Посвящается моей жене Норме.

# Оглавление

*Предисловие* 9

1 Наименьшее отсутствующее число 12

2 Превосходная задача 19

3 Улучшаем седловой поиск 24

4 Задача о выборке 35

5 Сортировка попарных сумм 42

6 Делаем сотню 49

7 Строим дерево минимальной высоты 58

8 Распутываем жадные алгоритмы 68

9 Поиск знаменитостей 75

10 Удаляем повторы 84

11 Вовсе не максимальная сумма сегмента 94

12 Ранжируем суффиксы 101

13 Преобразование Барроуза–Уилера 115

14 Последний хвост 128

15 Все общие префиксы 139

16 Алгоритм Бойера—Мура 145

17 Алгоритм Кнута—Морриса—Пратта 156

18 Планирование в «Час пик» 166

19 Простой алгоритм решения судоку 178

20 Задача «Обратного отсчёта» 189

21 Хиломорфизмы и нексусы 202

22 Три способа вычисления определителей 215

23 Внутри выпуклой оболочки 224

24 Рациональное арифметическое кодирование 236

25 Целочисленное арифметическое кодирование 247

26 Алгоритм Шора—Вейта 262

27 Упорядоченная вставка 274

28 Бесцикловые функциональные алгоритмы 287

29 Алгоритм Джонсона—Троттера 298

30 Прядение паутины для чайников 306

*Предметный указатель* 326

# Предисловие

В 1990 году, когда журнал *Journal of Functional Programming* (JFP) только планировался к выходу, бывшие тогда редакторами Саймон Пейтон Джонс (Simon Peyton Jones) и Филипп Вадлер (Philip Wadler) попросили меня взяться за постоянную колонку под названием *Функциональные жемчужины* (*Functional Pearls*). Их идея заключалась в подражании чрезвычайно успешной серии эссе Джона Бентли (Jon Bentley) «Жемчужины программирования», которые публиковались в 80-е годы в журнале *Communications of the ACM*. Вот что Бентли писал о своих жемчужинах:

> Как настоящие жемчужины вырастают из песчинок, раздражающих устриц, так и жемчужины программирования вырастают из реальных задач, мучающих программистов. Эти программы интересны, к тому же они учат важным программистским приёмам и фундаментальным принципам проектирования.

Мне кажется, что редакторы обратились именно ко мне, потому что я интересовался следующим подходом: я брал понятную, но неэффективную функциональную программу, выступавшую в роли спецификации к решаемой задаче, а затем посредством эквациональных рассуждений приходил к более эффективной версии. Одним из факторов, стимулирующих рост интереса к функциональным языкам в 90-е годы, было именно удобство применения эквациональных рассуждений. В самом деле, акроним в названии функционального языка GOFER, изобретённого Марком Джонсом (Mark Jones), отражает как раз эту идею. GOFER стал одним из языков, повлиявших на развитие языка Haskell, на котором основывается настоящая книга. Эквациональные рассуждения доминируют здесь над всем.

За последние 20 лет в JFP и изредка на таких конференциях как *International Conference of Functional Programming* (ICFP) и *Mathematics of Program Construction Conference* (MPC) появилось около 80 жемчужин. Из них я написал примерно четверть, большая же часть написана другими. Темы этих жемчужин включают интересные выводы программ, новые структуры данных, а также маленькие, но элегантные встроенные в Haskell и ML предметно-ориентированные языки для конкретных приложений.

Мне всегда были интересны алгоритмы и их проектирование. Отсюда название этой книги — *Жемчужины проектирования алгоритмов: функциональный подход* — вместо более общего *Функциональные жемчужины*. Многие, хотя и не все, жемчужины начинаются со спецификации на языке Haskell, а затем продолжаются выводом более эффективной реализации. При написании конкретно этих жемчужин мне хотелось выяснить, до какой степени проектирование алгоритма можно представить в виде традиционного в математике вычисления результата посредством применения общеизвестных принципов, теорем и законов. Хотя в математике вычисления обычно производятся для упрощения сложных выражений, в проектировании алгоритмов получается в точности наоборот: простые, но неэффективные программы преобразуются в более эффективные, но, возможно, неочевидные. Жемчужиной является не полученная в итоге программа, а скорее процесс её получения. Остальные жемчужины, в части из них совсем немного преобразований, посвящены попыткам дать простые объяснения некоторым интересным и довольно хитроумным алгоритмам. Объяснение идей, лежащих в основе алгоритма, при функциональном подходе оказывается гораздо проще, чем при императивном: составляющие алгоритм функции легче отделить, они коротки и отражают мощные вычислительные шаблоны.

Жемчужины в этой книге, появлявшиеся прежде в JFP и других местах, были неоднократно отшлифованы. Собственно, многие не слишком сильно напоминают оригиналы. Даже при этом их несложно отшлифовать ещё лучше. Золотым стандартом красоты в математике считается книга Айгнера (Aigner) и Циглера (Ziegler) Proofs from The Book (третье издание, Springer, 2003), содержащая совершенные доказательства математических теорем. Я всегда рассматриваю эту книгу как идеал, к которому следует стремиться.

Примерно треть жемчужин новые. С некоторыми явно обозначенными исключениями жемчужины можно читать в любом порядке, хотя главы были упорядочены до некоторой степени по темам, таким как «разделяй и властвуй», жадные алгоритмы, полный перебор и т.п. В жемчужинах присутствует небольшое повторение материала, по большей части отно-

сящегося к свойствам используемых библиотечных функций или к более общим законам, таким как законы слияния для разнообразных свёрток. При необходимости читатель может обратиться к добавленному к книге короткому предметному указателю.

Наконец, в этой книге собраны идеи многих людей. Некоторые жемчужины были изначально написаны в сотрудничестве с другими авторами. Я хотел бы поблагодарить Шэрон Кёртис (Sharon Curtis), Джереми Гиббонса (Jeremy Gibbons), Ральфа Хинзе (Ralf Hinze), Герэйнта Джонса (Geraint Jones) и Шин-Чень Му (Shin-Cheng Mu), моих соавторов, за щедрое разрешение переработать этот материал. Джереми Гиббонс прочитал окончательную версию черновика и сделал множество полезных предложений, направленных на улучшение изложения. Некоторые жемчужины досконально обсуждались на встречах исследовательской группы Algebra of Programming в Оксфорде. Хотя многие из недостатков и ошибок были исправлены, нет никаких сомнений в том, что при этом добавились новые. Помимо упомянутых ранее, я хотел бы выразить благодарность Стивену Дрейпу (Stephen Drape), Тому Харперу (Tom Harper), Дэниэлю Джеймсу (Daniel James), Джеффри Лейку (Jeffrey Lake), Менг Вонг (Meng Wang) и Николасу Ву (Nicholas Wu) за предложения по улучшению текста. Я также хотел бы поблагодарить Ламберта Мертенса (Lambert Meertens) и Уга де Мура (Oege de Moor) за плодотворное многолетнее сотрудничество. Наконец, я в долгу у Дэвида Транаха (David Tranah), моего редактора в издательстве Cambridge University Press, за содействие и поддержку, и в том числе за столь необходимые технические советы по подготовке окончательной версии.

Ричард Бёрд

# 1

# Наименьшее отсутствующее число

## Введение

Рассмотрим задачу отыскания наименьшего натурального числа, отсутствующего в заданном конечном множестве натуральных чисел $X$. Здесь мы имеем дело с упрощённой версией более общей программистской задачи, в которой числа соответствуют некоторым объектам, а $X$ — множество объектов, используемых в настоящий момент. Задача заключается в том, чтобы найти некоторый неиспользуемый объект, например, с наименьшим именем.

Разумеется, решение задачи зависит от способа представления множества $X$. Если $X$ задано списком без повторений, где элементы упорядочены в порядке возрастания, то решение очевидно: в последовательности элементов следует искать первый пропуск. Предположим, однако, что множество $X$ задано списком различных чисел в произвольном порядке, например:

$$[08, 23, 09, 00, 12, 11, 01, 10, 13, 07, 41, 04, 14, 21, 05, 17, 03, 19]$$

Как бы вы стали искать наименьшее число, отсутствующее в этом списке?

Не сразу становится очевидным, что имеется решение линейной сложности, ведь невозможно выполнить сортировку произвольного числового списка за линейное время. Тем не менее, такое решение существует, и целью этой жемчужины будет описание двух возможных стратегий: одна из них основана на использовании массивов языка Haskell, а вторая на методе «разделяй и властвуй».

## Решение с использованием массива

Определим спецификацию задачи с помощью функции *minfree*:

$$\begin{array}{lcl} \mathit{minfree} & :: & [\mathit{Nat}] \to \mathit{Nat} \\ \mathit{minfree}\ \mathit{xs} & = & \mathit{head}\ ([0\,..\,] \setminus\setminus \mathit{xs}) \end{array}$$

Выражение $us \setminus\setminus vs$ обозначает список тех элементов $us$, которые остаются после удаления всех элементов $vs$:

$$\begin{array}{lcl} (\setminus\setminus) & :: & \mathit{Eq}\ a \Rightarrow [a] \to [a] \to [a] \\ \mathit{us} \setminus\setminus \mathit{vs} & = & \mathit{filter}\ (\notin \mathit{vs})\ \mathit{us} \end{array}$$

Хотя функция *minfree* и работает, для списка длины $n$ она требует в худшем случае $\Theta(n^2)$ операций. К примеру, чтобы получить результат вызова $\mathit{minfree}\,[n-1, n-2\,..\,0]$ понадобится вычислить $i \notin [n-1, n-2\,..\,0]$, где $0 \leqslant i \leqslant n$, поэтому в конечном итоге получится $n(n+1)/2$ проверок на равенство.

Ключевой факт для обеих стратегий решения, с массивом и по методу «разделяй и властвуй», заключается в том, что в списке $xs$ содержатся не все элементы из промежутка $[0\,..\,\mathit{length}\ \mathit{xs}]$. Поэтому наименьшее число, отсутствующее в $xs$, является одновременно наименьшим числом, отсутствующим в $\mathit{filter}\ (\leqslant n)\ \mathit{xs}$, где $n = \mathit{length}\ \mathit{xs}$. Решение, основанное на массиве, использует этот факт для построения контрольного массива чисел, находящихся в $\mathit{filter}\ (\leqslant n)\ \mathit{xs}$. Контрольный массив с индексами от 0 до $n$ содержит $n+1$ логическое значение, инициализированное *False*. Для каждого элемента $x$ из $xs$, такого, что $x \leqslant n$, элемент контрольного массива в позиции $x$ устанавливается равным *True*. Наименьшее отсутствующее число находится после этого по первой же позиции, равной *False*. Таким образом, $\mathit{minfree} = \mathit{search} \cdot \mathit{checklist}$, где

$$\begin{array}{lcl} \mathit{search} & :: & \mathit{Array\ Int\ Bool} \to \mathit{Int} \\ \mathit{search} & = & \mathit{length} \cdot \mathit{takeWhile\ id} \cdot \mathit{elems} \end{array}$$

Функция *search* принимает на вход массив логических значений, преобразует массив в список и возвращает длину наибольшего начального сегмента, содержащего только истинные элементы. Полученная длина и будет индексом первого вхождения значения *False*.

Одним из возможных вариантов определения функции *checklist* с линейной сложностью является использование функции *accumArray* из модуля *Data.Array* стандартной библиотеки языка Haskell. Тип этой функции несколько пугающий:

$$Ix\ i \Rightarrow (e \to v \to e) \to e \to (i,i) \to [(i,v)] \to Array\ i\ e$$

Ограничение $Ix\,i$ требует, чтобы типовая переменная $i$ принадлежала классу *Index*, например, *Int* или *Char*, она используется для обозначения индексов или позиций в массиве. Первый аргумент это «суммирующая» функция, она перевычисляет элемент массива (типа $e$) на основе некоторого значения (типа $v$). Второй аргумент определяет начальное значение для элемента массива в очередной позиции. Третий аргумент это пара из начального и конечного индексов. Наконец, четвёртый аргумент это ассоциативный список пар индекс–значение. Функция *accumArray* строит массив, обрабатывая ассоциативный список слева направо, при этом она меняет определяемые очередным индексом элементы массива, присваивая им результаты вызова суммирующей функции для прежнего элемента и очередного значения из ассоциативного списка. Этот процесс выполняется за линейное по длине ассоциативного списка время в предположении, что суммированию достаточно константного.

Теперь функцию *checklist* можно реализовать как вызов функции *accumArray*:

$$\begin{array}{lcl} checklist & :: & [Int] \to Array\ Int\ Bool \\ checklist\ xs & = & accumArray\ (\vee)\ False\ (0,n) \\ & & \quad (zip\ (filter\ (\leqslant n)\ xs)\ (repeat\ True)) \\ & & \quad \textbf{where}\ n\ =\ length\ xs \end{array}$$

Эта реализация не требует, чтобы элементы списка *xs* не повторялись. Единственное ограничение в том, чтобы они были натуральными числами.

Интересно, что функцией *accumArray* можно воспользоваться для сортировки числового списка за линейное время при условии, что все его элементы принадлежат ограниченному диапазону $(0,n)$. Заменим *checklist* на *countlist*, где

$$\begin{array}{lcl} countlist & :: & [Int] \to Array\ Int\ Int \\ countlist\ xs & = & accumArray\ (+)\ (0,n)\ (zip\ xs\ (repeat\ 1)) \end{array}$$

Теперь $sort\ xs\ =\ concat\ [replicate\ k\ x \mid (x,k) \leftarrow countlist\ xs]$. В сущности, применяя *countlist* вместо *checklist*, функцию *minfree* можно определить как позицию первого вхождения нулевого элемента.

Приведённая выше реализация строит массив за один проход, пользуясь умной библиотечной функцией. Более прозаичный способ реализовать функцию *checklist* состоит в явной отметке всех вхождений шаг за шагом с

применением операции присваивания с константной сложностью. В языке Haskell это возможно, только если обработка массива выполняется в подходящей монаде, например, в монаде с состоянием. В следующем примере реализация *checklist* использует модуль *Data.Array.ST*:

```
checklist xs =
  runSTArray (do
    { a ← newArray (0, n) False;
      sequence [writeArray a x True | x ← xs, x ⩽ n];
      return a})
  where n = length xs
```

Впрочем, такое решение не удовлетворит чисто функционального программиста, поскольку оно эксплуатирует традиционную императивную парадигму, хотя и в функциональных одеждах.

## Решение по методу «разделяй и властвуй»

Обратимся теперь к стратегии «разделяй и властвуй». Идея в том, чтобы выразить $minfree\ (xs +\!\!+ ys)$ в терминах $minfree\ xs$ и $minfree\ ys$. Запишем некоторые свойства операции $\setminus\!\setminus$:

$$\begin{aligned}(as +\!\!+ bs) \setminus\!\setminus cs &= (as \setminus\!\setminus cs) +\!\!+ (bs \setminus\!\setminus cs)\\ as \setminus\!\setminus (bs +\!\!+ cs) &= (as \setminus\!\setminus bs) \setminus\!\setminus cs\\ (as \setminus\!\setminus bs) \setminus\!\setminus cs &= (as \setminus\!\setminus cs) \setminus\!\setminus bs\end{aligned}$$

Эти свойства аналогичны соответствующим свойствам теоретико-множественных операций, в которых объединение множеств $\bigcup$ заменяется на $+\!\!+$, а разность множеств $\setminus$ на $\setminus\!\setminus$. Предположим теперь, что $as$ и $vs$ не пересекаются, т.е. $as \setminus\!\setminus vs = as$, и что $bs$ и $vs$ также не пересекаются, т.е. $bs \setminus\!\setminus us = bs$. Из указанных свойств операций $+\!\!+$ и $\setminus\!\setminus$ следует, что

$$(as +\!\!+ bs) \setminus\!\setminus (us +\!\!+ vs) = (as \setminus\!\setminus us) +\!\!+ (bs \setminus\!\setminus vs)$$

Выберем теперь некоторое натуральное число $b$ и положим $as = [0\,..\,b-1]$ и $bs = [b\,..]$. Пусть, далее, $us = filter\ (<b)\ xs$ и $vs = filter\ (\geqslant b)\ xs$. Тогда $as$ и $vs$ окажутся непересекающимися, а значит, такими же будут $bs$ и $us$. Следовательно,

$$\begin{aligned}[0\,..] \setminus\!\setminus xs = {} & ([0\,..\,b-1] \setminus\!\setminus us) +\!\!+ ([b\,..] \setminus\!\setminus vs)\\ & \textbf{where}\ (us, vs) = partition\ (<b)\ xs\end{aligned}$$

Haskell обеспечивает эффективную реализацию функции *partition*, которая разбивает список на те элементы, которые удовлетворяют предикату $p$, и те, которые ему не удовлетворяют. Поскольку

$$head\ (xs +\!\!+ ys)\ =\ \textbf{if}\ null\ xs\ \textbf{then}\ head\ ys\ \textbf{else}\ head\ xs$$

получаем, по-прежнему для любого натурального числа $b$, что

$$\begin{array}{lcl} minfree\ xs & = & \textbf{if}\ null\ ([0\,..\,b-1] \setminus\!\setminus us) \\ & & \quad \textbf{then}\ head\ ([b\,..] \setminus\!\setminus vs) \\ & & \quad \textbf{else}\ head\ ([0\,..] \setminus\!\setminus us) \\ & & \textbf{where}\ (us, vs)\ =\ partition\ (<b)\ xs \end{array}$$

Следующий вопрос: можно ли реализовать проверку $null\ ([0\,..\,b-1] \setminus\!\setminus us)$ более эффективно, чем прямым вычислением, которое требует квадратичного времени по длине $us$? Да, так как на входе список неповторяющихся натуральных чисел, таковым же является $us$, причём каждый элемент $us$ меньше $b$. Поэтому

$$null\ ([0\,..\,b-1] \setminus\!\setminus us)\ \equiv\ length\ us == b$$

Заметим, что предыдущее решение не зависело от предположения, что данный список не содержит дубликатов, однако оно оказывается ключевым для эффективной реализации по методу «разделяй и властвуй».

Дальнейшее изучение кода *minfree* подсказывает, что следует обобщить *minfree* до функции, скажем, *minfrom*, которая определена так:

$$\begin{array}{lcl} minfrom & :: & Nat \to [Nat] \to Nat \\ minfrom\ a\ xs & = & head\ ([a\,..] \setminus\!\setminus xs) \end{array}$$

где предполагается, что каждый элемент $x$ больше или равен $a$. Тогда, при условии, что $b$ выбрано таким образом, чтобы длины $us$ и $vs$ были меньше длины $xs$, следующее рекурсивное определение *minfree* оказывается вполне обоснованным:

$$\begin{array}{lcl} minfree\ xs & = & minfrom\ 0\ xs \\ minfrom\ a\ xs & | & null\ xs \qquad\qquad = \ a \\ & | & length\ us == b - a\ =\ minfrom\ b\ vs \\ & | & \textbf{otherwise} \qquad\quad = \ minfrom\ a\ us \\ & & \textbf{where}\ (us, vs)\ =\ partition\ (<b)\ xs \end{array}$$

Остаётся выбрать $b$. Ясно, что нам нужно $b > a$. К тому же, хотелось бы иметь $b$ таким, чтобы максимум из длин $us$ и $vs$ был настолько малым,

насколько это возможно. Правильный выбор $b$, удовлетворяющего указанным требованиям, таков:

$$b \;=\; a + 1 + n \operatorname{div} 2$$

где $n \;=\; \mathit{length}\ \mathit{xs}$. Если $n \neq 0$ и $\mathit{length}\ \mathit{us} < b - a$, то

$$\mathit{length}\ \mathit{us} \leqslant n \operatorname{div} 2 < n$$

И, если $\mathit{length}\ \mathit{us} \;=\; b - a$, то

$$\mathit{length}\ \mathit{vs} \;=\; n - n \operatorname{div} 2 - 1 \leqslant n \operatorname{div} 2$$

При таком выборе параметра $b$ число операций $T(n)$, необходимое для вычисления $\mathit{minfrom}\ 0\ \mathit{xs}$, где $n \;=\; \mathit{length}\ \mathit{xs}$, удовлетворяет рекуррентному соотношению $T(n) = T(n \operatorname{div} 2) + \Theta(n)$, следовательно, $T(n) = \Theta(n)$.

В качестве последней оптимизации можно избежать постоянного перевычисления длины, уточнив представление данных, а именно, заменив $\mathit{xs}$ на пару $(\mathit{length}\ \mathit{xs}, \mathit{xs})$. Это приводит нас к окончательному варианту программы

$$\begin{array}{lcl}
\mathit{minfree}\ \mathit{xs} & = & \mathit{minfrom}\ 0\ (\mathit{length}\ \mathit{xs}, \mathit{xs}) \\
\mathit{minfrom}\ a\ (n, \mathit{xs}) & \mid & n == 0 \quad = \quad a \\
& \mid & m == b - a \;=\; \mathit{minfrom}\ b\ (n - m, \mathit{vs}) \\
& \mid & \textbf{otherwise} \;=\; \mathit{minfrom}\ a\ (m, \mathit{us}) \\
& & \textbf{where}\ (\mathit{us}, \mathit{vs}) \;=\; \mathit{partition}\ (<b)\ \mathit{xs} \\
& & \qquad\quad b \;=\; a + 1 + n \operatorname{div} 2 \\
& & \qquad\quad m \;=\; \mathit{length}\ \mathit{us}
\end{array}$$

Выясняется, что вышеприведённая программа примерно в два раза быстрее инкрементной программы на основе массива и на 20% быстрее, чем программа с использованием *accumArray*.

## Заключительные замечания

Это была простая задача с как минимум двумя простыми решениями. Второе решение основывалось на известном методе проектирования алгоритмов, стратегии «разделяй и властвуй». Идея разбиения списка на те элементы, которые меньше заданного значения, и все остальные возникает во многих алгоритмах, например, в быстрой сортировке Хоара (Quicksort). При поиске алгоритма со сложностью $\Theta(n)$, использующего список из $n$

элементов, довольно заманчиво сразу обратиться к методу, обрабатывающему каждый элемент списка за константное или хотя бы амортизированное константное время. Однако рекурсивный процесс, который делает $\Theta(n)$ операций для сведения задачи к той же задаче не более чем половинного размера, также достаточно хорош.

Одно из различий между проектировщиками чисто функциональных и императивных алгоритмов состоит в том, что первые не предполагают существования массивов с операцией изменения за константное время, по крайней мере без некоторого объёма подготовительной работы. Для чисто функционального программиста операция изменения массива требует логарифмического по размеру массива времени [1]. Это объясняет, почему иногда заметен логарифмический разрыв между функциональным и императивным решениями задачи. Но иногда, как здесь, этот разрыв при ближайшем рассмотрении исчезает.

[1] По правде говоря, программисты-императивщики отлично знают, что константное время индексирования и изменения возможно только для маленьких массивов.

# 2

# Превосходная задача

## Введение

В этой жемчужине мы будем выполнять маленькое упражнение по программированию от Мартина Рема (Rem, 1988a). В то время как решение Рема использует бинарный поиск, наше будет ещё одним применением метода «разделяй и властвуй». Говорят, что некоторый элемент массива *превосходит* данный, если он больше и расположен правее, т.е. $x[j]$ превосходит $x[i]$, если $i < j$ и $x[i] < x[j]$. *Числом превосходства* (*surpasser count*) элемента массива называют количество элементов, его превосходящих. Например, вот числа превосходства для букв слова ТЕЛЕГРАФИСТ:

| Т | Е | Л | Е | Г | Р | А | Ф | И | С | Т |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 4 | 5 | 5 | 3 | 4 | 0 | 2 | 1 | 0 |

Наибольшее число превосходства равно шести. Первое вхождение буквы Е имеет шесть превосходящих её элементов: буквы Л, Р, Ф, И, С и Т. Задача Рема заключается в вычислении наибольшего числа превосходства для массива длины $n > 1$ с помощью алгоритма со сложностью $O(n \log n)$.

## Спецификация

Будем предполагать, что на входе вместо массива имеется список. Функция *msc* (сокращение от maximum surpasser count) может быть специфицирована следующим образом:

$$
\begin{array}{lcl}
msc & :: & Ord\ a \Rightarrow [a] \to Int \\
msc\ xs & = & maximum\ [scount\ z\ zs \mid z : zs \leftarrow tails\ xs] \\
scount\ x\ xs & = & length\ (filter\ (x<)\ xs)
\end{array}
$$

Значение $scount\,x\,xs$ это число превосходства элемента $x$ относительно списка $xs$, функция $tails$ возвращает непустые хвосты непустого списка в порядке убывания их длин [1]:

$$
\begin{array}{lcl}
tails\ [\,] & = & [\,] \\
tails\ (x : xs) & = & (x : xs) : tails\ xs
\end{array}
$$

Данное выше определение функции $msc$ работает, но требует квадратичного времени.

## Разделяй и властвуй

Учитывая заданную сложность $O(n \log n)$, кажется разумным обратиться к алгоритму по методу «разделяй и властвуй». Если нам удастся найти такую функцию $join$, что

$$msc\ (xs +\!\!+\ ys) \ = \ join\ (msc\ xs)\ (msc\ ys)$$

и которую можно вычислить за линейное время, то временная сложность $T(n)$ алгоритма «разделяй и властвуй» на списке длины $n$ будет удовлетворять соотношению $T(n) = 2T(n/2) + O(n)$, а значит, $T(n) = O(n \log n)$. Однако довольно-таки очевидно, что такой функции $join$ не существует: единственное число $msc\ xs$ предоставляет слишком мало информации для любого подобного разбиения.

Наименьшим обобщением будет начать с таблицы всех чисел превосходства:

$$table\ xs \ = \ [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

Тогда $msc = maximum \cdot map\ snd \cdot table$. Теперь посмотрим, сможем ли мы найти линейную $join$, удовлетворяющую следующему:

$$table\ (xs +\!\!+\ ys) \ = \ join\ (table\ xs)\ (table\ ys)$$

Нам понадобится следующее свойство «разделяй и властвуй» для $tails$:

[1] В отличие от стандартной функции языка Haskell с тем же именем, которая возвращает возможно пустые хвосты возможно пустого списка.

$$tails\ (xs +\!\!+ ys)\ =\ map\ (+\!\!+ys)\ (tails\ xs) +\!\!+ tails\ ys$$

Проведём вычисления:

$$\begin{aligned}
&table\ (xs +\!\!+ ys)\\
=\quad &\{\ \text{определение}\ \}\\
&[(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ (xs +\!\!+ ys)]\\
=\quad &\{\ \text{свойство «разделяй и властвуй» для } tails\ \}\\
&[(z, scount\ z\ zs) \mid z : zs \leftarrow map\ (+\!\!+ys)\ (tails\ xs) +\!\!+ tails\ ys]\\
=\quad &\{\ \text{дистрибутивный закон для } \leftarrow \text{ по } +\!\!+\ \}\\
&[(z, scount\ z\ (zs +\!\!+ ys)) \mid z : zs \leftarrow tails\ xs] +\!\!+\\
&[(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys])\\
=\quad &\{\ \text{так как } scount\ z\ (zs +\!\!+ ys)\ =\ scount\ z\ zs + scount\ z\ ys\ \}\\
&[(z, scount\ z\ zs + scount\ z\ ys) \mid z : zs \leftarrow tails\ xs] +\!\!+\\
&[(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ ys])\\
=\quad &\{\ \text{определение функции } table \text{ и } ys\ =\ map\ fst\ (table\ ys)\ \}\\
&[(z, c + scount\ z\ (map\ fst\ (table\ ys))) \mid (z, c) \leftarrow table\ xs] +\!\!+ table\ ys
\end{aligned}$$

Следовательно, функцию *join* можно определить так:

$$\begin{aligned}
join\ txs\ tys\ &=\ [(z, c + tcount\ z\ tys) \mid (z, c) \leftarrow txs] +\!\!+ tys\\
tcount\ z\ tys\ &=\ scount\ z\ (map\ fst\ tys)
\end{aligned}$$

Однако проблема этого определения в том, что для вычисления *join txs tys* недостаточно линейного времени по длине *txs* и *tys*. Вычисление *tcount* можно было бы ускорить, если бы список *tys* был упорядочен по возрастанию первого компонента пары. В этом случае можно рассуждать так:

$$\begin{aligned}
&tcount\ z\ tys\\
=\quad &\{\ \text{определение } tcount \text{ и } scount\ \}\\
&length\ (filter\ (z<)\ (map\ fst\ tys))\\
=\quad &\{\ \text{так как } filter\ p \cdot map\ f\ =\ map\ f \cdot filter\ (p \cdot f)\ \}\\
&length\ (map\ fst\ (filter\ ((z<) \cdot fst)\ tys))\\
=\quad &\{\ \text{так как } length \cdot map\ f\ =\ length\ \}\\
&length\ (filter\ ((z<) \cdot fst)\ tys)\\
=\quad &\{\ \text{так как } tys \text{ упорядочен по первому компоненту}\ \}\\
&length\ (dropWhile\ ((z \geqslant) \cdot fst)\ tys)
\end{aligned}$$

Таким образом,

$$tcount\ z\ tys\ =\ length\ (dropWhile\ ((z \geqslant) \cdot fst)\ tys) \tag{2.1}$$

Это вычисление подсказывает, что было бы полезно поддерживать *table* в порядке возрастания первого компонента:

$$table\ xs\ =\ sort\ [(z, scount\ z\ zs) \mid z : zs \leftarrow tails\ xs]$$

Повторяя приведённое выше вычисление для упорядоченной версии *table*, получим, что

$$join\ txs\ tys\ =\ [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs] \mathbin{\Lambda\!\!\!\backslash} tys \tag{2.2}$$

где $\Lambda\!\!\!\backslash$ сливает два упорядоченных списка. Пользуясь этим определением, можно получить более эффективное рекурсивное определение *join*. Один из базовых случаев, $join\ [\,]\ tys\ =\ tys$, очевиден. Другой, а именно, $join\ txs\ [\,]\ =\ txs$, следует из того, что $tcount\ x\ [\,]\ =\ 0$. Рекурсивную часть можно упростить, сравнивая $x$ и $y$:

$$join\ txs@((x, c) : txs')\ tys@((y, d) : tys') \tag{2.3}$$

В языке Haskell символ @ вводит синоним, так что *txs* это синоним для $(x, c) : txs'$, аналогично для *tys*. Используя (2.2), выражение (2.3) можно свести к

$$((x, c + tcount\ x\ tys) : [(x, c + tcount\ x\ tys) \mid (x, c) \leftarrow txs']) \mathbin{\Lambda\!\!\!\backslash} tys$$

Чтобы узнать, какой элемент будет выдан операцией $\Lambda\!\!\!\backslash$ первым, нужно сравнить $x$ и $y$. Если $x < y$, то это элемент слева и, поскольку согласно (2.1) $tcount\ x\ tys\ =\ length\ tys$, выражение (2.3) сводится к

$$(x, c + length\ tys) : join\ txs'\ tys$$

Если $x = y$, необходимо сравнить $c + tcount\ x\ tys$ и $d$. Но $d\ =\ tcount\ x\ tys'$ по определению *table*, а $tcount\ x\ tys\ =\ tcount\ x\ tys'$ согласно (2.1). Поэтому (2.3) сводится к $(y, d) : join\ txs\ tys'$. Такой же результат будет получен и в оставшемся случае $x > y$.

Собирая всё вместе и добавляя к *join* во избежание перевычисления длины дополнительный аргумент $length\ tys$, приходим к следующему алгоритму вычисления *table* на основе метода «разделяй и властвуй»:

$$\begin{array}{ll} table\ [x] & =\ [(x, 0)] \\ table\ xs & =\ join\ (m - n)\ (table\ ys)\ (table\ zs) \end{array}$$

$$
\begin{array}{lcl}
\textbf{where}\ m & = & \mathit{length}\ \mathit{xs} \\
n & = & m \operatorname{div} 2 \\
(\mathit{ys}, \mathit{zs}) & = & \mathit{splitAt}\ n\ \mathit{xs}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{join}\ 0\ \mathit{txs}\ [\,] & = & \mathit{txs} \\
\mathit{join}\ n\ [\,]\ \mathit{tys} & = & \mathit{tys} \\
\mathit{join}\ n\ \mathit{txs}@((x, c) : \mathit{txs}')\ \mathit{tys}@((y, d) : \mathit{tys}') & & \\
\qquad \mid\ x < y & = & (x, c + n) : \mathit{join}\ n\ \mathit{txs}'\ \mathit{tys} \\
\qquad \mid\ x \geqslant y & = & (y, d) : \mathit{join}\ (n - 1)\ \mathit{txs}\ \mathit{tys}'
\end{array}
$$

Так как *join* выполняется за линейное время, список *table* вычисляется за $O(n \log n)$ операций, а значит такова же сложность *msc*.

## Заключительные замечания

Невозможно вычислить *table* с помощью алгоритма, более быстрого, чем $O(\log n)$. Причина в том, что если *xs* это список без повторяющихся элементов, то *table xs* предоставляет достаточно информации для определения такой сортирующей перестановки списка *xs*. Более того, никакие последующие сравнения между элементами списка не требуются. Фактически *table xs* соотносится с таблицей инверсий для перестановки из $n$ элементов; см. Кнута (Knuth, 1998): *table xs* это просто таблица инверсий для *reverse xs*. Так как основанная на сравнениях сортировка $n$ элементов требует $\Omega(n \log n)$ операций, то столько же требует и вычисление *table*.

Как было сказано во введении, решение Рема (Rem, 1988b) отличается тем, что оно использует итеративный алгоритм с применением бинарного поиска. Программист-императивщик также мог бы обратиться к алгоритму «разделяй и властвуй», но он, вероятно, предпочёл бы алгоритм с обработкой массива просто потому, что он требует меньше памяти.

## Литература

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, second edition. Reading, MA: Addison-Wesley. [*Имеется русский перевод:* Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 2-е изд. М.: Вильямс. 2007.]

Rem, M. (1988a). Small programming exercises 20. *Science of Computer Programming* **10** (1), 99–105.

Rem, M. (1988b). Small programming exercises 21. *Science of Computer Programming* **10** (3), 319–25.