ГЛАВА 15

Динамическое программирование

Динамическое программирование, как и метод разбиения, позволяет решать задачи, комбинируя решения вспомогательных задач. (Термин "программирование" в данном контексте означает табличный метод, а не составление компьютерного кода.) В главе 2 уже было показано, как в алгоритмах разбиения задача делится на несколько независимых подзадач, каждая из которых решается рекурсивно, после чего из решений вспомогательных задач формируется решение исходной задачи. Динамическое программирование, напротив, находит применение тогда, когда вспомогательные задачи не являются независимыми, т.е. когда разные вспомогательные задачи используют решения одних и тех же подзадач. В этом смысле алгоритм разбиения, многократно решая задачи одних и тех же типов, выполняет больше действий, чем было бы необходимо. В алгоритме динамического программирования каждая вспомогательная задача решается только один раз, после чего ответ сохраняется в таблице. Это позволяет избежать одних и тех же повторных вычислений каждый раз, когда встречается данная подзадача.

Динамическое программирование, как правило, применяется к ***задачам оптимизации*** (optimization problems). В таких задачах возможно наличие многих решений. Каждому варианту решения можно сопоставить какое-то значение, и нам нужно найти среди них решение с оптимальным (минимальным или максимальным) значением. Назовем такое решение *одним из возможных* оптимальных решений. В силу того, что таких решений с оптимальным значением может быть несколько, следует отличать их от *единственного* оптимального решения.

Процесс разработки алгоритмов динамического программирования можно разбить на четыре перечисленных ниже этапа.

1. Описание структуры оптимального решения.

2. Рекурсивное определение значения, соответствующего оптимальному решению.
3. Вычисление значения, соответствующего оптимальному решению, с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

Этапы 1–3 составляют основу метода динамического программирования. Этап 4 может быть опущен, если требуется узнать только значение, соответствующее оптимальному решению. На четвертом этапе иногда используется дополнительная информация, полученная на третьем этапе, что облегчает процесс конструирования оптимального решения.

В последующих разделах метод динамического программирования используется для решения некоторых задач оптимизации. В разделе 15.1 исследуется задача по составлению графика работы двух автосборочных конвейеров. По завершении каждого этапа сборки автомобиль либо остается на том же конвейере, либо перемещается на другой. В разделе 15.2 исследуется вопрос о том, в каком порядке следует выполнять перемножение нескольких матриц, чтобы свести к минимуму общее количество операций перемножения скаляров. На этих двух примерах в разделе 15.3 обсуждаются две основные характеристики, которыми должны обладать задачи, для которых подходит метод динамического программирования. Далее, в разделе 15.4 показано, как найти самую длинную общую подпоследовательность двух последовательностей. Наконец, в разделе 15.5 с помощью динамического программирования конструируется дерево бинарного поиска, оптимальное для заданного распределения ключей, по которым ведется поиск.

15.1 Расписание работы конвейера

В качестве первого примера динамического программирования рассмотрим решение производственной задачи. Некая автомобильная корпорация производит автомобили на заводе, оснащенном двумя конвейерами (рис. 15.1). На оба конвейера, на каждом из которых предусмотрено по несколько рабочих мест, поступают для сборки автомобильные шасси, после чего на каждом рабочем месте конвейера добавляются все новые детали. Наконец, собранный автомобиль покидает конвейер. На каждом конвейере имеется n рабочих мест, пронумерованных от 1 до n. Обозначим рабочее место под номером j на i-м конвейере (где i принимает значения 1 или 2) через $S_{i,j}$. На обоих конвейерах на рабочих местах с одинаковыми номерами выполняются один и те же операции. Однако конвейеры создавались в разное время и по разным технологиям, поэтому время выполнения одних и тех же операций на разных конвейерах различается. Обозначим время сборки на рабочем месте $S_{i,j}$ через $a_{i,j}$. Как видно из рис. 15.1, шасси поступает на первое

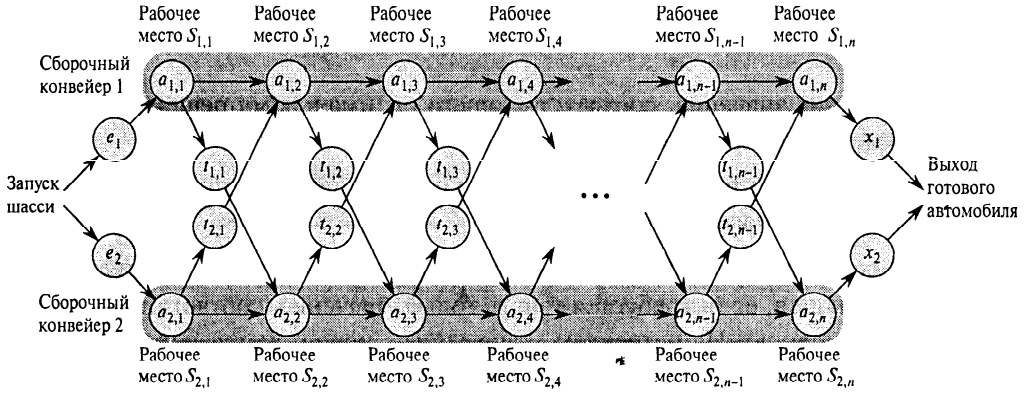


Рис. 15.1. Производственная задача по определению оптимального способа сборки

рабочее место одного из конвейеров, а затем переходит от одного рабочего места к другому. Постановка шасси на конвейер занимает время e_i, а снятие готового автомобиля с конвейера — время x_i.

Обычно поступившее на конвейер шасси проходит все этапы сборки на одном и том же конвейере. Временем перехода от одного рабочего места к другому, если этот переход выполняется в пределах одного конвейера, можно пренебречь. Однако иногда заказчик требует, чтобы автомобиль был собран за кратчайшее время, и такой порядок приходится нарушить. Для ускорения сборки шасси по-прежнему проходит все n рабочих мест в обычном порядке, однако менеджер может дать указание перебросить частично собранный автомобиль с одного конвейера на другой, причем такая переброска возможна на любом этапе сборки. Время, которое требуется для перемещения шасси с конвейера i на соседний после прохождения рабочего места $S_{i,j}$, равно $t_{i,j}$, $i = 1, 2, \ldots, n-1$ (поскольку после прохождения n-го рабочего места сборка завершается). Задача заключается в том, чтобы определить, какие рабочие места должны быть выбраны на первом конвейере, а какие — на втором, чтобы минимизировать полное время, затраченное на заводе на сборку одного автомобиля. В примере, проиллюстрированном на рис. 15.2 *а*, полное время получается наименьшим, если на первом конвейере выбираются рабочие места 1, 3 и 6, а на втором — места 2, 4 и 5. На рисунке указаны величины e_i, $a_{i,j}$, $t_{i,j}$ и x_i. Самый быстрый путь через фабрику выделен жирной линией. В части *б* рисунка приведены величины $f_i[j]$, f^*, $l_i[j]$ и l^*, соответствующие примеру, изображенному в части *а* (о том, что означают эти величины, будет сказано немного позже).

Очевидный метод перебора, позволяющий минимизировать время сборки на конвейерах с малым числом рабочих мест, становится неприменимым для большого количества рабочих мест. Если заданы списки рабочих мест, которые исполь-

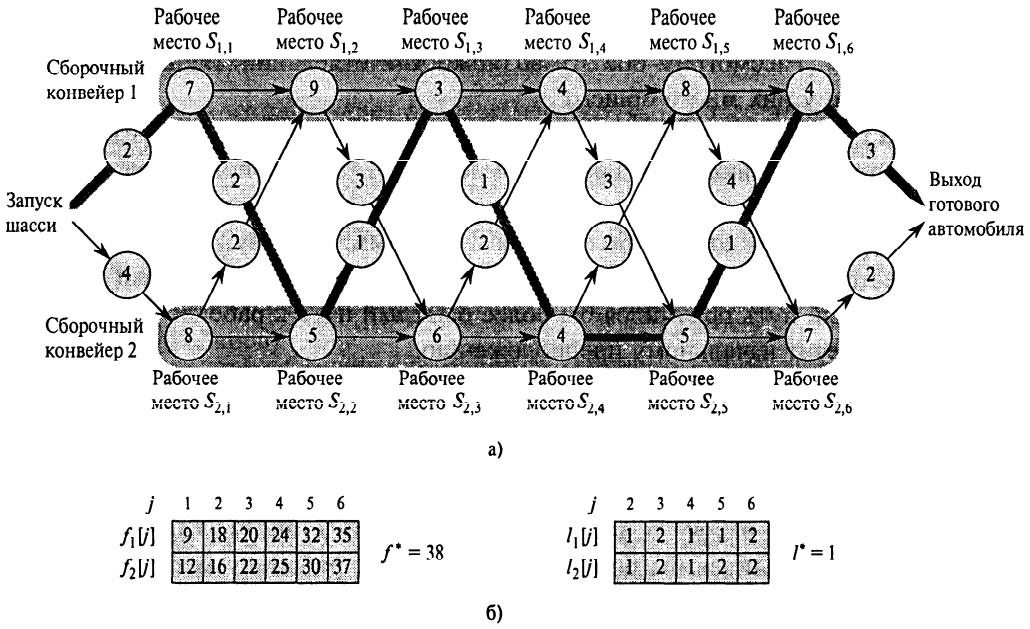


Рис. 15.2. Пример задачи на составление расписания сборочного конвейера

зуются на первом и втором конвейерах, то время полного прохождения шасси по конвейеру легко вычислить за время $\Theta(n)$. К сожалению, всего существует 2^n возможных комбинаций. Это легко понять, рассматривая множество рабочих мест первого конвейера как подмножество множества $\{1, 2, \ldots, n\}$; очевидно, что всего существует 2^n таких подмножеств. Таким образом, способ, при котором наиболее рациональное прохождение по конвейеру определяется методом перебора всех возможных вариантов, в результате чего определяется время сборки для каждого варианта, занял бы время, равное $\Omega(2^n)$. Очевидно, что при больших n этот способ неприемлем.

Первый этап: структура самой быстрой сборки

В парадигме динамического программирования первый этап заключается в том, чтобы охарактеризовать структуру оптимального решения. В задаче по составлению расписания работы конвейера этот шаг можно выполнить следующим образом. Рассмотрим способ, при котором шасси, поступившее на первый конвейер, попадет на рабочее место $S_{1,j}$ за кратчайшее время. Если $j = 1$, существует всего один способ прохождения такого отрезка пути, поэтому легко определить время достижения рабочего места $S_{1,j}$. Однако если $j = 2, 3, \ldots, n$, возможен один из двух вариантов: на рабочее место $S_{1,j}$ шасси могло попасть непосредственно с рабочего места $S_{1,j-1}$ или с $S_{2,j-1}$. В первом случае временем перехода

от одного рабочего места к другому можно пренебречь, а во втором переход займет время $t_{2,j-1}$. Рассмотрим обе эти возможности отдельно, хотя впоследствии станет ясно, что у них много общего.

Сначала предположим, что самый быстрый путь, проходящий через рабочее место $S_{1,j}$, проходит также через рабочее место $S_{1,j-1}$. Основной вывод, который можно сделать в этой ситуации, заключается в том, что отрезок пути от начальной точки до $S_{1,j-1}$ тоже должен быть самым быстрым. Почему? Если бы на это рабочее место можно было бы попасть быстрее, то при подстановке данного отрезка в полный путь получился бы более быстрый путь к рабочему месту $S_{1,j}$, а это противоречит начальному предположению.

Теперь предположим, что самый быстрый путь, проходящий через рабочее место $S_{1,j}$, проходит также через рабочее место $S_{2,j-1}$. В этом случае можно прийти к выводу, что на рабочее место $S_{2,j-1}$ шасси должно попасть за кратчайшее время. Объяснение аналогично предыдущему: если бы существовал более быстрый способ добраться до рабочего места $S_{2,j-1}$, то при подстановке данного отрезка в полный путь получился бы более быстрый путь к рабочему месту $S_{1,j}$, а это снова противоречит сделанному предположению.

Обобщая эти рассуждения, можно сказать, что задача по составлению оптимального расписания (определение самого быстрого пути до рабочего места $S_{i,j}$) содержит в себе оптимальное решение подзадач (нахождение самого быстрого пути до рабочего места $S_{1,j-1}$ или $S_{2,j-1}$). Назовем это свойство ***оптимальной подструктурой*** (optimal substructure). В разделе 15.3 мы убедимся, что наличие этого свойства — один из признаков применимости динамического программирования.

С помощью свойства оптимальной подструктуры можно показать, что определение оптимального решения задачи сводится к определению оптимального решения ее подзадач. В задаче по составлению расписания работы конвейера рассуждения проводятся следующим образом. Наиболее быстрый путь, проходящий через рабочее место $S_{1,j}$, должен также проходить через рабочее место под номером $j-1$, расположенное на первом или втором конвейере. Таким образом, если самый быстрый путь проходит через рабочее место $S_{1,j}$, то справедливо одно из таких утверждений:

- этот путь проходит через рабочее место $S_{1,j-1}$, после чего шасси попадает непосредственно на рабочее место $S_{1,j}$;
- этот путь проходит через рабочее место $S_{2,j-1}$, после чего автомобиль перебрасывается из второго конвейера на первый, а затем попадает на рабочее место $S_{1,j}$.

Из соображений симметрии для самого быстрого пути, проходящего через рабочее место $S_{2,j}$, справедливо одно из следующих утверждений:

- этот путь проходит через рабочее место $S_{2,j-1}$, после чего шасси попадает непосредственно на рабочее место $S_{2,j}$;
- этот путь проходит через рабочее место $S_{1,j-1}$, после чего автомобиль перебрасывается из первого конвейера на второй, а затем попадает на рабочее место $S_{2,j}$.

Чтобы решить задачу определения наиболее быстрого пути до рабочего места j, которое находится на любом из конвейеров, нужно решить вспомогательную задачу определения самого быстрого пути до рабочего места $j-1$ (также на любом из конвейеров).

Таким образом, оптимальное решение задачи об оптимальном расписании работы конвейера можно найти путем поиска оптимальных решений подзадач.

Второй этап: рекурсивное решение

Второй этап в парадигме динамического программирования заключается в том, чтобы рекурсивно определить оптимальное решение в терминах оптимальных решений подзадач. В задаче по составлению расписания работы конвейера роль подзадач играют задачи по определению самого быстрого пути, проходящего через j-е рабочее место на любом из двух конвейеров для $j = 2, 3, \ldots, n$. Обозначим через $f_i[j]$ минимально возможное время, в течение которого шасси проходит от стартовой точки до рабочего места $S_{i,j}$.

Конечная цель заключается в том, чтобы определить кратчайшее время f^*, в течение которого шасси проходит по всему конвейеру. Для этого автомобиль, который находится в сборке, должен пройти весь путь до рабочего места n, находящегося на первом или втором конвейере, после чего он покидает фабрику. Поскольку самый быстрый из этих путей является самым быстрым путем прохождения всего конвейера, справедливо следующее соотношение:

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2). \tag{15.1}$$

Легко также сделать заключение по поводу величин $f_1[1]$ и $f_2[1]$. Чтобы за кратчайшее время пройти первое рабочее место на любом из конвейеров, шасси должно попасть на это рабочее место непосредственно. Таким образом, можно записать уравнения

$$f_1[1] = e_1 + a_{1,1}, \tag{15.2}$$

$$f_2[1] = e_2 + a_{2,1}. \tag{15.3}$$

А теперь рассмотрим, как вычислить величину $f_i[j]$ при $j = 2, 3, \ldots, n$ (и $i = 1, 2$). Рассуждая по поводу $f_1[j]$, мы пришли к выводу, что самый быстрый путь до рабочего места $S_{1,j}$ является также либо самым быстрым путем до рабочего

места $S_{1,j-1}$, после чего шасси попадает прямо на рабочее место $S_{1,j}$, либо самым быстрым путем до рабочего места $S_{2,j-1}$, с последующим переходом со второго конвейера на первый. В первом случае можно записать соотношение $f_1[j] = f_1[j-1] + a_{1,j}$, а во втором — соотношение $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Таким образом, при $j = 2, 3, \ldots, n$ выполняется уравнение

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}). \quad (15.4)$$

Аналогично можно записать уравнение

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}). \quad (15.5)$$

Комбинируя уравнения (15.2)–(15.5), получим рекуррентные соотношения

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{при } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{при } j \geqslant 2. \end{cases} \quad (15.6)$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{при } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{при } j \geqslant 2. \end{cases} \quad (15.7)$$

На рис. 15.2*б* приведены величины $f_i[j]$, соответствующие примеру, изображенному на рис. 15.2*а*, и вычисленные по уравнениям (15.6) и (15.7), а также величина f^*.

Величины $f_i[j]$ являются значениями, соответствующими оптимальным решениям подзадач. Чтобы было легче понять, как конструируется оптимальное решение, обозначим через $l_i[j]$ номер конвейера (1 или 2), содержащего рабочее место под номером $j-1$, через которое проходит самый быстрый путь к рабочему месту $S_{i,j}$. Индексы i и j принимают следующие значения: $i = 1, 2$; $j = 2, 3, \ldots, n$. (Величины $l_i[1]$ не определяются, поскольку на обоих конвейерах отсутствуют рабочие места, предшествующие рабочему месту под номером 1.) Кроме того, обозначим через l^* номер конвейера, через n-е рабочее место которого проходит самый быстрый полный путь сборки. Величины $l_i[j]$ облегчат определение самого быстрого пути. С помощью приведенных на рис. 15.2*б* величин l^* и $l_i[j]$ самый быстрый способ сборки на заводе, изображенном на рис. 15.2*а*, выясняется путем таких рассуждений. Начнем с $l^* = 1$; при этом используется рабочее место $S_{1,6}$. Теперь посмотрим на величину $l_1[6]$, которая равна 2, из чего следует, что используется рабочее место $S_{2,5}$. Продолжая рассуждения, смотрим на величину $l_2[5] = 2$ (используется рабочее место $S_{2,4}$), величину $l_2[4] = 1$ (рабочее место $S_{1,3}$), величину $l_1[3] = 2$ (рабочее место $S_{2,2}$) и величину $l_2[2] = 1$ (рабочее место $S_{1,1}$).

Третий этап: вычисление минимальных промежутков времени

На данном этапе на основе уравнения (15.1) и рекуррентных соотношений (15.6) и (15.7) легко было бы записать рекурсивный алгоритм определения самого быстрого пути сборки автомобиля на заводе. Однако с таким алгоритмом связана одна проблема: время его работы экспоненциально зависит от n. Чтобы понять, почему это так, введем значения $r_i(j)$, обозначающие количество ссылок в рекурсивном алгоритме на величины $f_i[j]$. Из уравнения (15.1) получаем:

$$r_1(n) = r_2(n) = 1. \tag{15.8}$$

Из рекуррентных уравнений (15.6) и (15.7) следует соотношение

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1), \tag{15.9}$$

справедливое при $j = 1, 2, \ldots, n-1$. В упражнении 15.1-2 предлагается показать, что $r_i(j) = 2^{n-j}$. Таким образом, к величине $f_1[1]$ алгоритм обращается 2^{n-1} раз! В упражнении 15.1-3 предлагается показать, что полное количество обращений ко всем величинам $f_i[j]$ равно $\Theta(2^n)$.

Намного лучших результатов можно достичь, если вычислять величины $f_i[j]$ в порядке, отличном от рекурсивного. Обратите внимание, что при $j \geqslant 2$ каждое значение $f_i[j]$ зависит только от величин $f_1[j-1]$ и $f_2[j-1]$. Вычисляя значения $f_i[j]$ в порядке *увеличения* номеров рабочих мест j — слева направо (см. рис. 15.2*б*), — можно найти самый быстрый путь (и время его прохождения) по заводу в течение времени $\Theta(n)$. Приведенная ниже процедура FASTEST_WAY принимает в качестве входных данных величины $a_{i,j}$, $t_{i,j}$, e_i и x_i, а также количество рабочих мест на каждом конвейере n:

```
FASTEST_WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4      do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5            then f1[j] ← f1[j − 1] + a1,j
 6                 l1[j] ← 1
 7            else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                 l1[j] ← 2
 9         if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10            then f2[j] ← f2[j − 1] + a2,j
11                 l2[j] ← 2
12            else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
```

```
14 if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```

Процедура FASTEST_WAY работает следующим образом. В строках 1–2 с помощью уравнений (15.2) и (15.3) вычисляются величины $f_1[1]$ и $f_2[1]$. Затем в цикле **for** в строках 3–13, вычисляются величины $f_i[j]$ и $l_i[j]$ при $i = 1, 2$ и $j = 2, 3, \ldots, n$. В строках 4–8 с помощью уравнения (15.4) вычисляются величины $f_1[j]$ и $l_1[j]$, а в строках 9–13, с помощью уравнения (15.5), величины $f_2[j]$ и $l_2[j]$. Наконец, в строках 14–18 с помощью уравнения (15.1) вычисляются величины f^* и l^*. Поскольку строки 1–2 и 14–18 выполняются в течение фиксированного времени, и выполнение каждой из $n - 1$ итераций цикла **for**, заданного в строках 3–13, тоже длится в течение фиксированного времени, на выполнение всей процедуры требуется время, равное $\Theta(n)$.

Один из способов вычисления значений $f_i[j]$ и $l_i[j]$ — заполнение ячеек соответствующей таблицы. Как видно из рис. 15.2*б*, таблицы для величин $f_i[j]$ и $l_i[j]$ заполняются слева направо (и сверху вниз в каждом столбце). Для вычисления величины $f_i[j]$ понадобятся значения $f_1[j-1]$ и $f_2[j-1]$. Зная, что эти величины уже вычислены и занесены в таблицу, их можно просто извлечь из соответствующих ячеек таблицы, когда они понадобятся.

Четвертый этап: построение самого быстрого пути

После вычисления величин $f_i[j]$, f^*, $l_i[j]$ и l^* нужно построить последовательность решений, используемых в самом быстром пути сборки. Ранее было рассказано, как это можно сделать для примера, проиллюстрированного на рис. 15.2.

Приведенная ниже процедура выводит в порядке убывания номера использующихся рабочих мест. В упражнении 15.1-1 предлагается модифицировать эту процедуру так, чтобы номера рабочих мест выводились в ней в порядке возрастания.

```
PRINT_STATIONS(l, n)
1 i ← l*
2 print "Конвейер " i ", рабочее место " n
3 for j ← n downto 2
4     do i ← li[j]
5        print "Конвейер " i ", рабочее место " j − 1
```

В примере, приведенном на рис. 15.2, процедура PRINT_STATIONS выведет такую последовательность рабочих мест:

Конвейер 1, рабочее место 6
Конвейер 2, рабочее место 5
Конвейер 2, рабочее место 4
Конвейер 1, рабочее место 3
Конвейер 2, рабочее место 2
Конвейер 1, рабочее место 1

Упражнения

15.1-1. Покажите, как модифицировать процедуру PRINT_STATIONS, чтобы она выводила данные в порядке возрастания номеров рабочих мест. (*Указание*: используйте рекурсию.)

15.1-2. С помощью уравнений (15.8) и (15.9) и метода подстановки покажите, что количество обращений $r_i(j)$ к величинам $f_i[j]$ в рекурсивном алгоритме равно 2^{n-j}.

15.1-3. Воспользовавшись результатом, полученном при выполнении упражнения 15.1-2, покажите, что полное число обращений ко всем величинам $f_i[j]$, выражающееся двойной суммой $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$, равно $2^{n+1}-2$.

15.1-4. В таблицах, содержащих величины $f_i[j]$ и $l_i[j]$, общее количество ячеек равно $4n-2$. Покажите, как свести количество ячеек к $2n+2$ и при этом иметь возможность вычисления величины f^* и вывода информации о всех рабочих местах, составляющих самый быстрый путь прохождения завода.

15.1-5. Профессор предположил, что могут существовать значения e_i, $a_{i,j}$ и $t_{i,j}$, для которых процедура FASTEST_WAY дает такие значения $l_i[j]$, что для некоторого рабочего места j $l_1[j]=2$ и $l_2[j]=1$. Принимая во внимание, что все переходы выполняются в течение положительных промежутков времени $t_{i,j}$, покажите, что предположение профессора ложно.

15.2 Перемножение цепочки матриц

Следующий пример динамического программирования — алгоритм, позволяющий решить задачу о перемножении цепочки матриц. Пусть имеется последовательность (цепочка), состоящая из n матриц, и нужно вычислить их произведение

$$A_1 A_2 \cdots A_n. \tag{15.10}$$

Выражение (15.10) можно вычислить, используя в качестве подпрограммы стандартный алгоритм перемножения пар матриц. Однако сначала нужно расставить скобки, чтобы устранить все неоднозначности в порядке перемножения. Порядок

произведения матриц ***полностью определен скобками*** (fully parenthesized), если произведение является либо отдельной матрицей, либо взятым в скобки произведением двух подпоследовательностей матриц, в котором порядок перемножения полностью определен скобками. Матричное умножение обладает свойством ассоциативности, поэтому результат не зависит от расстановки скобок. Например, если задана последовательность матриц $\langle A_1, A_2, A_3, A_4 \rangle$, то способ вычисления их произведения можно полностью определить с помощью скобок пятью разными способами:

$$(A_1(A_2(A_3A_4))) ,$$
$$(A_1((A_2A_3)A_4)) ,$$
$$((A_1A_2)(A_3A_4)) ,$$
$$((A_1(A_2A_3))A_4) ,$$
$$(((A_1A_2)A_3)A_4) .$$

От того как расставлены скобки при перемножении последовательности матриц, может сильно зависеть время, затраченное на вычисление произведения. Сначала рассмотрим, как определить стоимость произведения двух матриц. Ниже приводится псевдокод стандартного алгоритма. Атрибуты $rows$ и $columns$ означают количество строк и столбцов матрицы.

```
MATRIX_MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "Несовместимые размеры"
3     else for i ← 1 to rows[A]
4               do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8        return C
```

Матрицы A и B можно перемножать, только если они ***совместимы***: количество столбцов матрицы A должно совпадать с количеством строк матрицы B. Если A — это матрица размера $p \times q$, а B — матрица размера $q \times r$, то в результате их перемножения получится матрица C размера $p \times r$. Время вычисления матрицы C преимущественно определяется количеством произведений скаляров (далее в главе для краткости будем называть эту операцию скалярным умножением — *Прим. перев.*), которое выполняется в строке 7. Это количество равно pqr. Итак, стоимость умножения матриц будет выражаться в количестве умножений скалярных величин.

Чтобы проиллюстрировать, как расстановка скобок при перемножении нескольких матриц влияет на количество выполняемых операций, рассмотрим при-

мер, в котором перемножаются три матрицы $\langle A_1, A_2, A_3 \rangle$. Предположим, что размеры этих матриц равны 10×100, 100×5 и 5×50 соответственно. Перемножая матрицы в порядке, заданном выражением $((A_1 A_2) A_3)$, необходимо выполнить $10 \cdot 100 \cdot 5 = 5000$ скалярных умножений, чтобы найти результат произведения $A_1 A_2$ (при этом получится матрица размером 10×5), а затем — еще $10 \cdot 5 \cdot 50 = 2500$ скалярных умножений, чтобы умножить эту матрицу на матрицу A_3. Всего получается 7500 скалярных умножений. Если вычислять результат в порядке, заданном выражением $(A_1 (A_2 A_3))$, то сначала понадобится выполнить $100 \cdot 5 \cdot 50 = 25000$ скалярных умножений (при этом будет найдена матрица $A_2 A_3$ размером 100×50), а затем еще $10 \cdot 100 \cdot 50 = 50000$ скалярных умножений, чтобы умножить A_1 на эту матрицу. Всего получается 75 000 скалярных умножений. Таким образом, для вычисления результата первым способом понадобится в 10 раз меньше времени.

Задачу о перемножении последовательности матриц (matrix-chain multiplication problem) можно сформулировать так: для заданной последовательности матриц $\langle A_1, A_2, \ldots, A_n \rangle$, в которой матрица A_i, $i = 1, 2, \ldots, n$ имеет размер $p_{i-1} \times p_i$, с помощью скобок следует полностью определить порядок умножений в матричном произведении $A_1 A_2 \cdots A_n$, при котором количество скалярных умножений сведется к минимуму.

Обратите внимание, что само перемножение матриц в задачу не входит. Наша цель — определить оптимальный порядок перемножения. Обычно время, затраченное на нахождение оптимального способа перемножения матриц, с лихвой окупается, когда выполняется само перемножение (как это было в рассмотренном примере, когда удалось обойтись всего 7 500 скалярными умножениями вместо 75 000).

Подсчет количества способов расстановки скобок

Прежде чем приступить к решению задачи об умножении последовательности матриц методами динамического программирования, заметим, что исчерпывающая проверка всех возможных вариантов расстановки скобок не является эффективным алгоритмом ее решения. Обозначим через $P(n)$ количество альтернативных способов расстановки скобок в последовательности, состоящей из n матриц. Если $n = 1$, то матрица всего одна, поэтому скобки в матричном произведении можно расставить всего одним способом. Если $n \geqslant 2$, то произведение последовательности матриц, в котором порядок перемножения полностью определен скобками, является произведением двух таких произведений подпоследовательностей матриц, в которых порядок перемножения тоже полностью определен скобками. Разбиение на подпоследовательности может производиться на границе k-й и $k+1$-й матриц для любого $k = 1, 2, \ldots, n-1$. В результате получаем

рекуррентное соотношение

$$P(n) = \begin{cases} 1 & \text{при } n = 1, \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{при } n \geqslant 2. \end{cases} \tag{15.11}$$

В задаче 12-4 предлагается показать, что решением аналогичного рекуррентного соотношения является последовательность ***чисел Каталана*** (Catalan numbers), возрастающая как $\Omega\left(4^n/n^{3/2}\right)$. Более простое упражнение (упражнение 15.2-3) заключается в том, чтобы показать, что решение рекуррентного соотношения (15.11) ведет себя как $\Omega(2^n)$. Таким образом, количество вариантов расстановки скобок экспоненциально увеличивается с ростом n, и метод прямого перебора всех вариантов не подходит для определения оптимальной стратегии расстановки скобок в матричном произведении.

Первый этап: структура оптимальной расстановки скобок

Первый этап применения парадигмы динамического программирования — найти оптимальную вспомогательную подструктуру, а затем с ее помощью сконструировать оптимальное решение задачи по оптимальным решениям вспомогательных задач. В рассматриваемой задаче этот этап можно осуществить следующим образом. Обозначим для удобства результат перемножения матриц $A_i A_{i+1} \cdots A_j$ через $A_{i..j}$, где $i \leqslant j$. Заметим, что если задача нетривиальна, т.е. $i < j$, то любой способ расстановки скобок в произведении $A_i A_{i+1} \cdots A_j$ разбивает это произведение между матрицами A_k и A_{k+1}, где k — целое, удовлетворяющее условию $i \leqslant k < j$. Таким образом, при некотором k сначала вычисляются матрицы $A_{i..k}$ и $A_{k+1..j}$, а затем они умножаются друг на друга, в результате чего получается произведение $A_{i..j}$. Стоимость, соответствующая этому способу расстановки скобок, равна сумме стоимости вычисления матрицы $A_{i..k}$, стоимости вычисления матрицы $A_{k+1..j}$ и стоимости вычисления их произведения.

Ниже описывается оптимальная вспомогательная подструктура для данной задачи. Предположим, что в результате оптимальной расстановки скобок последовательность $A_i A_{i+1} \ldots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}. Тогда расстановка скобок в "префиксной" подпоследовательности $A_i A_{i+1} \ldots A_k$ тоже должна быть оптимальной. Почему? Если бы существовал более экономный способ расстановки скобок в последовательности $A_i A_{i+1} \ldots A_k$, то его применение позволило бы перемножить матрицы $A_i A_{i+1} \ldots A_j$ еще эффективнее, что противоречит предположению об оптимальности первоначальной расстановки скобок. Аналогично можно прийти к выводу, что расстановка скобок в подпоследовательности матриц $A_{k+1} A_{k+1} \ldots A_j$, возникающей в результате

оптимальной расстановки скобок в последовательности $A_iA_{i+1}\dots A_j$, также должна быть оптимальной.

Теперь с помощью нашей оптимальной вспомогательной подструктуры покажем, что оптимальное решение задачи можно составить из оптимальных решений вспомогательных задач. Мы уже убедились, что для решения любой нетривиальной задачи об оптимальном произведении последовательности матриц всю последовательность необходимо разбить на подпоследовательности и что каждое оптимальное решение содержит в себе оптимальные решения подзадач. Другими словами, решение полной задачи об оптимальном перемножении последовательности матриц можно построить путем разбиения этой задачи на две подзадачи — оптимальную расстановку скобок в подпоследовательностях $A_iA_{i+1}\dots A_k$ и $A_{k+1}A_{k+2}\dots A_j$. После этого находятся оптимальные решения подзадач, из которых затем составляется оптимальное решение полной задачи. Необходимо убедиться, что при поиске способа перемножения матриц учитываются все возможные разбиения — только в этом случае можно быть уверенным, что найденное решение будет глобально оптимальным.

Второй этап: рекурсивное решение

Далее, рекурсивно определим стоимость оптимального решения в терминах оптимальных решений вспомогательных задач. В задаче о перемножении последовательности матриц в качестве вспомогательной задачи выбирается задача об оптимальной расстановке скобок в подпоследовательности $A_iA_{i+1}\dots A_j$ при $1 \leqslant i \leqslant j \leqslant n$. Путь $m[i,j]$ — минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Тогда в полной задаче минимальная стоимость матрицы $A_{1..n}$ равна $m[1,n]$.

Определим величину $m[i,j]$ рекурсивно следующим образом. Если $i = j$, то задача становится тривиальной: последовательность состоит всего из одной матрицы $A_{i..j} = A_j$, и для вычисления произведения матриц не нужно выполнять никаких скалярных умножений. Таким образом, при $i = 1,2,\dots,n$ $m[i,i] = 0$. Чтобы вычислить $m[i,j]$ при $i < j$, воспользуемся свойством подструктуры оптимального решения, исследованным на первом этапе. Предположим, что в результате оптимальной расстановки скобок последовательность $A_iA_{i+1}\dots A_j$ разбивается между матрицами A_k и A_{k+1}, где $i \leqslant k < j$. Тогда величина $m[i,j]$ равна минимальной стоимости вычисления частных произведений $A_{i..k}$ и $A_{k+1..j}$ плюс стоимость умножения этих матриц друг на друга. Если вспомнить, что каждая матрица A_i имеет размеры $p_{i-1} \times p_i$, то нетрудно понять, что для вычисления произведения матриц $A_{i..k}A_{k+1..j}$ понадобится $p_{i-1}p_kp_j$ скалярных умножений. Таким образом, получаем:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

В этом рекурсивном уравнении предполагается, что значение k известно, но на самом деле это не так. Для выбора этого значения всего имеется $j - i$ возможностей, а именно — $k = i, i + 1, \ldots, j - 1$. Поскольку в оптимальной расстановке скобок необходимо использовать одно из этих значений k, все, что нужно сделать, — проверить все возможности и выбрать среди них лучшую. Таким образом, рекурсивное определение оптимальной расстановки скобок в произведении $A_i A_{i+1} \ldots A_j$ принимает вид:

$$m[i, j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{при } i < j. \end{cases} \qquad (15.12)$$

Величины $m[i, j]$ равны стоимостям оптимальных решений вспомогательных задач. Чтобы легче было проследить за процессом построения оптимального решения, обозначим через $s[i, j]$ значение k, при котором последовательность $A_i A_{i+1} \ldots A_j$ разбивается на две подпоследовательности в процессе оптимальной расстановки скобок. Таким образом, величина $s[i, j]$ равна значению k, такому что $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Третий этап: вычисление оптимальной стоимости

На данном этапе не составляет труда написать на основе рекуррентного соотношения (15.12) рекурсивный алгоритм для вычисления минимальной стоимости $m[1, n]$ для произведения $A_1 A_2 \ldots A_n$. Однако в разделе 15.3 мы сможем убедиться, что время работы этого алгоритма экспоненциально зависит от n, что ничем не лучше метода прямого перебора, при котором проверяется каждый способ расстановки скобок в произведении.

Важное наблюдение, которое можно сделать на данном этапе, заключается в том, что у нас относительно мало подзадач: по одной для каждого выбора величин i и j, удовлетворяющих неравенству $1 \leqslant i \leqslant j \leqslant n$, т.е. всего $\binom{n}{2} + n = \Theta(n^2)$. В рекурсивном алгоритме каждая вспомогательная задача может неоднократно встречаться в разных ветвях рекурсивного дерева. Такое свойство перекрытия вспомогательных подзадач — вторая отличительная черта применимости метода динамического программирования (первая отличительная черта — наличие оптимальной подструктуры).

Вместо того чтобы рекурсивно решать рекуррентное соотношение (15.12), выполним третий этап парадигмы динамического программирования и вычислим оптимальную стоимость путем построения таблицы в восходящем направлении. В описанной ниже процедуре предполагается, что размеры матриц A_i равны $p_{i-1} \times p_i$ $(i = 1, 2, \ldots, n)$. Входные данные представляют собой последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$; длина данной последовательности равна $length[p] = n + 1$. В процедуре используется вспомогательная таблица $m[1..n, 1..n]$ для хранения

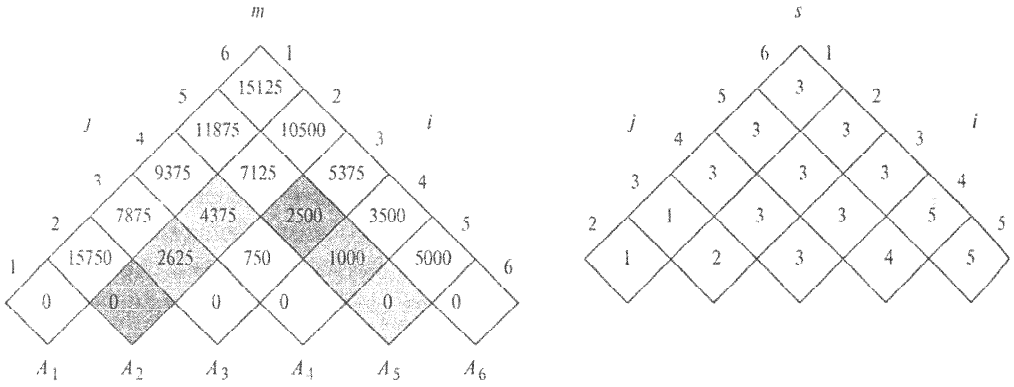
стоимостей $m[i,j]$ и вспомогательная таблица $s[1..n, 1..n]$, в которую заносятся индексы k, при которых достигаются оптимальные стоимости $m[i,j]$. Таблица s будет использоваться при построении оптимального решения.

Чтобы корректно реализовать восходящий подход, необходимо определить, с помощью каких записей таблицы будут вычисляться величины $m[i,j]$. Из уравнения (15.12) видно, что стоимость $m[i,j]$ вычисления произведения последовательности $j-i+1$ матриц зависит только от стоимости вычисления последовательностей матриц, содержащих менее $j-i+1$ матриц. Другими словами, при $k = i, i+1, \ldots, j-1$ матрица $A_{i..k}$ представляет собой произведение $k-i+1 < j-i+1$ матриц, а матрица $A_{k+1..j}$ — произведение $j-k < j-i+1$ матриц. Таким образом, в ходе выполнения алгоритма следует организовать заполнение таблицы m в порядке, соответствующем решению задачи о расстановке скобок в последовательностях матриц возрастающей длины:

```
MATRIX_CHAIN_ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n            ▷ l — длина последовательности
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m и s
```

Сначала в этом алгоритме (строки 2–3) вычисляются величины $m[i,i] \leftarrow 0$ $(i = 1, 2, \ldots, n)$, которые представляют собой минимальные стоимости для последовательностей единичной длины. Затем в первой итерации цикла в строках 4–12 с помощью рекуррентного соотношения (15.12) вычисляются величины $m[i, i+1]$ при $i = 1, 2, \ldots, n-1$ (минимальные стоимости для последовательностей длины $l = 2$). При втором проходе этого цикла вычисляются величины $m[i, i+2]$ при $i = 1, 2, \ldots, n-2$ (минимальные стоимости для последовательностей длины $l = 3$) и т.д. На каждом этапе вычисляемые в строках 9–12 величины $m[i,j]$ зависят только от уже вычисленных и занесенных в таблицу значений $m[i,k]$ и $m[k+1, j]$.

На рис. 15.3 описанный выше процесс проиллюстрирован для цепочки, состоящей из $n = 6$ матриц, размеры которых равны: $A_1 - 30 \times 35$, $A_2 - 35 \times 15$, $A_3 - 15 \times 5$, $A_4 - 5 \times 10$, $A_5 - 10 \times 20$, $A_6 - 20 \times 25$. Поскольку величины $m[i,j]$ опре-



Матрица	Размерность
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Рис. 15.3. Таблицы m и s, вычисленные процедурой MATRIX_CHAIN_ORDER для $n = 6$

делены только при $i \leqslant j$, используется только часть таблицы m, расположенная над ее главной диагональю. То же можно сказать и о таблице s. На рис. 15.3 таблица повернута так, чтобы ее главная диагональ была расположена горизонтально. В нижней части рисунка приведен список матриц, входящих в последовательность. На этой схеме легко найти минимальную стоимость $m[i, j]$ перемножения частичной последовательности матриц $A_i A_{i+1} \ldots A_j$. Она находится на пересечении линий, идущих от матрицы A_i вправо и вверх, и от матрицы A_j — влево и вверх. В каждой горизонтальной строке таблицы содержатся стоимости перемножения частных последовательностей, состоящих из одинакового количества матриц. В процедуре MATRIX_CHAIN_ORDER строки вычисляются снизу вверх, а элементы в каждой строке — слева направо. Величина $m[i, j]$ вычисляется с помощью произведений $p_{i-1} p_k p_j$ для $k = i, i+1, \ldots, j-1$ и величин внизу слева и внизу справа от $m[i, j]$. Из таблицы m видно, что минимальное количество скалярных умножений, необходимых для вычисления произведения шести матриц, равно $m[1, 6] = 15\,125$. Для пояснения приведем пример. При вычислении элемента $m[5, 2]$ использовались пары элементов, идущие от матриц A_2 и A_5

и имеющие одинаковый оттенок фона:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125 .$$

Несложный анализ структуры вложенных циклов в процедуре MATRIX_CHAIN_ORDER показывает, что время ее работы составляет $O(n^3)$. Глубина вложения циклов равна трем, а индексы в каждом из них (l, i и k) принимают не более $n - 1$ значений. В упражнении 15.2-4 предлагается показать, что время работы этого алгоритма фактически равно $\Omega(n^3)$. Для хранения таблиц m и s требуется объем, равный $\Theta(n^2)$. Таким образом, процедура MATRIX_CHAIN_ORDER намного эффективнее, чем метод перебора и проверки всевозможных способов расстановки скобок, время работы которого экспоненциально зависит от количества перемножаемых матриц.

Четвертый этап: конструирование оптимального решения

Несмотря на то, что в процедуре MATRIX_CHAIN_ORDER определяется оптимальное количество скалярных произведений, необходимых для вычисления произведения последовательности матриц, в нем не показано, как именно перемножаются матрицы. Оптимальное решение несложно построить с помощью информации, хранящейся в таблице s. В каждом элементе $s[i,j]$ хранится значение индекса k, где при оптимальной расстановке скобок в последовательности $A_i A_{i+1} \ldots A_j$ выполняется разбиение. Таким образом, нам известно, что оптимальное вычисление произведения матриц $A_{1..n}$ выглядит как $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Эти частные произведения матриц можно вычислить рекурсивно, поскольку элемент $s[1, s[1,n]]$ определяет матричное умножение, выполняемое последним при вычислении $A_{1..s[1,n]}$, а $s[s[1,n]+1, n]$ — последнее умножение при вычислении $A_{s[1,n]+1..n}$. Приведенная ниже рекурсивная процедура выводит оптимальный способ расстановки скобок в последовательности матриц $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, по таблице s, полученной в результате работы процедуры MATRIX_CHAIN_ORDER, и индексам i и j. В результате вызова процедуры PRINT_OPTIMAL_PARENS$(s, 1, n)$ выводится оптимальная расстановка скобок в последовательности $\langle A_1, A_2, \ldots, A_n \rangle$:

```
PRINT_OPTIMAL_PARENS(s, i, j)
1 if i = j
2    then print "A"_i
3    else print "("
```

```
4       PRINT_OPTIMAL_PARENS(s, i, s[i, j])
5       PRINT_OPTIMAL_PARENS(s, s[i, j] + 1, j)
6       print ")"
```

В примере, проиллюстрированном на рис. 15.3, в результате вызова процедуры PRINT_OPTIMAL_PARENS$(s, 1, 6)$ выводится строка $((A_1\,(A_2A_3))\,((A_4A_5)\,A_6))$.

Упражнения

15.2-1. Определите оптимальный способ расстановки скобок в произведении последовательности матриц, размеры которых равны $\langle 5, 10, 3, 12, 5, 50, 6\rangle$.

15.2-2. Разработайте рекурсивный алгоритм MATRIX_CHAIN_MULTIPLY(A, s, i, j), в котором оптимальным образом вычисляется произведение заданной последовательности матриц $\langle A_1A_2, \ldots, A_n\rangle$. На вход этого алгоритма также подаются индексы i и j, а также таблица s, вычисленная с помощью процедуры MATRIX_CHAIN_ORDER. (Начальный вызов этой процедуры выглядит следующим образом: MATRIX_CHAIN_MULTIPLY$(A, s, 1, n)$.)

15.2-3. Покажите с помощью метода подстановок, что решение рекуррентного соотношения (15.11) ведет себя как $\Omega\,(2^n)$.

15.2-4. Пусть $R\,(i, j)$ — количество обращений к элементу матрицы $m\,[i, j]$, которые выполняются в ходе вычисления других элементов этой матрицы в процедуре MATRIX_CHAIN_ORDER. Покажите, что полное количество обращений ко всем элементам равно:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} R\,(i, j) = \frac{n^3 - n}{3}.$$

(*Указание*: при решении может оказаться полезным уравнение (A.3).)

15.2-5. Покажите, что в полной расстановке скобок в n-элементном выражении используется ровно $n - 1$ пар скобок.

15.3 Элементы динамического программирования

Несмотря на рассмотренные в предыдущем разделе два примера, в которых применялся метод динамического программирования, возможно, вам все еще не совсем понятно, когда применим этот метод. В каких случаях задачу следует решать с помощью динамического программирования? В данном разделе рассматриваются два ключевых ингредиента, которые должны быть присущи задаче

оптимизации, чтобы к ней можно было применить метод динамического программирования: наличие оптимальной подструктуры и перекрывающиеся вспомогательные программы. Мы также рассмотрим разновидность метода, который называется *запоминанием* (memoization) и позволяет воспользоваться свойством перекрывающихся вспомогательных программ.

Оптимальная подструктура

Первый шаг решения задачи оптимизации методом динамического программирования состоит в том, чтобы охарактеризовать структуру оптимального решения. Напомним, что ***оптимальная подструктура*** проявляется в задаче в том случае, если в ее оптимальном решении содержатся оптимальные решения вспомогательных подзадач. Если в задаче выявлено наличие оптимальной подструктуры, это служит веским аргументом в пользу того, что к ней может быть применим метод динамического программирования. (Однако наличие этого свойства может также свидетельствовать о применимости жадных алгоритмов; см. главу 16.) В динамическом программировании оптимальное решение задачи конструируется из оптимальных решений вспомогательных задач. Следовательно, необходимо убедиться в том, что в число рассматриваемых подзадач входят те, которые используются в оптимальном решении.

Оптимальная подструктура была обнаружена в обеих задачах, которые исследовались в настоящей главе. В разделе 15.1 было установлено, что самый быстрый путь до рабочего места с номером j, расположенного на одном из двух конвейеров, включает в себя самый быстрый путь до рабочего места с номером $j-1$. В разделе 15.2 было продемонстрировано, что в оптимальной расстановке скобок, в результате которой последовательность матриц $A_i A_{i+1} \ldots A_j$ разбивается на подпоследовательности между матрицами A_k и A_{k+1}, содержатся оптимальные решения задач о расстановке скобок в подпоследовательностях $A_i A_{i+1} \ldots A_k$ и $A_{k+1} A_{k+2} \ldots A_j$.

Можно видеть, что поиск оптимальной подструктуры происходит по общему образцу.

1. На первом этапе следует показать, что в процессе решения задачи приходится делать выбор. В рассмотренных примерах это был выбор рабочего места на конвейере или выбор индекса, при котором разбивается последовательность матриц. После выбора остается решить одну или несколько вспомогательных задач.
2. На этом этапе мы исходим из того, что для поставленной задачи делается выбор, ведущий к оптимальному решению. Пока что не рассматривается, как именно следует делать выбор, а просто предполагается, что он найден.

3. Исходя из предположения предыдущего этапа, определяется, какие вспомогательные задачи получаются и как лучше охарактеризовать получающееся в результате пространство подзадач.
4. Показывается, что решения вспомогательных задач, возникающих в ходе оптимального решения задачи, сами должны быть оптимальными. Это делается методом "от противного": предполагая, что решение каждой вспомогательной задачи не оптимально, приходим к противоречию. В частности, путем "вырезания" неоптимального решения вспомогательной задачи и "вставки" оптимального демонстрируется, что можно получить лучшее решение исходной задачи, а это противоречит предположению, что уже имеется оптимальное решение. Если вспомогательных задач несколько, они обычно настолько похожи, что описанный выше способ рассуждения, примененный к одной из вспомогательных задач, легко модифицируется для остальных.

Характеризуя пространство подзадач, постарайтесь придерживаться такого практического правила: попытайтесь, чтобы это пространство было как можно проще, а потом расширьте его до необходимых пределов. Например, пространство в подзадачах, возникающих в задаче о составлении расписания работы конвейера, было образовано самым быстрым путем прохождения шасси от начала конвейера до рабочего места $S_{1,j}$ или $S_{2,j}$. Это подпространство оказалось вполне подходящим, и не возникло необходимости его расширять.

Теперь предположим, что в задаче о перемножении цепочки матриц производится попытка ограничить пространство подзадач произведением вида $A_1 A_2 \ldots A_j$. Как и раньше, оптимальная расстановка скобок должна разбивать произведение между матрицами A_k и A_{k+1} для некоторого индекса $1 \leqslant k < j$. Если k не всегда равно $j - 1$, мы обнаружим, что возникают вспомогательные задачи в виде $A_1 A_2 \ldots A_k$ и $A_{k+1} A_{k+2} \ldots A_j$ и что последняя подзадача не является подзадачей вида $A_1 A_2 \ldots A_j$. В этой задаче необходима возможность изменения обоих концов последовательности, т.е. чтобы во вспомогательной задаче $A_i A_{i+1} \ldots A_j$ могли меняться оба индекса, — и i, и j.

Оптимальная подструктура изменяется в области определения задачи в двух аспектах:

1. количество вспомогательных задач, которые используются при оптимальном решении исходной задачи;
2. количество выборов, возникающих при определении того, какая вспомогательная задача (задачи) используется в оптимальном решении.

В задаче о составлении расписания работы конвейера в оптимальном решении используется только одна вспомогательная задача, и для поиска оптимального решения необходимо рассмотрение двух вариантов. Чтобы найти самый быстрый

путь через рабочее место $S_{i,j}$, мы используем *либо* самый быстрый путь через рабочее место $S_{1,j-1}$, *либо* самый быстрый путь через рабочее место $S_{2,j-1}$. Каждый из вариантов представляет одну вспомогательную задачу, для которой необходимо найти оптимальное решение. Перемножение вспомогательной цепочки матриц $A_i A_{i+1} \ldots A_j$ — пример, в котором возникают две вспомогательные задачи и $j - i$ вариантов выбора. Если задана матрица A_k, у которой происходит разбиение произведения, то возникают две вспомогательные задачи — расстановка скобок в подпоследовательностях $A_i A_{i+1} \ldots A_k$ и $A_{k+1} A_{k+2} \ldots A_j$, причем для *каждой* из них необходимо найти оптимальное решение. Как только оптимальные решения вспомогательных задач найдены, выбирается один из $j - i$ вариантов индекса k.

Если говорить неформально, время работы алгоритма динамического программирования зависит от произведения двух множителей: общего количества вспомогательных задач и количества вариантов выбора, возникающих в каждой вспомогательной задаче. При составлении расписания работы сборочного конвейера у нас всего возникало $\Theta(n)$ вспомогательных задач и лишь два варианта выбора, которые нужно было проверить. В результате получается, что время работы алгоритма ведет себя как $\Theta(n)$. При перемножении матриц всего возникало $\Theta(n^2)$ вспомогательных задач, и в каждой из них — не более $n - 1$ вариантов выбора, поэтому время работы алгоритма ведет себя как $O(n^3)$.

Оптимальная подструктура используется в динамическом программировании в восходящем направлении. Другими словами, сначала находятся оптимальные решения вспомогательных задач, после чего определяется оптимальное решение поставленной задачи. Поиск оптимального решения задачи влечет за собой необходимость выбора одной из вспомогательных задач, которые будут использоваться в решении полной задачи. Стоимость решения задачи обычно определяется как сумма стоимостей решений вспомогательных задач и стоимости, которая затрачивается на определение правильного выбора. Например, при составлении расписания работы сборочного конвейера сначала решались вспомогательные задачи по определению самого быстрого пути через рабочие места $S_{1,j-1}$ и $S_{2,j-1}$, а потом одна из них выбиралась в качестве предыдущей для рабочего места $S_{i,j}$. Стоимость, которая относится к самому выбору, зависит от того, происходит ли переход от одного конвейера к другому между рабочими местами $j - 1$ и j. Если шасси остается на одном и том же конвейере, то эта стоимость равна $a_{i,j}$, а если осуществляется переход от одного конвейера к другому — она равна $t_{i',j-1} + a_{i,j}$, где $i' \neq i$. В задаче о перемножении цепочки матриц сначала был определен оптимальный способ расстановки скобок во вспомогательной цепочке $A_i A_{i+1} \ldots A_j$, а потом была выбрана матрица A_k, у которой выполняется разбиение произведения. Стоимость, которая относится к самому выбору, выражается членом $p_{i-1} p_k p_j$.

В главе 16 рассматриваются жадные алгоритмы, имеющие много общего с динамическим программированием. В частности, задачи, к которым применимы

жадные алгоритмы, обладают оптимальной подструктурой. Одно из характерных различий между жадными алгоритмами и динамическим программированием заключается в том, что в жадных алгоритмах оптимальная подструктура используется в нисходящем направлении. Вместо того чтобы находить оптимальные решения вспомогательных задач с последующим выбором одного из возможных вариантов, в жадных алгоритмах сначала делается выбор, который выглядит наилучшим на текущий момент, а потом решается возникшая в результате вспомогательная задача.

Некоторые тонкости

Следует быть особенно внимательным в отношении вопроса о применимости оптимальной подструктуры, когда она отсутствует в задаче. Рассмотрим такие две задачи, в которых имеется ориентированный граф $G = (V, E)$ и вершины $u, v \in V$.

Задача о кратчайшем невзвешенном пути[1]. Нужно найти путь от вершины u к вершине v, состоящий из минимального количества ребер. Этот путь обязан быть простым, поскольку в результате удаления из него цикла получается путь, состоящий из меньшего количества ребер.

Задача о самом длинном невзвешенном пути. Определите простой путь от вершины u к вершине v, состоящий из максимального количества ребер. Требование простоты весьма важно, поскольку в противном случае можно проходить по одному и тому же циклу сколько угодно раз, получая в результате путь, состоящий из произвольного количества ребер.

В задаче о кратчайшем пути возникает оптимальная подструктура. Это можно показать с помощью таких рассуждений. Предположим, что $u \neq v$ и задача является нетривиальной. В этом случае любой путь p от u к v должен содержать промежуточную вершину, скажем, w. (Заметим, что вершина w может совпадать с вершиной u или v.) Поэтому путь $u \overset{p}{\rightsquigarrow} v$ можно разложить на вспомогательные пути $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$. Очевидно, что количество ребер, входящих в путь p, равно сумме числа ребер в путях p_1 и p_2. Утверждается, что если путь p от вершины u к вершине v оптимальный (т.е. кратчайший), то p_1 должен быть кратчайшим путем от вершины u к вершине w. Почему? Аргумент такой: если бы существовал другой путь, соединяющий вершины u и v и состоящий из меньшего количества ребер, чем p_1, скажем, p_1', можно было бы вырезать путь p_1 и вставить путь p_1', в результате чего получился бы путь $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, состоящий из меньшего количества ребер, чем путь p. А это противоречит предположению об оптимальности пути p. Аналогично, p_2 должен быть кратчайшим путем от вершины w

[1] Термин "невзвешенный" используется, чтобы отличать эту задачу от той, при которой находится кратчайший путь на взвешенных ребрах, с которой мы ознакомимся в главах 24 и 25. Для решения задачи о невзвешенном пути можно использовать метод поиска в ширину, описанный в главе 26

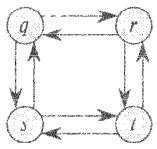


Рис. 15.4. Пример, демонстрирующий отсутствие оптимальной подструктуры в задаче о поиске самого длинного простого пути

к вершине v. Таким образом, кратчайший путь от вершины u к вершине v можно найти, рассмотрев все промежуточные вершины w, отыскав кратчайший путь от вершины u к вершине w и кратчайший путь от вершины w к вершине v, и выбрав промежуточную вершину w, через которую весь путь окажется кратчайшим. В разделе 25.2 один из вариантов этой оптимальной подструктуры используется для поиска кратчайшего пути между всеми парами вершин во взвешенном ориентированном графе.

Напрашивается предположение, что в задаче поиска самого длинного простого невзвешенного пути тоже проявляется оптимальная подструктура. В конце концов, если разложить самый длинный простой путь $u \overset{p}{\rightsquigarrow} v$ на вспомогательные пути $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, то разве не должен путь p_1 быть самым длинным простым путем от вершины u к вершине w, а путь p_2 — самым длинным простым путем от вершины w к вершине v? Оказывается, нет! Пример такой ситуации приведен на рис. 15.4. Рассмотрим путь $q \to r \to t$, который является самым длинным простым путем от вершины q к вершине t. Является ли путь $q \to r$ самым длинным путем от вершины q к вершине r? Нет, поскольку простой путь $q \to s \to t \to r$ длиннее. Является ли путь $r \to t$ самым длинным путем от вершины r к вершине t? Снова нет, поскольку простой путь $r \to q \to s \to t$ длиннее.

Этот пример демонстрирует, что в задаче о самых длинных простых путях не только отсутствует оптимальная подструктура, но и не всегда удается составить "законное" решение задачи из решений вспомогательных задач. Если сложить самые длинные простые пути $q \to s \to t \to r$ и $r \to q \to s \to t$, то получим путь $q \to s \to t \to r \to q \to s \to t$, который не является простым. Итак, на деле оказывается, что в задаче о поиске самого длинного невзвешенного пути не возникает никаких оптимальных подструктур. Для этой задачи до сих пор не найдено ни одного эффективного алгоритма, работающего по принципу динамического программирования. Фактически, это NP-полная задача, что означает — как будет показано в главе 34 — что вряд ли ее можно решить в течение полиномиального времени.

Что можно сказать по поводу структуры самого длинного простого пути, так отличающейся от структуры самого короткого пути? Несмотря на то, что

в решениях задач о поиске и самого короткого, и самого длинного пути возникают две вспомогательные задачи, вспомогательные задачи при определении самого длинного пути не являются ***независимыми***, в то время как в задаче о кратчайшем пути они независимы. Что подразумевается под независимостью вспомогательных задач? Подразумевается, что решение одной вспомогательной задачи не влияет на решение другой вспомогательной задачи, возникающей в той же задаче. В примере, проиллюстрированном на рис. 15.4, рассматривается задача о поиске самого длинного простого пути между вершинами q и t, в которой возникают две вспомогательные задачи: определение самых длинных простых путей между вершинами q и r и между вершинами r и t. Для первой из этих вспомогательных задач выбирается путь $q \to s \to t \to r$, в котором используются вершины s и t. Во второй вспомогательной задаче мы не сможем больше использовать эти вершины, поскольку в процессе комбинирования решений этих двух вспомогательных задач получился бы путь, который не является простым. Если во второй задаче нельзя использовать вершину t, то ее вообще невозможно будет решить, поскольку эта вершина должна быть в найденном пути, и это не та вершина, в которой "соединяются" решения вспомогательных задач (этой вершиной является r). Использование вершин s и t в решении одной вспомогательной задачи приводит к невозможности их применения в решении другой вспомогательной задачи. Однако для ее решения необходимо использовать хотя бы одну из них, а в оптимальное решение данной вспомогательной задачи входят обе эти вершины. Поэтому эти вспомогательные задачи не являются независимыми. Другими словами, использование ресурсов в решении одной вспомогательной задачи (в качестве ресурсов выступают вершины) делают их недоступными в другой вспомогательной задаче.

Почему же вспомогательные задачи остаются независимыми при поиске самого короткого пути? Ответ такой: по самой природе поставленной задачи возникающие в ней вспомогательные задачи не используют одни и те же ресурсы. Утверждается, что если вершина w находится на кратчайшем пути p от вершины u к вершине v, то можно соединить *любой* кратчайший путь $u \overset{p_1}{\rightsquigarrow} w$ с *любым* кратчайшим путем $w \overset{p_2}{\rightsquigarrow} v$, получив в результате самый короткий путь от вершины u к вершине v. Мы уверены в том, что пути p_1 и p_2 не содержат ни одной общей вершины, кроме w. Почему? Предположим, что имеется еще некоторая вершина $x \neq w$, принадлежащая путям p_1 и p_2, так что путь p_1 можно разложить как $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$, а путь p_2 — как $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. В силу оптимальной подструктуры этой задачи количество ребер в пути p равно сумме количеств ребер в путях p_1 и p_2. Предположим, что путь p содержит e ребер. Теперь построим путь $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ от вершины u к вершине v. В этом пути содержится не более $e - 2$ вершин, что противоречит предположению о том, что путь p — кратчайший. Таким образом,

мы убедились, что вспомогательные задачи, возникающие в задаче поиска кратчайшего пути, являются независимыми.

Подзадачи, возникающие в задачах, которые рассматриваются в разделах 15.1 и 15.2, являются независимыми. При перемножении цепочки матриц, вспомогательные задачи заключались в перемножении подцепочек $A_iA_{i+1}\dots A_k$ и $A_{k+1}A_{k+2}\dots A_j$. Это непересекающиеся подцепочки, которые не могут содержать общих матриц. При составлении расписания конвейера для определения самого быстрого пути через рабочее место $S_{i,j}$ осуществлялся поиск самого быстрого пути через рабочие места $S_{1,j-1}$ и $S_{2,j-1}$. Поскольку решение (т.е. самый быстрый путь через рабочее место $S_{i,j}$) содержит только одно из решений этих подзадач, данная подзадача автоматически независима от всех других подзадач, использующихся в этом решении.

Перекрытие вспомогательных задач

Вторая составляющая часть, наличие которой необходимо для применения динамического программирования, заключается в том, что пространство вспомогательных задач должно быть "небольшим" в том смысле, что в результате выполнения рекурсивного алгоритма одни и те же вспомогательные задачи решаются снова и снова, а новые вспомогательные задачи не возникают. Обычно полное количество различающихся вспомогательных задач выражается как полиномиальная функция от объема входных данных. Когда рекурсивный алгоритм снова и снова обращается к одной и той же задаче, говорят, что задача оптимизации содержит ***перекрывающиеся вспомогательные задачи*** (overlapping subproblems)[2]. В отличие от описанной выше ситуации, в задачах, решаемых с помощью алгоритма разбиения, на каждом шаге рекурсии обычно возникают полностью новые задачи. В алгоритмах динамического программирования обычно используется преимущество, заключающееся в наличии перекрывающихся вспомогательных задач. Это достигается путем однократного решения каждой вспомогательной задачи с последующим сохранением результатов в таблице, где при необходимости их можно будет найти за фиксированное время.

В разделе 15.1 было показано, что в рекурсивном решении задачи о составлении расписания работы конвейера осуществляется 2^{n-j} обращений к $f_i[j]$ для $j = 1, 2, \dots, n$. Путем использования таблицы, содержащей результаты решений вспомогательных задач, экспоненциальное время работы алгоритма удается свести к линейному.

[2]Может показаться странным, что вспомогательные задачи, использующиеся в динамическом программировании, являются и независимыми, и перекрывающимися. Эти требования могут показаться противоречащими друг другу, однако это не так, поскольку они относятся к разным понятиям. Две вспомогательные задачи одной и той же задачи независимы, если в них не используются общие ресурсы. Две вспомогательные задачи перекрываются, если на самом деле речь идет об одной и той же вспомогательной задаче, возникающей в разных задачах.

Чтобы подробнее проиллюстрировать свойство перекрывания вспомогательных задач, еще раз обратимся к задаче о перемножении цепочки матриц. Возвратимся к рис. 15.3. Обратите внимание, что процедура MATRIX_CHAIN_ORDER в процессе решения вспомогательных задач в более высоких строках постоянно обращается к решениям вспомогательных задач в более низких строках. Например, к элементу $m[3,4]$ осуществляется 4 обращения: при вычислении элементов $m[2,4]$, $m[1,4]$, $m[3,5]$ и $m[3,6]$. Если бы элемент $m[3,4]$ каждый раз приходилось каждый раз вычислять заново, а не просто находить в таблице, это привело бы к значительному увеличению времени работы. Чтобы продемонстрировать это, рассмотрим приведенную ниже (неэффективную) рекурсивную процедуру, в которой определяется величина $m[i,j]$, т.е. минимальное количество скалярных умножений, необходимых для вычисления произведения цепочки матриц $A_{i..j} = A_iA_{i+1}\ldots A_j$. Эта процедура основана непосредственно на рекуррентном соотношении (15.12):

```
RECURSIVE_MATRIX_CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        do q ← RECURSIVE_MATRIX_CHAIN(p, i, k)
                + RECURSIVE_MATRIX_CHAIN(p, k + 1, j)
                + p_{i−1} p_k p_j
6           if q < m[i, j]
7              then m[i, j] ← q
8  return m[i, j]
```

На рис. 15.5 показано рекурсивное дерево, полученное в результате вызова процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, 4)$. Каждый его узел обозначен величинами параметров i и j. Вычисления, которые производятся в части дерева, выделенной серым фоном, заменяются в процедуре MEMOIZED_MATRIX_CHAIN$(p, 1, 4)$. Обратите внимание, что некоторые пары значений встречаются много раз.

Можно показать, что время вычисления величины $m[1,n]$ в этой рекурсивной процедуре, как минимум, экспоненциально по n. Обозначим через $T(n)$ время, которое потребуется процедуре RECURSIVE_MATRIX_CHAIN для вычисления оптимального способа расстановки скобок в цепочке, состоящей из n матриц. Если считать, что выполнение каждой из строк 1–2 и 6–7 требует как минимум единичного интервала времени, то мы получим такое рекуррентное соотношение:

$$T(1) \geqslant 1,$$

$$T(n) \geqslant 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{при } n > 1.$$

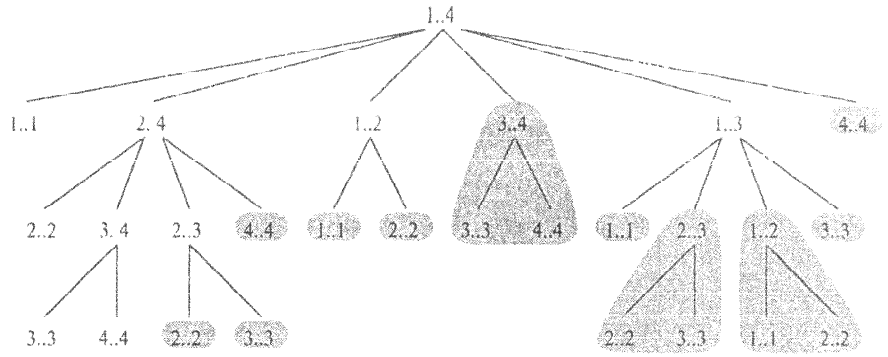


Рис. 15.5. Рекурсивное дерево, соответствующее вызову процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, 4)$

Если заметить, что при $i = 1, 2, \ldots, n-1$ каждое слагаемое $T(i)$ один раз появляется как $T(k)$ и один раз как $T(n-k)$, и просуммировать $n-1$ единиц с той, которая стоит слева от суммы, то рекуррентное соотношение можно переписать в виде

$$T(n) \geqslant 2\sum_{i=1}^{n-1} T(i) + n. \tag{15.13}$$

Докажем с помощью метода подстановок, что $T(n) = \Omega(2^n)$. В частности, покажем, что для всех $n \geqslant 1$ справедливо соотношение $T(n) \geqslant 2^{n-1}$. Очевидно, что базисное соотношение индукции выполняется, поскольку $T(1) \geqslant 1 = 2^0$. Далее, по методу математической индукции для $n \geqslant 2$ имеем

$$T(n) \geqslant 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n = 2\left(2^{n-1} - 1\right) + n = (2^n - 2) + n \geqslant 2^{n-1}.$$

На этом доказательство завершено. Таким образом, полное количество вычислений, которые выполняются при вызове процедуры RECURSIVE_MATRIX_CHAIN$(p, 1, n)$, как минимум, экспоненциально зависит от n.

Сравним этот нисходящий рекурсивный алгоритм с восходящим алгоритмом, построенным по принципу динамического программирования. Последний работает эффективнее, поскольку в нем используется свойство перекрывающихся вспомогательных задач. Всего возникает $\Theta\left(n^2\right)$ различных вспомогательных задач, и в алгоритме, основанном на принципах динамического программирования, каждая из них решается ровно по одному разу. В рекурсивном же алгоритме каждую вспомогательную задачу необходимо решать всякий раз, когда она возникает в рекурсивном дереве. Каждый раз, когда рекурсивное дерево для обычного рекурсивного решения задачи несколько раз включает в себя одну и ту же вспомогательную

задачу, полезно проверить, нельзя ли воспользоваться динамическим программированием.

Построение оптимального решения

На практике зачастую мы сохраняем сведения о том, какой выбор делается в каждой вспомогательной программе, так что впоследствии нам не надо дополнительно решать задачу восстановления этой информации. В задаче о составлении расписания работы сборочного конвейера в элементе $l_i[j]$ сохранялся номер рабочего места, предшествующего рабочему месту $S_{i,j}$ на самом быстром пути через это рабочее место. Другая возможность — заполнить всю таблицу $f_i[j]$. В этом случае можно было бы определить, какое рабочее место предшествует рабочему месту $S_{1,j}$ на самом быстром пути, проходящем через рабочее место $S_{1,j}$, но на это потребовалось бы больше усилий. Если $f_1[j] = f_1[j-1] + a_{1,j}$, то на самом быстром пути через рабочее место $S_{1,j}$ ему предшествует рабочее место $S_{1,j-1}$. В противном случае должно выполняться соотношение $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$, и рабочему месту $S_{1,j}$ предшествует рабочее место $S_{2,j-1}$. В задаче о составлении расписания работы сборочного конвейера на восстановление информации о предшествующем рабочем месте необходимо время, равное $O(1)$ (на каждое рабочее место), даже если отсутствует таблица $l_i[j]$.

Однако в задаче о перемножении цепочки матриц в процессе воссоздания оптимального решения, благодаря таблице $s[i,j]$ удается значительно уменьшить объем работы. Предположим, что таблица $s[i,j]$ не поддерживается, а заполняется только таблица $m[i,j]$, в которой содержатся оптимальные стоимости вспомогательных задач. При выборе вспомогательной задачи, использующейся в оптимальном решении задачи о расстановке скобок в произведении $A_iA_{i+1}\ldots A_j$, всего имеется $j-i$ вариантов выбора, и это количество не является константой. Поэтому на восстановление сведений о том, какие вспомогательные задачи выбираются в процессе решения данной задачи, потребуется время, равное $\Theta(j-i) = \omega(1)$. Сохраняя в элементе $s[i,j]$ индекс, где выполняется разбиение произведения $A_iA_{i+1}\ldots A_j$, данные о каждом выборе можно восстановить за время $O(1)$.

Запоминание

Одна из вариаций динамического программирования часто обеспечивает ту же степень эффективности обычного подхода динамического программирования, но при соблюдении нисходящей стратегии. Идея заключается в том, чтобы ***запомнить*** (memoize[3]) обычный неэффективный рекурсивный алгоритм. Как и при

[3]В данном случае это не опечатка, по-английски этот термин пишется именно так. Как поясняет в примечании автор книги, смысл не просто в том, чтобы запомнить информацию (memorize), а в том, чтобы вскоре ею воспользоваться как памяткой (memo). — *Прим. ред.*

обычном динамическом программировании, поддерживается таблица, содержащая решения вспомогательных задач. Однако структура, управляющая заполнением таблицы, больше напоминает рекурсивный алгоритм.

В рекурсивном алгоритме с запоминанием поддерживается элемент таблицы для решения каждой вспомогательной задачи. Изначально в каждом таком элементе таблицы содержится специальное значение, указывающее на то, что данный элемент еще не заполнен. Если в процессе выполнения рекурсивного алгоритма вспомогательная задача встречается впервые, ее решение вычисляется и заносится в таблицу. Каждый раз, когда снова будет встречаться эта вспомогательная задача, в таблице находится и возвращается ее решение[4].

Ниже приведена версия процедуры RECURSIVE_MATRIX_CHAIN с запоминанием:

```
MEMOIZED_MATRIX_CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3       do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP_CHAIN(p, 1, n)

LOOKUP_CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else for k ← i to j − 1
6             do q ← LOOKUP_CHAIN(p, i, k)
                     + LOOKUP_CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                if q < m[i, j]
8                   then m[i, j] ← q
9  return m[i, j]
```

В процедуре MEMOIZED_MATRIX_CHAIN, как и в процедуре MATRIX_CHAIN_ORDER, поддерживается таблица $m[1..n, 1..n]$, состоящая из вычисленных значений $m[i, j]$, которые представляют собой минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$. Изначально каждый элемент таблицы содержит значение ∞, указывающее на то, что данный элемент еще должен быть заполнен. Если при вызове процедуры LOOKUP_CHAIN(p, i, j) вы-

[4] В этом подходе предполагается, что известен набор параметров всех возможных вспомогательных задач и установлено соответствие между ячейками таблицы и вспомогательными задачами. Другой подход состоит в том, чтобы организовать запоминание с помощью хеширования, в котором в качестве ключей выступают параметры вспомогательных задач.

полняется условие $m[i,j] < \infty$ (строка 1), эта процедура просто возвращает ранее вычисленную стоимость $m[i,j]$ (строка 2). В противном случае эта стоимость вычисляется так же, как и в процедуре RECURSIVE_MATRIX_CHAIN, сохраняется в элементе $m[i,j]$, после чего возвращается. (Удобство использования значения ∞ для незаполненных элементов таблицы объясняется тем, что это же значение используется для инициализации элементов $m[i,j]$ в строке 3 процедуры RECURSIVE_MATRIX_CHAIN.) Таким образом, процедура LOOKUP_CHAIN(p,i,j) всегда возвращает значение $m[i,j]$, но оно вычисляется лишь в том случае, если это первый вызов данной процедуры с параметрами i и j.

Рис. 15.5 иллюстрирует, насколько эффективнее процедура MEMOIZED_MATRIX_CHAIN расходует время по сравнению с процедурой RECURSIVE_MATRIX_CHAIN. Затененные поддеревья представляют значения, которые вычисляются только один раз. При возникновении повторной потребности в этих значениях осуществляется их поиск в хранилище.

Время работы процедуры MEMOIZED_MATRIX_CHAIN, как и время выполнения алгоритма MATRIX_CHAIN_ORDER, построенного по принципу динамического программирования, равно $O(n^3)$. Каждая ячейка таблицы (а всего их $\Theta(n^2)$) однократно инициализируется в строке 4 процедуры MEMOIZED_MATRIX_CHAIN. Все вызовы процедуры LOOKUP_CHAIN можно разбить на два типа:

1. вызовы, в которых справедливо соотношение $m[i,j] = \infty$ и выполняются строки 3–9;
2. вызовы, в которых выполняется неравенство $m[i,j] < \infty$ и просто происходит возврат в строке 2.

Всего вызовов первого типа насчитывается $\Theta(n^2)$, по одному на каждый элемент таблицы. Все вызовы второго типа осуществляются как рекурсивные обращения из вызовов первого типа. Всякий раз, когда в некотором вызове процедуры LOOKUP_CHAIN выполняются рекурсивные обращения, общее их количество равно $O(n)$. Поэтому в общем итоге производится $O(n^3)$ вызовов второго типа, причем на каждый из них расходуется время $O(1)$. На каждый вызов первого типа требуется время $O(n)$ плюс время, затраченное на рекурсивные обращения в данном вызове первого типа. Поэтому общее время равно $O(n^3)$. Таким образом, запоминание преобразует алгоритм, время работы которого равно $\Omega(2^n)$, в алгоритм со временем работы $O(n^3)$.

В итоге получается, что задачу об оптимальном способе перемножения цепочки матриц можно решить либо с помощью алгоритма с запоминанием, работающего в нисходящем направлении, либо с помощью динамического программирования в восходящем направлении. При этом в обоих случаях потребуется время, равное $O(n^3)$. Оба метода используют преимущество, возникающее благодаря перекрыванию вспомогательных задач. Всего возникает только $\Theta(n^2)$ различных вспомогательных задач, и в каждом из описанных выше методов решение каждой

из них вычисляется один раз. Если не применять запоминание, то время работы обычного рекурсивного алгоритма станет экспоненциальным, поскольку уже решенные задачи придется неоднократно решать заново.

В общем случае, если все вспомогательные задачи необходимо решить хотя бы по одному разу, восходящий алгоритм, построенный по принципу динамического программирования, обычно работает быстрее, чем нисходящий алгоритм с запоминанием. Причины — отсутствие непроизводительных затрат на рекурсию и их сокращение при поддержке таблицы. Более того, в некоторых задачах после определенных исследований удается сократить время доступа к таблице или занимаемый ею объем. Если же можно обойтись без решения некоторых вспомогательных задач, содержащихся в пространстве подзадач данной задачи, запоминание результатов решений обладает тем преимуществом, что решаются только необходимые вспомогательные задачи.

Упражнения

15.3-1. Как эффективнее определить оптимальное количество скалярных умножений в задаче о перемножении цепочки матриц: перечислить все способы расстановки скобок в произведении матриц и вычислить количество скалярных умножений в каждом из них или запустить процедуру RECURSIVE_MATRIX_CHAIN? Обоснуйте ответ.

15.3-2. Нарисуйте рекурсивное дерево для процедуры MERGE_SORT, описанной в разделе 2.3.1, работающей с массивом из 16 элементов. Объясните, почему для повышения производительности работы хорошего алгоритма разбиения, такого как MERGE_SORT, использование запоминания не будет эффективным.

15.3-3. Рассмотрим разновидность задачи о перемножении цепочки матриц, цель которой — так расставить скобки в последовательности матриц, чтобы количество скалярных умножений стало не минимальным, а максимальным. Проявляется ли в этой задаче оптимальная подструктура?

15.3-4. Опишите, какие перекрывающиеся вспомогательные задачи возникают при составлении расписания работы конвейера.

15.3-5. Согласно сделанному выше утверждению, в динамическом программировании сначала решаются вспомогательные задачи, а потом выбираются те из них, которые будут использованы в оптимальном решении задачи. Профессор утверждает, что не всегда необходимо решать все вспомогательные задачи, чтобы найти оптимальное решение. Он выдвигает предположение, что оптимальное решение задачи о перемножении цепочки матриц можно найти, всегда выбирая матрицу A_k, после которой следует разбивать произведение $A_i A_{i+1} \ldots A_j$ (индекс k выбирается

так, чтобы минимизировать величину $p_{i-1}p_kp_j$) *перед* решением вспомогательных задач. Приведите пример задачи о перемножении цепочки матриц, в котором такой жадный подход приводит к решению, отличному от оптимального.

15.4 Самая длинная общая подпоследовательность

В биологических приложениях часто возникает необходимость сравнить ДНК двух (или большего количества) различных организмов. Стандартная ДНК состоит из последовательности молекул, которые называются ***основаниями*** (bases). К этим молекулам относятся: аденин (adenine), гуанин (guanine), цитозин (cytosine) и тимин (thymine). Если обозначить каждое из перечисленных оснований его начальной буквой латинского алфавита, то стандартную ДНК можно представить в виде строки, состоящей из конечного множества элементов $\{A, C, G, T\}$. (Определение строки см. в приложении В.) Например, ДНК одного организма может иметь вид S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, а ДНК другого – S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Одна из целей сравнения двух ДНК состоит в том, чтобы выяснить степень их сходства. Это один из показателей того, насколько тесно связаны между собой два организма. Степень подобия можно определить многими различными способами. Например, можно сказать, что два кода ДНК подобны, если один из них является подстрокой другого. (Алгоритмы, с помощью которых решается эта задача, исследуются в главе 32.) В нашем примере ни S_1, ни S_2 не является подстрокой другой ДНК. В этом случае можно сказать, что две цепочки молекул подобны, если для преобразования одной из них в другую потребовались бы только небольшие изменения. (Такой подход рассматривается в задаче 15-3.) Еще один способ определения степени подобия последовательностей S_1 и S_2 заключается в поиске третьей последовательности S_3, основания которой имеются как в S_1, так и в S_2; при этом они следуют в одном и том же порядке, но не обязательно одно за другим. Чем длиннее последовательность S_3, тем более схожи последовательности S_1 и S_2. В рассматриваемом примере самая длинная последовательность S_3 имеет вид GTCGTCGGAAGCCGGCCGAA.

Последнее из упомянутых выше понятий подобия формализуется в виде задачи о самой длинной общей подпоследовательности. Подпоследовательность данной последовательности — это просто данная последовательность, из которой удалили ноль или больше элементов. Формально последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ является ***подпоследовательностью*** (subsequence) последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$, если существует строго возрастающая последовательность $\langle i_1, i_2, \ldots, i_k \rangle$ индексов X, такая что для всех $j = 1, 2, \ldots, k$ выполняется соотношение $x_{i_j} = z_j$. Например, $Z = \langle B, C, D, B \rangle$ — подпоследовательность

последовательности $X = \langle A, B, C, B, D, A, B \rangle$, причем соответствующая ей последовательность индексов имеет вид $\langle 2, 3, 5, 7 \rangle$.

Говорят, что последовательность Z является ***общей подпоследовательностью*** (common subsequence) последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Например, если $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$, то последовательность $\langle B, C, A \rangle$ — общая подпоследовательность X и Y. Однако последовательность $\langle B, C, A \rangle$ — не *самая длинная* общая подпоследовательность X и Y, поскольку ее длина равна 3, а длина последовательности $\langle B, C, B, A \rangle$, которая тоже является общей подпоследовательностью X и Y, равна 4. Последовательность $\langle B, C, B, A \rangle$ — *самая длинная общая подпоследовательность* (longest common subsequence, LCS) последовательностей X и Y, как и последовательность $\langle B, D, A, B \rangle$, поскольку не существует общей подпоследовательности длиной 5 элементов или более.

В ***задаче о самой длинной общей подпоследовательности*** даются две последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$, и требуется найти общую подпоследовательность X и Y максимальной длины. В данном разделе показано, что эта задача эффективно решается методом динамического программирования.

Этап 1: характеристика самой длинной общей подпоследовательности

Решение задачи поиска самой длинной общей подпоследовательности "в лоб" заключается в том, чтобы пронумеровать все подпоследовательности последовательности X и проверить, являются ли они также подпоследовательностями Y, пытаясь отыскать при этом самую длинную из них. Каждая подпоследовательность последовательности X соответствует подмножеству индексов $\{1, 2, \ldots, m\}$ последовательности X. Всего имеется 2^m подпоследовательностей последовательности X, поэтому время работы алгоритма, основанного на этом подходе, будет экспоненциально зависеть от размера задачи, и для длинных последовательностей он становится непригодным.

Однако задача поиска самой длинной общей подпоследовательности обладает оптимальной подструктурой. Этот факт доказывается в сформулированной ниже теореме. Как мы увидим, естественно возникающие классы вспомогательных задач соответствуют парам "префиксов" двух входных последовательностей. Дадим точное определение этого понятия: i-м ***префиксом*** последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ для $i = 1, 2, \ldots, m$ является подпоследовательность $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. Например, если $X = \langle A, B, C, B, D, A, B \rangle$, то $X_4 = \langle A, B, C, B \rangle$, а X_0 — пустая последовательность.

Теорема 15.1 (Оптимальная подструктура LCS). Пусть имеются последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$, а $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — их самая длинная общая подпоследовательность.

1. Если $x_m = y_n$, то $z_k = x_m = y_n$ и Z_{k-1} — LCS последовательностей X_{m-1} и Y_{n-1}.
2. Если $x_m \neq y_n$, то из $z_k \neq x_m$ следует, что Z — самая длинная общая подпоследовательность последовательностей X_{m-1} и Y.
3. Если $x_m \neq y_n$, то из $z_k \neq y_n$ следует, что Z — самая длинная общая подпоследовательность последовательностей X и Y_{n-1}.

Доказательство. (1) Если бы выполнялось соотношение $z_k \neq x_m$, то к последовательности Z можно было бы добавить элемент $x_m = y_n$, в результате чего получилась бы общая подпоследовательность последовательностей X и Y длиной $k + 1$, а это противоречит предположению о том, что Z — *самая длинная* общая подпоследовательность последовательностей X и Y. Таким образом, должно выполняться соотношение $z_k = x_m = y_n$. Таким образом, префикс Z_{k-1} — общая подпоследовательность последовательностей X_{m-1} и Y_{n-1} длиной $k - 1$. Нужно показать, что это самая длинная общая подпоследовательность. Проведем доказательство методом от противного. Предположим, что имеется общая подпоследовательность W последовательностей X_{m-1} и Y_{n-1}, длина которой превышает $k - 1$. Добавив к W элемент $x_m = y_n$, получим общую подпоследовательность последовательностей X и Y, длина которой превышает k, что приводит к противоречию.

(2) Если $z_k \neq x_m$, то Z — общая подпоследовательность последовательностей X_{m-1} и Y. Если бы существовала общая подпоследовательность W последовательностей X_{m-1} и Y, длина которой превышает k, то она была бы также общей подпоследовательностью последовательностей $X_m \equiv X$ и Y, что противоречит предположению о том, что Z — самая длинная общая подпоследовательность последовательностей X и Y.

(3) Доказательство этого случая аналогично доказательству случая (2) с точностью до замены элементов последовательности X соответствующими элементами последовательности Y. ■

И теоремы 15.1 видно, что самая длинная общая подпоследовательность двух последовательностей содержит в себе самую длинную общую подпоследовательность их префиксов. Таким образом, задача о самой длинной общей подпоследовательности обладает оптимальной подструктурой. В рекурсивном решении этой задачи также возникают перекрывающиеся вспомогательные задачи, но об этом речь пойдет в следующем разделе.

Этап 2: рекурсивное решение

Из теоремы 15.1 следует, что при нахождении самой длинной общей подпоследовательности последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ возникает одна или две вспомогательные задачи. Если $x_m = y_n$, необходимо найти самую длинную общую подпоследовательность последовательностей X_{m-1} и Y_{m-1}. Добавив к ней элемент $x_m = y_n$, получим LCS последовательностей X и Y. Если $x_m \neq y_n$, необходимо решить две вспомогательных задачи: найти самые LSC последовательностей X_{m-1} и Y, а также последовательностей X и Y_{n-1}. Какая из этих двух подпоследовательностей окажется длиннее, та и будет самой длинной общей подпоследовательностью последовательностей X и Y.

В задаче поиска самой длинной общей подпоследовательности легко увидеть проявление свойства перекрывающихся вспомогательных задач. Чтобы найти LCS последовательностей X и Y, может понадобиться найти LCS последовательностей X_{m-1} и Y, а также LCS X и Y_{n-1}. Однако в каждой из этих задач возникает задача о поиске LCS последовательностей X_{m-1} и Y_{n-1}. Общая вспомогательная задача возникает и во многих других подзадачах.

Как и в задаче о перемножении цепочки матриц, в рекурсивном решении задачи о самой длинной общей подпоследовательности устанавливается рекуррентное соотношение для значений, характеризующих оптимальное решение. Обозначим через $c[i, j]$ длину самой длинной общей подпоследовательности последовательностей X_i и Y_j. Если $i = 0$ или $j = 0$, длина одной из этих последовательностей равна нулю, поэтому самая длинная их общая подпоследовательность имеет нулевую длину. Оптимальная вспомогательная подструктура задачи о самой длинной общей подпоследовательности определяется формулой:

$$c[i,j] = \begin{cases} 0 & \text{при } i = 0 \text{ или } j = 0, \\ c[i-1, j-1] + 1 & \text{при } i, j > 0 \text{ и } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{при } i, j > 0 \text{ и } x_i \neq y_j. \end{cases} \tag{15.14}$$

Обратите внимание, что в этой рекурсивной формулировке условия задачи ограничивают круг рассматриваемых вспомогательных задач. Если $x_i = y_j$, можно и нужно рассматривать вспомогательную задачу поиска самой длинной общей подпоследовательности последовательностей X_{i-1} и Y_{j-1}. В противном случае рассматриваются две вспомогательные задачи по поиску LCS последовательностей X_i и Y_{j-1}, а также последовательностей X_{i-1} и Y_j. В рассмотренных ранее алгоритмах, основанных на принципах динамического программирования (для задач о составлении расписания работы сборочного конвейера и о перемножении цепочки матриц), выбор вспомогательных задач не зависел от условий задачи. Задача о поиске LCS не единственная, в которой вид возникающих вспомогательных задач определяется условиями задачи. В качестве другого подобного примера можно привести задачу о расстоянии редактирования (см. задачу 15-3).

Этап 3: вычисление длины самой длинной общей подпоследовательности

На основе уравнения (15.14) легко написать рекурсивный алгоритм с экспоненциальным временем работы, предназначенный для вычисления длины LCS двух последовательностей. Однако благодаря наличию всего лишь $\Theta(mn)$ различных вспомогательных задач можно воспользоваться динамическим программированием для вычисления решения в восходящем направлении.

В процедуре LCS_LENGTH в роли входных данных выступают две последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Величины $c[i, j]$ хранятся в таблице $c[0..m, 0..n]$, элементы которой вычисляются по строкам (т.е. сначала слева направо заполняется первая строка, затем — вторая и т.д.). В процедуре также поддерживается таблица $b[1..m, 1..n]$, благодаря которой упрощается процесс построения оптимального решения. Наглядный смысл элементов $b[i, j]$ состоит в том, что каждый из них указывает на элемент таблицы, соответствующий оптимальному решению вспомогательной задачи, выбранной при вычислении элемента $c[i, j]$. Процедура возвращает таблицы b и c; в элементе $c[m, n]$ хранится длина LCS последовательностей X и Y.

```
LCS_LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                           then c[i, j] ← c[i − 1, j]
14                                b[i, j] ← "↑"
15                           else c[i, j] ← c[i, j − 1]
16                                b[i, j] ← "←"
17  return c и b
```

На рис. 15.6 показаны таблицы, полученные в результате выполнения процедуры LCS_LENGTH с входными последовательностями $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$. Время выполнения процедуры равно $O(mn)$, поскольку на вычисление каждого элемента таблицы требуется время, равное $O(1)$. Квадрат,

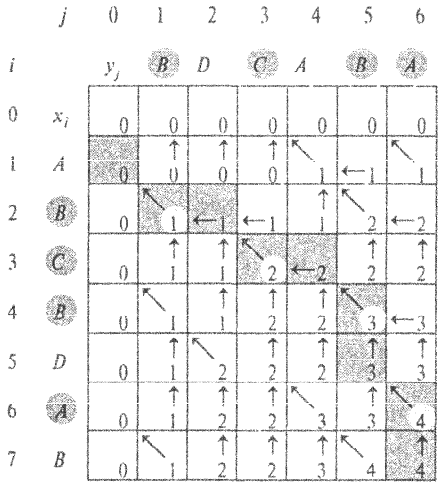


Рис. 15.6. Таблицы c и b, которые возвращаются процедурой LCS_LENGTH для входных последовательностей $X = \langle A, B, C, B, D, A, B\rangle$ и $Y = \langle B, D, C, A, B, A\rangle$

который находится на пересечении строки i и столбца j, содержит значение $c[i, j]$ и соответствующую стрелку, выступающую в роли значения $b[i, j]$. Элемент 4, который является значением содержащегося в правом нижнем углу элемента $c[7, 6]$, — длина самой длинной общей подпоследовательности последовательностей X и Y (в данном случае это $\langle B, C, B, A\rangle$). При $i, j > 0$ элемент $c[i, j]$ зависит только от того, выполняется ли соотношение $x_i = y_j$, и от значений элементов $c[i-1, j]$, $c[i, j-1]$ и $c[i-1, j-1]$, которые вычисляются перед значением $c[i, j]$. Чтобы восстановить элементы, из которых состоит самая длинная общая подпоследовательность, проследуем по стрелочкам $b[i, j]$, ведущим из правого нижнего угла. Полученный путь обозначен затенением серого цвета. Каждый элемент "↖" на этом пути соответствует (выделенному цветом) элементу, для которого $x_i = y_j$ является элементом самой длинной общей подпоследовательности.

Этап 4: построение самой длинной общей подпоследовательности

С помощью таблицы b, которая возвращается процедурой LCS_LENGTH, можно быстро сконструировать самую длинную общую подпоследовательность последовательностей $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$. Мы просто начинаем с элемента $b[m, n]$ и проходим таблицу по стрелкам. Если значением элемента $b[i, j]$ является "↖", то элемент $x_i = y_j$ принадлежит самой длинной

общей подпоследовательности. Элементы LCS, восстановленные этим методом, следуют в обратном порядке. Приведенная ниже рекурсивная процедура выводит элементы самой длинной общей подпоследовательности последовательностей X и Y в прямом порядке. Начальный вызов этой процедуры выглядит как PRINT_LCS$(b, X, length[X], length[Y])$:

```
PRINT_LCS(b, X, i, j)
1  if i = 0 или j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT_LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT_LCS(b, X, i − 1, j)
8  else PRINT_LCS(b, X, i, j − 1)
```

Для таблицы b, изображенной на рис. 15.6, эта процедура выводит последовательность "$BCBA$". Время работы процедуры равно $O(m + n)$, поскольку хотя бы один из индексов i или j уменьшается на каждой стадии рекурсии.

Улучшение кода

После разработки алгоритма часто оказывается, что можно улучшить время его работы или объем требуемой им памяти. Особенно это справедливо в отношении незамысловатых алгоритмов, основанных на принципах динамического программирования. Некоторые изменения могут упростить код и уменьшить постоянный множитель, но не приводят к повышению производительности в асимптотическом пределе. Другие же могут вызвать существенное асимптотическую экономию времени и пространства.

Например, можно обойтись без таблицы b. Каждый элемент $c[i, j]$ зависит только от трех других элементов этой же таблицы: $c[i-1, j-1]$, $c[i-1, j]$ и $c[i, j-1]$. Для данного элемента $c[i, j]$ в течение времени $O(1)$ можно определить, какое из этих трех значений было использовано для вычисления $c[i, j]$, не прибегая к таблице b. Таким образом, самую длинную общую подпоследовательность можно восстановить в течение времени $O(m + n)$. Для этого понадобится процедура, подобная процедуре PRINT_LCS (такую процедуру предлагается составить в упражнении 15.4-2). Несмотря на то, что в этом методе экономится объем памяти, равный $\Theta(mn)$, асимптотически количество памяти, необходимой для вычисления самой длинной общей подпоследовательности, не изменяется. Это объясняется тем, что таблица c все равно требует $\Theta(mn)$ памяти.

Однако можно уменьшить объем памяти, необходимой для работы процедуры PRINT_LCS, поскольку одновременно нужны лишь две строки этой таблицы:

вычисляемая и предыдущая. (Фактически для вычисления длины самой длинной общей подпоследовательности можно обойтись пространством, лишь немного превышающим объем, необходимый для одной строки матрицы c — см. упражнение 15.4-4.) Это усовершенствование работает лишь в том случае, когда нужно знать только длину самой длинной общей подпоследовательности. Если же необходимо воссоздать элементы этой подпоследовательности, такая "урезанная" таблица содержит недостаточно информации для того, чтобы проследить обратные шаги в течение времени $O(m+n)$.

Упражнения

15.4-1. Определите самую длинную общую подпоследовательность последовательностей $\langle 1,0,0,1,0,1,0,1\rangle$ и $\langle 0,1,0,1,1,0,1,1,0\rangle$.

15.4-2. Покажите, как в течение времени $O(m+n)$ воссоздать самую длинную общую подпоследовательность, не используя при этом таблицу b, если имеется таблица c и исходные последовательности $X = \langle x_1, x_2, \ldots, x_m\rangle$ и $Y = \langle y_1, y_2, \ldots, y_n\rangle$.

15.4-3. Приведите версию процедуры LCS_LENGTH с запоминанием, время работы которой равно $O(mn)$.

15.4-4. Покажите, как вычислить длину самой длинной общей подпоследовательности, используя при этом всего $2 \cdot \min(m,n)$ элементов таблицы c плюс $O(1)$ дополнительной памяти. Затем покажите, как это же можно сделать с помощью $\min(m,n)$ элементов таблицы и $O(1)$ дополнительной памяти.

15.4-5. Приведите алгоритм, предназначенный для вычисления самой длинной монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O(n^2)$.

★ 15.4-6. Приведите алгоритм, предназначенный для вычисления самой длинной монотонно неубывающей подпоследовательности данной последовательности, состоящей из n чисел. Время работы алгоритма должно быть равно $O(n \lg n)$. (*Указание*: обратите внимание, что последний элемент кандидата в искомую подпоследовательность по величине не меньше последнего элемента кандидата длиной $i-1$.)

15.5 Оптимальные бинарные деревья поиска

Предположим, что разрабатывается программа, предназначенная для перевода текстов с русского языка на украинский. Для каждого русского слова необходимо найти украинский эквивалент. Один из возможных путей поиска — построение

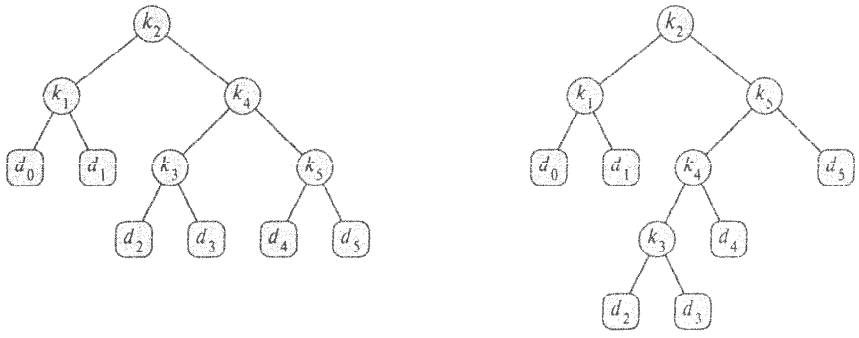
бинарного дерева поиска с n русскими словами, выступающими в роли ключей, и украинскими эквивалентами, играющими роль сопутствующих данных. Поскольку поиск с помощью этого дерева будет производиться для каждого отдельного слова из текста, полное затраченное на него время должно быть как можно меньше. С помощью красно-черного дерева или любого другого сбалансированного бинарного дерева поиска можно добиться того, что время каждого отдельного поиска будет равным $O(\lg n)$. Однако слова встречаются с разной частотой, и может получиться так, что какое-нибудь часто употребляемое слово (например, предлог или союз) находится далеко от корня, а такое редкое слово, как "контрвстреча", — возле корня. Такой способ организации привел бы к замедлению перевода, поскольку количество узлов, просмотренных в процессе поиска ключа в бинарном дереве, равно увеличенной на единицу глубине узла, содержащего данный ключ. Нужно сделать так, чтобы слова, которые встречаются в тексте часто, были размещены поближе к корню. Кроме того, в исходном тексте могут встречаться слова, для которых перевод отсутствует. Таких слов вообще не должно быть в бинарном дереве поиска. Как организовать бинарное дерево поиска, чтобы свести к минимуму количество посещенных в процессе поиска узлов, если известно, с какой частотой встречаются слова?

Необходимая нам конструкция известна как ***оптимальное бинарное дерево поиска*** (optimal binary search tree). Приведем формальное описание задачи. Имеется заданная последовательность $K = \langle k_1, k_2, \ldots, k_n \rangle$, состоящая из n различных ключей, которые расположены в отсортированном порядке (так что $k_1 < k_2 < \cdots < k_n$). Из этих ключей нужно составить бинарное дерево поиска. Для каждого ключа k_i задана вероятность p_i поиска этого ключа. Кроме того, может выполняться поиск значений, отсутствующих в последовательности K, поэтому следует предусмотреть $n+1$ фиктивных ключей $\langle d_0, d_1, \ldots, d_n \rangle$, представляющих эти значения. В частности, d_0 представляет все значения, меньшие k_1, а d_n — все значения, превышающие k_n. Фиктивный ключ d_i ($i = 1, 2, \ldots, n-1$) представляет все значения, которые находятся между k_i и k_{i+1}. Для каждого фиктивного ключа d_i задана соответствующая ей вероятность q_i. На рис. 15.7 показаны два бинарных дерева поиска для множества, состоящего из $n = 5$ ключей. Каждый ключ k_i представлен внутренним узлом, а каждый фиктивный ключ d_i является листом. Поиск может быть либо успешным (найден какой-то ключ k_i), либо неудачным (возвращается какой-то фиктивный ключ d_i), поэтому справедливо соотношение

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1. \tag{15.15}$$

Вероятности, соответствующие внутренним узлам p_i и листьям q_i, приведены в табл. 15.1.

Поскольку вероятность поиска каждого обычного и фиктивного ключа считается известной, можно определить математическое ожидание стоимости поиска



а) б)

Рис. 15.7. Два бинарных дерева поиска для множества из 5 элементов

Таблица 15.1. Вероятности поиска ключей в узлах бинарного дерева

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

по заданному бинарному дереву поиска T. Предположим, что фактическая стоимость поиска определяется количеством проверенных узлов, т.е. увеличенной на единицу глубиной узла на дереве T, в котором находится искомый ключ. Тогда математическое ожидание стоимости поиска в дереве T равно

$$\begin{aligned} \mathrm{E}[\text{Стоимость поиска в } T] &= \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i + \\ &+ \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i = \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \\ &+ \sum_{i=0}^{n} depth_T(d_i) \cdot q_i, \end{aligned} \tag{15.16}$$

где величина $depth_T(\,)$ обозначает глубину узла в дереве T. Последнее равенство следует из уравнения (15.15). В табл. 15.2 вычисляется математическое ожидание стоимости поиска для бинарного дерева, изображенного на рис. 15.7*а*.

Наша цель — построить для данного набора вероятностей бинарное дерево поиска, математическое ожидание стоимости поиска для которого будет

Таблица 15.2. Вычисление математического ожидания стоимости поиска

Узел	Глубина	Вероятность	Вклад
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Всего			2,80

минимальным. Такое дерево называется ***оптимальным бинарным деревом поиска***. На рис. 15.7*б* показано оптимальное бинарное дерево поиска для вероятностей, заданных в табл. 15.1. Математическое ожидание поиска в этом дереве равно 2.75. Этот пример демонстрирует, что оптимальное бинарное дерево поиска — это не обязательно дерево минимальной высоты. Кроме того, в оптимальном дереве ключ, которому соответствует максимальная вероятность, не всегда находится в корне. В данном случае вероятность имеет самую большую величину для ключа k_5, хотя в корне оптимального бинарного дерева расположен ключ k_2. (Минимальная величина математического ожидания для всевозможных бинарных деревьев поиска, в корне которых находится ключ k_5, равна 2.85.)

Как и в задаче о перемножении цепочки матриц, последовательный перебор всех возможных деревьев в данном случае оказывается неэффективным. Чтобы сконструировать бинарное дерево поиска, можно обозначить ключами $k_1, k_2, \ldots, k_n$ узлы бинарного дерева с n узлами, а затем добавить листья для фиктивных ключей. В задаче 12-4 было показано, что количество бинарных деревьев с n узлами равно $\Omega\left(4^n/n^{3/2}\right)$, так что количество бинарных деревьев, которые надо проверять при полном переборе, растет экспоненциально с ростом n. Не удивительно, что эта задача будет решаться методом динамического программирования.

Этап 1: структура оптимального бинарного дерева поиска

Чтобы охарактеризовать оптимальную подструктуру оптимального бинарного дерева поиска, исследуем его поддеревья. Рассмотрим произвольное поддерево бинарного дерева поиска. Оно должно содержать ключи, которые составляют непрерывный интервал $k_i, \ldots, k_j$ для некоторых $1 \leqslant i \leqslant j \leqslant n$. Кроме того, такое поддерево должно также содержать в качестве листьев фиктивные ключи $d_{i-1}, \ldots, d_j$.

Теперь можно сформулировать оптимальную подструктуру: если в состав оптимального бинарного дерева поиска T входит поддерево T', содержащее ключи $k_i, \ldots, k_j$, то это поддерево тоже должно быть оптимальным для вспомогательной подзадачи с ключами $k_i, \ldots, k_j$ и фиктивными ключами $d_{i-1}, \ldots, d_j$. Для доказательства этого утверждения применяется обычный метод "вырезания и вставки". Если бы существовало поддерево T'', математическое ожидание поиска в котором ниже, чем математическое ожидание поиска в поддереве T', то из дерева T можно было бы вырезать поддерево T' и подставить вместо него поддерево T''. В результате получилось бы дерево, математическое ожидание времени поиска в котором оказалось бы меньше, что противоречит оптимальности дерева T.

Покажем с помощью описанной выше оптимальной подструктуры, что оптимальное решение задачи можно воссоздать из оптимальных решений вспомогательных задач. Если имеется поддерево, содержащее ключи $k_i, \ldots, k_j$, то один из этих ключей, скажем, k_r ($i \leqslant r \leqslant j$) будет корнем этого оптимального поддерева. Поддерево, которое находится слева от корня k_r, будет содержать ключи $k_i, \ldots, k_{r-1}$ (и фиктивные ключи $d_{i-1}, \ldots d_{r-1}$), а правое поддерево — ключи $k_{r+1}, \ldots, k_j$ (и фиктивные ключи $d_r, \ldots, d_j$). Как только будут проверены все ключи k_r (где $i \leqslant r \leqslant j$), которые являются кандидатами на роль корня, и найдем оптимальные бинарные деревья поиска, содержащие элементы $k_i, \ldots, k_{r-1}$ и $k_{r+1}, \ldots, k_j$, мы гарантированно построим оптимальное бинарное дерево поиска.

Стоит сделать одно замечание по поводу "пустых" поддеревьев. Предположим, что в поддереве с ключами $k_i, \ldots, k_j$ в качестве корня выбран ключ k_i. Согласно приведенным выше рассуждениям, поддерево, которое находится слева от корня k_i, содержит ключи $k_i, \ldots, k_{i-1}$. Естественно интерпретировать эту последовательность как такую, в которой не содержится ни одного ключа. Однако следует иметь в виду, что поддеревья содержат помимо реальных и фиктивные ключи. Примем соглашение, согласно которому поддерево, состоящее из ключей $k_i, \ldots, k_{i-1}$, не содержит обычных ключей, но содержит один фиктивный ключ d_{i-1}. Аналогично, если в качестве корня выбран ключ k_j, то правое поддерево не содержит обычных ключей, но содержит один фиктивный ключ d_j.

Этап 2: рекурсивное решение

Теперь все готово для рекурсивного определения оптимального решения. В качестве вспомогательной задачи выберем задачу поиска оптимального бинарного дерева поиска, содержащего ключи $k_i, \ldots, k_j$, где $i \geqslant 1$, $j \leqslant n$ и $j \geqslant i-1$ (если $j = i-1$, то фактических ключей не существует, имеется только фиктивный ключ d_{i-1}). Определим величину $e[i,j]$ как математическое ожидание стоимости поиска в оптимальном бинарном дереве поиска с ключами $k_i, \ldots, k_j$. В конечном итоге нужно вычислить величину $e[1,n]$.

Если $j = i-1$, то все просто. В этом случае имеется всего один фиктивный ключ d_{i-1}, и математическое ожидание стоимости поиска равно $e[i, i-1] = q_{i-1}$.

Если $j \geqslant i$, то среди ключей $k_i, \ldots, k_j$ нужно выбрать корень k_r, а потом из ключей $k_i, \ldots, k_{r-1}$ составить левое оптимальное бинарное дерево поиска, а из ключей $k_{r+1}, \ldots, k_j$ — правое оптимальное бинарное дерево поиска. Что происходит с математическим ожиданием стоимости поиска в поддереве, когда оно становится поддеревом какого-то узла? Глубина каждого узла в поддереве возрастает на единицу. Согласно уравнению (15.16), математическое ожидание стоимости поиска в этом поддереве возрастает на величину суммы по всем вероятностям поддерева. Обозначим эту сумму вероятностей, вычисленную для поддерева с ключами $k_i, \ldots, k_j$, так:

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l. \tag{15.17}$$

Таким образом, если k_r — корень оптимального поддерева, содержащего ключи $k_i, \ldots, k_j$, то выполняется соотношение

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)).$$

Заметив, что

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j),$$

выражение для величины $e[i,j]$ можно переписать так:

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j). \tag{15.18}$$

Рекурсивное соотношение (15.18) предполагает, что нам известно, какой узел k_r используется в качестве корня. На эту роль выбирается ключ, который приводит к минимальному значению математического ожидания стоимости поиска. С учетом этого получаем окончательную рекурсивную формулу:

$$e[i,j] = \begin{cases} q_{i-1} & \text{при } j = i-1, \\ \min\limits_{i \leqslant r \leqslant j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{при } i \leqslant j. \end{cases} \tag{15.19}$$

Величины $e[i, j]$ — это математическое ожидание стоимостей поиска в оптимальных бинарных деревьях поиска. Чтобы было легче следить за структурой оптимального бинарного дерева поиска, обозначим через $root[i, j]$ (где $1 \leqslant i \leqslant j \leqslant n$) индекс r узла k_r, который является корнем оптимального бинарного дерева поиска, содержащего ключи $k_i, \ldots, k_j$. Скоро мы узнаем, как вычисляются величины $root[i, j]$, а способ восстановления из этих величин оптимального бинарного дерева поиска оставим до того момента, когда придет время выполнить упражнение 15.5-1.

Этап 3: вычисление математического ожидания стоимости поиска в оптимальном бинарном дереве поиска

На данном этапе некоторые читатели, возможно, заметили некоторое сходство между характеристиками задач об оптимальных бинарных деревьях поиска и о перемножении цепочки матриц. Во вспомогательных задачах обеих задач индексы элементов изменяются последовательно. Прямая рекурсивная реализация уравнения (15.19) может оказаться такой же неэффективной, как и прямая рекурсивная реализация алгоритма в задаче о перемножении цепочки матриц. Вместо этого будем сохранять значения $e[i, j]$ в таблице $e[1..n+1, 0..n]$. Первый индекс должен пробегать не n, а $n+1$ значений. Это объясняется тем, что для получения поддерева, в который входит только фиктивный ключ d_n, понадобится вычислить и сохранить значение $e[n+1, n]$. Второй индекс должен начинаться с нуля, поскольку для получения поддерева, содержащего лишь фиктивный ключ d_0, нужно вычислить и сохранить значение $e[1, 0]$. Мы будем использовать только те элементы $e[i, j]$, для которых $j \geqslant i - 1$. Кроме того, будет использована таблица $root[i, j]$, в которую будут заноситься корни поддеревьев, содержащих ключи $k_i, \ldots, k_j$. В этой таблице задействованы только те записи, для которых $1 \leqslant i \leqslant j \leqslant n$.

Для повышения эффективности понадобится еще одна таблица. Вместо того чтобы каждый раз при вычислении $e[i, j]$ вычислять значения $w(i, j)$ "с нуля", для чего потребуется $\Theta(j - i)$ операций сложения, будем сохранять эти значения в таблице $w[1..n+1, 0..n]$. В базовом случае вычисляются величины $w[i, i-1] = q_{i-1}$ для $1 \leqslant i \leqslant n+1$. Для $j \geqslant i$ вычисляются величины

$$w[i, j] = w[i, j-1] + p_j + q_j. \tag{15.20}$$

Таким образом, каждое из $\Theta(n^2)$ значений матрицы $w[i, j]$ можно вычислить за время $\Theta(1)$.

Ниже приведен псевдокод, который принимает в качестве входных данных вероятности $p_1, \ldots, p_n$ и $q_0, \ldots, q_n$ и размер n и возвращает таблицы e и $root$.

```
OPTIMAL_BST(p, q, n)
 1  for i ← 1 to n + 1
 2       do e[i, i − 1] ← q_{i−1}
 3          w[i, i − 1] ← q_{i−1}
 4  for l ← 1 to n
 5       do for i ← 1 to n − l + 1
 6               do j ← i + l − 1
 7                  e[i, j] ← ∞
 8                  w[i, j] ← w[i, j − 1] + p_j + q_j
 9                  for r ← i to j
10                       do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                          if t < e[i, j]
12                            then e[i, j] ← t
13                                 root[i, j] ← r
14  return e и root
```

Благодаря приведенному выше описанию и аналогии с процедурой MATRIX_CHAIN_ORDER, описанной в разделе 15.2, работа представленной выше процедуры должна быть понятной. Цикл **for** в строках 1–3 инициализирует значения $e[i, i-1]$ и $w[i, i-1]$. Затем в цикле **for** в строках 4–13 с помощью рекуррентных соотношений (15.19) и (15.20) вычисляются элементы матриц $e[i, j]$ и $w[i, j]$ для всех индексов $1 \leqslant i \leqslant j \leqslant n$. В первой итерации, когда $l = 1$, в этом цикле вычисляются элементы $e[i, i]$ и $w[i, i]$ для $i = 1, 2, \ldots, n$. Во второй итерации, когда $l = 2$, вычисляются элементы $e[i, i+1]$ и $w[i, i+1]$ для $i = 1, 2, \ldots, n-1$ и т.д. Во внутреннем цикле **for** (строки 9–13) каждый индекс r апробируется на роль индекса корневого элемента k_r оптимального бинарного дерева поиска с ключами $k_i, \ldots, k_j$. В этом цикле элементу $root[i, j]$ присваивается то значение индекса r, которое подходит лучше всего.

На рис. 15.8 показаны таблицы $e[i, j]$, $w[i, j]$ и $root[i, j]$, вычисленные с помощью процедуры OPTIMAL_BST для распределения ключей из табл. 15.1. Как и в примере с перемножением цепочки матриц, таблицы повернуты так, чтобы диагонали располагались горизонтально. В процедуре OPTIMAL_BST строки вычисляются снизу вверх, а в каждой строке заполнение элементов производится слева направо.

Время выполнения процедуры OPTIMAL_BST, как и время выполнения процедуры MATRIX_CHAIN_ORDER, равно $\Theta(n^3)$. Легко увидеть, что время работы составляет $O(n^3)$, поскольку циклы **for** этой процедуры трижды вложены друг в друга, и индекс каждого цикла принимает не более n значений. Далее, индексы циклов в процедуре OPTIMAL_BST изменяются не в тех же пределах, что и индексы циклов в процедуре MATRIX_CHAIN_ORDER, но во всех направлениях они принимают по крайней мере одно значение. Таким образом, процедура OPTIMAL_BST, как и процедура MATRIX_CHAIN_ORDER, выполняется в течение времени $\Omega(n^3)$.

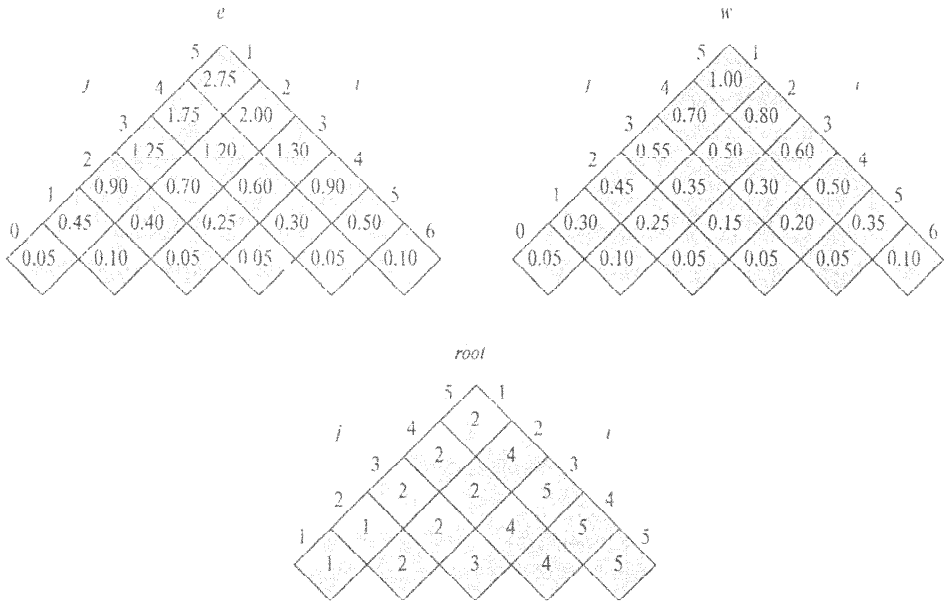


Рис. 15.8. Таблицы $e[i, j]$, $w[i, j]$ и $root[i, j]$, вычисленные процедурой OPTIMAL_BST для распределения ключей из табл. 15.1

Упражнения

15.5-1. Напишите псевдокод для процедуры CONSTRUCT_OPTIMAL_BST($root$), которая по заданной таблице $root$ выдает структуру оптимального бинарного дерева поиска. Для примера, приведенного на рис. 15.8, процедура должна выводить структуру, соответствующую оптимальному бинарному дереву поиска, показанному на рис. 15.7*б*:

k_2 — корень
k_1 — левый дочерний элемент k_2
d_0 — левый дочерний элемент k_1
d_1 — правый дочерний элемент k_1
k_5 — правый дочерний элемент k_2
k_4 — левый дочерний элемент k_5
k_3 — левый дочерний элемент k_4
d_2 — левый дочерний элемент k_3
d_3 — правый дочерний элемент k_3
d_4 — правый дочерний элемент k_4
d_5 — правый дочерний элемент k_5

15.5-2. Определите стоимость и структуру оптимального бинарного дерева поиска для множества, состоящего из $n = 7$ ключей, которым соответствуют следующие вероятности:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

15.5-3. Предположим, что вместо того, чтобы поддерживать таблицу $w[i, j]$, значение $w(i, j)$ вычисляется в строке 8 процедуры OPTIMAL_BST непосредственно из уравнения (15.17) и используется в строке 10. Как это изменение повлияет на асимптотическое поведение времени выполнения этой процедуры?

⋆ 15.5-4. Кнут [184] показал, что всегда существуют корни оптимальных поддеревьев, такие что $root[i, j-1] \leqslant root[i, j] \leqslant root[i+1, j]$ для всех $1 \leqslant i \leqslant j \leqslant n$. Используйте этот факт для модификации процедуры OPTIMAL_BST, при которой время ее выполнения станет равным $\Theta(n^2)$.

Задачи

15-1. **Битоническая евклидова задача о коммивояжере**

Евклидова задача о коммивояжере (euclidean travelling-salesman problem) — это задача, в которой определяется кратчайший замкнутый путь, соединяющий заданное множество, которое состоит из n точек на плоскости. На рис. 15.9*а* приведено решение задачи, в которой имеется семь точек. В общем случае задача является NP-полной, поэтому считается, что для ее решения требуется время, превышающее полиномиальное (см. главу 34).

Бентли (J.L. Bentley) предположил, что задача упрощается благодаря ограничению интересующих нас маршрутов ***битоническими*** (bitonic tour), т.е. такими, которые начинаются в крайней левой точке, проходят слева направо, а затем — справа налево, возвращаясь прямо к исходной точке. На рис. 15.9*б* показан кратчайший битонический маршрут, проходящий через те же семь точек. В этом случае возможна разработка алгоритма, время работы которого является полиномиальным.

Напишите алгоритм, предназначенный для определения оптимального битонического маршрута, время работы которого будет равным $O(n^2)$. Предполагается, что не существует двух точек, координаты x которых совпадают. (*Указание*: перебор вариантов следует организовать слева направо, поддерживая оптимальные возможности для двух частей, из которых состоит маршрут.)

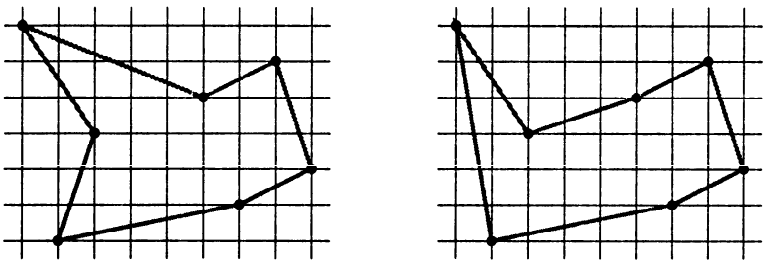


Рис. 15.9. *а*) кратчайший (не битонический) замкнутый маршрут длиной 24.89; *б*) кратчайший битонический маршрут длиной 25.58

15-2. **Вывод на печать с форматированием**

Рассмотрим задачу о форматировании параграфа текста, предназначенного для вывода на печать. Входной текст представляет собой последовательность, состоящую из n слов, длины которых равны $l_1, l_2, \ldots, l_n$ (длина слов измеряется в символах). При выводе на печать параграф должен быть аккуратно отформатирован и размещен в несколько строк, в каждой из которых помещается по M символов. Сформулируем критерий "аккуратного" форматирования. Если данная строка содержит слова от i-го до j-го и мы оставляем между словами ровно по одному пробелу, количество оставшихся пробелов, равное $M - j + i - \sum_{k=i}^{j} l_k$, должно быть неотрицательно, чтобы все слова поместились в строке. Мы хотим минимизировать сумму возведенных в куб остатков по всем строкам, кроме последней. Сформулируйте алгоритм, основанный на принципах динамического программирования, который аккуратно выводит на принтер параграф, состоящий из n слов. Проанализируйте время работы этого алгоритма и требуемое для его работы количество памяти.

15-3. **Расстояние редактирования**

Для того чтобы преобразовать исходную строку текста $x[1..m]$ в конечную строку $y[1..n]$, можно воспользоваться различными операциями преобразования. Наша цель заключается в том, чтобы для данных строк x и y определить последовательность преобразований, превращающих x в y. Для хранения промежуточных результатов используется массив z (предполагается, что он достаточно велик для хранения необходимых промежуточных результатов). Изначально массив z пуст, а в конце работы $z[j] = y[j]$ для всех $j = 1, 2, \ldots, n$. Поддерживается текущий индекс i массива x и индекс j массива z, и в ходе выполнения операций преобразования разрешено изменять элементы массива z и эти индексы. В начале работы алгоритма $i = j = 1$. В процессе преобразования необходимо

проверить каждый символ массива x. Это означает, что в конце выполнения последовательности операций значение i должно быть равно $m + 1$. Всего имеется шесть перечисленных ниже операций преобразования.

Копирование символа из массива x в массив z путем присвоения $z[j] \leftarrow x[i]$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Замена символа из массива x другим символом c путем присвоения $z[j] \leftarrow c$ с последующим увеличением индексов i и j. В этой операции проверяется элемент $x[i]$.

Удаление символа из массива x путем увеличения на единицу индекса i и сохранения индекса j прежним. В этой операции проверяется элемент $x[i]$.

Вставка символа c в массив z путем присвоения $z[j] \leftarrow c$ и увеличения на единицу индекса j при сохранении индекса i прежним. В этой операции не проверяется ни один элемент массива x.

Перестановка двух следующих символов путем копирования их из массива x в массив z в обратном порядке. Это делается путем присваиваний $z[j] \leftarrow x[i+1]$ и $z[j+1] \leftarrow x[i]$ с последующим изменением значений индексов $i \leftarrow i + 2$ и $j \leftarrow j + 2$. В этой операции проверяются элементы $x[i]$ и $x[i+1]$.

Удаление остатка массива x путем присвоения $i \leftarrow m + 1$. В этой операции проверяются все элементы массива x, которые не были проверены до сих пор. Если эта операция выполняется, то она должна быть последней.

В качестве примера приведем один из способов преобразования исходной строки `algorithm` в конечную строку `altruistic`. В ходе этого преобразования выполняется перечисленная ниже последовательность операций, в которых символы подчерка — это элементы $x[i]$ и $z[j]$ после выполнения очередной операции:

Операция	x	z
Исходные строки	algorithm	_
Копирование	algorithm	a_
Копирование	algorithm	al_
Замена символом t	algorithm	alt_
Удаление	algorithm	alt_
Копирование	algorithm	altr_
Вставка символа u	algorithm	altru_

Вставка символа i	algorithm	altrui_
Вставка символа s	algorithm	altruis_
Перестановка	algorithm	altruisti_
Вставка символа c	algorithm	altruistic_
Удаление остатка	algorithm_	altruistic_

Заметим, что существует несколько других последовательностей операций, позволяющих преобразовать строку `algorithm` в строку `altruistic`.

С каждой операцией преобразования связана своя стоимость, которая зависит от конкретного приложения. Однако мы предположим, что стоимость каждой операции — известная константа. Кроме того, предполагается, что стоимости отдельно взятых операций копирования и замены меньше суммарной стоимости операций удаления и вставки, иначе применять эти операции было бы нерационально. Стоимость данной последовательности операций преобразования представляет собой сумму стоимостей отдельных операций последовательности. Стоимость приведенной выше последовательности преобразований строки `algorithm` в строку `altruistic` равна сумме трех стоимостей копирования, стоимости замены, стоимости удаления, четырем стоимостям вставки, стоимости перестановки и удаления остатка.

а) Пусть имеется две последовательности $x[1..m]$ и $y[1..n]$, а также множество, элементами которого являются стоимости операций преобразования. ***Расстоянием редактирования*** (edit distance) от x до y называется минимальная стоимость последовательности операций преобразования x в y. Опишите основанный на принципах динамического программирования алгоритм, в котором определяется расстояние редактирования от $x[1..m]$ до $y[1..n]$ и выводится оптимальная последовательность операций редактирования. Проанализируйте время работы этого алгоритма и требуемую для него память.

Задача о расстоянии редактирования — это обобщение задачи об анализе двух ДНК (см., например, книгу Сетубала (Setubal) и Мейданиса (Meidanis) [272, раздел 3.2]). Имеется два метода, позволяющих оценить подобие двух ДНК путем их выравнивания. Один способ выравнивания двух последовательностей x и y заключается в том, чтобы вставлять пробелы в произвольных местах этих двух последовательностей (включая позиции, расположенные в конце одной из них). Получившиеся в результате последовательности x' и y' должны иметь одинаковую длину, однако они не могут содержать пробелы в одинаковых позициях (т.е. не существует

такого индекса j, чтобы и $x'[j]$, и $y'[j]$ были пробелами). Далее каждой позиции присваивается определенное количество баллов в соответствии со сформулированными ниже правилами:

- $+1$, если $x'[j] = y'[j]$ и ни один из этих элементов не является пробелом;
- -1, если $x'[j] \neq y'[j]$ и ни один из этих элементов не является пробелом;
- -2, если элемент $x'[j]$ или элемент $y'[j]$ является пробелом.

Стоимость выравнивания определяется как сумма баллов, полученных при сравнении отдельных позиций. Например, если заданы последовательности $x =$ GATCGGCAT и $y =$ CAATGTGAATC, то одно из возможных выравниваний имеет вид:

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Символ + под позицией указывает на то, что ей присваивается балл $+1$, символ - означает балл -1, а символ * — балл -2. Таким образом, полная стоимость равна $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

б) Объясните, как свести задачу о поиске оптимального выравнивания к задаче о поиске расстояния редактирования с помощью подмножества операций преобразования, состоящего из копирования, замены, удаления элемента, вставки, перестановки и удаления остатка.

15-4. **Вечеринка в фирме**

Профессор Тамада является консультантом президента корпорации по вопросам проведения вечеринок компании. Взаимоотношения между сотрудниками компании описываются иерархической структурой. Это означает, что отношение "начальник-подчиненный" образует дерево, во главе (в корне) которого находится президент. В отделе кадров каждому служащему присвоили рейтинг дружелюбия, представленный действительным числом. Чтобы на вечеринке все чувствовали себя раскованно, президент выдвинул требование, согласно которому ее не должны одновременно посещать сотрудник и его непосредственный начальник.

Профессору Тамаде предоставили дерево, описывающее структуру корпорации в представлении с левым дочерним и правым сестринским узлами, описанном в разделе 10.4. Каждый узел дерева, кроме указателей, содержит имя сотрудника и его рейтинг дружелюбия. Опишите алгоритм,

предназначенный для составления списка гостей, который бы давал максимальное значение суммы рейтингов дружелюбия гостей. Проанализируйте время работы этого алгоритма.

15-5. **Алгоритм Витерби**

Подход динамического программирования можно использовать в ориентированном графе $G = (V, E)$ для распознавания речи. Каждое ребро $(u, v) \in E$ графа помечено звуком $\sigma(u, v)$, который является членом конечного множества звуков Σ. Граф с метками представляет собой формальную модель речи человека, который разговаривает на языке, состоящем из ограниченного множества звуков. Все пути в графе, которые начинаются в выделенной вершине $v_0 \in V$, соответствуют возможной последовательности звуков, допустимых в данной модели. Метка направленного пути по определению является объединением меток, соответствующих ребрам, из которых состоит путь.

а) Разработайте эффективный алгоритм, который для заданного графа G с помеченными ребрами и выделенной вершиной v_0, и последовательности $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ символов из множества Σ возвращал бы путь в графе G, который начинается в вершине v_0 и имеет метку s (если таковой существует). В противном случае алгоритм должен возвращать строку NO-SUCH-PATH (нет такого пути). Проанализируйте время работы алгоритма. (*Указание*: могут оказаться полезными концепции, изложенные в главе 22.)

Теперь предположим, что каждому ребру $(u, v) \in E$ также сопоставляется неотрицательная вероятность $p(u, v)$ перехода по этому ребру из вершины u. При этом издается соответствующий звук. Сумма вероятностей по всем ребрам, исходящим из произвольной вершины, равна 1. По определению вероятность пути равна произведению вероятностей составляющих его ребер. Вероятность пути, берущего начало в вершине v_0, можно рассматривать как вероятность "случайной прогулки", которая начинается в этой вершине и продолжается по случайному пути. Выбор каждого ребра на этом пути в вершине u производится в соответствии с вероятностями ребер, исходящих из этой вершины.

б) Обобщите сформулированный в части *а*) ответ так, чтобы возвращаемый путь (если он существует) был *наиболее вероятным* из всех путей, начинающихся в вершине v_0 и имеющих метку s. Проанализируйте время работы этого алгоритма.

15-6. **Передвижение по шахматной доске**

Предположим, имеется шахматная доска размером $n \times n$ клеток и шашка. Необходимо провести шашку от нижнего края доски к верхнему. Ходы

можно делать в соответствии со сформулированным далее правилом. На каждом ходу шашка передвигается в одну из следующих клеток:

а) на соседнюю клетку, расположенную непосредственно над текущей;

б) на ближайшую клетку, расположенную по диагонали в направлении вверх и влево (если шашка не находится в крайнем левом столбце);

в) на ближайшую клетку, расположенную по диагонали в направлении вверх и вправо (если шашка не находится в крайнем правом столбце).

Каждый ход из клетки x в клетку y имеет стоимость $p(x, y)$, которая задается для всех пар (x, y), допускающих ход. При этом величины $p(x, y)$ не обязательно положительны.

Сформулируйте алгоритм, определяющий последовательность ходов, для которой стоимость полного пути от нижнего края доски к верхнему краю будет максимальной. В качестве начальной может быть выбрана любая клетка в нижнем ряду. Аналогично, маршрут может заканчиваться в любой клетке верхнего ряда. Чему равно время работы этого алгоритма?

15-7. **Расписание для получения максимального дохода**

Предположим, что имеется один компьютер и множество, состоящее из n заданий $a_1, a_2, \ldots, a_n$, которые нужно выполнить на этом компьютере. Каждому заданию a_j соответствует время обработки t_j, прибыль p_j и конечный срок выполнения d_j. Компьютер не способен выполнять несколько заданий одновременно. Кроме того, если запущено задание a_j, то оно должно выполняться без прерываний в течение времени t_j. Если задание a_j завершается до того, как истечет конечный срок его выполнения d_j, вы получаете доход p_j. Если же это произойдет после момента времени d_j, то полученный доход равен нулю. Сформулируйте алгоритм, который бы выдавал расписание для получения максимальной прибыли. При этом предполагается, что любое время выполнения задания выражается целым числом в интервале от 1 до n. Чему равно время работы этого алгоритма?

Заключительные замечания

Беллман (R. Bellman) приступил к систематическому изучению динамического программирования в 1955 году. Как в названии "динамическое программирование", так и в термине "линейное программирование" слово "программирование" относится к методу табличного решения. Методы оптимизации, включающие в себя элементы динамического программирования, были известны и раньше, однако именно Беллман дал строгое математическое обоснование этой области [34].

Ху (Hu) и Шинг (Shing) [159,160] сформулировали алгоритм, позволяющий решить задачу о перемножении матриц в течение времени $O(n \lg n)$.

По-видимому, алгоритм решения задачи о самой длинной общей подпоследовательности, время работы которого равно $O(mn)$, — плод "народного" творчества. Кнут (Knuth) [63] поставил вопрос о том, существует ли алгоритм решения этой задачи, время работы которого возрастало бы медленнее, чем квадрат размера. Масек (Masek) и Патерсон (Paterson) ответили на этот вопрос утвердительно, разработав алгоритм, время работы которого равно $O(mn/\lg n)$, где $n \leqslant m$, а последовательности составлены из элементов множества ограниченного размера. Шимански (Szimanski) [288] показал, что в особом случае, когда ни один элемент не появляется во входной последовательности более одного раза, эту задачу можно решить в течение времени $O((n+m)\lg(n+m))$. Многие из этих результатов были обобщены для задачи о вычислении расстояния редактирования (задача 15-3).

Ранняя работа Гильберта (Gilbert) и Мура (Moore) [114], посвященная бинарному кодированию переменной длины, нашла применение при конструировании оптимального бинарного дерева поиска для случая, когда все вероятности p_i равны 0. В этой работе приведен алгоритм, время работы которого равно $O(n^3)$. Ахо (Aho), Хопкрофт (Hopcroft) и Ульман (Ullman) [5] представили алгоритм, описанный в разделе 15.5. Упражнение 15.5-4 предложено Кнутом [184]. Ху и Такер (Tucker) [161] разработали алгоритм для случая, когда все вероятности p_i равны 0; время работы этого алгоритма равно $O(n^2)$, а количество необходимой для него памяти — $O(n)$. Впоследствии Кнуту [185] удалось уменьшить это время до величины $O(n \lg n)$.