Содержание

2.2. Сортировка слиянием

Алгоритмы, которые мы рассмотрим в данном разделе, основаны на простой операции *слияния* — объединения двух упорядоченных массивов для получения одного большего упорядоченного массива. Эта операция непосредственно приводит к простому рекурсивному методу сортировки, который называется *сортировкой слиянием*: для сортировки массива нужно поделить его пополам, отсортировать (рекурсивно) эти половины и слить результаты (рис. 2.2.1). Как мы увидим, одним из наиболее привлекательных свойств сортировки слиянием является гарантированное время сортировки любого массива из N элементов за время, пропорциональное $N\log N$. Основной ее недостаток — требование дополнительной памяти с объемом, пропорциональным N.

Исходные данные	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Сортировка левой половины	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Сортировка правой половины	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Слияние результатов	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.1. *Принцип работы сортировки слиянием*

Абстрактное слияние на месте

Очевидный способ реализации слияния — метод, который сливает два отдельных упорядоченных массива объектов `Comparable` в третий массив. Эту стратегию несложно реализовать: создайте выходной массив нужного размера, а затем последовательно выбирайте из двух входных массивов наименьший оставшийся в них элемент и добавляйте его в выходной массив.

Но при сортировке большого массива придется выполнять большое количество слияний, поэтому стоимость создания нового массива для каждой порции сливаемых данных может оказаться слишком большой. Гораздо привлекательнее иметь метод сортировки на месте, чтобы отсортировать на месте первую половину массива, потом отсортировать на месте вторую половину массива, а затем слить эти половины, перемещая элементы внутри массива и не используя значительного объема дополнительной памяти. Сейчас имеет смысл остановиться на минутку и подумать, как это можно сделать. С виду эту задачу решить совсем нетрудно, но известные решения довольно сложны, особенно в сравнении с теми, которые требуют дополнительной памяти.

Но *абстракция* слияния на месте все-таки полезна. Поэтому мы будем использовать сигнатуру `merge(a, lo, mid, hi)` для обозначения метода, который помещает результат слияния подмассивов `a[lo..mid]` и `a[mid+1..hi]` в единый упорядоченный массив `a[lo..hi]`. Код, приведенный в листинге 2.2.1, реализует этот метод слияния всего лишь в нескольких строках: он копирует все данные во вспомогательный массив, а затем сливает их в исходный массив. Другой способ описан в упражнении 2.2.10.

Листинг 2.2.1. Абстрактное слияние на месте

```
public static void merge(Comparable[] a, int lo, int mid, int hi)
{  // Слияние a[lo..mid] с a[mid+1..hi].
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)  // Копирование a[lo..hi] в aux[lo..hi].
     aux[k] = a[k];
   for (int k = lo; k <= hi; k++)  // Слияние назад в a[lo..hi].
     if      (i > mid)                 a[k] = aux[j++];
     else if (j > hi )                 a[k] = aux[i++];
     else if (less(aux[j], aux[i]))    a[k] = aux[j++];
     else                              a[k] = aux[i++];
}
```

Для выполнения слияния данный метод сначала копирует данные во вспомогательный массив `aux[]`, а затем сливает их обратно в `a[]` (рис. 2.2.2). В процессе слияния (второй цикл `for`) возможны четыре варианта: левая половина закончилась (берем данные из правой), правая половина закончилась (берем данные из левой), текущий ключ из правой половины меньше текущего ключа из левой половины (берем справа) и текущий ключ из правой половины больше или равен текущему ключу из левой половины (берем слева).

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
Входные данные		E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
Копия		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	**A**										0	6	E	**E**	**G**	**M**	**R**	**A**	**C**	**E**	**R**	**T**
	1	**A**	**C**									0	7	E	**E**	**G**	**M**	**R**		**C**	**E**	**R**	**T**
	2	**A**	**C**	**E**								1	7	**E**	**E**	**G**	**M**	**R**			**E**	**R**	**T**
	3	**A**	**C**	**E**	**E**							2	7		**E**	**G**	**M**	**R**			**E**	**R**	**T**
	4	**A**	**C**	**E**	**E**	**E**						2	8			G	**M**	**R**			**E**	**R**	**T**
	5	**A**	**C**	**E**	**E**	**E**	**G**					3	8			**G**	**M**	**R**				**R**	**T**
	6	**A**	**C**	**E**	**E**	**E**	**G**	**M**				4	8				**M**	**R**				**R**	**T**
	7	**A**	**C**	**E**	**E**	**E**	**G**	**M**	**R**			5	8					**R**				**R**	**T**
	8	**A**	**C**	**E**	**E**	**E**	**G**	**M**	**R**	**R**		5	9									**R**	**T**
	9	**A**	**C**	**E**	**E**	**E**	**G**	**M**	**R**	**R**	**T**	6	10										**T**
Результат слияния		A	C	E	E	E	G	M	R	R	T												

Рис. 2.2.2. *Трассировка абстрактного слияния на месте*

Нисходящая сортировка слиянием

Алгоритм 2.4 (см. листинг 2.2.2) представляет собой рекурсивную реализацию сортировки слиянием, основанную на этом абстрактном слиянии на месте. Это один из наиболее известных примеров применения парадигмы *разделяй и властвуй* для построения эффективных алгоритмов. На данном рекурсивном коде основано индуктивное доказательство, что алгоритм действительно упорядочивает массив: если он сортирует два подмассива, то он сортирует и весь массив, сливая их вместе.

Листинг 2.2.2. Нисходящая сортировка слиянием

```
public class Merge
{
  private static Comparable[] aux;     // Вспомогательный массив для слияний.
  public static void sort(Comparable[] a)
  {
    aux = new Comparable[a.length];    // Память выделяется один раз.
    sort(a, 0, a.length - 1);
  }
  private static void sort(Comparable[] a, int lo, int hi)
  {  // Сортировка a[lo..hi].
    if (hi <= lo) return;
    int mid = lo + (hi - lo)/2;
    sort(a, lo, mid);         // Сортировка левой половины.
    sort(a, mid+1, hi);       // Сортировка правой половины.
    merge(a, lo, mid, hi);    // Слияние результатов (листинг 2.2.1).
  }
}
```

Для упорядочения подмассива `a[lo..hi]` мы делим его на две части: `a[lo..mid]` и `a[mid+1..hi]`, сортируем их независимо (рекурсивными вызовами) и сливаем упорядоченные подмассивы для получения результата (рис. 2.2.3).

	a[]															
lo hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**0**,0,**1**)	E	M	**R**	**G**	**E**	**S**	**O**	**R**	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**2**,2,**3**)	**E**	**M**	G	R	**E**	**S**	**O**	**R**	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**0**,1,**3**)	E	G	M	R	**E**	**S**	**O**	**R**	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**4**,4,**5**)	**E**	**G**	**M**	**R**	E	S	**O**	**R**	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**6**,6,**7**)	**E**	**G**	**M**	**R**	**E**	**S**	O	R	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**4**,5,**7**)	**E**	**G**	**M**	**R**	E	O	R	S	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**0**,3,**7**)	E	E	G	M	O	R	R	S	**T**	**E**	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**8**,8,**9**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	E	T	**X**	**A**	**M**	**P**	**L**	**E**
merge(a,**10**,10,**11**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	**E**	**T**	A	X	**M**	**P**	**L**	**E**
merge(a,**8**,9,**11**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	T	X	**M**	**P**	**L**	**E**
merge(a,**12**,12,**13**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	**A**	**E**	**T**	**X**	M	P	**L**	**E**
merge(a,**14**,14,**15**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	**A**	**E**	**T**	**X**	**M**	**P**	E	L
merge(a,**12**,13,**15**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	**A**	**E**	**T**	**X**	E	L	M	P
merge(a,**8**,11,**15**)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	E	L	M	P	T	X
merge(a,**0**,7,**15**)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.3. *Трассировка результатов слияния при выполнении нисходящей сортировки слиянием*

Чтобы разобраться, как работает сортировка слиянием, полезно внимательно рассмотреть динамику вызовов методов, показанную на рис. 2.2.4. Чтобы упорядочить содержимое массива `a[0..15]`, метод `sort()` вызывает себя для сортировки `a[0..7]`, затем для сортировки `a[0..3]` и `a[0..1]` и, наконец, после вызовов с `a[0]` и `a[1]` приступает к их слиянию (для краткости мы опустили вызовы для простейших вызовов с одним элементом). После этого выполняются слияния `a[2]` с `a[3]`, `a[0..1]` с `a[2..3]` и т.д.

Из этой трассировки видно, что код сортировки просто организует последовательность вызовов метода `merge()`. Мы еще вернемся к этому наблюдению ниже в настоящем разделе. Рекурсивный код дает также основания для анализа времени выполнения сортировки слиянием. Мы проведем этот анализ максимально подробно, т.к. сортировка слиянием является прототипом общего принципа построения алгоритмов "разделяй и властвуй".

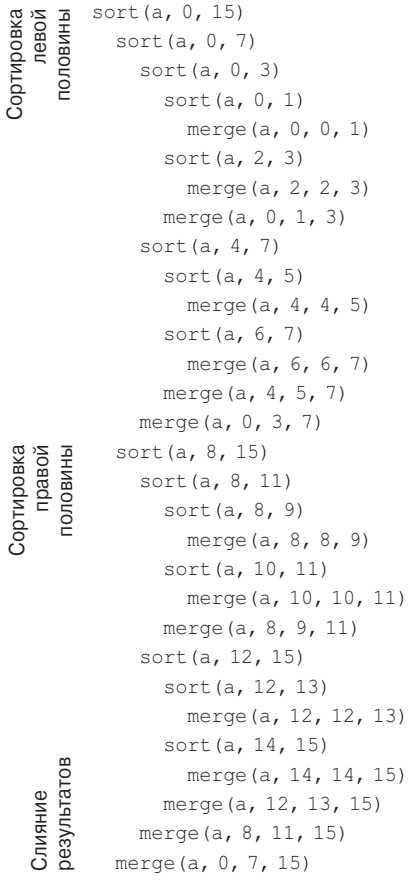

Рис. 2.2.4. *Трассировка вызовов нисходящей сортировки слиянием*

Утверждение Е. При упорядочении любого массива длиной N нисходящая сортировка слиянием использует от $\frac{1}{2}N\lg N$ до $N\lg N$ сравнений.

Доказательство. Пусть $C(N)$ — количество сравнений, необходимое для упорядочения массива длиной N. В крайних случаях $C(0) = C(1) = 0$, а для $N > 0$ можно записать рекуррентное отношение для верхней границы, которое непосредственно соответствует рекурсивному методу `sort()`:

$$C(N) \le C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + N$$

Первое слагаемое в правой части — количество сравнений для упорядочения левой половины массива, второе — количество сравнений для упорядочения правой половины, а третье — количество сравнений при сортировке. Нижняя граница

$$C(N) \ge C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + \lfloor N/2 \rfloor$$

следует из того, что количество сравнений при слиянии не меньше $\lfloor N/2 \rfloor$.

Мы выведем точное решение рекуррентного уравнения для случая точного равенства и N, равного степени 2 (т.е. $N = 2^n$). Во-первых, поскольку $\lfloor N/2 \rfloor = \lceil N/2 \rceil = 2^{n-1}$, то

$$C(2^n) = 2C(2^{n-1}) + 2^n$$

Поделив обе части этого равенства на 2^n, получим

$$C(2^n)/2^n = C(2^{n-1})/2^{n-1} + 1$$

Применение этого же равенства к первому слагаемому в правой части дает

$$C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1$$

И, повторив предыдущий шаг еще $n-1$ раз, получим

$$C(2^n)/2^n = C(2^0)/2^0 + n$$

откуда, после умножения обеих частей на 2^n, следует решение:

$$C(N) = C(2^n) = n2^n = N\lg N$$

Точные решения для произвольных N более сложны, но аналогичные рассуждения в отношении неравенств пригодны и для границ количества сравнений при произвольных значениях N. Это доказательство верно вне зависимости от входных значений и их порядка.

Для понимания утверждения Е удобно рассмотреть дерево, приведенное на рис. 2.2.5 — в нем каждый узел означает подмассив, для которого метод `sort()` выполняет операцию `merge()`. Это дерево состоит точно из n уровней. На k-м сверху уровне ($k = 0, ..., n-1$) имеются 2^k подмассивов, каждый длиной 2^{n-k}, и каждый требует для слияния 2^{n-k} сравнений. Поэтому для каждого из n уровней общая стоимость равна 2^n, а для всего дерева необходимо $n2^n = N\lg N$ сравнений.

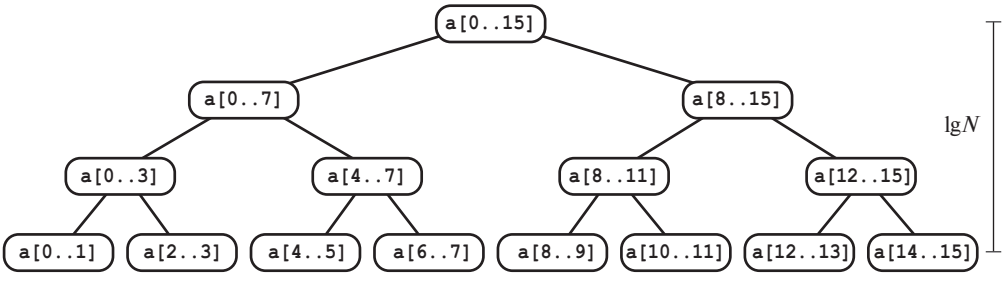

Рис. 2.2.5. *Дерево зависимостей подмассивов для сортировки слиянием при $N = 16$*

Утверждение Ж. Для упорядочения массива длиной N нисходящая сортировка слиянием использует не более $6N\lg N$ обращений к массиву.

Доказательство. Каждое слияние обращается к массиву максимум $6N$ раз: $2N$ для копирования, $2N$ — для записи назад и не более $2N$ — для сравнений. Результат следует из таких же рассуждений, как и для утверждения Е.

Из утверждений Е и Ж следует, что для сортировки слиянием можно ожидать время выполнения, пропорциональное $N\lg N$. Так что по сравнению с элементарными методами из раздела 2.1 мы переходим на другой уровень: теперь мы можем сортировать очень большие массивы, затрачивая лишь в логарифмическое количество раз больше времени, чем на просмотр каждого отдельного элемента. Сортировка слиянием позволяет сортировать многие миллионы элементов, но это невозможно с помощью сортировки вставками или выбором. Основным недостатком сортировки слиянием можно считать требование дополнительной памяти объемом, пропорциональным N, которая нужна для вспомогательного массива. Если память является дорогостоящим ресурсом, то необходим другой метод. Но, с другой стороны, можно существенно сократить время выполнения сортировки слиянием, подобрав для реализации специальные модификации.

Использование сортировки вставками для маленьких подмассивов

Большинство рекурсивных алгоритмов можно улучшить, обрабатывая небольшие случаи специальным образом. Рекурсия *гарантирует*, что небольшие случаи встречаются очень часто, поэтому усовершенствование их обработки приводит к усовершенствованию всего алгоритма. В случае сортировки мы знаем, что сортировка вставками (или сортировка выбором) проста и поэтому может работать быстрее для маленьких подмас-

сивов, чем сортировка слиянием. Как обычно, понять функционирование сортировки слиянием помогает визуальная трассировка. На рис. 2.2.6 демонстрируется работа реализации сортировки слиянием с отсечкой для небольших подмассивов. Переключение на сортировку вставками для небольших подмассивов (скажем, длиной 15 или меньше) может улучшить время выполнения типичной реализации сортировки слиянием на 10–15% (см. упражнение 2.2.23).

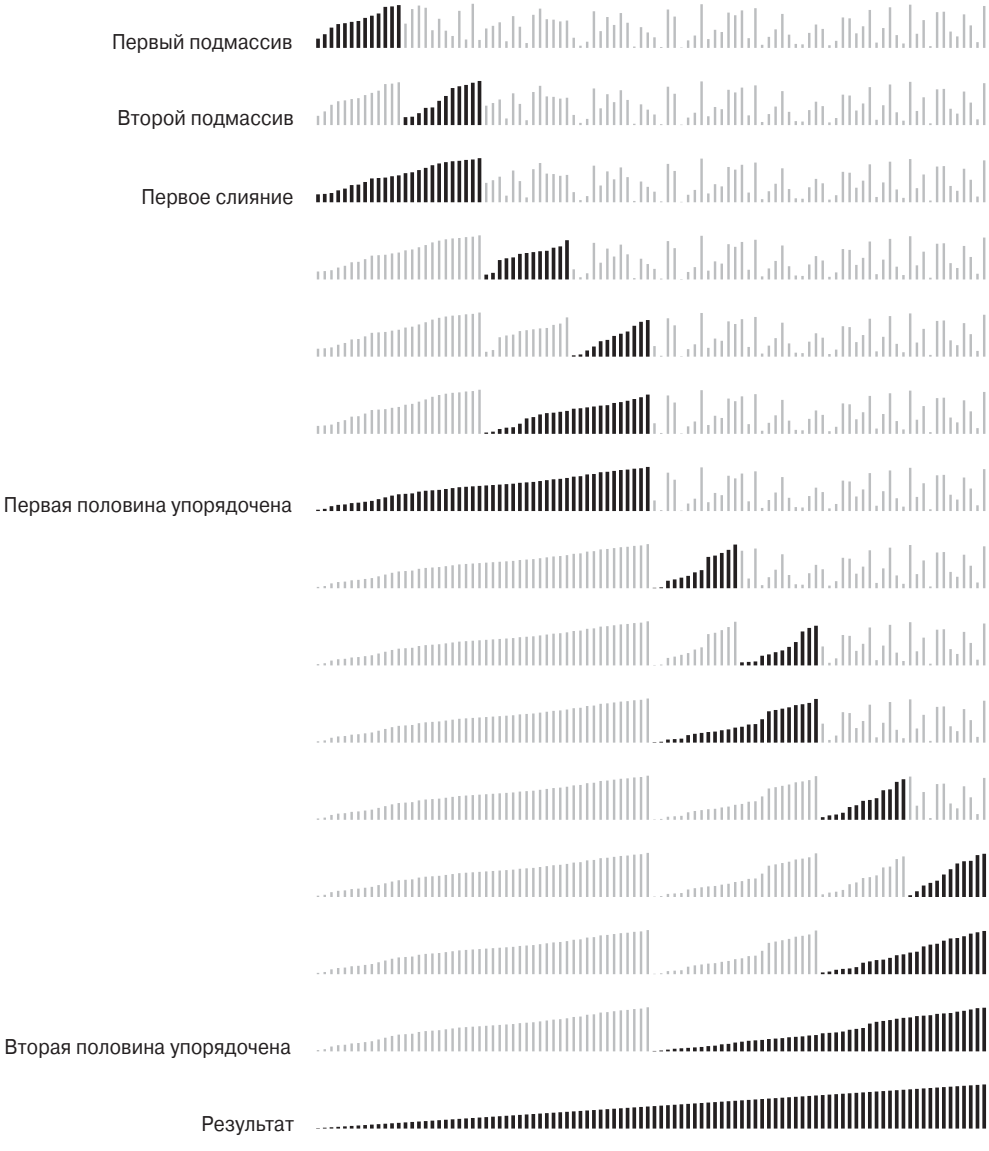

Рис. 2.2.6. *Визуальная трассировка нисходящей сортировки слиянием с отсечкой небольших подмассивов*

Проверка массива на изначальную упорядоченность

Время выполнения для уже упорядоченных массивов можно снизить до линейного, введя проверку на пропуск вызова `merge()`, если `a[mid]` меньше или равно `a[mid+1]`. При этом все рекурсивные вызовы также выполняются, но время выполнения для упорядоченных подмассивов уже линейно (см. упражнение 2.2.8).

Отказ от копирования во вспомогательный массив

Можно устранить время (но не память), необходимое для копирования во вспомогательный массив при слиянии. Для этого нужно воспользоваться двумя разновидностями метода сортировки: одна берет данные из указанного массива и помещает упорядоченные выходные данные во вспомогательный массив, а другая берет данные из вспомогательного массива и помещает упорядоченные выходные данные в исходный массив. С помощью этого приема и небольших рекурсивных хитростей можно так упорядочить рекурсивные вызовы, что входной и вспомогательный массивы будут меняться ролями на каждом уровне (см. упражнение 2.2.11).

Здесь уместно повторить то, о чем уже было сказано в главе 1, но что легко забыть. Локально каждый алгоритм в этой книге рассматривается как критический для какого-то применения. А глобально мы пытаемся нащупать общие выводы, чтобы рекомендовать какой-то подход. Наши описания таких усовершенствований не обязательно означают рекомендацию реализовывать их во всех ситуациях — скорее, это рекомендация не делать абсолютных выводов о производительности на основе первых реализаций. При рассмотрении новой задачи лучше всего использовать самую простую реализацию, с которой у вас не будет никаких проблем, а затем усовершенствовать ее, если она станет узким местом в приложении. Обычно не стоит тратить время на улучшения, которые уменьшают время выполнения только на небольшой постоянный множитель. Поэтому необходимо экспериментально проверять эффективность конкретных усовершенствований, как мы постоянно указываем в упражнениях.

В случае сортировки слиянием три перечисленных выше улучшения несложно реализовать, и они вполне пригодны, если в конкретной ситуации оптимален метод слияния — например, в ситуациях, описанных в конце данной главы.

Восходящая сортировка слиянием

Рекурсивная реализация сортировки слиянием представляет собой прототип принципа *разделяй и властвуй*, где для решения большая задача делится на части, затем решаются меньшие подзадачи, а на основе полученных решений выводится решение и всей исходной задачи (рис. 2.2.7). Даже если мы обдумываем слияние двух больших подмассивов, на самом деле бо́льшая часть времени уходит на слияния маленьких подмассивов. Но слияния можно организовать и другим способом: сначала выполнить их для маленьких подмассивов, на следующем проходе попарно слить эти подмассивы и продолжать так, пока слияние не охватит весь массив. Код для такого метода даже короче, чем стандартная рекурсивная реализация. Сначала выполняется проход слияний 1 с 1 (отдельные элементы считаются подмассивами размера 1), потом проход слияний 2 с 2 (сливаются подмассивы размера 2 и получаются подмассивы размера 4), затем проход слияний 4 с 4 и т.д. В последнем слиянии на каждом проходе второй подмассив может быть короче первого (что не представляет сложности для метода `merge()`), но в остальных случаях все слияния обрабатывают подмассивы одинакового размера и удваивают размер упорядоченных подмассивов для следующего прохода.

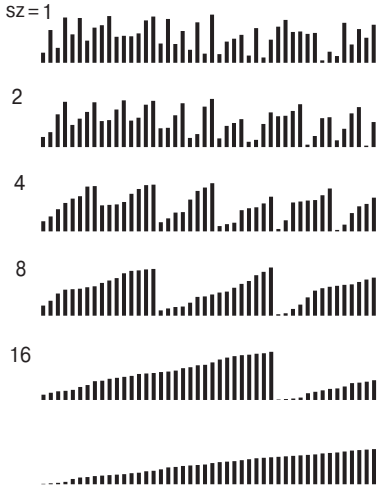

Рис. 2.2.7. Визуальная трассировка восходящей сортировки слиянием

Листинг 2.2.3. Восходящая сортировка слиянием

```
public class MergeBU
{
  private static Comparable[] aux;     // вспомогательный массив для слияний
  // Код merge() см. в листинге 2.2.1.
  public static void sort(Comparable[] a)
  {  // Выполнение lgN проходов попарных слияний.
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz)        // sz — размер подмассива
      for (int lo = 0; lo < N-sz; lo += sz+sz) // lo — индекс в подмассиве
        merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
  }
}
```

Восходящая сортировка слиянием состоит из последовательности проходов по всему массиву, на каждом их которых выполняются слияния `sz` с `sz`. Сначала `sz` = 1, а на каждом проходе это значение удваивается. Последний подмассив имеет размер `sz` только в том случае, если размер массива кратен удвоенному `sz` (иначе он меньше `sz`) (рис. 2.2.8).

Утверждение 3. Для упорядочения массива длиной N восходящая сортировка слиянием использует от $\frac{1}{2}N\lg N$ до $N\lg N$ сравнений и не более $6N\lg N$ обращений к массиву.

Доказательство. Количество проходов по массиву в точности равно $\lfloor \lg N \rfloor$ (а это значение в точности равно такому n, что $2^n \le N < 2^{n+1}$). На каждом проходе количество обращений к массиву в точности равно $6N$, а количество сравнений не больше N и не меньше $N/2$.

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz=1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a,10,10,11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a,12,12,13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a,14,14,15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz=2																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9,11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a,12,13,15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz=4																
merge(a,0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a,8,11,15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz=8																
merge(a,0,7,15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.8. *Трассировка результатов слияния при выполнении восходящей сортировки слиянием*

Если длина массива равна степени 2, то нисходящая и восходящая сортировки слиянием выполняют совершенно одинаковые сравнения и обращения к элементам массива, только в разном порядке. Если длина массива не равна степени 2, то последовательности сравнений и обращений к массиву для этих двух алгоритмов будут различными (см. упражнение 2.2.5).

Восходящая сортировка слиянием удобна при организации данных в виде *связных списков*. Сначала сортируемый массив рассматривается как последовательность подсписков длиной 1, потом проход по массиву создает связанные между собой упорядоченные подмассивы длиной 2, затем длиной 4 и т.д. Этот метод переупорядочивает ссылки и поэтому может сортировать список *на месте* (без создания новых узлов списка).

И нисходящий, и восходящий подходы к реализации алгоритмов типа “разделяй и властвуй” интуитивно понятны. Из реализаций сортировки слиянием можно сделать следующий вывод: когда рассматривается алгоритм, основанный на одном из этих подходов, полезно рассмотреть и другой подход. Задачу иногда удобнее решать, разбивая ее на меньшие задачи (с рекурсивным их решением), как в методе `Merge.sort()`, а иногда — объединяя маленькие решения в большие, как в методе `MergeBU.sort()`.

Сложность сортировки

Сортировка слиянием важна, в частности, тем, что она используется как базис для доказательства фундаментального результата в области *вычислительной сложности*, который помогает понять сложность сортировки вообще. Как правило, вычислительная сложность играет важную роль в проектировании алгоритмов, и этот результат непосредственно влияет на проектирование алгоритмов сортировки, поэтому сейчас мы займемся этим вплотную.

Первое, что нужно сделать при изучении сложности — определиться с моделью вычислений. Обычно исследователи стараются найти простейшую модель, которая все-таки соотносится с задачей. В случае сортировки мы рассматриваем класс алгоритмов, *основанных на сравнениях*, которые принимают решения об элементах только на основе сравнения их ключей. Основанный на сравнениях алгоритм может выполнить произвольный объем вычислений между сравнениями, но он не может получить какую-либо информацию о ключе кроме его сравнения с другим ключом. Поскольку мы ограничиваемся API-интерфейсом `Comparable`, все алгоритмы в настоящей главе находятся в этом классе (с игнорированием стоимости обращений к массиву) — как, впрочем, и многие другие алгоритмы, которые мы можем себе представить. В главе 5 мы рассмотрим алгоритмы, которые не опираются на свойство `Comparable` элементов.

Утверждение И. Ни один алгоритм сортировки, основанный на сравнениях, не может гарантированно упорядочить N элементов, выполнив менее $\lg(N!) \sim N\lg N$ сравнений.

Доказательство. Первым делом, мы будем считать, что все ключи различны, т.к. такую сортировку должен уметь выполнить любой алгоритм. Тогда для описания последовательности сравнений можно использовать бинарное дерево. Каждый *узел* в этом дереве представляет собой либо *лист* ($i_0\ i_1\ i_2 \ldots i_{N-1}$), который означает, что сортировка завершена и исходные данные находятся в порядке $a[i_0]$, $a[i_1]$, ..., $a[i_{N-1}]$, либо *внутренний узел* (i:j) — такой узел соответствует операции сравнения элементов `a[i]` и `a[j]`, и его левое поддерево соответствует последовательности сравнений в случае, если `a[i]` меньше `a[j]`, а правое — случаю, если `a[i]` больше `a[j]`. Путь от корня до любого листа соответствует последовательности сравнений, выполненных алгоритмом для формирования упорядоченности, сформулированной в листе. Например, на рис. 2.2.9 показано дерево сравнений для $N=3$.

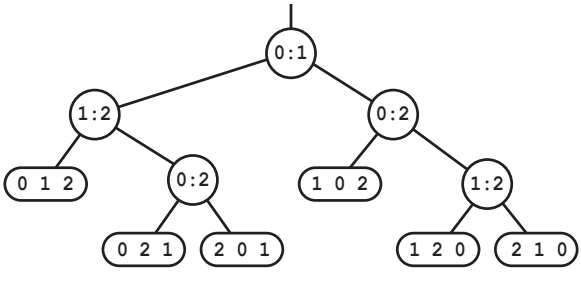

Рис. 2.2.9. *Дерево сравнений при сортировке трех элементов*

Мы никогда не будем явно создавать такие деревья — это просто математический аппарат для описания сравнений, используемых любым алгоритмом.

Первое заключение, важное для доказательства: дерево должно содержать не менее $N!$ листьев, т.к. существуют $N!$ различных перестановок из N различных ключей. Если листьев меньше $N!$, то, значит, какие-то перестановки ими не охвачены, и алгоритм не сможет работать для таких сочетаний ключей.

Количество внутренних узлов на пути от корня к любому листу дерева равно количеству сравнений, используемых алгоритмом для некоторых входных данных.

Нас интересует длина самого длинного такого пути в дереве (*высота* дерева), поскольку она равна количеству сравнений, выполняемых алгоритмом в худшем случае. Одно из основных комбинаторных свойств бинарных деревьев гласит, что дерево высотой h не может иметь более 2^h листьев: дерево высотой h с максимальным количеством листьев полностью сбалансировано, т.е. *полно*. Пример для $h = 4$ приведен на рис. 2.2.10.

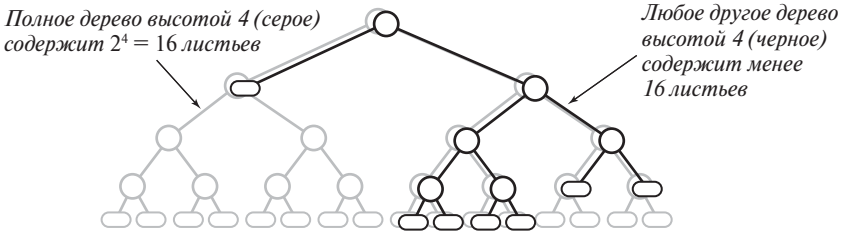

Рис. 2.2.10. *Полное и произвольное деревья высотой 4*

Из последних двух абзацев следует, что любой алгоритм сортировки, основанный на сравнениях, соответствует дереву сравнений высотой h, такой, что

$$N! \le \textit{количество листьев} \le 2^h$$

Это проиллюстрировано на рис. 2.2.11.

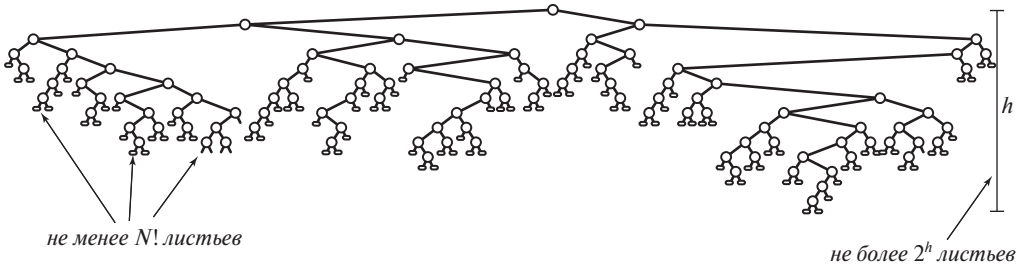

Рис. 2.2.11. *Количество листьев в дереве сравнений*

Значение h в точности равно количеству сравнений в худшем случае, поэтому можно взять логарифм (по основанию 2) от крайних частей этой формулы и получить, что количество сравнений, используемых любым алгоритмом, не может быть меньше $\lg(N!)$. Приблизительное значение $\lg(N!) \sim N\lg N$ следует из аппроксимации Стирлинга для функции факториала (см. табл. 1.4.5).

Этот результат — указание, чего следует ожидать при проектировании алгоритма сортировки. Например, не имея такого результата, кто-то мог бы долго искать алгоритм сортировки на основе сравнений, который использует в худшем случае вдвое меньше сравнений, чем сортировка слиянием. Нижняя граница в утверждении И показывает, что подобные усилия не увенчаются успехом: *такой алгоритм невозможен*. Это очень сильное высказывание, применимое к любому мыслимому алгоритму на основе сравнений.

Утверждение З гласит, что количество сравнений, выполняемых сортировкой слиянием в худшем случае, равно $\sim N\lg N$. Этот результат — *верхняя граница* сложности задачи сортировки, в том смысле, что лучшему алгоритму пришлось бы гарантировать выполнение меньшего количества сравнений. Утверждение И гласит, что ни один алгоритм сортировки не может гарантировать выполнение меньше $\sim N\lg N$ сравнений. Это *нижняя граница* сложности задачи сортировки в том смысле, что даже самый наилучший алгоритм должен использовать не менее такого количества сравнений в худшем случае. Давайте посмотри, что означает все это вместе.

Утверждение К. Сортировка слиянием является асимптотически оптимальным алгоритмом сортировки на основе сравнений.

Доказательство. Это утверждение в точности означает, что *количество сравнений, выполняемых сортировкой слиянием в худшем случае, и минимальное количество сравнений, которое может гарантировать любой алгоритм сортировки на основе сравнений, равны* $\sim N\lg N$. Эти факты уже установлены в утверждениях З и И.

Здесь важно понимать, что, как и в случае с моделью вычислений, необходимо четко определить, *что* мы понимаем под оптимальным алгоритмом. Например, можно ужесточить определение оптимальности и требовать, чтобы оптимальный алгоритм сортировки выполнял *в точности* $\lg(N!)$ сравнений. Мы не будем этого делать, т.к. при больших N мы не сможем заметить разницу между таким алгоритмом и (к примеру) сортировкой слиянием. Либо можно расширить определение оптимальности, чтобы ему соответствовал любой алгоритм сортировки, для которого количество сравнений в худшем случае равно $N\lg N$ *с некоторым постоянным коэффициентом*. Мы не будем делать и этого, поскольку разница между таким алгоритмом и сортировкой слиянием будет хорошо заметна при больших N.

Вопрос вычислительной сложности может показаться несколько абстрактным, но относительно фундаментальных исследований сложности, присущей решению вычислительных задач, сомнения обычно не возникают. Более того, в случае, когда приходится рассматривать этот вопрос, именно он помогает разработке хорошего ПО. Во-первых, хорошие верхние границы позволяют проектировщикам ПО получить гарантии производительности: описано много случаев, когда поиск причин низкой производительности приводил к обнаружению квадратичной сортировки вместо линейно-логарифмической. Во-вторых, хорошо определенные нижние границы позволяют не тратить силы на поиски повышения производительности, которое в принципе недоступно.

Однако оптимальность сортировки слиянием еще не означает, что рассказ о сортировке завершен и нет смысла рассматривать другие методы. Дело в том, что теория, изложенная в данном разделе, имеет ряд ограничений, например:

- сортировка слиянием не оптимальна в смысле использования памяти;
- худший случай не обязательно возникает на практике;
- могут быть важны и другие операции, отличные от сравнений (например, обращения к массиву);
- некоторые виды данных можно упорядочивать, *вообще* не выполняя сравнений.

Поэтому в настоящей книге мы рассмотрим и несколько других методов сортировки.

Вопросы и ответы

Вопрос. **Сортировка слиянием быстрее, чем сортировка Шелла?**

Ответ. На практике времена их выполнения отличаются друг от друга на небольшой постоянный множитель (если в сортировке Шелла применяется тщательно проверенная последовательность шагов, вроде приведенной в алгоритме 2.3), поэтому относительная производительность зависит от реализаций.

```
% java SortCompare Слияние Шелла 100000
Для 100000 случайных Double
  Слияние в 1.2 раза быстрее, чем Шелла
```

Пока никто не смог теоретически доказать, что сортировка Шелла является линейно-логарифмической для случайных данных, так что существует шанс, что асимптотический рост средней производительности сортировки Шелла выше. Такой разрыв доказан для производительности в худшем случае, но он не имеет практического значения.

Вопрос. **Почему бы не сделать массив `aux[]` локальным в методе `merge()`?**

Ответ. Чтобы не было излишних расходов на создание массива при каждом слиянии, даже небольшого размера. Иначе эти затраты доминировали бы во времени выполнения сортировки слиянием (см. упражнение 2.2.26). Более правильное решение (которое мы не рассматривали, чтобы не загромождать код) — сделать массив `aux[]` локальным в методе `sort()` и передавать его в качестве аргумента в `merge()` (см. упражнение 2.2.9).

Вопрос. **Как ведет себя сортировка слиянием при наличии в массиве одинаковых значений?**

Ответ. Если все элементы имеют одинаковое значение, время выполнения линейно (при наличии дополнительной проверки и пропуска слияния, если массив уже упорядочен). Но если имеется несколько повторяющихся значений, то такой выигрыш в производительности достигается не всегда. Например, предположим, что входной массив содержит N элементов с одним значением в нечетных позициях и N элементов с другим значением в четных позициях. Для такого массива время выполнения является линейно-логарифмическим (в соответствии с рекуррентными соотношениями для элементов со всеми различными значениями), а не линейным.

Упражнения

2.2.1. Приведите трассировку, в стиле показанной на рис. 2.2.2, для слияния ключей `A E Q S U Y E I N O S` с помощью абстрактного метода `merge()` слияния на месте.

2.2.2. Приведите трассировки, в стиле показанной на рис. 2.2.3, для сортировки ключей `E A S Y Q U E S T I O N` с помощью нисходящей сортировки слиянием.

2.2.3. Выполните упражнение 2.2.2 для восходящей сортировки слиянием.

2.2.4. Правда ли, что абстрактное слияние на месте генерирует правильные выходные данные тогда и только тогда, когда два входных подмассива упорядочены? Обоснуйте свой ответ или приведите контрпример.

2.2.5. Приведите последовательность размеров подмассивов в слияниях, выполняемых алгоритмами нисходящей и восходящей сортировки слиянием для $N = 39$.

2.2.6. Напишите программу для вычисления точного количества обращений к массиву, выполняемых нисходящей и восходящей сортировками слиянием. С помощью этой программы начертите график для N от 1 до 512 и сравните полученные значения с верхней границей $6N\lg N$.

2.2.7. Покажите, что количество сравнений, которое использует сортировка слиянием, монотонно увеличивается ($C(N + 1) > C(N)$ для всех $N > 0$).

2.2.8. Предположим, что в алгоритм 2.4 добавлен пропуск вызова `merge()`, если `a[mid] <= a[mid+1]`. Докажите, что количество сравнений, используемых при сортировке слиянием уже упорядоченного массива, линейно.

2.2.9. В библиотечном ПО не рекомендуется задействовать статические массивы наподобие `aux[]`, т.к. данный класс могут одновременно использовать несколько клиентов. Приведите реализацию `Merge`, в которой не применяется статический массив. *Не* делайте массив `aux[]` локальным в методе `merge()` (см. "Вопросы и ответы" в данном разделе). *Совет*: передавайте вспомогательный массив в качестве аргумента в рекурсивный метод `sort()`.

Творческие задачи

2.2.10. *Более быстрое слияние.* Реализуйте версию метода `merge()`, которая копирует вторую половину массива `a[]` в массив `aux[]` в *убывающем порядке*, а затем выполняет слияние назад в `a[]`. Это изменение позволяет не использовать во внутреннем цикле код проверки, не закончилась ли каждая половина. *Примечание*: такая сортировка работает неустойчиво (см. подраздел "Устойчивость" в разделе 2.5).

2.2.11. *Усовершенствования.* Реализуйте три усовершенствования для сортировки слиянием, которые описаны в данном разделе после утверждения Ж: добавьте отсечку для небольших подмассивов, проверяйте упорядоченность входного массива и не выполняйте копирование, переключая аргументы в рекурсивном коде.

2.2.12. *Сублинейный объем дополнительной памяти.* Разработайте реализацию слияния, которое снижает потребность в дополнительной памяти до $\max(M, N/M)$ с помощью следующего приема. Разбейте массив на N/M блоков размером M (для простоты предположим, что N кратно M). Затем, (1) считая блоки элементами с ключом сортировки, равным ключу первого элемента в блоке, отсортируйте их с помощью сортировки выбором; (2) выполните проход по массиву, сливая первый блок со вторым, потом второй с третьим и т.д.

2.2.13. *Нижняя граница для среднего случая.* Докажите, что *ожидаемое* количество сравнений, которое выполняет любой алгоритм сортировки на основе сравнений, должно быть не менее $\sim N\lg N$ (если все возможные упорядочения входных данных равновероятны). *Подсказка*: ожидаемое количество сравнений не меньше длины внешнего пути дерева сравнений (суммы длин путей от корня до всех листьев), которое минимально в сбалансированном случае.

2.2.14. *Слияние упорядоченных очередей.* Разработайте статический метод, который принимает в качестве аргументов две очереди упорядоченных элементов и возвращает очередь, полученную слиянием исходных очередей в порядке возрастания.

2.2.15. *Восходящая сортировка слиянием для очередей.* Разработайте реализацию восходящей сортировки слиянием на основе следующего принципа. Из N исходных элементов создайте N очередей, каждая из одного элемента. Создайте очередь из этих N очередей. Затем многократно выполняйте операцию слияния из упражнения 2.2.14 для первых двух очередей с помещением слитой очереди в конец. Повторяйте до тех пор, пока очередь очередей не будет содержать только одну очередь.

2.2.16. *Естественная сортировка слиянием.* Напишите версию восходящей сортировки слиянием, которая использует упорядоченность массива, выполняя следующую процедуру, когда ей нужны для слияния два массива: она находит упорядоченный подмассив (увеличивая указатель, пока не встретится элемент, меньший, чем его предшественник в массиве), потом находит следующий, затем сливает их. Выразите время выполнения этого алгоритма через размер массива и количество максимальных увеличивающихся последовательностей в массиве.

2.2.17. *Сортировка связных списков.* Реализуйте естественную сортировку слиянием для связных списков. (Это самый удобный способ для сортировки связных списков, потому что он не требует дополнительной памяти и гарантированно выполняется за линейно-логарифмическое время.)

2.2.18. *Тасование связного списка.* Разработайте и реализуйте алгоритм типа "разделяй и властвуй", который случайным образом тасует связный список за линейно-логарифмическое время, используя дополнительную память логарифмического объема.

2.2.19. *Вставки.* Разработайте и реализуйте линейно-логарифмический алгоритм для вычисления количества перестановок в заданном массиве (количество обменов, необходимое сортировке вставками для этого массива — см. раздел 2.1). Эта величина связана с *тау-расстоянием Кенделла*; см. раздел 2.5.

2.2.20. *Косвенная сортировка.* Разработайте и реализуйте версию сортировки слиянием, которая не переупорядочивает массив, а возвращает массив `perm` типа `int[]` — такой, что `perm[i]` содержит индекс `i`-го наименьшего элемента в массиве.

2.2.21. *Трипликаты.* Пусть имеются три списка, содержащие каждый по N имен. Разработайте линейно-логарифмический алгоритм для определения, имеется ли имя, которое находится во всех трех списках. Если да, то нужно возвратить первое такое имя.

2.2.22. *3-частная сортировка слиянием.* Допустим, что вместо деления массива пополам на каждом шаге, он делится на трети, потом эти трети сортируются и объединяются с помощью 3-частного слияния. Каков порядок роста общего времени выполнения для этого алгоритма?

Эксперименты

2.2.23. *Усовершенствования.* Экспериментально оцените эффективность каждого из трех усовершенствований сортировки слиянием, которые описаны в тексте раздела (см. упражнение 2.2.11). Также сравните производительность реализации слияния, приведенного в тексте, со слиянием, описанным в упражнении 2.2.10. В частности, эмпирически определите наилучшее значение размера небольших подмассивов, при котором надо переключаться на сортировку вставками.

2.2.24. *Усовершенствование с помощью проверки на упорядоченность.* Экспериментально оцените для больших случайно упорядоченных массивов эффективность модификации, описанной в упражнении 2.2.8 для случайных данных. В частности, сформулируйте гипотезу о среднем количестве проверок, завершившихся успешно (т.е. массив уже упорядочен), в виде функции от N (размер исходного сортируемого массива).

2.2.25. *Многочастная сортировка слиянием.* Разработайте реализацию сортировки слиянием с помощью k-частных слияний (а не 2-частных, как обычно). Проанализируйте полученный алгоритм, сформулируйте гипотезу о лучшем значении k и экспериментально проверьте эту гипотезу.

2.2.26. *Создание массива.* Используйте программу `SortCompare` для получения грубой оценки влияния на производительность на вашем компьютере создания массива `aux[]` в методе `merge()`, а не в `sort()`.

2.2.27. *Длины подмассивов.* С помощью выполнения сортировки слиянием для больших случайных массивов эмпирически определите среднюю длину другого подмассива, когда в первом подмассиве заканчиваются данные, в виде функции от N (суммы размеров двух подмассивов для данного слияния).

2.2.28. *Нисходящие и восходящие слияния.* Используйте программу `SortCompare` для сравнения нисходящей и восходящей сортировки слиянием для $N = 10^3$, 10^4, 10^5 и 10^6.

2.2.29. *Естественная сортировка слиянием.* Эмпирически определите количество проходов, необходимое для естественной сортировки слиянием (см. упражнение 2.2.16) для случайных ключей `Long` и $N = 10^3$, 10^6 и 10^9. *Подсказка*: для выполнения этого упражнения не обязательно реализовывать сортировку (и даже генерировать полные 64-битовые ключи).