Функциональное программирование

Ричард Бёрд

Жемчужины проектирования алгоритмов

Функциональный подход

С примерами на языке Haskell

ДМК издательство

УДК 004.021+004.421
ББК 32.973-018
Б11

Б11 **Ричард Бёрд**
Жемчужины проектирования алгоритмов: функциональный подход / Пер. с англ. В. Н. Брагилевского и А. М. Пеленицына. – М.: ДМК Пресс, 2013. — 330 с.: ил.
ISBN 978-5-94074-867-0

В этой книге Ричард Бёрд представляет принципиально новый подход к проектированию алгоритмов, а именно проектирование посредством формального вывода. Основное содержание книги разделено на 30 коротких глав, называемых жемчужинами, в каждой из которых решается конкретная программистская задача. Эти задачи, некоторые из них абсолютно новые, происходят из таких разнообразных источников, как игры и головоломки, захватывающие комбинаторные построения и более традиционные алгоритмы сжатия данных и сопоставления строк.

Каждая жемчужина начинается с постановки задачи, формулируемой на функциональном языке программирования Haskell, чрезвычайно мощном и в то же время лаконичном, позволяющем легко и просто выражать алгоритмические идеи. Новшество книги состоит в том, что каждое решение формально вычисляется из исходной постановки задачи посредством обращения к законам функционального программирования.

Издание предназначено для программистов, увлекающихся функциональным программированием, студентов, аспирантов и преподавателей, интересующихся принципами проектирования алгоритмов, а также всех, кто желает приобрести и развить навыки рассуждений в эквациональном стиле применительно к программам и алгоритмам.

УДК 004.021+004.421
ББК 32.973-018

ISBN 978-0-521-51338-8 (англ.)
ISBN 978-5-94074-867-0 (рус.)

Оглавление

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 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

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

где $/\!\!\setminus\!\!\setminus$ 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

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

Чтобы узнать, какой элемент будет выдан операцией $/\!\!\setminus\!\!\setminus$ 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»:

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

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

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

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.

3

Улучшаем седловой поиск

Действие происходит на занятии по проектированию функциональных алгоритмов. В классе четверо студентов: Анна, Иван, Мария и Фёдор.

Учитель. Доброе утро, ребята. Сегодня я бы хотел, чтобы вы спроектировали функцию *invert* с двумя параметрами: функция f, действующая из множества пар натуральных чисел во множество натуральных чисел, и натуральное число z. Значение $invert\ f\ z$ должно быть списком всех пар (x, y), удовлетворяющих равенству $f(x, y) = z$. Вы можете предполагать только то, что f строго возрастает по каждому из аргументов, и ничего иного.

Иван. Кажется, это простая задача. Так как f действует на натуральных числах и возрастает по каждому из аргументов, то мы знаем, что из равенства $f(x, y) = x$ следует, что $x \leqslant z$ и $y \leqslant z$. Поэтому можно определить *invert* простым поиском всех возможных пар значений:

$$invert\ f\ z = [(x, y) \mid x \leftarrow [0\,..\,z], y \leftarrow [0\,..\,z], f(x, y) == z]$$

Разве это не решение?

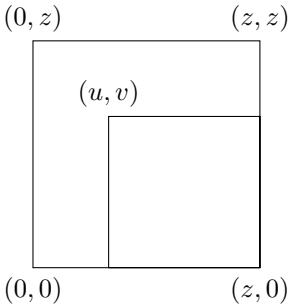
Учитель. Да, это решение, но оно требует $(z + 1)^2$ вычислений значения функции f. Поскольку вычисление значения f может оказаться очень долгим, я бы хотел увидеть решение, требующее как можно меньшее количество вызовов f.

Фёдор. Ну, нетрудно уменьшить количество вычислений вдвое. Так как при условии, что f возрастает, $f(x,y) \geqslant x+y$, то поиск можно проводить только по тем значениям, которые расположены на или ниже диагонали квадрата:

$$\mathit{invert}\, f\, z \;=\; [(x,y) \mid x \leftarrow [0\,..\,z], y \leftarrow [0\,..\,z-x], f(x,y) == z]$$

А если задуматься, то обе верхние границы можно вообще заменить на $z - f(0,0)$ и $z - x - f(0,0)$. Тогда, если $z < f(0,0)$, то поиск сразу останавливается.

Анна. В предположении, что порядок найденных решений не важен, я думаю, результат можно ещё улучшить. Метод Ивана выполняет поиск от начала координат в левом нижнем углу квадрата размера $z+1$, обходя его столбец за столбцом. Мы же можем ускориться, если начнём в левом верхнем углу квадрата в точке $(0,z)$. На каждом этапе область поиска ограничена прямоугольником с левым верхним углом в точке (u,v) и правым нижним углом в точке $(z,0)$. Вот картинка:



Давайте определим функцию

$$\mathit{find}\, (u,v)\, f\, z \;=\; [(x,y) \mid x \leftarrow [u\,..\,z], y \leftarrow [v, v-1\,..\,0], f(x,y) == z]$$

Таким образом, $\mathit{invert}\, f\, z \;=\; \mathit{find}\, (0,z)\, f\, z$. Теперь нетрудно найти более эффективную реализацию функции *find*.

Во-первых, если $u > z$ или $v < 0$, то, очевидно, $\mathit{find}\, (u,v)\, f\, z \;=\; [\,]$. В противном случае будем рассматривать различные возможные значения $f(u,v)$. Если $f(u,v) < z$, то остаток столбца u можно отбросить, так как при $v' < v$ имеет место неравенство $f(u,v') < f(u,v) < z$. Если $f(u,v) > z$, то аналогично отбрасываем остаток строки v. Наконец, если $f(u,v) = z$, то можно запомнить (u,v) и отбросить как столбец u, так и строку v.

Вот улучшенная версия функции *invert*:

$$
\begin{array}{l}
\mathit{invert}\ f\ z\ =\ \mathit{find}\ (0,z)\ f\ z \\
\mathit{find}\ (u,v)\ f\ z \\
\quad \mid u > z \vee v < 0\ =\ [\,] \\
\quad \mid z' < z\ =\ \mathit{find}\ (u+1,v)\ f\ z \\
\quad \mid z' == z\ =\ (u,v) : \mathit{find}\ (u+1,v-1)\ f\ z \\
\quad \mid z' > z\ =\ \mathit{find}\ (u,v-1)\ f\ z \\
\quad\ \ \textbf{where}\ z'\ =\ f(u,v)
\end{array}
$$

В худшем случае, если *find* обходит периметр квадрата от левого верхнего угла до правого нижнего, она делает $2z+1$ вызов функции f. В лучшем случае, если *find* идёт напрямую либо к нижней, либо к правой границе, ей требуется лишь $z+1$ вызов.

Фёдор. Ты можешь сократить область поиска ещё сильнее, потому что исходный квадрат с левым верхним углом $(0,z)$ и правым нижним углом $(z,0)$ является слишком избыточной оценкой области нахождения требуемого значения. Предположим, что мы предварительно вычислили такие m и n, что

$$
\begin{array}{lcl}
m & = & \mathit{maximum}\ (\mathit{filter}\ (\lambda y \to f(0,y) \leqslant z)\ [0\,..\,z]) \\
n & = & \mathit{maximum}\ (\mathit{filter}\ (\lambda x \to f(x,0) \leqslant z)\ [0\,..\,z])
\end{array}
$$

Тогда можно определить $\mathit{invert} f\, z = \mathit{find}(0,m) f\, z$, где *find* имеет в точности ту же форму, в какой её определила Анна, за тем исключением, что первое охранное выражение становится таким: $u > n \vee v < 0$. Другими словами, вместо того, чтобы анализировать весь квадрат $(z+1) \times (z+1)$, можно ограничиться квадратом размера $(m+1) \times (n+1)$.

Решающее соображение состоит в том, что m и n можно вычислить бинарным поиском. Пусть g — возрастающая функция на множестве натуральных чисел, предположим также, что x, y и z удовлетворяют условию $g\ z \leqslant z < g\ y$. Для отыскания единственного значения m, где $m = \mathit{bsearch}\ g\ (x,y)\ z$, на отрезке $x \leqslant m < y$ и с учётом $g\ m \leqslant z < g\ (m+1)$ можно использовать инварианты $g\ a \leqslant z < g\ b$ и $x \leqslant a < b \leqslant y$. Таким образом, получаем функцию

$$
\begin{array}{l}
\mathit{bsearch}\ g\ (a,b)\ z \\
\quad \mid a+1 == b\ =\ a \\
\quad \mid g\ m \leqslant z\ =\ \mathit{bsearch}\ g\ (m,b)\ z \\
\quad \mid \textbf{otherwise}\ =\ \mathit{bsearch}\ g\ (a,m)\ z \\
\quad\ \ \textbf{where}\ m\ =\ (a+b)\ \mathrm{div}\ 2
\end{array}
$$

Так как $a+1<b \Rightarrow a<m<y$, то ни $g\ x$, ни $g\ y$ алгоритмом не вычисляются, поэтому они могут быть фиктивными значениями. В частности, имеем

$$\begin{array}{rcl} m & = & bsearch\ (\lambda y \to f(0,y))\ (-1,z+1)\ z \\ n & = & bsearch\ (\lambda x \to f(x,0))\ (-1,z+1)\ z \end{array}$$

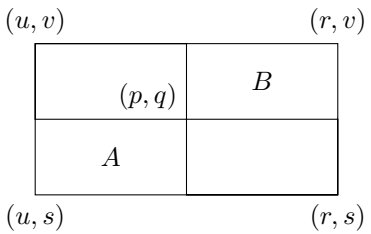
где f дополнена фиктивными значениями $f(0,-1)=0$ и $f(-1,0)=0$.

Этой версии *invert* требуется примерно $2\log z+m+n$ вызовов f в худшем случае и $2\log z+m$ min n в лучшем. Так как m и n могут оказаться значительно меньше z, к примеру, если $f(x,y)=2^x+3^y$, то мы получаем алгоритм, требующий в худшем случае только $O(\log z)$ операций.

Учитель. Мои поздравления, Анна и Фёдор, вы переоткрыли важную стратегию поиска, названную Дэвидом Грисом (David Gries) *седловым поиском*; см. работы Бэкхауза (Backhouse, 1986), Дейкстры (Dijkstra, 1985) и Гриса (Gries, 1981). Мне представляется, что Грис назвал его так, потому что форма трёхмерного графика f с наименьшим элементом в левом нижнем углу, наибольшим в правом верхнем и двумя крыльями напоминает седло. Основная идея, как заметила Анна, в том, чтобы начать поиск на кончике одного из крыльев, а не в точках с наименьшим или наибольшим значением. Рассматривая эту задачу, Дейкстра (Dijkstra, 1985) также указал на преимущества использования логарифмического поиска для определения подходящего начального прямоугольника.

Мария. А что если мы попробуем решение по методу «разделяй и властвуй»? В смысле, почему бы в начале не смотреть на элемент в середине прямоугольника? Уверена, что имеет смысл применить двумерный аналог бинарного поиска.

Предположим, мы ограничили поиск прямоугольником с левым верхним углом (u,v) и правым нижним (r,s). Почему бы вместо того, чтобы смотреть на $f(u,v)$, не проанализировать $f(p,q)$, где $p = (u+r)$ div 2, а $q = (v+s)$ div 2? Вот картинка:



Если $f(p,q) < z$, то мы можем отбросить все элементы левого нижнего прямоугольника A. Аналогично, если $f(p,q) > z$, можно отбросить правый верхний прямоугольник B. Если же $f(p,q) = z$, то можно отбросить оба.

Я знаю, что эта стратегия нарушает свойство, предложенное Анной, о том, что область поиска всегда прямоугольник. Вместо этого получаем либо два прямоугольника, либо Г-образную форму. Но мы же функциональные программисты и можем не ограничивать себя простыми циклами: алгоритм «разделяй и властвуй» столь же лёгок в реализации, что и итеративный, потому что оба можно выразить рекурсией.

Иван. Тебе всё равно придётся иметь дело с Г-образной формой. Конечно, её можно разбить на два прямоугольника. Фактически это можно сделать двумя способами: либо горизонтальным разрезом, либо вертикальным. Позволь мне дать грубую оценку. Рассмотрим прямоугольник размера $m \times n$ и обозначим $T(m,n)$ количество требуемых для поиска по нему вычислений f. Если $m = 0$ или $n = 0$, то искать нечего. Если $m = 1$ или $n = 1$, имеем

$$\begin{array}{lcl} T(1,n) & = & 1 + T(1, \lceil n/2 \rceil) \\ T(m,1) & = & 1 + T(\lceil m/2 \rceil, 1) \end{array}$$

В противном случае, если $m \geqslant 2$ и $n \geqslant 2$, можно отбросить прямоугольник размера, как минимум, $\lfloor m/2 \rfloor \times \lfloor n/2 \rfloor$. Если мы делаем горизонтальный разрез, то остаётся два прямоугольника: один размера $\lfloor m/2 \rfloor \times \lceil n/2 \rceil$, а второй $\lceil m/2 \rceil \times n$. Таким образом,

$$T(m,n) = 1 + T(\lfloor m/2 \rfloor, \lceil n/2 \rceil) + T(\lceil m/2 \rceil, n)$$

Если же мы делаем вертикальный разрез, то имеем

$$T(m,n) = 1 + T(\lceil m/2 \rceil, \lfloor n/2 \rfloor) + T(m, \lceil n/2 \rceil)$$

Я, правда, не вижу решения этих рекуррентных соотношений.

Фёдор. Если сделать одновременно и горизонтальный, и вертикальный разрезы, то останется три прямоугольника, так что при $m \geqslant 2$ и $n \geqslant 2$ будем иметь

$$T(m,n) = 1 + T(\lceil m/2 \rceil, \lfloor n/2 \rfloor) + T(\lceil m/2 \rceil, \lceil n/2 \rceil) + T(\lfloor m/2 \rfloor, \lceil n/2 \rceil)$$

Я могу решить это соотношение. Положим $U(i,j) = T(2^i, 2^j)$, теперь

$$\begin{array}{lcl} U(i,0) & = & i \\ U(0,j) & = & j \\ U(i+1,j+1) & = & 1 + 3U(i,j) \end{array}$$

Решением будет $U(i,j) = 3^k(|j-i| + 1/2) - 1/2$, где $k = i \min j$, это можно проверить индукцией. Следовательно, если $m \leqslant n$, то

$$T(m,n) \leqslant 3^{\log m} \log(2n/m) = m^{1.59} \log(2n/m)$$

Это лучше, чем $m+n$, при условии, что m значительно меньше n.

Иван. Я не думаю, что решение с тремя прямоугольниками столь же хорошее, что и с двумя. Следуя твоему подходу, Фёдор, давай положим $U(i,j) = T(2^i, 2^j)$. В предположении, что $i \leqslant j$, выполняя горизонтальный разрез, получаем

$$\begin{array}{lcl} U(0,j) & = & j \\ U(i+1,j+1) & = & 1 + U(i,j) + U(i,j+1) \end{array}$$

Решением является $U(i,j) = 2^i(j - i/2 + 1) - 1$, что также можно проверить индукцией. Поэтому

$$T(m,n) \leqslant m \log(2n/\sqrt{m})$$

Если $i \geqslant j$, то следует делать вертикальный разрез вместо горизонтального. Тогда мы получим алгоритм, требующий самое большее $n \log(2m/\sqrt{n})$ вызовов f. В любом случае, если один из параметров m или n значительно меньше другого, то мы имеем алгоритм лучше, чем при седловом поиске.

Анна. Пока вы двое решали рекуррентные соотношения, я думала над проблемой нижней границы сложности для функции *invert*. Рассмотрим различные возможные результаты при поиске в прямоугольнике размера $m \times n$. Предположим, что имеется $A(m,n)$ различных решений. Каждая проверка $f(x,y)$ относительно z допускает три возможных результата, поэтому высота h тернарного дерева проверок должна удовлетворять условию $h \geqslant \log_3 A(m,n)$. Считая, что $A(m,n)$ достижимо, получаем нижнюю границу для числа проверок, которые необходимо выполнить. Ситуация похожа на сортировку n элементов двоичными сравнениями: имеется $n!$ возможных результатов, поэтому любой алгоритм сортировки должен в худшем случае провести как минимум $\log_2 n!$ сравнений.

$A(m,n)$ достичь нетрудно: каждый список пар (x,y), где $0 \leqslant x < n$ и $0 \leqslant y < m$, удовлетворяющих $f(x,y) = z$, находится в соотношении один к одному со ступенчатой функцией, график которой идёт из левого верхнего угла прямоугольника размера $m \times n$ в его правый нижний угол. Значение z появляется на внутренних углах ступенек. Разумеется, этот график

необязательно задаёт именно тот путь, по которому идёт функция *find*. Количество таких путей $\binom{m+n}{n}$, а значит, таким же будет значение $A(m,n)$.

По другому этот результат можно получить, если предположить, что имеется k решений. Значение z можно разместить в m строках k раз ровно $\binom{m}{k}$ способами, и всякий раз будет $\binom{n}{k}$ возможных вариантов для столбцов. Следовательно,

$$A(m,n) = \sum_{k=0}^{m} \binom{m}{k}\binom{n}{k} = \binom{m+n}{n}$$

поскольку это суммирование является примером свёртки Вандермонда; см. (Graham, 1989b). Беря логарифмы, получаем нижнюю границу

$$\log A(m,n) = \Omega(m\log(1+n/m) + n\log(1+m/n))$$

Это приближение показывает, что если $m = n$, то не удастся сделать менее, чем $\Omega(m+n)$ шагов. Но если $m \leqslant n$, то $m \leqslant n\log(1+m/n)$, поскольку $x \leqslant \log(1+x)$ при $0 \leqslant x \leqslant 1$. Таким образом, $A(m,n) = \Omega(m\log(n/m))$. Решение Ивана не слишком близко к этой границе, потому что в его алгоритме в случае $m \leqslant n$ число операций оценивается как $O(m\log(n/\sqrt{m}))$.

Мария. Я не думаю, что в решении Ивана действительно необходимо применять метод «разделяй и властвуй». Есть и другие способы использовать бинарный поиск. Например, можно просто выполнить m бинарных поисков, по одному для каждой строки. Так мы получим решение со сложностью $O(m\log n)$. Но мне кажется, что её можно улучшить, достигнув оптимальную асимптотическую сложность $O(m\log(n/m))$ в предположении, что $m \leqslant n$.

Допустим, как и прежде, что поиск ограничен прямоугольником с левым верхним углом (u,v) и правым нижним (r,s). Таким образом, имеется $r-u$ столбцов и $s-v$ строк. Далее, предположим, что $v-s \leqslant r-u$, т.е. столбцов как минимум столько же, сколько и строк. Пусть мы выполняем бинарный поиск по средней строке, $q = (v+s)\operatorname{div} 2$, и ищем такое p, что $f(p,q) \leqslant z < f(p+1,q)$. Если $f(p,q) < z$, то нужно продолжать поиск только в двух прямоугольниках $((u,v),(p,q+1))$ и $((p+1,q-1),(r,s))$. Если $f(p,q) = z$, то можно вырезать столбец p и продолжить поиск в прямоугольниках $((u,v),(p-1,q+1))$ и $((p+1,q-1),(r,s))$. Рассуждения в случае, когда строк больше, чем столбцов, аналогичны. В результате, можно отбросить половину элементов массива с помощью логарифмического числа испытаний.

Вот алгоритм, который я имею в виду: реализуем *invert*

$$
\begin{array}{l}
\mathit{find}\ (u,v)\ (r,s)\ f\ z \\
\quad \mid u > r \vee v < s \ = \ [\,] \\
\quad \mid v - s \leqslant r - u \ = \ \mathit{rfind}\ (\mathit{bsearch}\ (\lambda x \to f(x,q))\ (u-1, r+1)\ z) \\
\quad \mid \textbf{otherwise} \ = \ \mathit{cfind}\ (\mathit{bsearch}\ (\lambda y \to f(p,y))\ (s-1, v+1)\ z) \\
\quad \textbf{where} \\
\qquad p \ = \ (u+r)\ \mathrm{div}\ 2 \\
\qquad q \ = \ (v+s)\ \mathrm{div}\ 2 \\
\qquad \mathit{rfind}\ p \ = \ (\textbf{if}\ f(p,q) == z\ \textbf{then}\ (p,q) : \mathit{find}\ (u,v)\ (p-1, q+1)\ f\ z \\
\qquad\qquad\qquad \textbf{else}\ \mathit{find}\ (u,v)\ (p, q+1)\ f\ z) +\!\!+ \\
\qquad\qquad\qquad \mathit{find}\ (p+1, q-1)\ (r,s)\ f\ z \\
\qquad \mathit{cfind}\ q \ = \ \mathit{find}\ (u,v)\ (p-1, q+1)\ f\ z +\!\!+ \\
\qquad\qquad\qquad (\textbf{if}\ f(p,q) == z\ \textbf{then}\ (p,q) : \mathit{find}\ (p+1, q-1)\ (r,s)\ f\ z \\
\qquad\qquad\qquad \textbf{else}\ \mathit{find}\ (p+1, q)\ (r,s)\ f\ z)
\end{array}
$$

Рис. 3.1: пересмотренное определение функции *find*

$$
\begin{array}{l}
\mathit{invert}\ f\ z \ = \ \mathit{find}\ (0,m)\ (n,0)\ f\ z \\
\quad \textbf{where}\ m \ = \ \mathit{bsearch}\ (\lambda y \to f(0,y))\ (-1, z+1)\ z \\
\qquad\qquad n \ = \ \mathit{bsearch}\ (\lambda x \to f(x,0))\ (-1, z+1)\ z
\end{array}
$$

где *find* $(u,v)\ (r,s)\ f\ z$, приведённая на рис. 3.1, выполняет поиск по прямоугольнику с левым верхним углом (u,v) и правым нижним углом (r,s).

Что же касается анализа, пусть $T(m,n)$ снова обозначает количество вызовов, необходимых для поиска по прямоугольнику $m \times n$. Предположим, что $m \leqslant n$. В лучшем случае, когда каждый бинарный поиск по строке возвращает самый левый или самый правый элемент, имеем $T(m,n) = \log n + T(m/2, n)$, т.е. $T(m,n) = O(\log m \times \log n)$. В худшем случае, когда бинарный поиск возвращает средний элемент, получается

$$T(m,n) = \log n + 2T(m/2, n/2)$$

Чтобы решить это соотношение, положим $U(i,j) = T(2^i, 2^j)$. Тогда имеем

$$U(i,j) = \sum_{k=0}^{i-1} 2^k (j-k) = O(2^i (j-i))$$

Следовательно, $T(m,n) = O(m \log(n/m))$, что асимптотически оптимально согласно полученной Анной нижней оценке.

Учитель. Отличная работа, все четверо! Удивительно, что примерно за 25 лет, в которые седловой поиск использовался в качестве примера формального построения программ, никто, как кажется, не заметил, что это алгоритм с не самой лучшей асимптотикой.

Послесловие

Реальная история, стоящая за этой жемчужиной, такова. Я решил использовать седловой поиск в качестве упражнения для кандидатов, проходящих собеседование на поступление в Оксфорд. Им давался двумерный числовой массив, возрастающий вдоль каждой строки и каждого столбца, и предлагалось найти систематический способ обнаружения всех вхождений заданного числа. Я хотел убедить их, что поиск из левого верхнего угла в правый нижний является хорошей стратегией. Однако те кандидаты, который изучали в школе информатику, пытались использовать бинарный поиск либо из середины каждой строки, либо из центра прямоугольника. Полагая, что седловой поиск является золотым стандартом для решения данной задачи, я намекал им, что следует поменять ход мысли. Только спустя некоторое время я заинтересовался тем, могли ли они быть правы.

Помимо описания нового алгоритма для старой задачи, как мне кажется, есть два других методологических аспекта, достойных упоминания. Во-первых, формальное построение программ испытывает сильное влияние со стороны доступных в целевом языке программирования методов вычисления. Последнее решение, приведённое Марией, вряд ли кто-то назовёт элегантным, тем не менее, оно достаточно элементарно в ситуации, когда рекурсия и конкатенация списков являются базовыми операциями, однако его реализация в языке с массивами и циклами может оказаться довольно затруднительной. Во-вторых, как совершенно уверены проектировщики алгоритмов, формальное построение программ должно сопровождаться идеями относительно возможных путей улучшения эффективности. Такие идеи могут возникнуть частично благодаря решению рекуррентных соотношений, а частично из определения нижних границ.

Впервые эта жемчужина появилась в (Bird, 2006). Один из рецензентов исходной работы указал следующее:

> Сложная реализация помимо собственно эффективности привносит и некоторые накладные расходы, которые так часто упускают из виду при анализе, подобном представленному в этой работе. Если автор действительно хочет убедить нас, что