно получить, зеркально отразив эти диаграммы относительно вертикальной оси.) Большими прямоугольниками α , β , γ , δ обозначены поддеревья с соответствующими высотами. Случай 1 имеет место, если новый элемент увеличил высоту правого поддерева узла B с h до $h + 1$, а случай 2 — когда новый элемент увеличивает высоту левого поддерева узла B . Во втором случае мы имеем либо $h = 0$ (и тогда сам узел X является новым узлом), либо узел X имеет два поддерева с соответственными высотами $(h - 1, h)$ или $(h, h - 1)$.

Простые преобразования восстанавливают баланс в обоих случаях, сохраняя в то же время симметричный порядок узлов A , B и X .

Picture: Повороты

В случае 1 мы просто поворачиваем дерево налево, прикрепляя β к A вместо B . Это преобразование подобно применению ассоциативного закона к алгебраической формуле, когда мы заменяем $\alpha(\beta\gamma)$ на $(\alpha\beta)\gamma$. В случае 2 это делается дважды: сначала (X, B) поворачивается направо, затем (A, X) — налево. В обоих случаях нужно изменить в дереве лишь несколько ссылок. Далее новые деревья имеют высоту $h + 2$, в точности ту же, что и до вставки элемента; следовательно, часть дерева, расположенная над узлом A (если таковая имеется), остается сбалансированной.

Picture: Дерево рис. 20, сбалансированное после вставки нового ключа R

Например, если мы вставляем новый узел на место [17] (рис. 20), то после поворота получим сбалансированное дерево, изображенное на рис. 21 (случай 1). Заметьте, что некоторые из показателей сбалансированности изменились.

Детали этой процедуры вставки можно разработать различными способами. На первый взгляд без вспомогательного стека не обойтись, так как необходимо запоминать узлы, которые будут затронуты вставкой. Ниже приводится алгоритм, в котором, прибегнув к маленькой хитрости, мы обходимся без стека, выигрывая при этом в скорости.

Алгоритм А. (Поиск, с вставкой по сбалансированному дереву.) Имеется таблица записей, образующих сбалансированное бинарное дерево. Алгоритм позволяет произвести поиск данного аргумента K . Если K в таблице нет, в подходящем месте в дерево вставляется новый узел, содержащий K . При необходимости производится балансировка дерева.

Предполагается (как и в алгоритме 6.2.2T), что узлы содержат поля KEY, LLINK и RLINK. Кроме того, имеется новое поле $B(P)$ = показатель сбалансированности узла $NODE(P)$, т. е. разность высот правого и левого поддеревьев; это поле всегда содержит $+1$, 0 или -1 . По адресу HEAD расположен специальный головной узел; RLINK (HEAD) указывает на корень дерева, а LLINK (HEAD) используется для хранения полной высоты дерева. Для данного алгоритма высота не имеет значения, но знание ее полезно для процедуры конкатенации, обсуждающейся ниже. Мы предполагаем, что дерево *непусто*, т. е. что $RLINK(HEAD) \neq \Lambda$.

В целях удобства описания в алгоритме используется обозначение $LINK(a, P)$ как синоним $LLINK(P)$ при $a = -1$ и как синоним $RLINK(P)$ при $a = +1$.

A1 [Начальная установка.] Установить $T \leftarrow HEAD$, $S \leftarrow P \leftarrow RLINK(HEAD)$. [Указательная переменная P будет двигаться вниз по дереву; S будет указывать на место, где может потребоваться балансировка; T всегда указывает на отца S .]

A2 [Сравнение.] Если $K < KEY(P)$, то перейти на **A3**; если $K > KEY(P)$, то перейти на **A4**; если $K = KEY(P)$, поиск удачно завершен.

A3 [Шаг влево.] Установить $Q \leftarrow LLINK(P)$. Если $Q = \Lambda$, выполнить $Q \leftarrow AVAIL$ и $LLINK(P) \leftarrow Q$; затем идти на **A5**. В противном случае, если $B(Q) \neq 0$, установить $T \leftarrow P$ и $S \leftarrow Q$. Наконец, установить $P \leftarrow Q$ и вернуться на **A2**.

A4 [Шаг вправо.] Установить $Q \leftarrow RLINK(P)$. Если $Q = \Lambda$, выполнить $Q \leftarrow AVAIL$ и $RLINK(P) \leftarrow Q$; затем идти на **A5**.

В противном случае, если $B(Q) \neq 0$, установить $T \leftarrow P$ и $S \leftarrow Q$. Наконец, установить $P \leftarrow Q$ и вернуться на **A2**. (Последнюю часть этого шага можно объединить с последней частью шага **A3**.)

A5 [Вставка.] (Мы только что присоединили новый узел $NODE(Q)$ к дереву; теперь его поля нуждаются в начальной установке.) Установить $KEY(Q) \leftarrow K$, $LLINK(Q) \leftarrow RLINK(Q) \leftarrow \Lambda$, $B(Q) \leftarrow 0$.

A6 [Корректировка показателей сбалансированности.] (Теперь нулевые показатели сбалансированности между S и Q нужно заменить на ± 1 .) Если $K < KEY(S)$, установить $R \leftarrow P \leftarrow LLINK(S)$; в противном случае установить $R \leftarrow P \leftarrow RLINK(S)$. Затем нужно 0 или более раз повторять следующую операцию, пока P не станет равным Q : если $K < KEY(P)$, установить $B(P) \leftarrow -1$ и $P \leftarrow LLINK(P)$; если $K > KEY(P)$, установить $B(P) \leftarrow +1$ и $P \leftarrow RLINK(P)$. (Если $K = KEY(P)$, значит, $P = Q$, и можно перейти к следующему шагу.)

A7 [Проверка сбалансированности.] Если $K < KEY(S)$, установить $a \leftarrow -1$; в противном случае $a \leftarrow +1$. Теперь возможны три случая:

- i) Если $B(S) = 0$ (дерево стало выше), установить $B(S) \leftarrow a$, $LLINK(HEAD) \leftarrow LLINK(HEAD) + 1$; алгоритм завершен.
- ii) Если $B(S) = -a$ (дерево стало более сбалансированным), установить $B(S) \leftarrow 0$; алгоритм завершен.
- iii) Если $B(S) = a$ (дерево перестало быть сбалансированным), при $B(R) = a$ идти на **A8**, при $B(R) = -a$ идти на **A9**.

(Случай (iii) соответствует ситуации, изображенной на диаграмме (1), при $a = +1$; S и R указывают соответственно на узлы A и B , а $LINK(-a, S)$ указывает на α и т.д.)

A8 [Однократный поворот.] Установить $P \leftarrow R$, $LINK(a, S) \leftarrow LINK(-a, R)$, $LINK(-a, R) \leftarrow S$, $B(S) \leftarrow B(R) \leftarrow 0$. Перейти на **A10**.

A9 [Двукратный поворот.] Установить $P \leftarrow LINK(-a, R)$, $LINK(-a, R) \leftarrow LINK(a, P)$, $LINK(a, P) \leftarrow R$, $LINK(a, S) \leftarrow LINK(-a, P)$, $LINK(-a, P) \leftarrow S$. Теперь установить

$$(B(S), B(R)) \leftarrow \begin{cases} (-a, 0), & \text{если } B(P) = a; \\ (0, 0), & \text{если } B(P) = 0; \\ (0, a), & \text{если } B(P) = -a; \end{cases} \quad (3)$$

затем $B(P) \leftarrow 0$.

A10 [Последний штрих.] [Мы завершили балансирующее преобразование от (1) к (2), P указывает на новый корень, а T — на отца старого корня.] Если $S = RLINK(T)$, то установить $RLINK(T) \leftarrow P$; в противном случае $LLINK(T) \leftarrow P$. ■

Этот алгоритм довольно длинный, но разделяется на три простые части: шаги **A1–A4** (поиск), шаги **A5–A7** (вставка нового узла), шаги **A8–A10** (балансировка дерева, если она нужна).

Picture: 22. Поиск с вставкой по сбалансированному дереву

Мы знаем, что для работы алгоритма требуется около $C \log N$ единиц времени при некотором C , но чтобы знать, при каких N выгодно использовать сбалансированные деревья, нужно оценить величину C . Анализ следующей MIX-программы позволяет подойти к решению этого вопроса.

Программа А. (Поиск с вставкой по сбалансированному дереву.) Эта реализация алгоритма **A** использует следующий формат узлов дерева:

Picture: Формат узла АВЛ-дерева

$rA \equiv K$, $rI1 \equiv P$, $rI2 \equiv Q$, $rI3 \equiv R$, $rI4 \equiv S$, $rI5 \equiv T$. Программа для шагов **A7–A9** дублируется, так что величина a в явном виде в программе не фигурирует.

B	EQU	0:1		
LLINK	EQU	2:3		
RLINK	EQU	4:5		
START	LDA	K	1	A1. Начальная установка.
	ENT5	HEAD	1	$T \leftarrow \text{HEAD}$.
	LD2	0,5 (RLINK)	1	$Q \leftarrow \text{RLINK}(\text{HEAD})$.
	JMP	2F	1	На A2 с $S \leftarrow P \leftarrow Q$
4H	LD2	0,1 (RLINK)	C2	A4. Шаг вправо. $Q \leftarrow \text{RLINK}(P)$
	J2Z	5F	C2	На A5, если $Q = \Lambda$.
1H	LDX	0,2 (B)	C - 1	$rX \leftarrow B(Q)$.
	JXZ	*+3	C - 1	Переход, если $B(Q) = 0$.
	ENT5	0,1	D - 1	$T \leftarrow P$.
2H	ENT4	0,2	D	$S \leftarrow Q$.
	ENT1	0,2	C	$P \leftarrow Q$.
	CMPA	1,1	C	A2. Сравнение.
	JG	4B	C	На A4, если $K > \text{KEY}(P)$.
	JE	SUCCESS	C1	Выход, если $K = \text{KEY}(P)$.
	LD2	0,1 (LLINK)	C1 - S	A3. Шаг влево. $Q \leftarrow \text{LLINK}(P)$.
	J2NZ	1B	C1 - S	Переход, если $Q \neq \Lambda$.
20-29 5H (скопировать здесь строки 14-23 программы 6.2.2 T) A5. Вставка.				
6H	CMPA	1,4	1 - S	A6. Коррект. показат. сбалансир.
	JL	*+3	1 - S	Переход, если $K < \text{KEY}(S)$.
	LDS	0,4 (RLINK)	E	$R \leftarrow \text{RLINK}(S)$.
	JMP	*+2	E	
	LD3	0,4 (LLINK)	1 - S - E	$R \leftarrow \text{LLINK}(S)$.
	ENT1	0,3	1 - S	$P \leftarrow R$.
	ENTX	-1	1 - S	$rX \leftarrow -1$.
	JMP	1F	1 - S	На цикл сравнения.
4H	JE	7F	F2 + 1 - S	На A7, если $K = \text{KEY}(P)$.
	STX	0,1 (1:1)	F2	$B(P) \leftarrow +1$ (он был +0).
	LD1	0,1 (RLINK)	F2	$P \leftarrow \text{RLINK}(P)$.
1H	CMPA	1,1	F + 1 - S	
	JGE	4B	F + 1 - S	Переход, если $K \geq \text{KEY}(P)$.
	STX	0,1 (B)	F1	$B(P) \leftarrow -1$.
	LD1	0,1 (LLINK)	F1	$P \leftarrow \text{LLINK}(P)$.
	JMP	1B	F1	На цикл сравнения.
7H	LD2	0,4(B)	1 - S	A7. Проверка сбалансир. $rI2 \leftarrow B(S)$.
	STZ	0,4 (B)	1 - S	$B(S) \leftarrow 0$.
	CMPA	1,4	1 - S	
	JG	A7R	1 - S	На $a = +1$ подпрограмму, если $K > \text{KEY}(S)$.

(здесь еще код в 2 колонки — приделать)

лиону, их будет примерно 20. Но если разделить дерево на "страницы" по 7 узлов в каждой, как показано пунктиром на рис. 29, и если в каждый момент времени доступна лишь одна страница, то потребуется примерно в три раза меньше обращений, т. е. поиск ускорится в три раза!

Группировка узлов в страницы, по существу, преобразует бинарные деревья в октарные, разветвляющиеся в каждой странице-узле на 8 путей. Если допустимы страницы больших размеров, разветвляющиеся на 128 путей после каждого обращения к диску, то можно находить требуемый ключ в таблице из миллиона элементов, просмотрев лишь три страницы. Можно постоянно хранить корневую страницу во внутренней памяти; тогда потребуются лишь два обращения к диску, хотя в любой момент времени во внутренней памяти будет не более 254 ключей.

Разумеется, не следует делать страницы очень большими, так как размеры внутренней памяти ограничены и чтение большей страницы занимает больше времени. Предположим, например, что чтение страницы, допускающей разветвление на m путей, занимает $72.5 + 0.05m$ мс. Время на внутреннюю обработку каждой страницы составит около $a + b \log m$ мс, где a мало по сравнению с 72.5, так что полное время, требующееся на поиск в большой таблице, примерно пропорционально $\log N$, умноженному на

$$(72.5 + 0.05m) / \log m + b.$$

Эта величина достигает минимума при $m \approx 350$; в действительности этот минимум очень "широк". Значения, очень близкие к оптимальному, достигаются при m от 200 до 500. На практике получение подобного диапазона хороших значений m зависит от характеристик используемых внешних запоминающих устройств и от длины записей в таблице.

У. И. Ландауэр [IEE Trans., EC-12 (1963), 863–871] предложил строить m -арные деревья следующим образом: размещать узлы на уровне $l + 1$, лишь если уровень l почти заполнен. Эта схема требует довольно сложной системы поворотов, так как для вставки одного нового элемента может потребоваться основательная перестройка дерева; Ландауэр исходил из предположения, что поиск приходится производить гораздо чаще вставок и удалений.

Если файл хранится на диске и является объектом сравнительно редких вставок и удалений, то для его представления подходит трехуровневое дерево, где первый уровень разветвления определяет, какой цилиндр будет использоваться, следующий уровень разветвления определяет соответствующую дорожку на этом цилиндре, а третий уровень содержит собственно записи. Такой метод называется *индексно-последовательной* организацией файла [ср. JACM, 16 (1969), 569–571].

Р. Мюнц и Р. Узгалис [Proc. Princeton Conf. on Inf. Sciences and Systems, 4 (1970), 345–349] предложили модификацию алгоритма 6.2.2Т, где все вставки порождают узлы, принадлежащие той же странице, что и их предшественник (если только это возможно); если страница полностью занята, заводится новая страница (если таковая имеется). При неограниченном количестве страниц и случайном порядке поступающих данных среднее число обращений, к диску, как можно показать, приближенно равно $H_N / (H_m - 1)$, что лишь немногим больше числа обращений в случае наилучшего возможного m -арного дерева (см. упр. 10).

В-деревья. В 1970 г. Р. Бэйер и Э. Мак-Крэйт [Acta Informatica, (1972), 173–189] и независимо от них примерно в то же время М. Кауфман [неопубликовано] предложили новый подход к внешнему поиску посредством сильно ветвящихся деревьев. Их идея основывается на подвижности нового типа структур данных, названных *В-деревьями*, и позволяет производить поиск или изменять большой файл с "гарантированной" эффективностью в наихудшем случае, используя при этом сравнительно простые алгоритмы.

В-деревом порядка m называется дерево, обладающее следующими свойствами:

- i) Каждый узел имеет не более m сыновей.
- ii) Каждый узел, кроме корня и листьев, имеет не менее $m/2$ сыновей.
- iii) Корень, если он не лист, имеет не менее 2 сыновей.
- iv) Все листья расположены на одном уровне и не содержат информации.
- v) Нелистовой узел с k сыновьями содержит $k - 1$ ключей.

(Как и обычно, лист — концевой узел, у него нет преемников. Так как листья не содержат информации, их можно рассматривать как внешние узлы, которых в действительности нет в дереве, так что Λ — это указатель на лист.)

На рис. 30 показано В-дерево порядка 7. Каждый узел (кроме корня и листьев) имеет от $\lceil 7/2 \rceil$ до 7 преемников и содержит 3, 4, 5 или 6 ключей. Корневой узел может содержать от 1 до 6 ключей (в данном случае 2). Все листья находятся на третьем уровне. Заметьте, что (а) ключи расположены в возрастающем порядке слева направо, если использовать естественное обобщение понятия симметричного порядка, и (б) количество листьев ровно на единицу больше количества ключей.

В-деревья порядка 1 и 2, очевидно, неинтересны; поэтому мы рассмотрим лишь случай $m \geq 3$. Заметьте, что (3-2)-деревья,

Picture: Рис. 30. В-дерево порядка 7

определенные в конце п. 6.2.3, являются В-деревьями порядка 3; и обратно, В-дерево порядка 3 есть (3-2)-дерево.

Узел, содержащий j ключей и $j + 1$ указателей, можно представить в виде

Picture: Узел В-дерева

где $K_1 < K_2 < \dots < K_j$, а P_i указывает на поддерево ключей, больших K_i и меньших K_{i+1} . Следовательно, поиск в В-дереве абсолютно прямолинеен: после того как узел (1) размещен во внутренней памяти, мы ищем данный аргумент среди ключей K_1, K_2, \dots, K_j . (При больших j , вероятно, производится бинарный поиск, а при малых j наилучшим будет последовательный поиск.) Если поиск удачен, нужный ключ найден; если поиск неудачен в силу того, что аргумент лежит между K_i и K_{i+1} , мы "подкачиваем" (вызываем в оперативную память) узел, указанный P_i , и продолжаем процесс. Указатель P_0 используется, если аргумент меньше K_1 , а P_j используется, если аргумент больше K_j . При $P_i = \Lambda$ поиск неудачен.

Привлекательной чертой В-деревьев является исключительная простота, с которой производятся вставки. Рассмотрим, например, рис. 30; каждый лист соответствует месту возможной вставки. Если нужно вставить новый ключ 337, мы просто меняем узел

Picture: Вставка в В-дерево

С другой стороны, при попытке вставить новый ключ 071 мы обнаруживаем, что в соответствующем узле уровня 2 нет места—он уже "полон". Эту трудность можно преодолеть, расщепив узел на две части по три ключа в каждой и подняв средний ключ на уровень 1:

Picture: Расщепление узла при вставке

Вообще, если нужно вставить новый элемент в В-дерево порядка m , все листья которого расположены на уровне l , новый ключ вставляют в подходящий узел уровня $l - 1$. Если узел теперь содержит m ключей, т. е. имеет вид (1) с $j = m$, его расщепляют на два узла

Picture: Расщепление узла: общий вид

и вставляют ключ $K_{\lceil m/2 \rceil}$ в узел-отца. (Таким образом, указатель P в нем заменяется последовательностью $P, K_{\lceil m/2 \rceil}, P'$.) Эта вставка в свою очередь может привести к расщеплению узла-отца. (Ср. с рис. 27, где изображен случай $m = 3$.) Если нужно расщепить корневой узел, который не имеет отца, то просто создают новый корень и помещают в него единственный ключ $K_{\lceil m/2 \rceil}$, в таком случае дерево становится на единицу выше.

Описанная процедура вставки сохраняет все свойства В-деревьев; чтобы оценить всю красоту идеи, читатель должен выполнить упр. 1. Заметьте, что дерево мало-помалу растет вверх от верхней части, а не вниз от нижней части, так как единственная причина, способная вызвать рост дерева,—расщепление корня.

Удаления из В-деревьев лишь немногим сложнее вставок (см. упр. 7).

Гарантированная эффективность. Рассмотрим, к скольким узлам будет происходить обращение в наихудшем случае при поиске в В-дереве порядка m . Предположим, что имеется N ключей, а на уровне l расположено $N + 1$ листьев. Тогда число узлов на уровнях 1, 2, 3, ... не менее 2, $2^{\lceil m/2 \rceil}$, $2^{\lceil m/2 \rceil^2}, \dots$; следовательно,

$$N + 1 \geq 2^{\lceil m/2 \rceil^{l-1}}. \quad (5)$$

Иными словами,

$$l \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{N + 1}{2} \right); \quad (6)$$

это означает, например, что при $N = 1999998$ и $m = 199$ имеем $l \leq 3$. Так как при поиске происходит обращение самое большее к l узлам, эта формула гарантирует малое время работы алгоритма.

При вставке нового ключа может понадобиться расщепить l узлов. Однако среднее число расщепляемых узлов много меньше, так как общее количество расщеплений при построении дерева равно на единицу меньше количества узлов, которое мы обозначим через p . В дереве не менее $1 + (\lceil m/2 \rceil - 1)(p - 1)$