Динамическое программирование

Наумов Д.А., доц. каф. КТ

Алгоритмы и структуры данных, 2021

Содержание лекции

Динамическое программирование, как правило, применяется к задачам оптимизации (optimization problems). Такая задача может иметь много возможных решений. С каждым вариантом решения можно сопоставить какое-то значение, и нам нужно найти среди них решение с оптимальным (минимальным или максимальным) значением.

1. Описание структуры оптимального решения.
2. Определение значения, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

Этапы 1-3 составляют основу метода динамического программирования для решения задач. Этап 4 может быть опущен, если требуется узнать только значение, соответствующее оптимальному решению.

Постановка задачи

Компания покупает длинные стальные стержни, режет их на куски и продает. Сама порезка стержней не стоит компании ни копейки. Руководство компании хочет знать, как лучше всего разрезать стержни на части.

Длина i	1	2	3	4	5	6	7	8	9	10
Цена p_i	1	5	8	9	10	17	17	20	24	30

Рис.: Пример таблицы цен отрезков стержня

Имеются стержень длиной n и таблица цен p_i для $i = 1, 2, ..., n$. Необходимо найти максимальную прибыль r_n, получаемую при разрезании стержня и продаже полученных кусков. Если цена p_n стержня длиной n достаточно велика, оптимальное решение может состоять в продаже стержня целиком, без разрезов.

Разрезание стержня

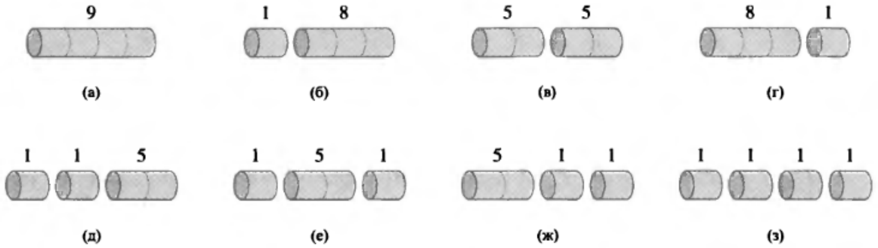


Рис.: Восемь способов разрезания стержня

Разрезание стержня

Стержень длиной n можно разрезать 2^{n-1} разными способами, поскольку мы можем независимо выбирать, резать его или нет на расстоянии i от левого конца, где $i = 1, 2, ..., n-1$.

$$
\begin{aligned}
r_1 &= 1 && \text{из решения } 1 = 1 \quad \text{(без разрезов)}\,, \\
r_2 &= 5 && \text{из решения } 2 = 2 \quad \text{(без разрезов)}\,, \\
r_3 &= 8 && \text{из решения } 3 = 3 \quad \text{(без разрезов)}\,, \\
r_4 &= 10 && \text{из решения } 4 = 2 + 2\,, \\
r_5 &= 13 && \text{из решения } 5 = 2 + 3\,, \\
r_6 &= 17 && \text{из решения } 6 = 6 \quad \text{(без разрезов)}\,, \\
r_7 &= 18 && \text{из решения } 7 = 1 + 6 \;\; \text{или} \;\; 7 = 2 + 2 + 3\,, \\
r_8 &= 22 && \text{из решения } 8 = 2 + 6\,, \\
r_9 &= 25 && \text{из решения } 9 = 3 + 6\,, \\
r_{10} &= 30 && \text{из решения } 10 = 10 \quad \text{(без разрезов)}\,.
\end{aligned}
$$

Рис.: Расчет максимальной прибыли

Разрезание стержня

В общем случае можно записать значения r_n для $n > 1$ через оптимальные прибыли от более коротких стержней:

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_n)$$

- Первый аргумент, p_n соответствует продаже стержня длиной n как есть, без разрезов.
- Прочие $n - 1$ аргументов функции max соответствуют максимальным доходам, получаемым при первоначальном разрезании стержня на две части размерами i и $n - i$, для каждого $i = 1, 2, ..., n - 1$, с последующим оптимальным разрезанием второй части.

Придется рассмотреть все возможные значения i и выбрать из них то, которое максимизирует доход.

Кроме того, возможно, следует не выбирать ни одного значения i вообще, а предпочесть продавать стержни неразрезанными.

Разрезание стержня

1. Для решения исходной задачи размером n мы решаем меньшие задачи того же вида.
2. Как только мы сделали первый разрез, мы можем рассматривать две части стержня как независимые экземпляры задачи разрезания стержня.
3. Общее оптимальное решение включает оптимальные решения двух связанных подзадач, максимизирующих доходы от каждой из двух частей стержня.

Задача разрезания стержня

демонстрирует оптимальную подструктуру: оптимальное решение задачи включает оптимальные решения подзадач, которые могут быть решены независимо.

Разрезание стержня

- Мы рассматриваем разрезание стержня как состоящее из первой части длиной i и остатка длиной $n - i$. Первая часть далее не разрезается; резать можно только остаток.
- Таким образом, мы можем рассматривать любое разрезание стержня длиной n как первую часть и некоторое разделение на части остатка стержня без первой части.
- При этом допускается и решение без разрезов, если первая часть имеет размер $i = n$ и дает прибыль r_n, а остаток длиной 0 дает нулевой доход.

В итоге мы получаем более простую версию уравнения:

$$r_n = max(r_i + r_{n-i}), i = 1..n$$

Разрезание стержня

```
Cut-Rod(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n − i))
return q
```

Рис.: Простой рекусивный способ вычислений

Разрезание стержня

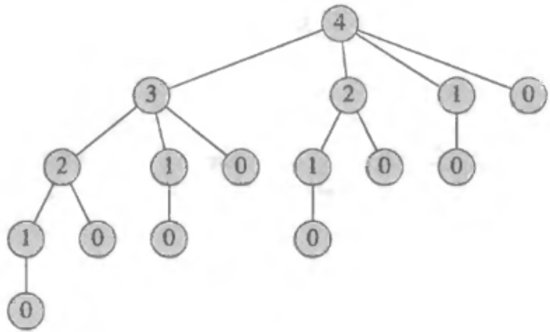


Рис.: Дерево рекурсии

Процедура явным образом рассматривает все 2^{n-1} возможных способа разрезания стержня длиной n.

Разрезание стержня

- Имеющееся рекурсивное решение неэффективно из-за того, что многократно решаются одни и те же подзадачи, мы будем сохранять их решения, тем самым добиваясь только однократного решения подзадач.
- Если позже нам вновь придется решать такую подзадачу, мы просто найдем ее ответ, не решая ее заново.
- Таким образом, динамическое программирование использует дополнительную память для экономии времени вычисления; это один из примеров **пространственно-временного компромисса**.

Нисходящая рекурсия

Нисходящая рекурсия с запоминанием

1. При таком подходе мы пишем процедуру рекурсивно, как обычно, но модифицируем ее таким образом, чтобы она запоминала решение каждой подзадачи (обычно в массиве или хеш-таблице).
2. Теперь процедура первым делом проверяет, не была ли эта задача решена ранее. Если была, то возвращается сохраненное значение (и экономятся вычисления на данном уровне).
3. Если же подзадача еще не решалась, процедура вычисляет возвращаемое значение, как обычно. Мы говорим, что данная рекурсивная процедура с запоминанием — она запоминает вычисленный ею результат.

Разрезание стержня

```
MEMOIZED-CUT-ROD(p, n)
r[0..n] — новый массив
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Рис.: Нисходящий подход

Восходящий подход

Восходящий подход

1. Восходящий подход зависит от некоторого естественного понятия “размера” подзадачи, такого, что решение любой конкретной подзадачи зависит только от решения “меньших” подзадач.
2. Мы сортируем подзадачи по размерам в возрастающем порядке. При решении определенной подзадачи необходимо решить все меньшие подзадачи, от которых она зависит, и сохранить полученные решения.
3. Каждую подзадачу мы решаем только один раз, и к моменту, когда мы впервые с ней сталкиваемся, все необходимые для ее решения подзадачи уже решены.

Разрезание стержня

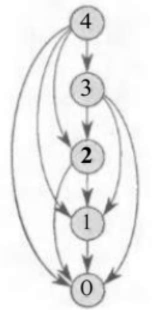
```
MEMOIZED-CUT-ROD(p, n)
r[0..n] — новый массив
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Рис.: Восходящий подход

Граф подзадач

ориентированный граф, содержащий по одной вершине для каждой из различных подзадач.

Граф подзадач содержит дугу, идущую от вершины подзадачи x к вершине подзадачи y, если определение оптимального решения подзадачи x непосредственно включает поиск оптимального решения для подзадачи y.



Восстановление решения

Можно расширить подход динамического программирования и записывать не только вычисленное оптимальное значение каждой подзадачи, но и выбор, который приводит к этому оптимальному значению.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
r[0..n] и s[0..n] — новые массивы
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r и s
```

Восстановление решения

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Рис.: Результаты работы алгоритмов

Постановка задачи 2: перемножение матриц

Пусть имеется последовательность (цепочка)

$$A_1, A_2, \ldots, A_n,$$

состоящая из n матриц, и нужно вычислить их произведение.

Если задана последовательность четырех матриц, то способ вычисления их произведения можно полностью определить с помощью скобок пятью разными способами:

$$\begin{aligned}
&(A_1(A_2(A_3A_4)))\ ,\\
&(A_1((A_2A_3)A_4))\ ,\\
&((A_1A_2)(A_3A_4))\ ,\\
&((A_1(A_2A_3))A_4)\ ,\\
&(((A_1A_2)A_3)A_4)\ .
\end{aligned}$$

Постановка задачи

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error “несовместимые размеры матриц”
else C — новая матрица размером A.rows × B.columns
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

Если A — это матрица размером pxq, а B — матрица размером qxr, то в результате их перемножения получится матрица C размером r.
Время вычисления матрицы преимущественно определяется количеством произведений скаляров, которое равно $p \cdot q \cdot r$.

Постановка задачи

От того, как расставлены скобки при перемножении последовательности матриц, может сильно зависеть время, затраченное на вычисление произведения.

$$A_1(10, 100)$$

$$A_2(100, 5)$$

$$A_3(5, 50)$$

Определить количество операций умножения для двух вариантов расстановки скобок.

Для заданной последовательности n матриц $A_1, A_2, ..., A_n$, в которой матрица $A_i, i = 1..n$, имеет размер $_{i-1}\,_i$, с помощью скобок следует полностью определить порядок умножений, при котором количество скалярных умножений сведется к минимуму.

Подсчет количества способов расстановки скобок

$$P(n) = \begin{cases} 1, & \text{если n=1} \\ \sum_{=1}^{n-1} P(k)P(n-k) & \text{если n>=2} \end{cases}$$

1. Описание структуры оптимального решения.
2. Определение значения, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.

1. Структура оптимальной расстановки скобок

Обозначим для удобства результат перемножения матриц $A_i A_{i+1} \ldots A_j$ через $A_{i..j}$, где $i < j$.

Заметим, что если задача нетривиальна, т.е. $i < j$, то любой способ расстановки скобок в произведении $A_i \cdot \ldots \cdot A_j$ разбивает это произведение между матрицами A_k и A_{k+1}, где k - целое, удовлетворяющее условию $i < k < j$.

Таким образом, при некотором k сначала выполняется вычисление матриц $A_{i..k}$ и и A_{k_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 матриц $A_{i..k}$, A_{k_j} и их произведения.

2. Рекурсивное решение

Путь $m[i,j]$ - минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i..j}$.
Тогда в полной задаче минимальная стоимость матрицы $_{1..n}$ равна $m[1,n]$.

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

В этой рекурсивной формуле предполагается, что значение k известно, но на самом деле это не так. Для выбора этого значения всего имеется $j-i$ возможностей - $k = i, i+1, ..., j$.

$$m[i,j] = \begin{cases} 0, & \text{если i=j} \\ \min(m[i,k] + m[k+1,j+p_{i-1}p_kp_j), i <= k <= j & \text{если i<j} \end{cases}$$

Обозначим через $s[i,j]$ значение k, в котором последовательность $A_{i..j}$ разбивается на две подпоследовательности в процессе оптимальной расстановки скобок.

3. Вычисление оптимальных стоимостей

Размеры матриц A_i равны $p_{i-1}xp_i(i = 1, 2, ..., n)$.

Входные данные представляют собой последовательность $p = (p_0, p_1, ..., p_n)$, длина данной последовательности равна $n + 1$.

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 $m[i, j]$.

При $k = i, i + 1, .., j - 1$ матрица $A_{i..k}$ представляет собой произведение $k - i + 1 < j - i + 1$ матриц, а матрица $A_{k..j}$ - произведение $j - \ < j - i + 1$ матриц. Таким образом, в ходе выполнения алгоритма следует организовать заполнение таблицы m в порядке возрастания длин последовательностей матриц.

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
m[1..n, 1..n] и s[1..n − 1, 2..n] — новые таблицы
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l — длина цепочки
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m и s
```

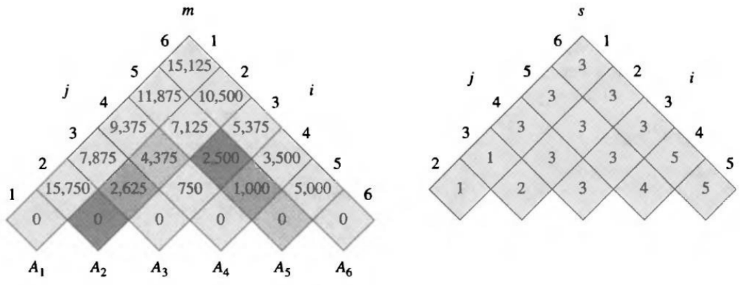


Рис.: Вычисленные значения таблиц m и s (n=6) для Matrix-Change-Order

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

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13\,000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11\,375 \end{cases}$$
$$= 7125\,.$$

4. Построение оптимального решения

Оптимальное вычисление произведения матриц $A_{1..n}$ выглядит как $A_{1..s[1,n]}A_{s[1,n+1]..n}$.

Все предшествующие произведения матриц можно вычислить рекурсивно, поскольку элемент $s[1, s[1, n]]$ определяет матричное умножение, выполняемое последним при вычислении при вычислении $A_{1..s[1,n]}$, а $s[s[1, n] + 1, n]$ - последнее умножение при вычислении $A_{s[1,n]+1..n}$.

4. Построение оптимального решения

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Результат вызова процедуры Print-Optimal-Parens(s, 1, 6):

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Наидлиннейшая общая последовательность

Стандартная ДНК состоит из последовательности молекул, которые называются основаниями (bases).
К этим молекулам относятся:

- аденин (adenine),
- гуанин (guanine),
- цитозин (cytosine)
- тимин (thymine).

Стандартную ДНК можно представить в виде строки, состоящей из конечного множества элементов { A, C, G , T }.

Пример: 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.

Как сравнивать?

1. два кода ДНК подобны, если один из них является подстрокой другого. В нашем примере ни S1, ни S2 не является подстрокой другой ДНК.
2. можно сказать, что две цепочки молекул подобны, если для преобразования одной из них в другую потребовались бы только небольшие изменения.
3. еще один способ определения степени подобия последовательностей S1 и S2 заключается в поиске третьей последовательности S3, основания которой имеются как в S1, так и в S2; при этом они следуют в одном и том же порядке, но не обязательно одно за другим. Чем длиннее последовательность S3, тем более схожи последовательности S1 и S2.

Подпоследовательность

Последовательность $Z = (z_1, z_2, ..., z_m)$ является **подпоследовательностью** последовательности $X = (x_1, x_2, ..., x_m)$, если существует строго возрастающая последовательность $(i_1, i_2, ..., i_k)$ индексов X, такая, что для всех $j = 1, 2, .., k$ выполняется соотношение $x_{i_j} = z_j$.

Например, $Z = (B, C, D, B)$ представляет собой подпоследовательность последовательности $X = (A, B, C, B, D, A, B)$, причем соответствующая ей последовательность индексов имеет вид $(2, 3, 5, 7)$.

Последовательность Z является общей подпоследовательностью последовательностей X и Y, если Z является подпоследовательностью как X, так и Y.

Например, если $X = (A, B, C, B, D, A, B)$ и $Y = (B, B, C, A, B, A)$, то последовательность (B, C, A) является общей

Постановка задачи

В задаче о наидлиннейшей общей подпоследовательности задаются две последовательности, $X = (x_1, x_2, ..., x_n)$ и $Y = (y_1, y_2, ..., y_m)$, и требуется найтиv общую подпоследовательность X и Y максимальной длины.

i-m префиксом последовательности

$X = (x_1, x_2, ..., x_m)$ для $i = 1, 2, ..., m$ является $= (x_1, x_2, ..., x_i)$.

Например, если $X = (A, B, C, B, D, A, B)$, то $X_4 = (A, B, C, D)$, а X_0 представляет собой пустую последовательность.

1. Оптимальная структура LCS

Пусть имеются последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ и пусть $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — наидлиннейшая общая подпоследовательность X и Y.

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 $z_k \neq x_m$ вытекает, что Z представляет собой LCS последовательностей X_{m-1} и Y.
3. Если $x_m \neq y_n$, то из $z_k \neq y_n$ вытекает, что Z представляет собой LCS последовательностей X и Y_{n-1}.

2. Рекурсивное решение

Обозначим через $c[i, j]$ длину наидлиннейшей общей подпоследовательности последовательностей X_i и Y_j.

Если $i = 0$ или $j = 0$, длина одной из этих последовательностей равна нулю, поэтому наидлиннейшая их общая подпоследовательность имеет нулевую длину.
Оптимальная вспомогательная подструктура задачи о наидлиннейшей общей подпоследовательности определяется рекурсивной формулой:

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

3. Вычисление длины наидлиннейшей общей подпоследовательности

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
b[1..m, 1..n] и c[0..m, 0..n] — новые таблицы
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c и b
```

3. Пример вычисления длины наидлиннейшей общей подпоследовательности

i \ j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

4. Построение наидлиннейшей общей подпоследовательности

С помощью таблицы b, которая возвращается процедурой LCS-LENGTH, можно быстро построить самую длинную общую подпоследовательность последовательностей X и Y.

Мы просто начинаем с элемента $b[m, n]$ и проходим таблицу по стрелкам.

Если значением элемента $b[i, j]$ является $\nwarrow$ то элемент $X_i = Y_j$ принадлежит наидлиннейшей общей подпоследовательности.

Элементы LCS, восстановленные этим методом, следуют в обратном порядке.

4. Пример вычисления длины наидлиннейшей общей подпоследовательности

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```