МЕТОД «РАЗДЕЛЯЙ И ВЛАСТВУЙ»

Метод «разделяй и властвуй» состоит из следующих трех этапов.

1. «*Разделение*». Задача разбивается на независимые подзадачи, которые не пересекаются (две задачи назовем независимыми, если они не имеют общих подподзадач).
2. «*Покорение*». Каждая подзадача решается отдельно (рекурсивным методом). Когда объем возникающих подзадач достаточно мал, то подзадачи решаются непосредственно.
3. «*Комбинирование*». Из отдельных решений подзадач строится решение исходной задачи.

Заметим, что если бы подзадачи пересекались, т. е. имели общие подподзадачи, то метод «разделяй и властвуй» делал бы лишнюю работу, решая некоторые подподзадачи по нескольку раз.

Если алгоритм рекурсивно обращается сам к себе, то время его работы описывается с помощью рекуррентного соотношения, в котором время, требуемое для решения всей задачи, выражается через время, необходимое для решения вспомогательных подзадач. Решая данное рекуррентное соотношение, получим функцию, задающую время работы алгоритма (выразив функцию через размерность задачи, получим трудоемкость разработанного алгоритма). Для алгоритмов, основанных на методе «разделяй и властвуй», рекуррентное соотношение часто может быть получено в следующем виде:

$$\begin{cases} T(n) = c_1, n \le c, \\ T(n) = D(n) + aT\left(\dfrac{n}{b}\right) + F(n), n > c, \end{cases}$$

где $D(n)$ – время, затраченное на этап «разделения» исходной задачи на a подзадач размером $1/b$ от размера исходной задачи (не всегда $a = b$); $T(n/b)$ – время, необходимое для решения подзадачи; $F(n)$ – время, затраченное на этап «комбинирования». Начальное условие говорит о том, что когда размер подзадач удовлетворяет неравенству $n \le c$, то данная подзадача решается непосредственно за время $\Theta(1)$.

При разбиении задачи на подзадачи полезен *принцип балансировки*, который предполагает, что задача разбивается на подзадачи приблизительно равных размерностей, т. е. идет поддержание равновесия. Обычно такая стратегия приводит к разделению исходной задачи пополам и обработке каждой из его частей тем же способом до тех пор, пока части не станут настолько малыми, что их можно будет обрабатывать непосредственно.

Часто такой процесс приводит к логарифмическому множителю в формуле, описывающей трудоемкость алгоритма.

Таким образом, в основе техники рассматриваемого метода лежит процедура разделения. Если разделение удается произвести без слишком больших затрат, то может быть построен эффективный алгоритм.

Пример. Отсортировать массив из n элементов, используя технику «разделяй и властвуй» (сортировку слиянием и сортировку Хоара).

Рассмотрим сначала этапы *сортировки слиянием*:

1) «*Разделение*». Сортируемая последовательность элементов разбивается на два сегмента равных размерностей (принцип балансировки), элементы которых не пересекаются. Число арифметических операций этого этапа: $D(n) = c_1, c_1$ – константа.

2) «*Покорение*». Элементы каждого сегмента упорядочиваются. Число арифметических операций этого этапа: $2T\left(n/2\right), n \geq 2$ $(a = b = 2)$. Если $n = 1$, то сегмент отсортирован, т. е. $T(1) = c_3, c_3$ – константа.

3) «*Комбинирование*». Происходит слияние отсортированных сегментов в один. Число арифметических операций этого этапа: $F(n) = c_2 \cdot n,$ – где c_2 – константа.

Таким образом, рекуррентное уравнение для оценки времени работы алгоритма имеет следующий вид:

$$\begin{cases} T(1) = c_3, \\ T(n) = c_1 + 2T\left(\dfrac{n}{2}\right) + c_2 \cdot n, n > 1. \end{cases}$$

Решая данное рекуррентное уравнение, получаем время работы алгоритма $\mathrm{O}(n \log n)$.

Принцип «разделяй и властвуй» лежит в основе еще одного рассмотренного нами ранее эффективного алгоритма сортировки массива – *алгоритма быстрой сортировки Хоара*. Его этапы:

1) «*Разделение*». Сортируемая последовательность элементов разбивается на два сегмента, которые не пересекаются. Если при сортировке слиянием процедура разделения производилась простым делением сортируемой последовательности на два сегмента, то в сортировке Хоара она производится таким образом, что значения элементов из первого сегмента не больше каждого значения из второго сегмента. Число арифметических операций этого этапа: $D(n) = c_1 \cdot n, c_1$ – константа.

2) «*Покорение*». Элементы каждого сегмента упорядочиваются этим же алгоритмом. Число арифметических операций этого этапа:

- в случае выполнения принципа балансировки: $2T\left(n/2\right), n \geq 2$ ($a = b = 2$).
- иначе: $T\left(n_1\right) + T\left(n_2\right), n_1 + n_2 = n$.

Если $n = 1$, то сегмент отсортирован, т. е. $T\left(1\right) = c_3, c_3$ – константа.

3) «*Комбинирование*». Происходит слияние отсортированных сегментов в один путем присоединения сегментов. Число арифметических операций этого этапа: $F\left(n\right) = c_2,$ где c_2 – константа.

Таким образом, рекуррентное соотношение для оценки времени работы алгоритма быстрой сортировки Хоара при выполнении принципа балансировки имеет следующий вид:

$$\begin{cases} T\left(1\right) = c_3, \\ T\left(n\right) = c_1 \cdot n + 2T\left(\dfrac{n}{2}\right) + c_2, n > 1, \end{cases}$$

и время работы алгоритма $\mathrm{O}\left(n \log n\right)$.

В худшем случае, когда процедура разделения разбивает элементы массива на сегменты размерности 1 и $n-1,$ поэтому рекуррентное соотношение для числа операций алгоритма быстрой сортировки Хоара имеет следующий вид:

$$\begin{cases} T\left(1\right) = c_3, \\ T\left(n\right) = c_1 + T\left(1\right) + T\left(n-1\right) + c_2 \cdot n, n > 1, \end{cases}$$

и получаем, что время работы алгоритма $\mathrm{O}\left(n^2\right)$.

Таким образом, пример явно иллюстрирует преимущества принципа балансировки в технике «разделяй и властвуй».

Упражнение. В основе каких алгоритмов внутренней сортировки лежит техника «разделяй и властвуй» (кроме рассмотренных ранее алгоритмов сортировки слиянием и Хоара)? Для каждого из алгоритмов указать этапы метода «разделяй и властвуй» и выписать рекуррентное уравнение для числа арифметических операций.

Пример. Найти максимальный и минимальный элементы массива двумя различными алгоритмами: используя технику «разделяй и властвуй» и без ее использования. Оценить данные алгоритмы по числу операций сравнения в предположении, что в массиве *n* элементов.

Рассмотрим сначала алгоритм решения поставленной задачи, который не основан на технике «разделяй и властвуй». Выбираем из первых двух элементов массива максимальный и минимальный элементы (одно срав-

нение), затем, последовательно просматривая элементы массива, сравниваем каждый последующий из $n-2$ элементов с максимальным и минимальным элементами ($2\cdot(n-2)$ сравнения).

Количество сравнений, которое выполняет алгоритм, есть

$$T(n)=2\cdot(n-2)+1=2n-3.$$

Второй алгоритм решения поставленной задачи основан на технике «разделяй и властвуй». Его этапы:

1) «*Разделение*». Делим массив на две части одинаковых размерностей (балансировка).

2) «*Покорение*». Находим максимальный и минимальный элементы для каждой из частей (рекурсивно).

3) «*Комбинирование*». Выбираем из максимальных элементов наибольший и из минимальных элементов – наименьший.

Рекуррентное уравнение данного алгоритма имеет следующий вид:

$$\begin{cases} T(n)=2T\left(\dfrac{n}{2}\right)+2, \\ T(2)=1. \end{cases}$$

Построим точное решение данного уравнения, используя метод рекурсивных деревьев (рис. 1):

$$T(n)=\sum_{i=1}^{k}2^i+2^k=\frac{2\cdot\left(2^k-1\right)}{2-1}+2^k=\left[\frac{n}{2^k}=2;\quad 2^k=\frac{n}{2};\quad\right]=3\cdot\frac{n}{2}-2.$$

Таким образом, количество сравнений, которое выполняет описанный выше алгоритм, есть

$$T(n)=\frac{3}{2}n-2.$$

Заметим, что асимптотические сложности в первом и втором алгоритмах совпадают и равны $O(n)$, но применение техники «разделяй и властвуй» во втором алгоритме дает меньшую константу при n.

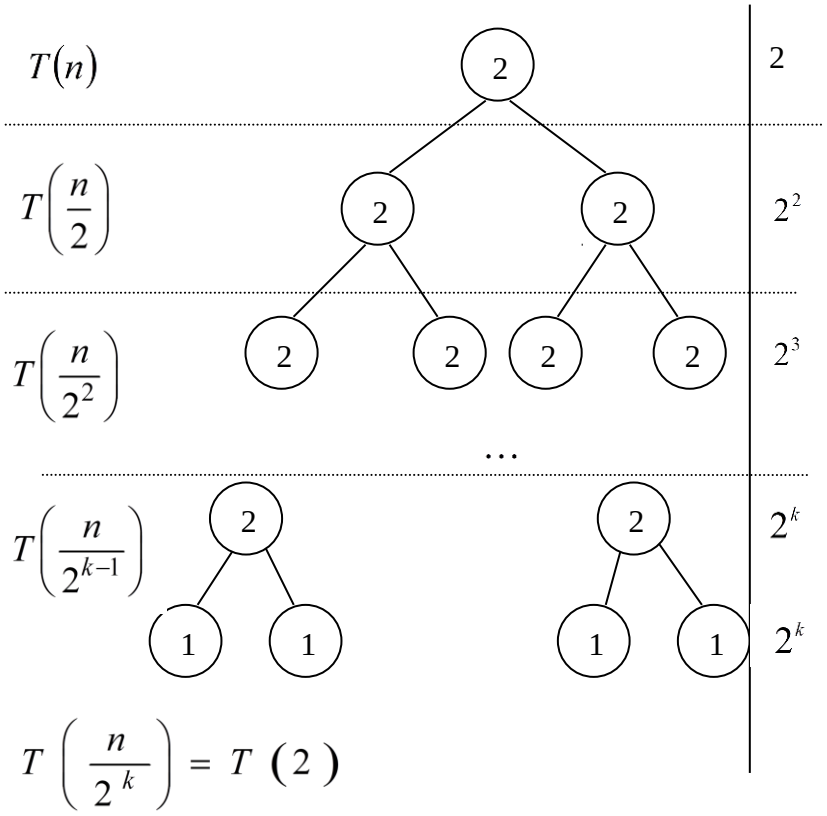

Рис. 1

ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Динамическое программирование, как правило, применяется к задачам оптимизации или к задачам, в которых нужно что-то подсчитать. Например, подсчитать число способов подняться по ступенькам при заданном способе подъема, число способов размещения *k* единиц в строке длины *n*, вычислить F_n число Фиббоначи и др. В оптимизационных задачах существует много решений, каждому из которых поставлено в соответствие некоторое значение, и нам необходимо найти среди всех возможных решений одно с оптимальным (наибольшим или наименьшим) значением. Например, во взвешенном графе между заданной парой вершин существует несколько маршрутов, каждый маршрут характеризуется своей длиной, и нам необходимо найти маршрут кратчайшей длины.

Процесс разработки алгоритмов с использованием метода динамического программирования можно разделить на следующие этапы.

1) Задача погружается в семейство вспомогательных подзадач той же природы. В отличие от метода «разделяй и властвуй», данные подзадачи могут являться зависимыми, т. е. могут пересекаться (различные вспомогательные задачи могут использовать при своем решении оптимальные

решения одних и тех же подзадач). Подзадачи должны удовлетворять следующим двум требованиям:

- Подзадачи должны быть более простыми по отношению к исходной задаче. Понятие «более простая задача» может в разных случаях пониматься по-разному. Задача может быть проще из-за того, что опущены некоторые ограничения. Она может быть проще из-за того, что некоторые ограничения добавлены. Однако, как бы ни была изменена задача, если это изменение приводит к решению более простой задачи, то, возможно, удастся, опираясь на эту более простую, решить и исходную.
- Оптимальное решение исходной задачи определяется через оптимальные решения подзадач. В этом случае говорят, что задача обладает свойством *оптимальной подструктуры*, и это один из аргументов в пользу применения для ее решения метода «динамического программирования».

2) Каждая вспомогательная подзадача решается (рекурсивно) только один раз. Значения оптимальных решений возникающих подзадач запоминаются в таблице (иногда данный метод называют *табличным*), что позволяет не решать повторно ранее решенные подзадачи.

3) Для исходной задачи строится возвратное соотношение, связывающее значение оптимального решения исходной задачи со значениями оптимальных решений вспомогательных подзадач (т. е. методом восходящего анализа от простого к сложному вычисляем значение оптимального решения исходной задачи).

4) Данный этап выполняется в том случае, когда требуется помимо значения оптимального решения получить и само это решение. Часто для этого требуется некоторая вспомогательная информация, полученная на предыдущих этапах метода.

Таким образом, стратегия метода динамического программирования – попытка свести рассматриваемую задачу к более простым задачам, тогда как стратегия предыдущей техники – «разделяй и властвуй». Данная техника находит свое применение, когда все значения оптимальных решений подзадач помещаются в память. Вычисление идет от малых подзадач к большим, и ответы запоминаются в таблице. Одна из клеток таблицы и дает значение оптимального решения исходной задачи.

При реализации принципа динамического программирования (ДП) часто говорят про ДП «*вперёд*», ДП «*назад*» и «*ленивое*» ДП. Продемонстрируем данные подходы на примерер вычисления чисел Фибоначчи. Предположим, что нам нужно вычислить F_n -ое число Фибоначчи.

При ДП «*вперёд*» в качестве базы ДП: $F_1 = 1$, а остальные элементы $F_i, i = 2, \dots, n$, полагаем равными 0. Последовательно просматриваем все состояния i, начиная от 2 и до $n - 1$. Из текущего состояния i подформировываем те состояния, в которые мы можем прийти из него:

$$F_{i+1} = F_{i+1} + F_i,$$
$$F_{i+2} = F_{i+2} + F_i.$$

При ДП «*назад*» в качестве базы ДП берём $F_1 = 1, F_2 = 1$. Последовательно просматриваем все состояния i от 3 до n и формируем текущее состояние i на основании тех состояний, из которых мы можем прийти в него:

$$F_i = F_{i-1} + F_{i-2}.$$

При «*ленивом*» ДП в качестве базы ДП берём $F_1 = 1, F_2 = 1$ и начинаем рекурсивно вычислять F_n. Если на некотором этапе рекурсии доходим до элемента, значение которого уже вычислено, то берем решение из таблицы. Как только удается вычислить некоторое состояние, то сохраняем его значение в таблице, чтобы не решать повторно одну и ту же подзадачу.

Пример. Заданы две строки символов: $A = \{a_1, a_2, \ldots, a_n\}$ и $B = \{b_1, b_2, \ldots, b_m\}$.

Необходимо определить $d(A, B)$ – минимальное число вставок в строку B, удалений и замен символа в строке A, требуемое для преобразования строки A в строку B.

Решение. Рассмотрим все возможные подзадачи данной задачи A_iB_j, которые заключаются в преобразовании некоторой строки $A_i = \{a_1, a_2, \ldots, a_i\}$ в строку $B_j = \{b_1, b_2, \ldots, b_j\}$, и пусть $d[i, j]$ – минимальное количество операций для выполнения этого преобразования.

В табл. 1 указаны все подзадачи A_iB_j при $1 \le i \le n = 2$ и $1 \le j \le m = 3$.

Таблица 1

	$j = 0$	$j = 1$	$j = 2$	$j = 3$
$i = 0$	{ø} {ø}	$\{b_1\}$	{ø} $\{b_1, b_2\}$	{ø} $\{b_1, b_2, b_3\}$
$i = 1$	$\{a_1\}$ {ø}	$\{a_1\}$ $\{b_1\}$	$\{a_1\}$ $\{b_1, b_2\}$	$\{a_1\}$ $\{b_1, b_2, b_3\}$
$i = 2$	$\{a_1, a_2\}$ {ø}	$\{a_1, a_2\}$ $\{b_1\}$	$\{a_1, a_2\}$ $\{b_1, b_2\}$	$\{a_1, a_2\}$ $\{b_1, b_2, b_3\}$

Построим матрицу $D = d[i, j]$ размером $(n+1) \times (m+1)$, $0 \le i \le n$, $0 \le j \le m$. Очевидно, что в матрице $d[0, j] = j, 0 \le j \le m$ – только вставки, и количество вставок равно длине цепочки B_j; $d[i, 0] = i, 0 \le i \le n$ – только удаления, и количество удалений равно длине цепочки A_i.

Рассмотрим смежные задачи:

$\{a_1, a_2, \ldots, a_{i-1}\}$ $\{b_1, b_2, \ldots, b_{j-1}\}$	$\{a_1, a_2, \ldots, a_{i-1}\}$ $\{b_1, b_2, \ldots, b_j\}$
$\{a_1, a_2, \ldots, a_i\}$ $\{b_1, b_2, \ldots, b_{j-1}\}$	$\{a_1, a_2, \ldots, a_i\}$ $\{b_1, b_2, \ldots, b_j\}$

Предположим, что оптимальные решения задач $A_{i-1}B_{j-1}$, A_iB_{j-1}, $A_{i-1}B_j$ уже построены. Покажем, как связаны решения этих задач с решением задачи A_iB_j.

- Если удалить в задаче A_iB_j элемент a_i, то приходим к задаче $A_{i-1}B_j$. Полагаем $d[i, j] = 1 + d[i-1, j]$, где первое слагаемое означает одну операцию удаления.
- Решим задачу A_iB_{j-1}, после чего добавим к строке B_{j-1} элемент b_j. Получаем решение задачи A_iB_j. Полагаем $d[i, j] = d[i, j-1] + 1$, где второе слагаемое означает одну операцию добавления.
- Решим задачу $A_{i-1}B_{j-1}$, после чего одна операция замены элемента a_i на b_j, которая необходима лишь в случае, когда эти элементы не совпадают, приводит к решению задачи A_iB_j. Полагаем

$$d[i, j] = d[i-1, j-1] + \delta(a_i, b_j), \text{ где } \delta(a_i, b_j) = \begin{cases} 0, & \text{åñëè } a_i = b_j, \\ 1, & \text{åñëè } a_i \neq b_j. \end{cases}$$

Таким образом, получаем следующее возвратное соотношение:

$$d[i, j] = \min\{1 + d[i-1, j], d[i, j-1] + 1, d[i-1, j-1] + \delta(a_i, b_j)\}.$$

Проиллюстрируем работу алгоритма на следующем примере:

$$A = \{p, t, s, l, d, d, f\},\ B = \{t, s, g, l, d, d, s\}.$$

		g	***t***	***s***	***g***	***l***	***d***	***d***	***s***
	i\j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0	0	1	2	3	4	5	6	7
p	**1**	1	1	2	3	4	5	6	7
t	**2**	2	1	2	3	4	5	6	7
s	**3**	3	2	1	2	3	4	5	6
l	**4**	4	3	2	2	2	3	4	5
d	**5**	5	4	3	3	3	2	3	4
d	**6**	6	5	4	4	4	3	2	3
f	**7**	7	6	5	5	5	4	3	3

Получаем, что $d[7, 7] = 3$ – минимальное количество требуемых операций для преобразования строки *A* в строку *B*.

Если запоминать задачи, на которых достигался минимум (соответствующие элементы матрицы *D* выделены подчеркиванием), то можно восстановить цепочку операций:

- *p t s l d d f* (удаление *p*) $\Rightarrow$ *t s l d d f*;
- *t s l d d f* (вставка *g*) $\Rightarrow$ *t s g l d d f*;
- *t s g l d d f* (замена *f* на *s*) $\Rightarrow$ *t s g l d d s*.

Пример. Задана группа матриц $A_1, A_1, \ldots, A_s$. Каждая матрица A_i задана размерами n_i, m_i, причем $m_i = n_{i+1}, 1 \le i \le s-1$. Определить, какое минимальное число операций умножения требуется для перемножения *s* матриц, причем перемножать можно любые две рядом стоящие матрицы.

Решение. Порядок перемножения всех *s* матриц неоднозначен. Чтобы устранить неоднозначность, нужно расставить скобки. Порядок расстановки скобок однозначно определит последовательность перемножаемых матриц. Поскольку матричное произведение ассоциативно, то результат не зависит от расстановки скобок, но порядок перемножения может существенно повлиять на время работы алгоритма.

Предположим, что у нас есть четыре матрицы:

$$A_1[10\times 20],\ A_2[20\times 50],\ A_3[50\times 1],\ A_4[1\times 100].$$

Известно, что для перемножения двух матриц размерами $n\times m$ и $m\times k$ требуется $n\cdot m\cdot k$ операций умножения. Возможны пять способов перемножения.

1. $A_1\big((A_2A_3)A_4\big)$ – требуется 23 000 операций умножения.
2. $A_1\big(A_2(A_3A_4)\big)$ – требуется 125 000 операций умножения.
3. $(A_1A_2)(A_3A_4)$ – требуется 65 000 операций умножения.
4. $\big(A_1(A_2A_3)\big)A_4$ – требуется 2 200 операций умножения.
5. $\big((A_1A_2)A_3\big)A_4$ – требуется 11 500 операций умножения.

Из вычислений следует, что способ, который является наименьшим по количеству выполненных операций умножения, – $\big(A_1(A_2A_3)\big)A_4$.

Если не использовать принцип динамического программирования, то процесс перебора всех порядков, в которых можно вычислить рассматриваемое произведение матриц с целью минимизации количества операций умножения, имеет экспоненциальную сложность. Действительно, число способов расстановки скобок в выражении, содержащем *n* множителей равно *n – 1* числу Каталана.

$$\mathrm{C}_{n-1} = \frac{4^{n-1}}{\sqrt{\pi}(n-1)^{3/2}}$$

Динамическое программирование дает алгоритм время работы которого $O\left(n^3\right)$.

Заметим, что перемножать группы матриц с номерами от i до j можно различными способами. На последнем этапе у нас останутся две матрицы. Как можно было получить эти матрицы? Для некоторого выбранного k, $i \le k \le j-1$, сначала перемножаются наилучшим образом матрицы с номерами от i до k (получаем матрицу размером $n_i \times m_k$), затем перемножаются наилучшим образом матрицы с номерами от $k+1$ до j (получаем матрицу размером $n_{k+1} \times m_j$) и затем перемножаем матрицы, полученные на предыдущих шагах за $n_i \cdot m_k \cdot m_j$ операций умножения.

Построим матрицу $F[s \times s]$, где $f[i, j]$, $j \ge i$, – минимальное количество операций умножения для перемножения матриц с номерами от i до j. Тогда

$$\begin{cases} f[i, j] = \min\limits_{i \le k \le j-1} \left(f[i, k] + f[k+1, j] + n_i \cdot m_k \cdot m_j \right), \\ f[i, i] = 0. \end{cases} \tag{6.1}$$

При динамическом программировании $f[i, j]$ вычисляются в порядке возрастания разностей индексов, т. е. начинаем с вычисления $f[i, i] = 0$ для всех $1 \le i \le s$, затем $f[i, i+1]$ для всех $1 \le i \le s-1$, затем $f[i, i+2]$ для всех $1 \le i \le s-2$ и т. д. Это необходимо для того, чтобы $f[i, k]$ и $f[k+1, j]$, $i \le k \le j-1$, уже были известны к моменту вычисления $f[i, j]$. Матрица F задается верхним треугольником, а результат решения задачи соответствует величине $f[1, s]$.

Приведем псевдокод описанного выше алгоритма (реализовано ДП назад). Под величиной ∞ понимается максимальное из возможных оптимальных значений подзадач.

```
для i = 1 до s f[i, j] = 0 ;
для p = 1 до s – 1
  для i = 1 до s – p
      { x = ∞
        j = i + p
        для k = i до j – 1
          если x > (f[i, k] + f[k + 1, j] + n_i · m_k · m_j ) то
                    x = f[i, k] + f[k + 1, j] + n_i · m_k · m_j ;
        f[i, j] : = kol;
      }
```

Для примера, сформируем матрицу F, если

$$A_1[20\times 5],\ A_2[5\times 35],\ A_3[35\times 4],\ A_4[4\times 25].$$

При построении матрицы наряду со значением оптимального решения $f[i,\ j]$ будем хранить то значение k, при котором $f[i,\ j]$ получило свое значение по формуле (6.1).

Первоначально по алгоритму будет сформирована только главная диагональ:

i\j	**1**	**2**	**3**	**4**
1	0			
2		0		
3			0	
4				0

На втором шаге будут сформированы элементы, стоящие над главной диагональю $f[i, i+1]=n_i\cdot m_i\cdot m_{i+1}, 1\le i\le 3$:

i\j	**1**	**2**	**3**	**4**
1	0	3500 *k* = 1		
2		0	700 *k* = 2	
3			0	3500 *k* = 3
4				0

На третьем шаге будут сформированы элементы, стоящие над главной диагональю $f[i, i+2], 1\le i\le 2$:

$$d[1,3]=\min\begin{cases}700+20\cdot 5\cdot 4,\ (k=1)\\ 3500+20\cdot 35\cdot 4,\ (k=2)\end{cases}=1100\ (k=1),$$

$$d[2,4]=\min\begin{cases}3500+35\cdot 4\cdot 25,\ (k=2)\\ 700+5\cdot 4\cdot 25,\ (k=3)\end{cases}=1200\ (k=3).$$

i\j	**1**	**2**	**3**	**4**
1	0	3500 *k* = 1	1100 *k* = 1	
2		0	700 *k* = 2	1200 *k* = 2
3			0	3500 *k* = 3
4				0

На последнем шаге будет сформирован элемент $f[1,4]$:

$$d[1,4]=\min\begin{cases}1200+20\cdot 5\cdot 25,\ (k=1)\\ 3500+3500+20\cdot 35\cdot 25,\ (k=2)\\ 1100+0+20\cdot 4\cdot 25,\ (k=3)\end{cases}=3100\ (k=3).$$

i\j	**1**	**2**	**3**	**4**
1	0	3500 *k* = 1	1100 *k* = 1	3100 *k* = 3
2		0	700 *k* = 2	1200 *k* = 2
3			0	3500 *k* = 3
4				0

Таким образом, минимальное число операций умножения, которое требуется для перемножения матриц

$$A_1[20\times 5],\ A_2[5\times 35],\ A_3[35\times 4],\ A_4[4\times 25],$$

равно 3100.

Используя информацию, полученную в процессе формирования матрицы *F* (сохраненные значения *k*), укажем оптимальный порядок перемножения:

$$\left(A_1\left(A_2A_3\right)\right)A_4.$$

Пример. Рассмотрим задачу распознавания образов. Задан текст *x*, необходимо заменить слово *y* этого текста каким-либо другим словом. Другими словами, необходимо найти вхождение слова *y* в строке *x* (эта задача часто возникает при редактировании текстов, при контекстной замене, при поиске вируса в памяти ЭВМ).

Рассмотрим некоторую строку $x=\{x_1,x_2,\ldots,x_n\}$. Для данной строки определим целочисленную *функцию отказов F*, которая для некоторого $i\le n$ есть наибольшее число $k<i$ такое, что первые *k* символов подстроки $\{x_1,x_2,\ldots,x_i\}$ совпадают с последними ее *k* символами:

$$\{x_1,x_2,\ldots,x_k\}=\{x_{i-k+1},x_{i-k=2},\ldots,x_i\}. \qquad (6.2)$$

В качестве примера посроим функцию отказов для строки $x=\{a,b,a,b,c,a,b,a\}$:

i	1	2	3	4	5	6	7	8
x[*i*]	*a*	*b*	*a*	*b*	*c*	*a*	*b*	*a*
F[*i*]	0	0	1	2	0	1	2	3

Рассмотрим алгоритм построения функции отказов. Предположим, что функция определена для $i = 1, 2, \ldots, k$ и необходимо определить $F[i]$ для $i = k + 1$. Пусть $F[k] = k_0$, т. е. k_0 – первая по длине цепочка, которую хотелось бы увеличить. Поэтому сравниваем x_{k_0+1} с x_i, и если они совпадают, то полагаем $F[i] = k_0 + 1$.

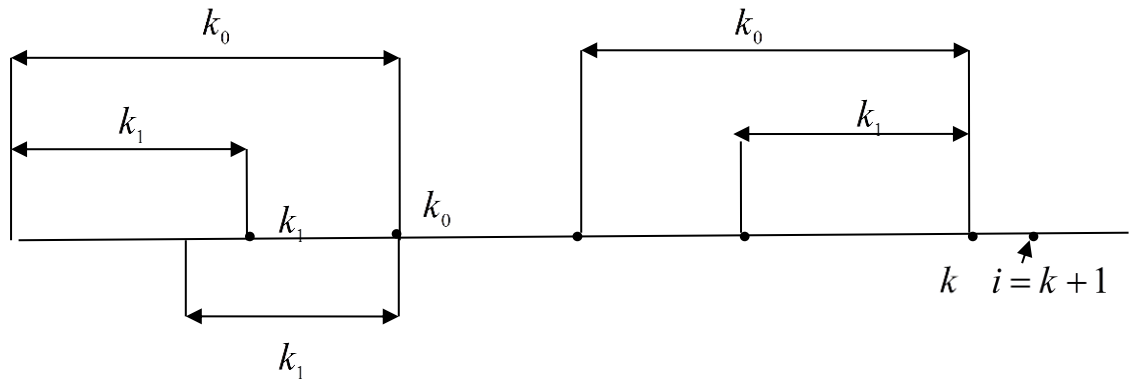

Рис. 6.2

В противном случае найдем вторую по длине цепочку длины $k_1 < k_0$, для которой выполняется свойство (6.2). Из рис. 6.2 видно, что $k_1 = F[k_0]$. Описанные действия повторяем до тех пор, пока:

- найдется $j \geq 0$, для которого выполняется свойство (6.2) и верно равенство $x_{k_j+1} = x_i$; тогда $F[i] = k_j + 1$;
- $k_j = 0$ и $x_1 \neq x_i$; тогда $F[i] = 0$.

Алгоритм построения функции отказов может быть реализован следующим образом.

```
F[1] =0;
 для i  = 2 до n
     { k = F[i – 1]
       пока  (x_{k+1} ≠ x_i) and (k > 0)  k  = F[k]
       если  (k = 0) and (x_1 ≠ x_i) то  F[i] = 0  иначе  F[i] : = k + 1
      }
```

Оценим время работы описанного выше алгоритма. Для некоторого i значение функции отказов $F[i]$ либо увеличивается по сравнению с предыдущим значением $F[i-1]$ на единицу, либо уменьшается. Причем значение функции F уменьшается (не обязательно на 1) столько раз, сколько раз сработал цикл while. Но функция F ограничена снизу нулем, поэтому общее количество уменьшений функции (оно происходит, если

$x_{F[i-1]+1} \neq x_i$, и начинает уменьшаться со значения $F[i]$) не превосходит общего количества увеличений. Количество увеличений функции F не превосходит n. Следовательно, количество уменьшений функции F не будет превосходить n. Таким образом, трудоемкость алгоритма равна количеству увеличений и уменьшений функции отказов F, т. е. $O(n)$.

Используем описанный выше алгоритм для решения задачи определения всех вхождений слова y в строку x. Сначала сцепляем слово y и строку x, вставляя между ними некоторый разделитель (разделитель – символ, который отсутствует как в x, так и в y). Затем для сцепленной строки $y+'*'+x$ строим функцию отказов F. Количество индексов i, для которых $F[i]=\|y\|$ (знак $\|z\|$ обозначает длину строки z), и есть число вхождений слова y в строку x. При этом если $F[i]=\|y\|$, то $j=i-2\cdot\|y\|$ – индекс начала вхождения слова y в строке x.

Для примера, найдем все вхождений слова y = «фпми» в строку x = «фпфпмифпмифп». Для этого сцепим слово y и строку x и построим функцию отказов:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$y + '*' + x$	ф	п	м	и	*	ф	п	ф	п	м	и	ф	п	м	и	ф	п
$F[i]$	0	0	0	0	0	1	2	1	2	3	**4**	1	2	3	**4**	1	2

Поскольку $F[i]=\|y\|=4$ выполняется для $i=11$ и $i=15$, то имеем два вхождения слова y в строку x с третьей и седьмой позиции:

i	1	2	3	4	5	6	7	8	9	10	11	12
$x[i]$	ф	п	**ф**	**п**	**м**	**и**	**ф**	**п**	**м**	**и**	ф	п

Если ставится задача найти в строке x мощности n подстроку y мощности m, то применение техники динамического программирования позволяет получить алгоритм время работы которого $O(n+m)$.

Упражнение. Решить описанную в примере задачу в предположении, что строка y может содержать символы «*» и «?», где «*» – означает любую подпоследовательность символов, а «?» – некоторый одиночный символ.

Упражнение. Доказать, что задача поиска наименьшего элементарного пути обладает свойством оптимальной подструктуры. Решить задачу, используя принцип динамического программирования.

Упражнение. Доказать, что задача поиска наибольшего элементарного пути не обладает свойством оптимальной подструктуры.