Содержание

Глава 8

Слияние и сортировка слиянием

Семейство алгоритмов быстрой сортировки, рассмотренное в главе 7, основано на операции *выборки*, т.е. на нахождении k-го минимального элемента в файле. Мы убедились в том, что выполнение операции выборки аналогично делению файла на две части: часть, содержащую k меньших элементов, и часть, содержащую $N-k$ бóльших элементов. В этой главе мы рассмотрим семейство алгоритмов сортировки, основанных на противоположном процессе, *слиянии*, т.е. объединении двух отсортированных файлов в один файл большего размера. На слиянии основан простой алгоритм сортировки вида "разделяй и властвуй" (см. раздел 5.2), а также его двойник — алгоритм восходящей сортировки слиянием, при этом оба алгоритма достаточно просто реализуются.

Выборка и слияние — противоположные операции в том смысле, что выборка разбивает файл на два независимых файла, а слияние объединяет два независимых файла в один. Контраст между этими двумя операциями становится очевидным при применении принципа "разделяй и властвуй" для создания конкретных методов сортировки. Можно переупорядочить файл таким образом, что если отсортировать обе части файла, становится упорядоченным и весь файл; и наоборот, можно разбить файл на две части, отсортировать их, а затем объединить упорядоченные части и получить весь файл в упорядоченном виде. Мы уже видели, что получается в первом случае: это быстрая сортировка, состоящая из процедуры выборки, за которой следуют два рекурсивных вызова. В этой главе мы рассмотрим *сортировку слиянием* (mergesort), которая является противоположностью быстрой сортировки, поскольку состоит из двух рекурсивных вызовов с последующей процедурой слияния.

Одним из наиболее привлекательных свойств сортировки слиянием является тот факт, что она сортирует файл, состоящий из N элементов, за время, пропорциональное $N \log N$, независимо от характера входных данных. В главе 9 мы познакомимся с еще одним алгоритмом, время выполнения которого гарантированно пропорционально $N \log N$; этот алгоритм носит название *пирамидальной сортировки* (heapsort). Основной недостаток сортировки слиянием заключается в том, что простые реализации этого алгоритма требуют объема дополнительной памяти, пропорционального N. Это препятствие можно преодолеть, однако способы сделать это настолько сложны, что практически неприменимы, особенно если учесть, что можно воспользоваться пирамидальной сортировкой. Кодирование сортировки слиянием не труднее кодирования пирамидальной сортировки, а длина ее внутреннего цикла находится между аналогичными показателями быстрой сортировки и пирамидальной сортировки; так что сортировка методом

слияния достойна внимания, если важно быстродействие, недопустимо ухудшение производительности на "неудобных" входных данных и доступна дополнительная память.

Гарантированное время выполнения, пропорциональное $N \log N$, может быть и недостатком. Например, в главе 6 были описаны методы, которые могут быть адаптированы таким образом, что в некоторых особых ситуациях время их выполнения может быть линейным — например, при достаточно высокой упорядоченности файла либо при наличии лишь нескольких различных ключей. В противоположность этому время выполнения сортировки слиянием зависит главным образом от числа ключей входного файла и практически не чувствительно к их упорядоченности.

Сортировка слиянием является устойчивой, и это склоняет чашу весов в ее пользу в тех приложениях, в которых устойчивость важна. Конкурирующие с ней методы, такие как быстрая сортировка или пирамидальная сортировка, не относятся к числу устойчивых. Различные приемы, обеспечивающие устойчивость этих методов, обычно требуют дополнительной памяти; поэтому если на первый план выдвигается устойчивость, то требование дополнительной памяти для сортировки слиянием становится менее важным.

У сортировки слиянием есть еще одно, иногда очень важное, свойство: она обычно реализуется таким образом, что обращается к данным в основном последовательно (один элемент за другим). Поэтому сортировка слиянием удобна для упорядочения связных списков, для которых применим только метод последовательного доступа. По тем же причинам, как мы увидим в главе 11, на слиянии часто основываются методы сортировки для специализированных и высокопроизводительных машин, поскольку в таких вычислительных средах последовательный доступ к данным является самым быстрым.

8.1. Двухпутевое слияние

При наличии двух упорядоченных входных файлов их можно объединить в один упорядоченный выходной файл, просто повторяя цикл, в котором меньший из двух элементов, наименьших в своих файлах, переносится в выходной файл; и так до исчерпания обоих входных файлов. В этом и следующем разделах будут рассмотрены несколько реализаций этой базовой абстрактной операции. Время выполнения линейно зависит от количества элементов в выходном файле, если на каждую операцию поиска следующего наименьшего элемента в любом входном файле затрачивается постоянное время, а это верно, если отсортированные файлы представлены структурой данных, поддерживающей последовательный доступ за постоянное время, наподобие связного списка или массива. Эта процедура представляет собой *двухпутевое слияние* (two-way merging); в главе 11 мы подробно изучим *многопутевое слияние*, в котором принимают участие более двух файлов. Наиболее важным приложением многопутевого слияния является внешняя сортировка, которая подробно рассматривается там же.

Для начала предположим, что имеются два отдельных упорядоченных массива целых чисел `a[0]`, ..., `a[N-1]` и `b[0]`, ..., `b[M-1]`, которые нужно объединить в третий массив `c[0]`, ..., `c[N+M-1]`. Легко реализуемая очевидная стратегия заключается в том, чтобы последовательно выбирать в `c` наименьший элемент из оставшихся в `a` и `b`, как показано в программе 8.1. Эта простая реализация обладает двумя важными характеристиками, которые мы сейчас рассмотрим.

Во-первых, в данной реализации предполагается, что массивы не пересекаются. В частности, если `a` и `b` — большие массивы, то для размещения выходных данных необходим третий, тоже большой, массив `c`. Было бы хорошо не задействовать дополнительную память, пропорциональную размеру выходного файла, а применить такой метод,

Программа 8.1. Слияние

Чтобы объединить два упорядоченных массива a и b в упорядоченный массив c, используется цикл for, который на каждой итерации помещает в массив c очередной элемент. Если массив a исчерпан, элемент берется из b; если исчерпан b, то элемент берется из a; если же элементы есть и в том, и в другом массиве, то в c переносится наименьший из оставшихся элементов в a и b. Предполагается, что оба входных массива упорядочены, и что массив c не пересекается (т.е. не перекрывает и не использует совместную память) с массивами a и b.

```
template <class Item>
void mergeAB(Item c[], Item a[], int N, Item b[], int M )
  { for (int i = 0, j = 0, k = 0; k < N+M; k++)
      {
        if (i == N) { c[k] = b[j++]; continue; }
        if (j == M) { c[k] = a[i++]; continue; }
        c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
      }
  }
```

который объединяет два упорядоченных файла a[1], ..., a[m] и a[m+1], ..., a[r] в один упорядоченный файл, просто перемещая элементы a[1], ..., a[r] без использования существенного объема дополнительной памяти. Здесь стоит остановиться и подумать о том, как это можно сделать. На первый взгляд кажется, что эту задачу решить просто, однако на самом деле все известные до сих пор решения достаточно сложны, особенно по сравнению с программой 8.1. Оказывается, довольно трудно разработать алгоритм обменного (т.е. на том же месте) слияния, который обошел бы по производительности альтернативные *обменные сортировки*. Мы еще вернемся к этому вопросу в разделе 8.2.

Слияние, как операция, имеет свою собственную область применения. Например, в типичной среде обработки данных может оказаться нужным использовать большой (упорядоченный) файл данных, в который регулярно добавляются новые элементы. Один из подходов заключается в *пакетном добавлении* новых элементов в главный (намного больший) файл и последующей сортировке всего файла. Однако эта ситуация как будто специально создана для слияния: гораздо эффективнее отсортировать (небольшой) пакет новых элементов и потом слить полученный небольшой файл с большим главным файлом. Слияние используется во многих аналогичных приложениях, так что изучать его, несомненно, стоит. Поэтому основное внимание в данное главе будет уделяться методам сортировки, в основу которых положено слияние.

Упражнения

8.1. Предположим, что упорядоченный файл размером N нужно объединить с неупорядоченным файлом размером M, причем M намного меньше N. Допустим, что у нас имеется программа сортировки, которая упорядочивает файл размером N за $c_1 N \lg N$ секунд, и программа слияния, которая может слить файл размером N с файлом размером M за $c_2(N + M)$ секунд, при $c_1 \approx c_2$. Во сколько раз быстрее, чем сортировка заново, работает предложенный метод, основанный на слиянии, если его рассматривать как функцию от M, при $N = 10^3$, 10^6 и 10^9 ?

8.2. Сравните сортировку всего файла методом вставок и два метода, представленные в упражнении 8.1. Считайте, что меньший файл упорядочен случайным образом, так что каждая вставка проходит примерно полпути в большом файле, а время выполнения сортировки имеет порядок $c_3MN/2$, при этом константа c_3 примерно равна другим константам.

8.3. Опишите, что произойдет, если попробовать воспользоваться программой 8.1 для обменного слияния с помощью вызова `merge(a, a, N/2, a+N/2, N-N/2)` для ключей `A E Q S U Y E I N O S T`.

○ **8.4.** Верно ли, что программа 8.1, вызванная так, как описано в упражнении 8.3, дает правильный результат тогда и только тогда, когда оба входных подмассива отсортированы? Обоснуйте свой ответ или приведите контрпример.

8.2. Абстрактное обменное слияние

Хотя реализация слияния требует дополнительной памяти, все же *абстракция* обменного слияния полезна при реализации изучаемых здесь методов сортировки. В нашей следующей реализации слияния это будет подчеркнуто с помощью сигнатуры функции `merge(a, l, m, r)`, что означает, что подпрограмма `merge` помещает результат слияния `a[1]`, ..., `a[m]` и `a[m+1]`, ..., `a[r]` в объединенный упорядоченный массив `a[1]`, ..., `a[r]`. Эту программу слияния можно было бы реализовать, сначала скопировав все входные данные во вспомогательный массив и затем применив базовый метод из программы 8.1, однако пока мы не будем делать этого, а сначала внесем в данный подход одно усовершенствование. И хотя выделения дополнительной *памяти* для вспомогательного массива, по-видимому, на практике не избежать, в разделе 8.4 мы рассмотрим дальнейшие улучшения, которые позволят избежать дополнительных затрат *времени* на копирование массива.

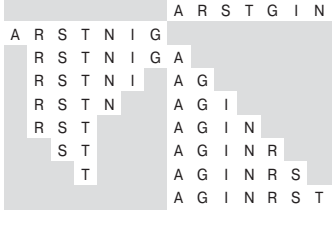


Рис. 8.1. Слияние без сигнальных ключей

Для слияния двух упорядоченных по возрастанию файлов они копируются в другой файл, причем второй файл копируется сразу за первым в обратном порядке. Тогда можно следовать следующему простому правилу: в выходной файл выбирается левый или правый элемент — тот, у которого ключ меньше. Максимальный ключ выполняет роль сигнального для обоих файлов, где бы он ни находился. На данном рисунке показано слияние файлов `A R S T` *и* `G I N`.

Вторая заслуживающая внимания основная характеристика базового слияния заключается в том, что внутренний цикл содержит две проверки для определения, исчерпаны ли входные массивы. Понятно, что обычно эти проверки дают отрицательный результат, и так и напрашивается использование сигнальных ключей, позволяющих отказаться от них. То есть если в конец массива `a` и массива `aux` добавить элементы с ключами, бóльшими значений всех других ключей, эти проверки можно удалить: если массив `a` (`b`) будет исчерпан, сигнальный ключ заставляет выбирать следующие элементы и помещать их в массив `c` только из массива `b` (`a`), вплоть до окончания слияния.

Однако, как было показано в главах 6 и 7, сигнальными ключами не всегда удобно пользоваться: либо потому, что не всегда легко определить наибольшее значение ключа, либо потому, что сигнальный ключ трудно вставить в массив. В случае слияния существует достаточно простое средство, которое показано на рис. 8.1.

В основу этого метода положена следующая идея: если при реализации обменной абстракции все-таки не отказываться от копирования массивов, то при копировании второго файла можно просто изменить порядок его элементов (без дополнительных затрат) — чтобы связанный с ним индекс перемещался справа налево. При таком упорядочении наибольший элемент, в каком бы он файле не находился, служит сигнальным ключом для другого массива. Программа 8.2 является эффективной реализацией абстрактного обменного слияния, основанной на этой идее, и служит отправной точкой для разработки алгоритмов сортировки, которые рассматриваются далее в этой главе. В ней все же используется вспомогательный массив с размером, пропорциональным выходному файлу, но она более эффективна, чем примитивная реализация, поскольку в ней не нужны проверки на окончание сливаемых массивов.

Последовательность ключей, которая сначала увеличивается, а затем уменьшается (или сначала уменьшается, а затем увеличивается), называется *битонической* (bitonic) последовательностью. Сортировка битонической последовательности эквивалентна слиянию, но иногда удобно представить задачу слияния в виде задачи битонической сортировки; рассмотренный метод, позволяющий избежать вставки сигнальных ключей, как раз и служит простым примером этого.

Одно из важных свойств программы 8.1 заключается в том, что реализуемое ею слияние устойчиво: оно сохраняет относительный порядок элементов с одинаковыми ключами. Эту характеристику нетрудно проверить, и при реализации абстрактного обменного слияния часто имеет смысл убедиться в сохранении устойчивости, т.к., как будет показано в разделе 8.3, устойчивое слияние приводит к устойчивым методам *сортировки*. Свойство устойчивости не всегда просто сохранить: например, программа 8.2 не обеспечивает устойчивости (см. упражнение 8.6). Это обстоятельство еще больше усложняет проблему разработки алгоритма по-настоящему обменного слияния.

Программа 8.2. Абстрактное обменное слияние

Данная программа выполняет слияние двух файлов без использования сигнальных ключей, для чего второй массив копируется во вспомогательный массив `aux` в обратном порядке, сразу за концом первого массива (т.е. устанавливая в `aux` *битонический* порядок). Первый цикл `for` пересылает первый массив и оставляет `i` равным `l`, т.е. готовым для начала слияния. Второй цикл `for` пересылает второй массив, после чего `j` равно `r`. Затем в процессе слияния (третий цикл `for`) наибольший элемент служит сигнальным ключом независимо от того, в каком массиве он находится. Внутренний цикл этой программы достаточно короткий (пересылка в `aux`, сравнение, пересылка обратно в `a`, увеличение значения `i` или `j` на единицу, увеличение и проверка значения `k`).

```
template <class Item>
void merge(Item a[], int l, int m, int r)
  { int i, j;
    static Item aux[maxN];
    for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
    for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
    for (int k = l; k <= r; k++)
      if (aux[j] < aux[i])
        a[k] = aux[j--];
      else
        a[k] = aux[i++];
  }
```

Упражнения

▷ **8.5.** Покажите в стиле диаграммы 8.1, как программа 8.1 выполняет слияние ключей A E Q S U Y E I N O S T.

○ **8.6.** Объясните, почему программа 8.2 не является устойчивой, и разработайте устойчивую версию этой программы.

8.7. Что получится, если программу 8.2 применить к ключам E A S Y Q U E S T I O N?

○ **8.8.** Верно ли, что программа 8.2 правильно сливает входные массивы тогда и только тогда, когда они отсортированы? Обоснуйте ваш ответ или приведите контрпример.

8.3. Нисходящая сортировка слиянием

Имея в своем распоряжении процедуру слияния, нетрудно положить ее в основу рекурсивной процедуры сортировки. Чтобы отсортировать заданный файл, нужно разделить его на две части, рекурсивно отсортировать обе половины и затем слить их. Реализация этого алгоритма представлена в программе 8.3; а пример показан на рис. 8.2. Как было сказано в главе 5, этот алгоритм является одним из самых известных примеров использования принципа “разделяй и властвуй” для разработки эффективных алгоритмов.

Нисходящая сортировка слиянием аналогична принципу управления сверху вниз, при котором руководитель разбивает большую задачу на подзадачи, которые должны независимо решать его подчиненные. Если каждый руководитель будет просто разбивать свою задачу на две равные части, а потом объединять решения, полученные его подчиненными, и передавать результат своему начальству, то получится процесс, аналогичный сортировке слиянием. Работа практически не продвигается, пока не получит свою задачу кто-то, не имеющий подчиненных (в рассматриваемом случае это слияние двух файлов размером 1); но потом руководство выполняет значительную работу, объединяя результаты работы подчиненных.

Сортировка слиянием играет важную роль благодаря простоте и оптимальности заложенного в нее метода (время ее выполнения пропорционально $N \log N$), который

Программа 8.3. Нисходящая сортировка слиянием

Эта базовая реализация сортировки слиянием является примером рекурсивной программы, основанной на принципе “разделяй и властвуй”. Для упорядочения массива `a[1]`, ..., `a[r]` он разбивается на две части `a[1]`, ..., `a[m]` и `a[m+1]`, ..., `a[r]`, которые сортируются независимо друг от друга (через рекурсивные вызовы) и вновь сливаются для получения отсортированного исходного файла. Функции `merge` может потребоваться вспомогательный файл, достаточно большой для помещения копии входного файла, однако эту абстрактную операцию удобно рассматривать как обменное слияние (см. текст).

```
template <class Item>
void mergesort(Item a[], int l, int r)
  { if (r <= l) return;
    int m = (r+l)/2;
    mergesort(a, l, m);
    mergesort(a, m+1, r);
    merge(a, l, m, r);
  }
```

допускает возможность устойчивой реализации. Эти утверждения сравнительно нетрудно доказать.

Как было показано в главе 5 (и для быстрой сортировки в главе 7), для визуализации структуры рекурсивных вызовов рекурсивного алгоритма можно воспользоваться древовидными структурами, которые позволяют лучше понять все варианты рассматриваемого алгоритма и провести его анализ. Для сортировки слиянием структура рекурсивных вызовов зависит только от размера входного массива. Для любого заданного N определяется дерево, получившее название *дерева "разделяй и властвуй"*, которое описывает размер подфайлов, обрабатываемых во время выполнения программы 8.3 (см. упражнение 5.73): если N равно 1, то это дерево состоит из одного узла с меткой 1; иначе дерево состоит из корневого узла, содержащего файл размером N, поддерева, представляющего левый подфайл размером $\lfloor N / 2 \rfloor$ и поддерева, представляющего правый подфайл размером $\lceil N / 2 \rceil$. Таким образом, каждый узел этого дерева соответствует вызову метода `mergesort`, а его метка показывает размер задачи, соответствующей этому рекурсивному вызову. Если N равно степени 2, это построение дает полностью сбалансированное дерево со степенями 2 во всех узлах и единицами во всех внешних узлах. Если N не является степенью 2, вид дерева усложняется. На рис. 8.3 представлены примеры обоих случаев. Ранее мы уже сталкивались с такими деревьями — при изучении в разделе 5.2 алгоритма с такой же структурой рекурсивных вызовов, как и у сортировки слиянием.

Рис. 8.2. Пример нисходящей сортировки слиянием

В каждой строке показан результат вызова функции `merge` *при выполнении нисходящей сортировки слиянием. Вначале сливаются* `A` *и* `S`*, и получается* `A S`*; потом сливаются* `O` *и* `R`*, и получается* `O R`*. Затем сливаются* `O R` *и* `A S`*, и получается* `A O R S`*. После этого сливаются* `I T` *и* `G N`*, и получается* `G I N T`*, потом этот результат сливается с* `A O R S`*, и получается* `A G I N O R S T`*, и т.д. Метод рекурсивно объединяет меньшие упорядоченные файлы в бóльшие.*

Структурные свойства деревьев "разделяй и властвуй" имеют непосредственное отношение к анализу сортировки слиянием. Например, общее количество сравнений, выполняемых алгоритмом, в точности равно сумме всех меток узлов.

Лемма 8.1. *Для сортировки любого файла из N элементов сортировка слиянием выполняет порядка $N\lg N$ сравнений.*

В реализациях, описанных в разделах 8.1 и 8.2, для каждого слияния двух подмассивов размером $N/2$ нужно N сравнений (это значение может отличаться на 1 или 2, в зависимости от способа использования сигнальных ключей). Следовательно, общее количество сравнений для всей сортировки может быть описано стандартным рекуррентным соотношением вида "разделяй и властвуй": $M_N = M_{\lfloor N/2 \rfloor} + M_{\lceil N/2 \rceil} + N$, при $M_1 = 0$. Это рекуррентное соотношение описывает также сумму меток узлов и длину внешнего пути дерева "разделяй и властвуй" с N узлами (см. упражнение 5.73). Данное утверждение нетрудно проверить, когда N является степенью числа 2 (см. формулу 2.4), и доказать методом индукции для произвольного N. Непосредственное доказательство содержится в упражнениях 8.12–8.14. ■

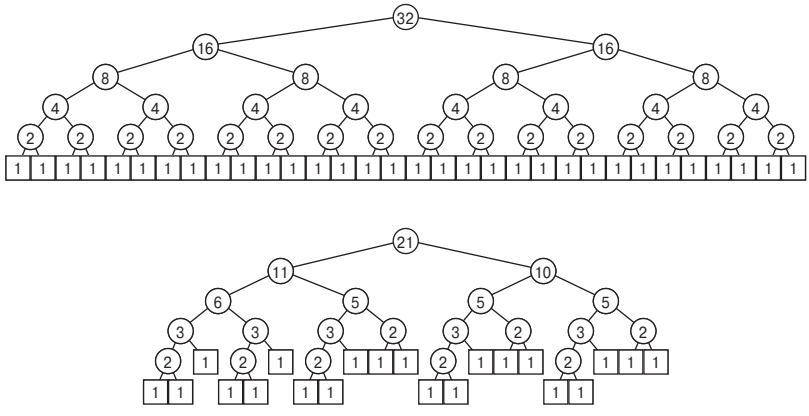


Рис. 8.3. Деревья "разделяй и властвуй"

На этих диаграммах показаны размеры подзадач, создаваемых нисходящей сортировкой слиянием. В отличие от, скажем, деревьев, соответствующих быстрой сортировке, эти структуры зависят только от размера первоначального файла и не зависят от значений ключей. На верхней диаграмме показана сортировка файла из 32 элементов. Сначала выполняется (рекурсивное) упорядочение двух файлов из 16 элементов, а затем их слияние. Файлы из 16 элементов сортируются (рекурсивно) с помощью (рекурсивной) сортировки файлов из 8 элементов и т.д. Для файлов, размер которых не равен степени двух, получается более сложная структура, пример которой приведен на нижней диаграмме.

Лемма 8.2. *Сортировке слиянием нужен объем дополнительной памяти, пропорциональный N.*

Это факт очевиден из обсуждения в разделе 8.2. Можно кое-что сделать для уменьшения размера дополнительной памяти — за счет существенного усложнения алгоритма (см., например, упражнение 8.21). Как будет показано в разделе 8.7, сортировка слиянием эффективна и в том случае, если сортируемый файл организован в виде связного списка. В этом случае данное свойство выполняется, но нужна дополнительная память для ссылок. В случае массивов, как было сказано в разделе 8.2 и будет сказано в разделе 8.4, можно выполнять слияние на месте (обсуждение этой темы будет продолжено в разделе 8.4), однако эта стратегия вряд ли применима на практике. ■

Лемма 8.3. *Сортировка слиянием устойчива, если устойчив используемый при этом метод слияния.*

Это утверждение легко проверить методом индукции. Для реализации метода слияния, наподобие предложенного в программе 8.1, легко показать, что относительное расположение повторяющихся ключей не нарушается. Однако, чем сложнее алгоритм, тем выше вероятность того, что эта устойчивость будет нарушена (см. упражнение 8.6). ■

Лемма 8.4. *Требования к ресурсам сортировки слиянием не зависят от исходной упорядоченности входных данных.*

В наших реализациях от входных данных зависит только порядок, в котором элементы обрабатываются во время слияний. Каждый проход требует памяти и числа шагов, пропорциональных размеру подфайла, из-за пересылки данных во

вспомогательный массив. Две ветви оператора `if` из-за особенностей компиляции могут выполняться за слегка различное время, что может привести к некоторой зависимости времени выполнения от характера входных данных, однако число сравнений и других операций над входными данными не зависит от того, как упорядочен входной файл. Обратите внимание на то, что это отнюдь *не эквивалентно* утверждению, что алгоритм не адаптивный (см. раздел 6.1) — ведь последовательность сравнений зависит от упорядоченности входных данных. ■

Упражнения

▷ **8.9.** Приведите последовательность слияний, выполняемых программой 8.3 при сортировке ключей `E A S Y Q U E S T I O N`.

8.10. Начертите деревья "разделяй и властвуй" для $N = 16$, 24, 31, 32, 33 и 39.

● **8.11.** Реализуйте рекурсивную сортировку слиянием для массивов, используя идею *трехпутевого*, а не двухпутевого слияния.

○ **8.12.** Докажите, что все узлы с метками 1 в деревьях "разделяй и властвуй" расположены на двух нижних уровнях.

○ **8.13.** Докажите, что метки в узлах на каждом уровне сбалансированного дерева размером N в сумме дают N, за исключением, возможно, нижнего уровня.

○ **8.14**. Используя упражнения 8.12 и 8.13, докажите, что количество сравнений, необходимых для выполнения сортировки слиянием, находится в пределах между $N\lg N$ и $N\lg N + N$.

● **8.15.** Найдите и докажите зависимость между количеством сравнений, используемых сортировкой слиянием, и количеством битов в $\lceil \lg N \rceil$-разрядных положительных числах, меньших N.

8.4. Усовершенствования базового алгоритма

Как мы уже видели на примере быстрой сортировки, большую часть рекурсивных алгоритмов можно усовершенствовать, обрабатывая файлы небольших размеров специальным образом. В силу рекурсивного характера функции часто вызываются именно для небольших файлов, поэтому улучшение их обработки приводит к улучшению всего алгоритма. Следовательно, как и для быстрой сортировки, переключение на сортировку вставками подфайлов небольших размеров даст улучшение времени выполнения типичной реализации сортировки слиянием на 10–15%.

Следующее полезное усовершенствование — это устранение времени копирования данных во вспомогательный массив, используемый слиянием. Для этого следует так организовать рекурсивные вызовы, что на каждом уровне процесс вычисления меняет ролями входной и вспомогательный массивы. Один из способов реализации такого подхода заключается в создании двух вариантов программ: одного для входных данных в массиве `aux` и выходных данных в массиве `a`, а другого — для входных данных в массиве `a` и выходных данных в массиве `aux`, обе эти версии поочередно вызывают одна другую. Другой подход продемонстрирован в программе 8.4, которая вначале создает копию входного массива, а затем использует программу 8.1 и переключает аргументы в рекурсивных вызовах, устраняя таким образом операцию явного копирования массива. Вместо нее программа поочередно переключается между выводом результата слияния то во вспомогательный, то во входной файл. (Это достаточно хитроумная программа.)

Программа 8.4. Сортировка слиянием без копирования

Данная рекурсивная программа сортирует массив b, помещая результат сортировки в массив a. Поэтому рекурсивные вызовы написаны так, что их результаты остаются в массиве b, а для их слияния в массив a используется программа 8.1. Таким образом, все пересылки данных выполняются во время слияний.

```
template <class Item>
void mergesortABr(Item a[], Item b[], int l, int r)
  { if (r-l <= 10) { insertion(a, l, r); return; }
    int m = (l+r)/2;
    mergesortABr(b, a, l, m);
    mergesortABr(b, a, m+1, r);
    mergeAB(a+l, b+l, m-l+1, b+m+1, r-m);
  }
template <class Item>
void mergesortAB(Item a[], int l, int r)
  { static Item aux[maxN];
    for (int i = l; i <= r; i++) aux[i] = a[i];
    mergesortABr(a, aux, l, r);
  }
```

Данный метод позволяет избежать копирования массива ценой возвращения во внутренний цикл проверок исчерпания входных файлов. (Вспомните, что устранение этих проверок в программе 8.2 преобразовало этот файл во время копирования в битонический.) Положение можно восстановить с помощью рекурсивной реализации той же идеи: нужно реализовать две программы как слияния, так и сортировки слиянием: одну для вывода массива по возрастанию, а другую — для вывода массива по убыванию. Это позволяет снова использовать битоническую стратегию и устранить необходимость в сигнальных ключах во внутреннем цикле.

Поскольку при этом используются четыре копии базовых программ и закрученные рекурсивные переключения аргументов, такая супероптимизация может быть рекомендована только экспертам (ну или студентам), но все-таки она существенно ускоряет сортировку слиянием. Экспериментальные результаты, которые будут рассмотрены в разделе 8.6, показывают, что сочетание всех предложенных выше усовершенствований ускоряет сортировку слиянием процентов на 40, однако все же она процентов на 25 медленнее быстрой сортировки. Эти показатели зависят от реализации и от машины, но в разных ситуациях результаты похожи.

Другие реализации слияния, использующие явные проверки исчерпания первого файла, могут привести к более (но не очень) заметным колебаниям времени выполнения в зависимости от характера входных данных. Для случайно упорядоченных файлов после исчерпания подфайла размер другого подфайла будет небольшим, а затраты на пересылку во вспомогательный файл все так же пропорциональны размеру этого подфайла. Можно еще попытаться улучшить производительность сортировки слиянием в тех случаях, когда файл в значительной степени упорядочен, и пропускать вызов merge при полной упорядоченности файла, однако для многих типов файлов данная стратегия неэффективна.

Упражнения

8.16. Реализуйте абстрактное обменное слияние, использующее дополнительный объем памяти, пропорциональный размеру меньшего из сливаемых файлов. (Этот метод должен сократить наполовину потребность сортировки в памяти.)

8.17. Выполните сортировку слиянием больших случайно упорядоченных файлов и экспериментально определите среднюю длину другого подфайла на момент исчерпания первого подфайла как функцию от N (сумма длин двух сливаемых подфайлов).

8.18. Предположим, программа 8.3 модифицирована так, что не вызывает метод `merge` при `a[m]<a[m+1]`. Сколько сравнений экономится в этом случае, если сортируемый файл уже упорядочен?

8.19. Выполните модифицированный алгоритм, предложенный в упражнении 8.18, для больших случайно упорядоченных файлов. Экспериментально определите среднее количество пропусков вызова `merge` в зависимости от N (размер исходного сортируемого файла).

8.20. Допустим, что сортировка слиянием должна быть выполнена на h-сортированном файле для небольшого значения h. Какие изменения нужно внести в подпрограмму `merge`, чтобы воспользоваться этим свойством входных данных? Поэкспериментируйте с гибридами сортировки методом Шелла и сортировки слиянием, основанными на этой подпрограмме.

8.21. Разработайте реализацию слияния, уменьшающую требование дополнительной памяти до $\max(M, N/M)$ за счет следующей идеи. Разбейте массив на N/M блоков размером M (для простоты предположим, что N кратно M). Затем, (1) рассматривая эти блоки как записи, первые ключи которых являются ключами сортировки, отсортируйте их с помощью сортировки выбором, и (2) выполните проход по массиву, сливая первый блок со вторым, затем второй блок с третьим и так далее.

8.22. Докажите, что метод, описанный в упражнении 8.21, выполняется за линейное время.

8.23. Реализуйте битоническую сортировку слиянием без копирования.

8.5. Восходящая сортировка слиянием

Как было сказано в главе 5, у каждой рекурсивной программы имеется нерекурсивный аналог, который хотя и выполняет эквивалентные действия, но может делать это в другом порядке. Нерекурсивные реализации сортировки слиянием заслуживают детального изучения в качестве образцов философии алгоритмов “разделяй и властвуй”.

Рассмотрим последовательность слияний, выполняемую рекурсивным алгоритмом. Из примера, приведенного на рис. 8.2, видно, что файл размером 15 сортируется следующей последовательностью слияний:

```
1-и-1  1-и-1  2-и-2  1-и-1  1-и-1  2-и-2  4-и-4
1-и-1  1-и-1  2-и-2  1-и-1  2-и-1  4-и-3  8-и-7.
```

Порядок выполнения слияний определяется рекурсивной структурой алгоритма. Но подфайлы обрабатываются независимо, поэтому слияния могут выполняться в различном порядке. На рис. 8.4 показана восходящая стратегия, при которой последовательность слияний такова:

```
1-и-1  1-и-1  1-и-1  1-и-1  1-и-1  1-и-1  1-и-1
2-и-2  2-и-2  2-и-2  2-и-1  4-и-4  4-и-3  8-и-7.
```

Рис. 8.4. Пример восходящей сортировки слиянием
В каждой строке показан результат вызова метода `merge` *при выполнении восходящей сортировки слиянием. Вначале выполняются слияния 1-и-1: при слиянии* `A` *и* `S` *получается* `A S`*; при слиянии* `O` *и* `R` *получается* `O R` *и т.д. Из-за нечетности размера файла последнее* `E` *не принимает участие в слиянии. На втором проходе выполняются слияния 2-и-2:* `A S` *сливается с* `O R`*, и получается* `A O R S` *и т.д., до последнего слияния 2-и-1. После этого выполняются слияния 4-и-4, 4-и-3 и завершающее 8-и-7.*

В обоих случаях выполняются семь слияний 1-и-1, три слияния 2-и-2 и по одному слиянию 2-и-1, 4-и-4, 4-и-3 и 8-и-7, но они выполняются в различном порядке. Восходящая стратегия предлагает сливать наименьшие из оставшихся файлов, проходя по массиву слева направо.

Последовательность слияний, выполняемая рекурсивным алгоритмом, определяется деревом "разделяй и властвуй", показанным на рис. 8.3: мы просто выполняем обратный проход по этому дереву. Как было показано в главе 3, можно разработать нерекурсивный алгоритм, использующий явный стек, который даст ту же последовательность слияний. Однако совсем не обязательно ограничиваться только обратным порядком: *любой* проход по дереву, при котором обход поддеревьев узла завершается перед посещением самого узла, дает правильный алгоритм. Единственное ограничение заключается в том, что сливаемые файлы должны быть предварительно отсортированы. В случае сортировки слиянием удобно сначала выполнять все слияния 1-и-1, затем все слияния 2-и-2, затем все 4-и-4, и так далее. Такая последовательность соответствует обходу дерева по уровням, который поднимается по дереву снизу вверх.

В главе 5 мы уже видели на нескольких примерах, что при рассуждении в стиле снизу-вверх имеет смысл переориентировать мышление в сторону стратегии "объединяй и властвуй", когда сначала решаются небольшие подзадачи, а затем они объединяются для получения решения большей задачи. В частности, нерекурсивный вариант вида "объединяй и властвуй" сортировки слиянием в программе 8.5 получается следующим образом: вначале все элементы файла рассматриваются как упорядоченные подсписки длиной 1. Потом для них выполняются слияния 1-и-1, и получаются упорядоченные подсписки размером 2, затем выполняется серия слияний 2-и-2, что дает упорядоченные подсписки размером 4, и так далее до упорядочения всего списка. Если размер файла не является степенью 2, то последний подсписок не всегда имеет тот же размер, что и все другие, но его все равно можно слить.

Если размер файла является степенью 2, то *множество* слияний, выполняемых восходящей сортировкой слиянием, в точности совпадает с *множеством* слияний, выполняемым рекурсивной сортировкой слиянием, однако *последовательность* слияний будет другой. Восходящая сортировка слиянием соответствует обходу дерева "разделяй и властвуй" по уровням, снизу вверх. В противоположность этому, рекурсивный алгоритм называется *нисходящей сортировкой слиянием* — в силу обратного обхода дерева сверху вниз.

Если размер файла не равен степени 2, восходящий алгоритм дает другое множество слияний, как показано на рис. 8.5. Восходящий алгоритм соответствует дереву "объединяй и властвуй" (см. упражнение 5.75), отличному от дерева "разделяй и властвуй", которое соответствует нисходящему алгоритму. Можно сделать так, чтобы последова-

тельность слияний, выполняемых рекурсивным методом, была такой же, как и для нерекурсивного метода, однако для этого нет особых причин, поскольку разница в производительности невелика по отношению к общим затратам.

Леммы 8.1–8.4 справедливы и для восходящей сортировки слиянием, при этом имеют место следующие дополнительные леммы:

Лемма 8.5. *Все слияния на каждом проходе восходящей сортировки слиянием манипулируют файлами, размер которых равен степени 2, за исключением, возможно, размера последнего файла.*

Это факт легко доказать методом индукции. ■

Лемма 8.6. *Количество проходов при восходящей сортировке слиянием по файлу из N элементов в точности равно числу битов в двоичном представлении N (без ведущих нулей).*

Размер подсписков после k проходов равен 2^k, т.к. на каждом проходе восходящей сортировки слиянием размер упорядоченных подфайлов удваивается. Значит, количество проходов, необходимое для сортировки файла из N элементов, есть наименьшее k такое, что $2k \geq N$, что в точности равно $\lceil \lg N \rceil$, т.е. количеству битов в двоичном представлении N. Этот результат можно доказать и методом индукции или с помощью анализа структурных свойств деревьев "объединяй и властвуй". ■

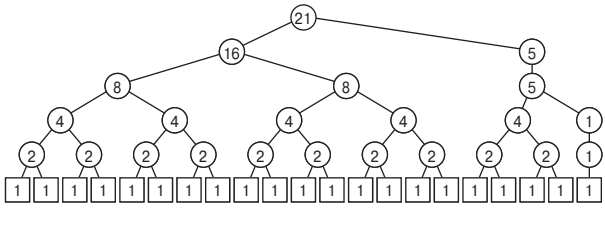
Программа 8.5. Восходящая сортировка слиянием

Восходящая сортировка слиянием состоит из последовательности проходов по всему файлу с выполнением слияний вида m-и-m и с удвоением m на каждом проходе. Последний подфайл имеет размер m лишь тогда, когда размер файла является четным кратным m, так что последнее слияние имеет тип m-и-*x*, для некоторого *x*, меньшего или равного m.

```
inline int min(int A, int B)
  { return (A < B) ? A : B; }
template <class Item>
void mergesortBU(Item a[], int l, int r)
  { for (int m = 1; m <= r-l; m = m+m)
      for (int i = l; i <= r-m; i += m+m)
        merge(a, i, i+m-1, min(i+m+m-1, r));
  }
```

Рис. 8.5. Размеры файлов при восходящей сортировке слиянием

Если размер файла не равен степени 2, то структуры слияний для восходящей сортировки слиянием совершенно не похожи на структуры слияний для нисходящей сортировки (рис. 8.3). При восходящей сортировке размеры всех файлов (возможно, за исключением последнего) равны степени 2. Эти различия помогают понять базовую структуру алгоритмов, но почти не влияют на производительность.



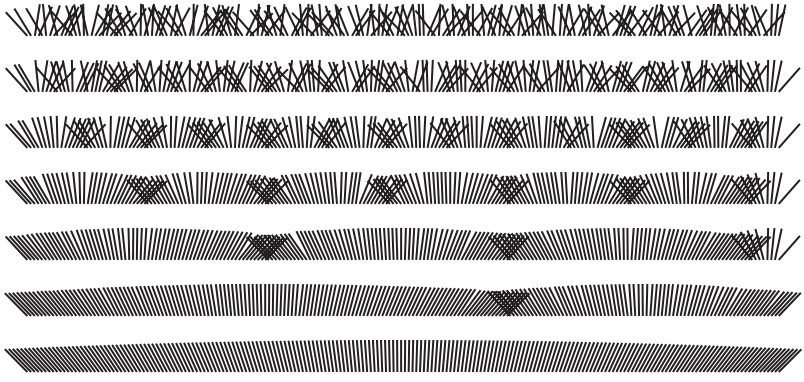


Рис. 8.6. Восходящая сортировка слиянием

Для упорядочения файла из 200 элементов восходящая сортировка слиянием выполняет лишь семь проходов по файлу. Каждый проход вдвое уменьшает количество упорядоченных подфайлов и удваивает их длину (кроме, возможно, последнего).

Процесс выполнения восходящей сортировки слиянием большого файла показан на рис. 8.6. Сортировка 1 миллиона элементов выполняется за 20 проходов по данным, 1 миллиарда — за 30 проходов и т.д.

Подводя итоги, отметим, что нисходящая и восходящая сортировки — это два простых алгоритма сортировки, основанных на операции слияния двух упорядоченных подфайлов в результирующий объединенный упорядоченный файл. Оба алгоритма тесно связаны между собой и даже выполняют одно и то же множество слияний, если размер исходного файла является степенью 2, но они отнюдь не идентичны. На рис. 8.7 демонстрируются различия динамических характеристик алгоритмов на примере большого файла. Каждый алгоритм может использоваться на практике, если экономия памяти не важна, и желательно гарантированное время выполнения в худшем случае. Оба алгоритма представляют интерес как прототипы общих принципов построения алгоритмов: “разделяй и властвуй” и “объединяй и властвуй”.

Упражнения

8.24. Покажите, какие слияния выполняет восходящая сортировка слиянием (программа 8.5) для ключей `E A S Y Q U E S T I O N`.

8.25. Реализуйте восходящую сортировку слиянием, которая начинает с сортировки вставками блоков по M элементов. Определите эмпирическим путем значение M, для которого разработанная программа быстрее всего сортирует произвольно упорядоченные файлы из N элементов, при $N = 10^3$, 10^4, 10^5 и 10^6.

8.26. Нарисуйте деревья, которые отображают слияния, выполняемые программой 8.5 для $N = 16$, 24, 31, 32, 33 и 39.

8.27. Напишите программу рекурсивной сортировки слиянием, выполняющую те же слияния, что и восходящая сортировка слиянием.

8.28. Напишите программу восходящей сортировки слиянием, выполняющую те же слияния, что и нисходящая сортировка слиянием. (Это упражнение намного труднее, чем упражнение 8.27).

8.29. Предположим, что размер файла является степенью 2. Удалите рекурсию из нисходящей сортировки слиянием так, чтобы получить нерекурсивную сортировку слиянием, выполняющую ту же *последовательность* слияний.

8.30. Докажите, что количество проходов, выполняемых нисходящей сортировкой слиянием, *также* равно количеству битов в двоичном представлении числа N (см. лемму 8.6).

8.6. Производительность сортировки слиянием

В таблице 8.1 показана относительная эффективность различных рассмотренных нами усовершенствований. Как это часто бывает, оказывается, что если направить усилия на оптимизацию внутреннего цикла алгоритма сортировки, то время выполнения сортировки можно сократить наполовину и даже больше.

Помимо усовершенствований, рассмотренных в разделе 8.2, можно добиться дальнейшего повышения производительности, поместив наименьшие элементы обоих массивов в простые переменные или машинные регистры процессора и избежав таким образом лишних обращений к массивам. Тогда внутренний цикл сортировки слиянием можно свести к сравнению (с условным переходом), увеличению на единицу значений двух счетчиков (`k` и либо `i`, либо `j`) и проверке условия завершения цикла с условным переходом. Общее количество команд в таком внутреннем цикле несколько больше, чем для быстрой сортировки, но эти команды выполняются всего лишь $N \lg N$ раз, в то время как команды внутреннего цикла быстрой сортировки выполняются на 39% чаще (или на 29% для варианта с вычислением медианы из трех). Для более точного сравнения этих двух алгоритмов в различных средах нужна их тщательная реализация и подробный анализ. Однако точно известно, что внутренний цикл сортировки слиянием несколько длиннее внутреннего цикла быстрой сортировки.

Как обычно, мы считаем своим долгом предостеречь, что погоня за подобными усовершенствованиями, столь заманчивая для многих программистов, часто приносит лишь незначительные выгоды, и в нее стоит пускаться только после разрешения более важных вопросов. В данном случае сортировка слиянием обладает несомненным преимуществом перед быстрой сортировкой в том, что она устойчива и гарантирует высокую скорость (вне зависимости от характера входных данных), но проигрывает в том, что использует дополнительный объем памяти, пропорциональный размеру массива.

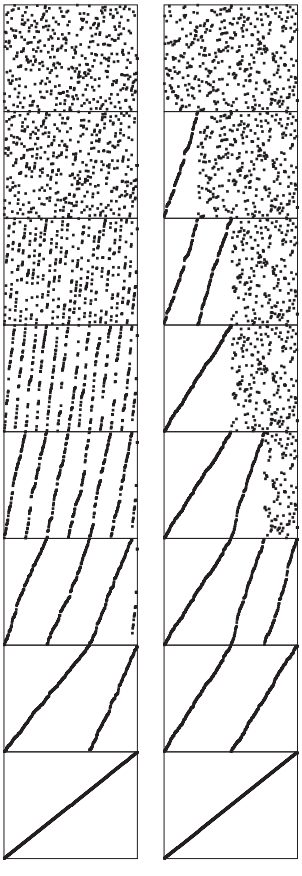


Рис. 8.7. Сравнение восходящей и нисходящей сортировки слиянием

Восходящая сортировка слиянием (слева) выполняет серию проходов по файлу, которые сливают упорядоченные подфайлы, пока не останется только один. Каждый элемент файла, за исключением, возможно, последнего, участвует в каждом проходе. В отличие от этого, нисходящая сортировка слиянием (справа) вначале упорядочивает первую половину файла, а затем берется за вторую половину (рекурсивно), поэтому диаграмма ее работы существенно отличается.

Таблица 8.1. Эмпирическое сравнение алгоритмов сортировки слиянием

Представленные здесь относительные временные показатели различных видов сортировки для случайно упорядоченных файлов чисел с плавающей точкой различного размера N показывают, что: стандартная быстрая сортировка примерно в два раза быстрее стандартной сортировки слиянием; добавление отсечения небольших файлов снижает время выполнения и нисходящей, и восходящей сортировок слиянием примерно на 15%; для указанных в таблице размеров файлов быстродействие нисходящей сортировки слиянием приблизительно на 10% выше, чем восходящей; даже если устранить затраты на копирование файла, то и в этом случае сортировка слиянием случайно упорядоченных файлов на 50–60% медленнее обычной быстрой сортировки (см. табл. 7.1).

		Нисходящая			*Восходящая*	
N	Q	T	T*	O	B	B*
12500	2	5	4	4	5	4
25000	5	12	8	8	11	9
50000	11	23	20	17	26	23
100000	24	53	43	37	59	53
200000	52	111	92	78	127	110
400000	109	237	198	168	267	232
800000	241	524	426	358	568	496

Обозначения:

Q	Стандартная быстрая сортировка (программа 7.1)
T	Стандартная нисходящая сортировка слиянием (программа 8.1)
T*	Нисходящая сортировка слиянием с отсечением небольших файлов
O	Нисходящая сортировка слиянием с отсечением и без копирования массива
B	Стандартная восходящая сортировка слиянием (программа 8.5)
B*	Восходящая сортировка слиянием с отсечением небольших файлов

Если эти факторы склоняются в пользу сортировки слиянием (и быстродействие имеет большое значение), то рассмотренные нами усовершенствования заслуживают внимательного рассмотрения — наряду с тщательным изучением компилированного кода, особенностей архитектуры компьютера и пр.

С другой стороны, уместно повторить и обычное предостережение о том, что программисты никогда не должны упускать из виду вопросы производительности, чтобы не допустить совершенно неоправданных издержек. Всем программистам (как и авторам!) совершенно не нравится, когда несущественные, но вовремя не замеченные свойства реализации перекрывают все другие хитроумные механизмы. Часто после такого тщательного анализа удается сократить время выполнения наполовину. Наиболее эффективной защитой против таких неприятных сюрпризов, возникающих в самый неподходящий момент, является многократное тестирование.

Эти моменты уже были рассмотрены в главе 5, однако привлекательность преждевременной оптимизации настолько сильна, что уместно напоминать об этом каждый раз при детальном изучении методов улучшения производительности. В случае сортировки слиянием можно чувствовать себя вполне спокойно, поскольку леммы 8.1–8.4

описывают наиболее важные характеристики производительности и верны для всех рассмотренных нами реализаций: время их выполнения пропорционально $N \log N$ при любой организации входных данных (см. рис. 8.8), они используют дополнительную память и могут быть реализованы с сохранением устойчивости. Сохранение этих свойств в процессе оптимизации обычно не является трудной задачей.

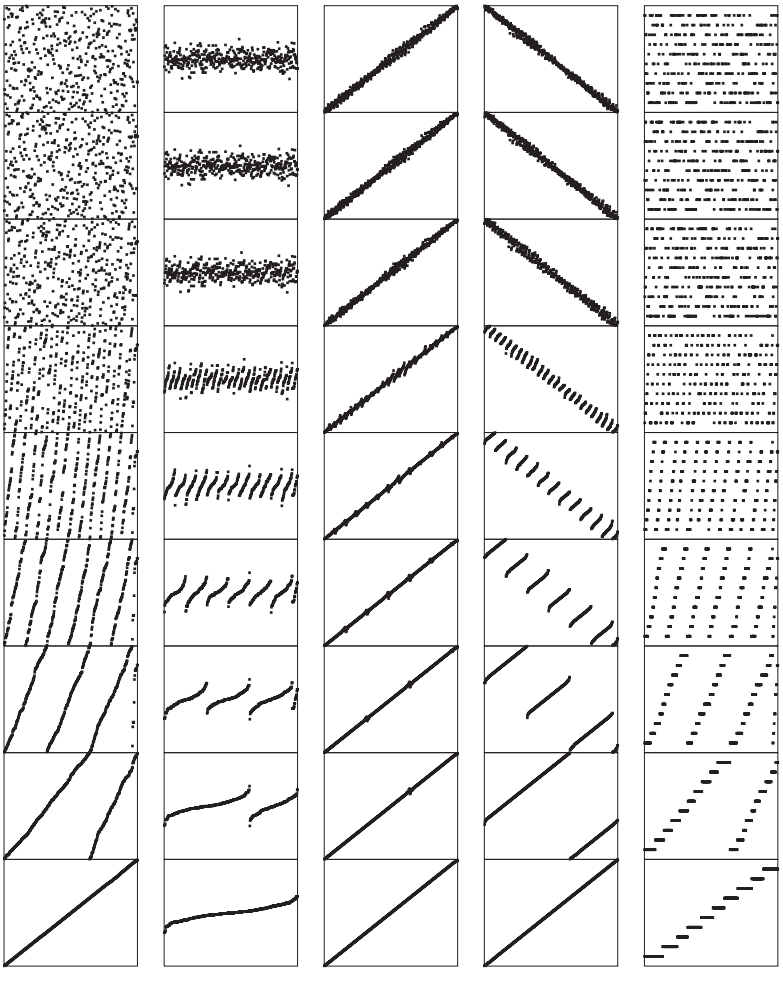


Рис. 8.8. Упорядочение различных видов файлов с помощью восходящей сортировки слиянием

Время выполнения сортировки слиянием не зависит от входных данных. На этих диаграммах показано количество проходов, выполняемых при восходящей сортировке файлов с равномерно распределенными случайными данными, нормально распределенными случайными данными, почти упорядоченных файлов, почти обратно упорядоченных данных и случайно упорядоченных с 10 различными значениями ключей (слева направо). Видно, что это количество зависит только от размера файла и никак не связано со входными значениями. Это поведение разительно отличается от поведения быстрой сортировки и многих других алгоритмов.

Упражнения

8.31. Реализуйте восходящую сортировку слиянием без копирования массивов.

8.32. Разработайте программу трехуровневой гибридной сортировки, использующую быструю сортировку, сортировку слиянием и сортировку вставками, чтобы получился метод, настолько же эффективный, как и самый быстрый вариант быстрой сортировки (даже для небольших файлов), но гарантирующий лучшее, чем квадратичное, время выполнения в худшем случае.

8.7. Реализации сортировки слиянием для связных списков

Раз уж для практической реализации сортировки слиянием все равно требуется дополнительная память, то можно рассмотреть и реализацию для связных списков. То есть вместо использования дополнительной памяти на вспомогательный массив можно применить ее для хранения ссылок. А кто-то может сразу столкнуться с проблемой сортировки связного списка (см. раздел 6.9). Оказывается, сортировка слиянием очень удобна для связных списков. Полная реализация метода сортировки слиянием связных списков представлена в программе 8.6. Обратите внимание, что код самого слияния почти так же прост, как и для слияния массивов (программа 8.2).

Имея в своем распоряжении эту функцию слияния, легко получить рекурсивную нисходящую сортировку слиянием списков. Программа 8.7 является прямой рекурсивной реализацией функции, которая принимает в качестве параметра указатель на неупорядоченный список и возвращает указатель на список, содержащий те же элементы в отсортированном порядке. Эта программа выполняет свою работу, переупорядочивая узлы списка и не требуя создания ни временных узлов, ни вспомогательных списков. Для определения середины списка программа 8.7 использует хитрый прием; другие реализации могут передавать длину списка в рекурсивную программу либо в виде параметра, либо в самом списке. В силу рекурсивной формулировки эта программа проста для понимания, несмотря на достаточно сложный алгоритм.

Программа 8.6. Слияние связных списков

Данная программа сливает список, на который указывает `a`, со списком, на который указывает `b`, с помощью вспомогательного указателя `c`. Операция сравнения ключей в функции `merge` включает и равенство, так что слияние будет устойчивым, если по условию список `b` следует за списком `a`. Для простоты здесь принято, что все списки завершаются пустой ссылкой, но пригодны и другие соглашения относительно окончания списков (см. табл. 3.1). Что еще важнее, в коде *не используются* ведущие узлы списков, которых иначе было бы очень много.

```
link merge(link a, link b)
  { node dummy(0); link head = &dummy, c = head;
    while ((a != 0) && (b != 0))
      if (a->item < b->item)
        { c->next = a; c = a; a = a->next; }
      else
        { c->next = b; c = b; b = b->next; }
    c->next = (a == 0) ? b : a;
    return head->next;
  }
```

Программа 8.7. Сортировка списков слиянием сверху вниз

Для выполнения сортировки эта программа разбивает список, на который указывает c, на две части, на которые указывают, соответственно, указатели a и b, рекурсивно сортирует эти части и получает окончательный результат с помощью функции merge (программа 8.6). Входной список должен заканчиваться пустой ссылкой (следовательно, так же должен заканчиваться и список b), а в конец списка a пустую ссылку заносит специальный оператор c->next = 0.

```
link mergesort(link c)
  { if (c == 0 || c->next == 0) return c;
    link a = c, b = c->next;
    while ((b != 0) && (b->next != 0))
      { c = c->next; b = b->next->next; }
    b = c->next; c->next = 0;
    return merge(mergesort(a), mergesort(b));
  }
```

К сортировке слиянием связных списков можно применить и восходящий подход "объединяй и властвуй", хотя необходимость отслеживать ссылки делает его более сложным, чем кажется на первый взгляд. Как упоминалось при изучении нисходящих методов сортировки массивов в разделе 8.3, при разработке алгоритма восходящей сортировки слиянием списков не существует веских причин придерживаться точно того же набора слияний, что и для рекурсивной версии или версии, использующей массивы.

Сам собой напрашивается интересный вариант восходящей сортировки слиянием связных списков: помещаем элементы списка в *циклический* список, после чего проходим по списку, сливая пары упорядоченных подфайлов до полного завершения. Этот метод концептуально прост, однако (как это часто бывает в случае низкоуровневых программ обработки связных списков) его реализация может оказаться весьма запутанной (см. упражнение 8.36). Другой вариант восходящей сортировки слиянием связных списков, основанный на той же идее, представлен в программе 8.8: в нем все сортируемые списки хранятся в АТД очереди.

Программа 8.8. Восходящая сортировка слиянием связных списков

Данная программа реализует восходящую сортировку слиянием, используя АТД очереди (программа 4.18). Элементы очереди представляют собой упорядоченные связные списки. После инициализации очереди списками единичной длины программа просто извлекает из очереди два списка, сливает их и возвращает полученный результат в эту же очередь — и так до тех пор, пока не останется только один список. Этот способ, как и в случае восходящей сортировки слиянием, соответствует последовательности проходов по всем элементам, с удвоением на каждом проходе длины упорядоченных списков.

```
link mergesort(link t)
  { QUEUE<link> Q(max);
    if (t == 0 || t->next == 0) return t;
    for (link u = 0; t != 0; t = u)
      { u = t->next; t->next = 0; Q.put(t); }
    t = Q.get();
    while (!Q.empty())
      { Q.put(t); t = merge(Q.get(), Q.get()); }
    return t;
  }
```

Этот метод также концептуально прост, однако (как и для многих других высокоуровневых программ, использующих АТД) его реализация *также* может потребовать хитроумного программирования.

Одно из важных свойств этого метода заключается в том, что он использует упорядоченность, которая может присутствовать в файле. В самом деле, количество проходов по списку равно не $\lceil \lg N \rceil$, а $\lceil \lg S \rceil$, где S — количество упорядоченных подфайлов в исходном файле. Такой способ иногда называется *естественной* (natural) сортировкой слиянием. Для случайно упорядоченных файлов естественная сортировка слиянием не дает существенного выигрыша, разве что один-два прохода (фактически этот метод, видимо, будет медленнее нисходящего метода — из-за затрат на проверку упорядоченности файла), однако на практике достаточно часто встречаются файлы, состоящие из блоков упорядоченных подфайлов; и в таких ситуациях данный метод будет достаточно эффективен.

Упражнения

- **8.33.** Разработайте реализацию нисходящей сортировки слиянием связных списков, которая передает рекурсивной процедуре длину списка в качестве параметра и использует ее для определения места разбиения списков.
- **8.34.** Разработайте реализацию нисходящей сортировки слиянием для связных списков, которые содержат собственные длины в ведущих узлах, и использующей эти длины для определения места разбиения списков.

8.35. Добавьте в программу 8.7 отсечение небольших подфайлов. Определите предельный размер отсекаемых файлов, который ускоряет выполнение программы.

- **8.36.** Реализуйте восходящую сортировку слиянием, использующую циклический связный список, как описано в тексте.

8.37. Добавьте в восходящую сортировку слиянием циклических списков из упражнения 8.36 отсечение небольших подфайлов. Определите предельный размер отсекаемых файлов, который ускоряет выполнение программы.

8.38. Добавьте в программу 8.8 отсечение небольших подфайлов. Определите предельный размер отсекаемых файлов, который ускоряет выполнение программы.

- **8.39.** Нарисуйте дерево “объединяй и властвуй”, которое отображает слияния, выполняемые программой 8.8, для $N = 16$, 24, 31, 32, 33 и 39.

8.40. Нарисуйте дерево “объединяй и властвуй”, которое отображает слияния, выполняемые сортировкой слиянием циклического списка (упражнение 8.38), для $N = 16$, 24, 31, 32, 33 и 39.

8.41. Проведите эмпирические исследования, и на их основе выдвиньте гипотезу о количестве упорядоченных подфайлов в файле из N случайных 32-разрядных двоичных целых чисел.

- **8.42.** Экспериментально определите количество проходов, необходимых для выполнения естественной сортировки слиянием случайных 64-разрядных двоичных ключей, при $N = 10^3$, 10^4, 10^5 и 10^6. *Подсказка*: для выполнения этого упражнения не обязательно реализовать сортировку (и даже генерировать полные 64-разрядные ключи).
- **8.43.** Преобразуйте программу 8.8 в программу естественной сортировки слиянием с помощью первоначального заполнения очереди упорядоченными подфайлами из входного файла.
- **8.44.** Реализуйте естественную сортировку слиянием для массивов.

8.8. Возврат к рекурсии

Программы, представленные в данной главе, и быстрая сортировка, рассмотренная в предыдущей главе — это типичные реализации алгоритмов "разделяй и властвуй". В последующих главах мы ознакомимся с еще несколькими алгоритмами подобной структуры, так что стоит более подробно рассмотреть основные характеристики этих реализаций.

Возможно, быструю сортировку было бы точнее назвать алгоритмом "*властвуй и разделяй*": в рекурсивных реализациях при каждом обращении бóльшая часть работы выполняется *перед* рекурсивными вызовами. А вот рекурсивная сортировка слиянием более соответствует принципу "разделяй и властвуй": вначале файл делится на две части, и затем каждая часть обрабатывается отдельно. Сортировка слиянием сначала выполняется для небольших задач, а в заключение обрабатывается самый большой подфайл. Быстрая сортировка начинается с обработки наибольшего подфайла и завершается обработкой подфайлов небольших размеров. Любопытно сравнение этих алгоритмов по аналогии с управлением коллективом сотрудников, упомянутой в начале данной главы: быстрая сортировка соответствует тому, что каждый руководитель затрачивает свои усилия на правильное разбиение задачи на подзадачи, так что после выполнения всех подзадач работа будет успешно выполнена; сортировка слиянием соответствует тому, что каждый руководитель быстро и произвольно делит задачу пополам, а затем, после решения подзадач, затрачивает свои усилия на объединение результатов.

Это различие ясно видно в нерекурсивных реализациях. Быстрой сортировке нужен стек — для хранения больших подзадач, разбиение на которые зависит от входных данных. Сортировка слиянием допускает простые нерекурсивные реализации, поскольку способ разбиения файла на части не зависит от данных, и становится возможным изменение очередности решения подзадач, что позволяет упростить программу.

Существует точка зрения, что быструю сортировку естественнее рассматривать как нисходящий алгоритм, поскольку он начинает работу на вершине дерева рекурсии, а затем для завершения сортировки опускается вниз. Можно обдумать и нерекурсивный вариант быстрой сортировки, которая обходит дерево рекурсии сверху вниз, но по уровням. Таким образом, сортировка многократно проходит по массиву, разбивая файлы на меньшие подфайлы. Для массивов этот метод не годится из-за большого объема затрат на хранение информации о подфайлах, а для связных списков он аналогичен восходящей сортировке слиянием.

Сортировка слиянием и быстрая сортировка отличаются по признаку устойчивости. Если подфайлы при сортировке слиянием упорядочены с соблюдением устойчивости, то вполне достаточно обеспечить устойчивость слияния, что сделать совсем нетрудно. Рекурсивная структура алгоритма автоматически приводит к индуктивному доказательству устойчивости. Но для реализации быстрой сортировки с использованием массивов нет очевидного простого способа разбиения файлов, обеспечивающего устойчивость, так что устойчивость исключается еще до вступления в дело рекурсии. Хотя примитивная реализация быстрой сортировки для связных списков является устойчивой (см. упражнение 7.4).

Как было показано в главе 5, алгоритмы с одним рекурсивным вызовом могут быть сведены к циклу, но алгоритмы с двумя рекурсивными циклами, наподобие сортировки слиянием или быстрой сортировки, открывают дверь в мир алгоритмов вида "разделяй и властвуй" и древовидных структур, к которому принадлежат и наши лучшие алгоритмы. Сортировка слиянием и быстрая сортировка заслуживают тщательного изучения не только из-за их важного практического значения, но и потому, что они позволяют лучше уяснить сущность рекурсии для разработки и понимания других рекурсивных алгоритмов.

Упражнения

- **8.45.** Допустим, сортировка слиянием реализована таким образом, что разбиение файла выполняется в *произвольном* месте, а не точно в середине файла. Сколько в среднем сравнений выполнит этот метод для упорядочения N элементов?
- **8.46.** Проведите анализ эффективности сортировки слиянием при упорядочении строк. Сколько в среднем сравнений символов выполняется при сортировке большого файла?
- **8.47.** Проведите эмпирические исследования по сравнению производительности быстрой сортировки связных списков (см. упражнение 7.4) и нисходящей сортировки слиянием связных списков (программа 8.7).