4... Нижняя граница времени сортировки и как ее превзойти

В предыдущей главе вы познакомились с четырьмя алгоритмами для сортировки n элементов в массиве. Два из них, сортировка выбором и сортировка вставкой, имеют время работы в наихудшем случае, равное $\Theta(n^2)$, что не очень-то хорошо. Один из алгоритмов, алгоритм быстрой сортировки, в наихудшем случае имеет время работы $\Theta(n^2)$, но в среднем случае выполняет сортировку только за время $\Theta(n \lg n)$. Сортировка слиянием имеет время работы $\Theta(n \lg n)$ во всех случаях. На практике наиболее эффективной является быстрая сортировка, но если вы хотите абсолютно гарантированно защититься от наихудшего случая, следует использовать сортировку слиянием.

Но насколько хорошим является время $\Theta(n \lg n)$? Нельзя ли разработать алгоритм сортировки, который в наихудшем случае превзойдет время $\Theta(n \lg n)$? Ответ зависит от правил игры: как алгоритм сортировки может использовать ключи сортировки для определения порядка сортировки?

В этой главе мы увидим, что при определенном наборе правил превзойти время работы $\Theta(n \lg n)$ невозможно. Затем мы рассмотрим два алгоритма сортировки, сортировку подсчетом и карманную сортировку, которые используют другие правила, а потому в состоянии выполнять сортировку за время всего лишь $\Theta(n)$.

Правила сортировки

Если рассмотреть, как четыре алгоритма из предыдущей главы используют ключи сортировки, то можно увидеть, что они определяют порядок сортировки, основываясь только на сравнении пары ключей. Все принимаемые ими решения имеют вид "если ключ сортировки этого элемента меньше, чем ключ сортировки другого элемента, то то-то, а в противном случае либо сделать что-то еще, либо ничего не делать". Вы можете подумать, что алгоритм сортировки может принимать решения *только* такого вида. А какие еще виды решений он в состоянии принимать?

Чтобы убедиться в том, что возможны и другие виды решений, давайте рассмотрим очень простой пример. Предположим, что мы знаем две вещи о сортируемых элементах, а именно — что каждый ключ сортировки является либо единицей, либо двойкой и что элементы состоят только из ключей сортировки, не имея никаких сопутствующих данных. В этой простой ситуации можно сортировать n элементов за время $\Theta(n)$, превзойдя алгоритмы со временем работы $\Theta(n \lg n)$ из предыдущей главы. Каким образом? Начнем с того, что пройдем по всем элементам и подсчитаем, сколько среди них единиц, — скажем, это k элементов. Тогда можно вновь пройти через массив, устанавливая значение 1 в первых k позициях, а затем устанавливая значение 2 в остальных $n-k$ позициях. Вот как выглядит соответствующая процедура.

Процедура REALLY-SIMPLE-SORT(A,n)

Вход:

- A: массив, все элементы которого имеют значения 1 или 2.
- n: количество сортируемых элементов A.

Результат: элементы A отсортированы в неубывающем порядке.

1. Установить k равным нулю.
2. Для $i = 1$ до n:
   A. Если $A[i] = 1$, увеличить k на единицу.
3. Для $i = 1$ до k:
   A. Установить $A[i]$ равным 1.
4. Для $i = k + 1$ до n:
   A. Установить $A[i]$ равным 2.

Шаги 1 и 2 подсчитывают количество единиц, увеличивая счетчик k для каждого элемента $A[i]$, равного 1. Шаг 3 заполняет подмассив $A[1..k]$ единицами, а шаг 4 заполняет остальные позиции, $A[k+1..n]$, двойками. Легко видеть, что эта процедура выполняется за время $\Theta(n)$: первый цикл выполняет n итераций, как и два последних цикла вместе; каждая итерация каждого цикла выполняется за постоянное время.

Обратите внимание, что процедура REALLY-SIMPLE-SORT никогда не сравнивает два элемента массива один с другим. Она сравнивает каждый элемент массива со значением 1, но не с другим элементом массива. Так что, как видите, в такой ограниченной ситуации возможна сортировка без сравнения пар ключей сортировки.

Нижняя граница сортировки сравнением

Теперь, когда у вас есть некоторое представление о том, как могут варьироваться правила игры, давайте рассмотрим, насколько быстрой может быть сортировка.

Определим ***сортировку сравнением*** как любой алгоритм сортировки, который определяет порядок сортировки только путем сравнения пар элементов. Сортировкой сравнением являются четыре алгоритма сортировки из предыдущей главы, но не алгоритм REALLY-SIMPLE-SORT.

Вот нижняя граница сортировки сравнением.

> В наихудшем случае любой алгоритм сортировки сравнением требует для сортировки n элементов $\Omega(n \lg n)$ сравнений пар элементов.

Вспомним, что Ω-обозначение дает нижнюю границу, так что мы, по сути, говорим “для достаточно больших n любой алгоритм сортировки сравнением требует в наихудшем случае выполнения по крайней мере $cn \lg n$ сравнений для некоторой константы c”. Поскольку каждое сравнение выполняется по меньшей мере за постоянное время, это дает

нам нижнюю границу $\Omega(n \lg n)$ времени сортировки *n* элементов при условии, что используется алгоритм сортировки сравнением.

Важно иметь ясное представление о нижней границе. Первое — она говорит что-то только о наихудшем случае. Вы всегда можете сделать алгоритм сортировки работающим за линейное время в наилучшем случае: просто заявить, что наилучший случай — это когда массив уже отсортирован, и просто проверить, что каждый элемент (за исключением последнего) не превышает его преемника в массиве. Это легко сделать за время $\Theta(n)$. Если вы обнаружите, что каждый элемент не превышает его преемника, то сортировка выполнена. Однако в наихудшем случае $\Omega(n \lg n)$ сравнений являются необходимыми. Мы называем эту нижнюю границу ***экзистенциальной*** нижней границей, потому что она утверждает, что существуют входные данные, которые требуют $\Omega(n \lg n)$ сравнений. Другой тип нижней границы — ***универсальная*** нижняя граница, которая применима ко всем входным данным. В случае сортировки единственной универсальной нижней границей является $\Omega(n)$, поскольку мы должны взглянуть на каждый элемент по крайней мере один раз. Обратите внимание, что в предыдущем предложении я не сказал, к *чему* относится $\Omega(n)$. Имел ли я в виду $\Omega(n)$ сравнений или время работы $\Omega(n)$? Я подразумевал время, поскольку мы должны проверить каждый элемент, даже если не сравниваем элементы попарно.

Вторая важная вещь, которую следует сказать о нижней границе, действительно замечательна: это то, что нижняя граница не зависит от конкретного алгоритма, лишь бы этот алгоритм являлся алгоритмом сортировки сравнением. Нижняя граница применяется к *любому* алгоритму сортировки сравнением, независимо от того, насколько простым или сложным он является. Нижняя граница применима ко всем алгоритмам сортировки сравнением, которые уже были изобретены или еще только будут открыты в будущем. Она справедлива даже для тех алгоритмов сортировки сравнением, которые никогда не будут обнаружены человечеством!

Сортировка подсчетом

Мы уже видели, как превзойти нижнюю границу при очень ограниченных условиях, когда имеется только два возможных значения ключа сортировки, а каждый элемент состоит только из ключа сортировки, без сопутствующих данных. В этом ограниченном случае *n* элементов можно отсортировать за время $\Theta(n)$, без сравнения пар элементов.

Метод Really-Simple-Sort можно обобщить на случай *m* различных возможных значений ключей сортировки, которые являются целыми числами из диапазона, представляющего собой *m* последовательных целых чисел (скажем, от 0 до $m-1$), а элементы при этом могут иметь сопутствующие данные.

Вот в чем заключается идея. Предположим, мы знаем, что ключами сортировки являются целые числа в диапазоне от 0 до $m-1$. Кроме того, предположим, мы знаем, что ровно у трех элементов ключ сортировки равен 5 и что у шести элементов ключи сортировки меньше 5 (т.е. находятся в диапазоне от 0 до 4). Тогда мы знаем, что в отсортированном массиве элементы с ключом сортировки, равным 5, должны занимать позиции 7, 8 и 9.

Обобщая, если мы знаем, что у k элементов ключи сортировки равны x, а у l элементов ключи сортировки меньше x, то элементы с ключами сортировки, равными x, в отсортированном массиве должны занимать позиции от $l+1$ до $l+k$. Таким образом, нам надо для каждого возможного значения ключа сортировки вычислить, у какого количества элементов ключи сортировки ключей меньше этого значения и сколько имеется элементов с данным значением ключа сортировки.

Мы можем вычислить, у скольких элементов ключи сортировки меньше каждого из возможных значений ключа, если начнем с того, что вычислим, у какого количества элементов ключи сортировки равны заданному значению. Начнем нашу работу с решения этой задачи.

Процедура COUNT-KEYS-EQUAL(A,n,m)

Вход:

- A: массив целых чисел в диапазоне от 0 до $m-1$.
- n: количество элементов в массиве A.
- m: определяет диапазон значений в массиве A.

Выход: массив $equal[0..m-1]$, такой, что $equal[j]$ содержит количество элементов массива A, равных j, для $j = 0,1,2,\ldots,m-1$.

1. Пусть $equal[0..m-1]$ представляет собой новый массив.
2. Установить все значения массива *equal* равными нулю.
3. Для $i=1$ до n:

   A. Установить значение переменной *key* равным $A[i]$.

   B. Увеличить $equal[key]$ на единицу.
4. Вернуть массив *equal*.

Обратите внимание, что процедура COUNT-KEYS-EQUAL никогда не сравнивает ключи сортировки один с другим. Она использует ключи сортировки только в качестве индекса в массиве *equal*. Поскольку первый цикл (неявный цикл на шаге 2) делает m итераций, второй цикл (на шаге 3) делает n итераций, и каждая итерация каждого цикла выполняется за константное время, процедура COUNT-KEYS-EQUAL выполняется за время $\Theta(n+m)$. Если m является константой, то время работы COUNT-KEYS-EQUAL равно $\Theta(n)$.

Теперь мы можем использовать массив *equal* для выяснения, у какого количества элементов ключи сортировки меньше каждого возможного значения.

Процедура COUNT-KEYS-LESS($equal$,m)

Вход:

- *equal*: массив, возвращаемый вызовом процедуры COUNT-KEYS-EQUAL.
- m: определяет диапазон индексов массива *equal* — от 0 до $m-1$.

Выход: массив $less[0..m-1]$, такой, что для $j = 0,1,2,\ldots,m-1$ элемент $less[j]$ содержит сумму $equal[0] + equal[1] + \cdots + equal[j-1]$.

1. Пусть $less[0..m-1]$ представляет собой новый массив.
2. Установить $less[0]$ равным нулю.
3. Для $j = 1$ до $m-1$:

   A. Установить $less[j]$ равным $less[j-1] + equal[j-1]$.
4. Вернуть массив *less*.

В предположении, что $equal[j]$ дает точное количество элементов, ключи сортировки которых равны j, для $j = 0,1,\ldots,m-1$, можно использовать следующий инвариант цикла, чтобы показать, что по завершении работы процедуры Count-Keys-Less значение $less[j]$ равно количеству ключей сортировки, меньших j.

> В начале каждой итерации цикла на шаге 3 значение $less[j-1]$ равно количеству ключей сортировки, меньших $j-1$.

Расписать части инициализации, сохранения и завершения я предоставляю читателю. Можно легко увидеть, что процедура Count-Keys-Less выполняется за время $\Theta(m)$. И она, определенно, не выполняет ни одного сравнения ключей одного с другим.

Рассмотрим пример. Предположим, что $m = 7$, так что все ключи сортировки являются целыми числами в диапазоне от 0 до 6, и у нас есть следующий массив A с $n = 10$ элементами: $A = \langle 4,1,5,0,1,6,5,1,5,3 \rangle$. Тогда $equal = \langle 1,3,0,1,1,3,1 \rangle$ и $less = \langle 0,1,4,4,5,6,9 \rangle$. Поскольку $less[5] = 6$, а $equal[5] = 3$ (вспомните, что индексы массивов *less* и *equal* начинаются с 0, а не с 1), по окончании сортировки позиции с 1 по 6 должны содержать значения ключей, меньшие, чем 5, а в позициях 7, 8 и 9 должно содержаться значение ключа, равное 5.

Когда у нас есть массив *less*, мы можем создать отсортированный массив, хотя и не на месте.

Процедура Rearrange(A,less,n,m)

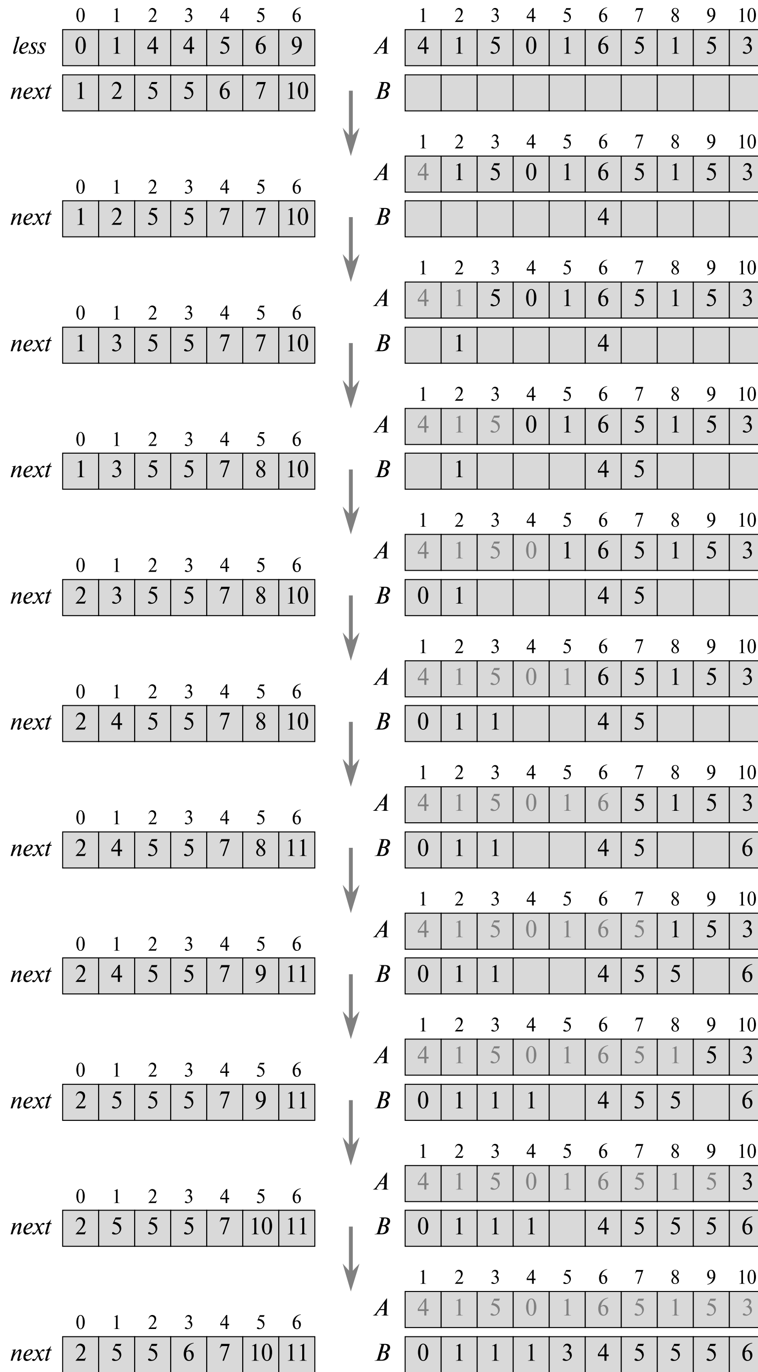
Вход:

- A: массив целых чисел в диапазоне от 0 до $m-1$.
- *less*: массив, возвращаемый процедурой Count-Keys-Less.
- n: количество элементов в массиве A.
- m: определяет диапазон значений элементов в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

1. Пусть $B[1..n]$ и $next[0..m-1]$ — новые массивы.
2. Для $j = 0$ до $m-1$:

   A. Установить $next[j]$ равным $less[j]+1$.



Algorithms_Unlocked.indb 74 31.10.2013 15:15:09

3. Для $i = 1$ до n:
   A. Установить значение *key* равным $A[i]$.
   B. Установить значение *index* равным $next[key]$.
   C. Установить $B[index]$ равным $A[i]$.
   D. Увеличить значение $next[key]$ на единицу.
4. Вернуть массив *B*.

Приведенный далее рисунок показывает, как процедура REARRANGE перемещает элементы из массива *A* в массив *B* так, чтобы они в конечном итоге оказались в массиве *B* в отсортированном порядке. В верхней части рисунка показаны массивы *less*, *next*, *A* и *B* перед первой итерацией цикла на шаге 3, а в последующих частях показаны массивы *next*, *A* и *B* после каждой очередной итерации. Серым цветом показаны элементы, скопированные в массив *B*.

Идея заключается в том, что при проходе по массиву *A* от начала до конца $next[j]$ указывает индекс элемента в массиве *B*, в который должен быть помещен очередной элемент массива *A* с ключом *j*. Вспомните, что если *l* элементов имеют ключи сортировки, меньшие, чем *x*, то *k* элементов с ключом сортировки, равным *x*, должны занимать позиции от $l+1$ до $l+k$. Цикл на шаге 2 выполняет установку значений массива *next* так, что изначально $next[j] = l+1$, где $l = less[j]$. Цикл на шаге 3 обходит массив *A* от начала до конца. Для каждого элемента $A[i]$ шаг 3A сохраняет значение $A[i]$ в переменной *key*, шаг 3B вычисляет значение *index*, которое представляет собой индекс в массиве *B*, где должно быть сохранено значение $A[i]$, а шаг 3C переносит $A[i]$ в эту позицию в массиве *B*. Поскольку следующий элемент в массиве *A*, который имеет тот же ключ сортировки, что и $A[i]$ (если таковой имеется), должен быть сохранен в следующей позиции массива *B*, шаг 3D увеличивает значение $next[key]$ на единицу.

Каково время работы процедуры REARRANGE? Цикл на шаге 2 выполняется за время $\Theta(m)$, а цикл на шаге 3 — за время $\Theta(n)$. Следовательно, процедура REARRANGE, как и процедура COUNT-KEYS-EQUAL, имеет время работы $\Theta(n+m)$, что равно $\Theta(n)$, если *m* является константой.

Теперь мы можем собрать все три процедуры вместе для создания процедуры ***сортировки подсчетом***.

Процедура COUNTING-SORT(*A*,*n*,*m*)

Вход:

- *A*: массив целых чисел в диапазоне от 0 до $m-1$.
- *n*: количество элементов в массиве *A*.
- *m*: определяет диапазон значений в массиве *A*.

Выход: массив *B*, содержащий элементы массива *A* в отсортированном порядке.

1. Вызвать процедуру COUNT-KEYS-EQUAL(*A*,*n*,*m*) и сохранить ее результат как массив *equal*.
2. Вызвать процедуру COUNT-KEYS-LESS(*equal*,*m*) и сохранить ее результат как массив *less*.
3. Вызвать процедуру REARRANGE(*A*,*less*,*n*,*m*) и сохранить ее результат как массив *B*.
4. Вернуть массив *B*.

Исходя из времени работы процедур COUNT-KEYS-EQUAL ($\Theta(n+m)$), COUNT-KEYS-LESS ($\Theta(m)$) и REARRANGE ($\Theta(n+m)$), можно сделать вывод, что процедура COUNTING-SORT выполняется за время $\Theta(n+m)$, или просто $\Theta(n)$, если m представляет собой константу. Сортировка подсчетом превосходит нижнюю границу $\Omega(n\lg n)$ сортировки сравнением, потому что она никогда не сравнивает ключи сортировки один с другим. Вместо этого она использует ключи сортировки для индексирования массивов, что вполне реально, когда ключи сортировки являются небольшими целыми значениями. Если ключи сортировки представляют собой действительные числа с дробной частью или, например, строки символов, то в таком случае использовать сортировку подсчетом нельзя.

Вы можете заметить, что процедура предполагает наличие в элементах только ключей сортировки без каких-либо сопутствующих данных. Да, я обещал, что в отличие от REALLY-SIMPLE-SORT процедура COUNTING-SORT допускает наличие сопутствующих данных. Этого легко добиться, достаточно только модифицировать шаг 3C процедуры REARRANGE так, чтобы он копировал весь элемент, а не только ключ сортировки. Вы могли также обратить внимание на то, что мои процедуры несколько неэффективно используют массивы. Да, массивы *equal*, *less* и *next* можно объединить в один массив, но эту задачу я оставляю в качестве задания читателю.

Я постоянно отмечал, что время работы равно $\Theta(n)$, если m является константой. Но когда m является константой? Один из примеров — сортировка студентов по уровню успеваемости. Например, уровень успеваемости может принимать значения от 0 до 10, но количество студентов при этом варьируется. Я вполне мог бы использовать сортировку подсчетом для n студентов за время $\Theta(n)$, так как значение $m=11$ (вспомните, что сортируемый диапазон значений — от 0 до $m-1$) является константой.

На практике однако сортировка подсчетом оказывается полезной в качестве части другого алгоритма сортировки — поразрядной сортировки. В дополнение к линейному времени работы при константном значении m сортировка подсчетом имеет еще одно важное свойство: она является ***устойчивой***. В случае устойчивой сортировки элементы с одним и тем же ключом сортировки оказываются в выходном массиве в том же порядке, что и во входном. Другими словами, устойчивая сортировка, встречая два элемента с равными ключами, разрешает неоднозначность, помещая в выходной массив первым тот элемент, который появляется первым во входном массиве. Понять, почему сортировка подсчетом является устойчивой, можно глядя на цикл на шаге 3 процедуры REARRANGE. Если два элемента A имеют один и тот же ключ сортировки, скажем, *key*, то процедура увеличивает $next[key]$ сразу же после переноса в массив B элемента, который ранее был в A. Таким

образом, к моменту перемещения элемента, который появляется в A позже, этот элемент будет помещаться в массив B в позицию с б*о*льшим индексом.

Поразрядная сортировка

Предположим, что вам нужно отсортировать строки символов некоторой фиксированной длины. Например сейчас я пишу этот раздел, сидя в самолете, и когда я делал заказ билета, мой код подтверждения был XI7FS6. Все коды подтверждения авиакомпании имеют вид строк из шести символов, каждый из которых является буквой или цифрой. Каждый символ может принимать 36 значений (26 букв плюс 10 цифр), так что всего имеется $36^6 = 2\,176\,782\,336$ возможных кодов подтверждения. Хотя это и константа, но она слишком велика, чтобы авиакомпания использовала для сортировки кодов сортировку подсчетом.Чтобы получить для каждого кода конкретное число, можно перевести каждый из 36 символов в числовой код, имеющий значение от 0 до 35. Код цифры является самой этой цифрой (так, код цифры 5 представляет собой число 5), а коды для букв начинаются с 10 для A и заканчиваются 35 для Z.

Теперь давайте немного упростим задачу и предположим, что каждый код подтверждения состоит только из двух символов (не беспокойтесь: мы вскоре вернемся к шести символам). Хотя можно воспользоваться сортировкой подсчетом с $m = 36^2 = 1296$, мы вместо этого используем ее *дважды* с $m = 36$. В первый раз в качестве ключа сортировки используем *правый* символ, а затем вновь отсортируем результат, но теперь в качестве ключа используем *левый* символ. Мы выбираем сортировку подсчетом, потому что она хорошо работает при относительно небольших m и потому что она устойчива.

Предположим, например, что у нас есть двухсимвольные коды <F6, E5, R6, X6, X2, T5, F2, T3>. После сортировки подсчетом по правому символу мы получаем коды в следующем порядке: <X2, F2, T3, E5, T5, F6, R6, X6>. Обратите внимание, что, поскольку сортировка подсчетом устойчива, а X2 в исходном порядке идет до F2, после сортировки по правому символу X2 продолжает находиться перед F2. Теперь отсортируем результат по левому символу, вновь используя сортировку подсчетом, и получим то, что хотели: <E5, F2, F6, R6, T3, T5, X2, X6>.

Что бы произошло, если бы мы сначала выполнили сортировку по левому символу? После сортировки подсчетом по левому символу мы бы получили <E5, F6, F2, R6, T5, T3, X6, X2>, а затем после сортировки подсчетом по правому символу был бы получен неверный окончательный результат <F2, X2, T3, E5, T5, F6, R6, X6>.

Почему работа справа налево приводит к правильному результату? Важное значение имеет использование устойчивой сортировки; это может быть сортировка подсчетом или любая иная, но устойчивая сортировка. Предположим, что мы работаем с символами в i-й позиции и что по всем $i-1$ позициям справа массив был отсортирован. Рассмотрим две любые сортировки ключей. Если они отличаются в i-й позиции, то их $i-1$ позиций справа значений не имеют: устойчивый алгоритм сортировки по i-й позиции разместит их в верном порядке. Если же, с другой стороны, они имеют один и тот же символ в i-й позиции,

то первым должен быть код, символ которого идет первым в $i-1$ позиции. Но применение метода устойчивой сортировки гарантирует получение именно этого результата.

Вернемся к нашим шестизначным кодам подтверждения и посмотрим, как будут отсортированы коды, которые изначально находились в порядке <XI7FS6, PL4ZQ2, JI8FR9, XL8FQ6, PY2ZR5, KV7WS9, JL2ZV3, KI4WR2>. Пронумеруем символы справа налево от 1 до 6. Тогда последовательность кодов после выполнения устойчивой сортировки по *i*-му символу имеет следующий вид.

i	Последовательность после сортировки
1	<PL4ZQ2, KI4WR2, JL2ZV3, PY2ZR5, XI7FS6, XL8FQ6, JI8FR9, KV7WS9>
2	<PL4ZQ2, XL8FQ6, KI4WR2, PY2ZR5, JI8FR9, XI7FS6, KV7WS9, JL2ZV3>
3	<XL8FQ6, JI8FR9, XI7FS6, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, JL2ZV3>
4	<PY2ZR5, JL2ZV3, KI4WR2, PL4ZQ2, XI7FS6, KV7WS9, XL8FQ6, JI8FR9>
5	<KI4WR2, XI7FS6, JI8FR9, JL2ZV3, PL4ZQ2, XL8FQ6, KV7WS9, PY2ZR5>
6	<JI8FR9, JL2ZV3, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, XI7FS6, XL8FQ6>

Обобщая, в алгоритме ***поразрядной сортировки*** мы предполагаем, что каждый ключ сортировки можно рассматривать как *d*-значное число, каждая цифра которого находится в диапазоне от 0 до $m-1$. Мы поочередно используем устойчивую сортировку для каждой цифры справа налево. Если в качестве устойчивой применяется сортировка подсчетом, то время сортировки по одной цифре составляет $\Theta(m+n)$, а время сортировки по всем *d* цифрам — $\Theta(d(m+n))$. Если *m* является константой (как в примере с кодами подтверждения $m=36$), то время работы поразрядной сортировки становится равным $\Theta(dn)$. Если *d* также представляет собой константу (например, 6 в случае кодов подтверждения), то время работы поразрядной сортировки превращается в просто $\Theta(n)$.

Когда поразрядная сортировка использует сортировку подсчетом для упорядочения по каждой цифре, она никогда не сравнивает два ключа сортировки один с другим. Она использует отдельные цифры для индексирования массивов в сортировке подсчетом. Вот почему поразрядная сортировка, как и сортировка подсчетом, преодолевает нижнюю границу $\Omega(n\lg n)$ сортировки сравнением.

Дальнейшее чтение

В главе 8 в CLRS [4] весь материал данной главы охвачен более подробно и широко.