Лекция 4-5. Динамическое программирование

Как строится алгоритм, основанный на динамическом программировании? Надо:

1. описать структуру оптимальных решений,
2. выписать рекуррентное соотношение, связывающее оптимальные значения параметра для подзадач,
3. двигаясь снизу вверх, вычислить оптимальное значение параметра,
4. пользуясь полученной информацией, построить оптимальное решение.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Мы хотим найти произведение

$$A_1 A_2 \ldots A_n \qquad (1)$$

последовательности n матриц $\langle A_1, A_2, \ldots, A_n \rangle$. Мы будем пользоваться стандартным алгоритмом перемножения двух матриц в качестве подпрограммы. Но прежде надо расставить скобки в (.1), чтобы указать порядок умножений. Будем говорить, что в произведении матриц **полностью расставлены скобки** (product is fully parenthesized), если это произведение либо состоит из одной-единственной матрицы, либо является заключенным в скобки произведением двух произведений с полностью расставленными скобками. Поскольку умножение матриц ассоциативно, конечный результат вычислений не зависит от расстановки скобок. Например, в произведении $A_1 A_2 A_3 A_4$ можно полностью расставить скобки пятью разными способами:

$$(A_1(A_2(A_3A_4))); \quad (A_1((A_2A_3)A_4)); \quad ((A_1A_2)(A_3A_4));$$
$$((A_1(A_2A_3))A_4); \quad (((A_1A_2)A_3)A_4);$$

во всех случаях ответ будет один и тот же.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error «умножить нельзя»
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 число столбцов у A равно числу строк у B. Если A — это $p \times q$-матрица, а B — это $q \times r$-матрица, то их произведение C является $p \times r$-матрицей. При выполнении этого алгоритма делается pqr умножений (строка 7) и примерно столько же сложений. Для простоты мы будем учитывать только умножения.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Чтобы увидеть, как расстановка скобок может влиять на стоимость, рассмотрим последовательность из трех матриц $\langle A_1, A_2, A_3 \rangle$ размеров 10×100, 100×5 и 5×50 соответственно. При вычислении $((A_1 A_2) A_3)$ нужно $10 \cdot 100 \cdot 5 = 5000$ умножений, чтобы найти 10×5-матрицу $A_1 A_2$, а затем $10 \cdot 5 \cdot 50 = 2500$ умножений, чтобы умножить эту матрицу на A_3. Всего 7500 умножений. При расстановке скобок $(A_1 (A_2 A_3))$ мы делаем $100 \times 5 \times 50 = 25\,000$ умножений для нахождения 100×50-матрицы $A_2 A_3$, плюс ещё $10 \times 100 \times 50 = 50\,000$ умножений (умножение A_1 на $A_2 A_3$), итого 75 000 умножений. Тем самым, первый способ расстановки скобок в 10 раз выгоднее.

Задача об умножении последовательности матриц (matrix-chain multiplication problem) может быть сформулирована следующим образом: дана последовательность из n матриц $\langle A_1, A_2, \ldots, A_n \rangle$ заданных размеров (матрица A_i имеет размер $p_{i-1} \times p_i$); требуется найти такую (полную) расстановку скобок в произведении $A_1 A_2 \ldots A_n$, чтобы вычисление произведения требовало наименьшего числа умножений.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Количество расстановок скобок

Прежде чем применять динамическое программирование к задаче об умножении последовательности матриц, стоит убедиться, что простой перебор всех возможных расстановок скобок не даст эффективного алгоритма. Обозначим символом $P(n)$ количество полных расстановок скобок в произведении n матриц. Последнее умножение может происходить на границе между k-й и $(k+1)$-й матрицами. До этого мы отдельно вычисляем произведение первых k и остальных $n-k$ матриц. Поэтому

$$P(n) = \begin{cases} 1, & \text{если } n = 1, \\ \sum\limits_{k=1}^{n-1} P(k)P(n-k), & \text{если } n \geqslant 2. \end{cases}$$

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Шаг 1: строение оптимальной расстановки скобок

Если мы собираемся воспользоваться динамическим программированием, то для начала должны описать строение оптимальных решений. Для задачи об умножении последовательности матриц это выглядит следующим образом. Обозначим для удобства через $A_{i..j}$ матрицу, являющуюся произведением $A_i A_{i+1} \dots A_j$. Оптимальная расстановка скобок в произведении $A_1 A_2 \dots A_n$ разрывает последовательность между A_k и A_{k+1} для некоторого k, удовлетворяющего неравенству $1 \leqslant k < n$. Иными словами, при вычислении произведения, диктуемом этой расстановкой скобок, мы сначала вычисляем произведения $A_{1..k}$ и $A_{k+1..n}$, а затем перемножаем их и получаем окончательный ответ $A_{1..n}$. Стало быть, стоимость этой оптимальной расстановки равна стоимости вычисления матрицы $A_{1..k}$, плюс стоимость вычисления матрицы $A_{k+1..n}$, плюс стоимость перемножения этих двух матриц.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Шаг 2: рекуррентное соотношение

Теперь надо выразить стоимость оптимального решения задачи через стоимости оптимальных решений её подзадач. Такими подзадачами будут задачи об оптимальной расстановке скобок в произведениях $A_{i..j} = A_i A_{i+1} \dots A_j$ для $1 \leqslant i \leqslant j \leqslant n$. Обозначим через $m[i,j]$ минимальное количество умножений, необходимое для вычисления матрицы $A_{i..j}$; в частности, стоимость вычисления всего произведения $A_{1..n}$ есть $m[1,n]$.

Поскольку для вычисления произведения $A_{i..k}A_{k+1..j}$ требуется $p_{i-1}p_k p_j$ умножений,

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

В этом соотношении подразумевается, что оптимальное значение k нам известно; на деле это не так. Однако число k может принимать всего лишь $j-i$ различных значений: $i, i+1, \dots, j-1$. Поскольку одно из них оптимально, достаточно перебрать эти значения k и выбрать наилучшее. Получаем рекуррентную формулу:

$$m[i,j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{при } i < j. \end{cases} \qquad (2)$$

Числа $m[i,j]$ — стоимости оптимальных решений подзадач.

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Шаг 3: вычисление оптимальной стоимости

Пользуясь соотношениями (2), теперь легко написать рекурсивный алгоритм, определяющий минимальную стоимость вычисления произведения $A_1 A_2 \ldots A_n$ (т.е. число $m[1, n]$). Однако время работы такого алгоритма экспоненциально зависит от n, так что этот алгоритм не лучше полного перебора. Настоящий выигрыш во времени мы получим, если воспользуемся тем, что подзадач относительно немного: по одной задаче для каждой пары (i, j), для которой $1 \leqslant i \leqslant j \leqslant n$, а всего $C_n^2 + n = \Theta(n^2)$. Экспоненциальное время работы возникает потому, что рекурсивный алгоритм решает каждую из подзадач по многу раз, на разных ветвях дерева рекурсии. Такое «перекрытие подзадач» — характерный признак задач, решаемых методом динамического программирования.

Вместо рекурсии мы вычислим оптимальную стоимость «снизу вверх». В нижеследующей программе предполагается, что матрица A_i имеет размер $p_{i-1} \times p_i$ при $i = 1, 2, \ldots, n$. На вход подаётся последовательность $p = \langle p_0, p_1, \ldots, p_n \rangle$, где $length[p] = n + 1$. Программа использует вспомогательные таблицы $m[1..n, 1..n]$ (для хранения стоимостей $m[i, j]$) и $s[1..n, 1..n]$ (в ней отмечается, при каком k достигается оптимальная стоимость при вычислении $m[i, j]$).

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

```
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
 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, s
```

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

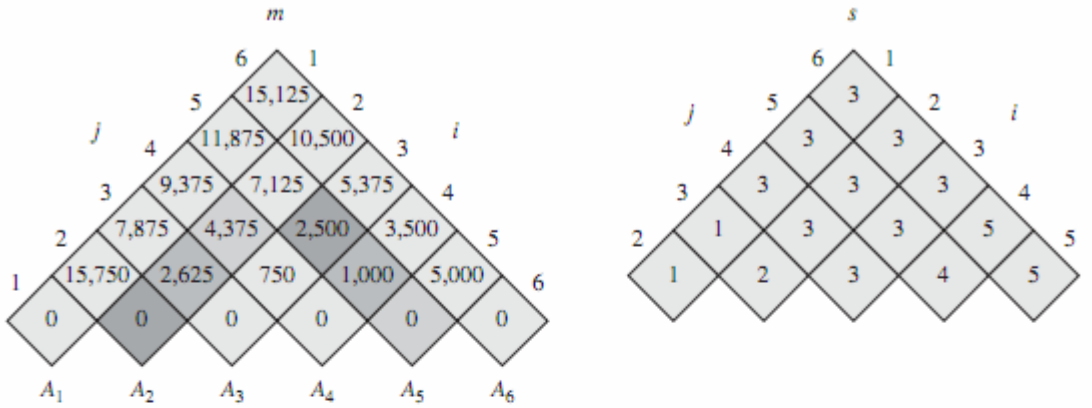

Рис. 1 Таблицы m и s, вычисляемые процедурой MATRIX-CHAIN-ORDER для $n = 6$ и матриц следующего размера:

матрица	размер
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Таблицы повёрнуты так, что главная диагональ горизонтальна. В таблице m используются только клетки, лежащие не ниже главной диагонали, в таблице s — только клетки, лежащие строго выше. Минимальное количество умножений, необходимое для перемножения всех шести матриц, равно $m[1, 6] = 15\,125$. Пары клеточек, заштрихованных одинаковой светлой штриховкой, совместно входят в правую часть формулы в процессе вычисления $m[2, 5]$ (строка 9 процедуры MATRIX-CHAIN-ORDER):

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

Лекция 4-5. Динамическое программирование

Перемножение нескольких матриц

Шаг 4: построение оптимального решения

Алгоритм MATRIX-CHAIN-ORDER находит минимальное число умножений, необходимое для перемножения последовательности матриц. Осталось найти расстановку скобок, приводящую к такому числу умножений.

MATRIX-CHAIN-MULTIPLY(A, s, i, j)

```
1  if j > i
2     then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
3          Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
4          return MATRIX-MULTIPLY(X, Y)
5     else return A_i
```

В примере на рис. 1 вызов MATRIX-CHAIN-MULTIPLY$(A, s, 1, 6)$ вычислит произведение шести матриц в соответствии с расстановкой скобок

$$((A_1(A_2A_3))((A_4A_5)A_6)). \qquad (3)$$

Лекция 4-5. Динамическое программирование

Когда применимо динамическое программирование

Оптимальность для подзадач

При решении оптимизационной задачи с помощью динамического программирования необходимо сначала описать структуру оптимального решения. Будем говорить, что задача **обладает свойством оптимальности для задач** (has optimal substructure), если оптимальное решение задачи содержит оптимальные решения её подзадач. Если задача обладает этим свойством, то динамическое программирование может оказаться полезным для её решения (а возможно, применим и жадный алгоритм).

В разделе 16.1 мы видели, что задача перемножения матриц обладает свойством оптимальности для подзадач: каждая скобка в оптимальном произведении указывает оптимальный способ перемножения входящих в неё матриц. Чтобы убедиться, что задача обладает этим свойством, надо ( ) показать, что, улучшая решение подзадачи, мы улучшим и решение исходной задачи.

Лекция 4-5. Динамическое программирование

Когда применимо динамическое программирование

Перекрывающиеся подзадачи

Второй свойство задач, необходимое для использования динамического программирования, — малость множества подзадач. Благодаря этому при рекурсивном решении задачи мы всё время выходим на одни и те же подзадачи. В таком случае говорят, что у оптимизационной задачи имеются **перекрывающиеся подзадачи** (overlapping subproblems). В типичных случаях количество подзадач полиномиально зависит от размера исходных данных.

В задачах, решаемых методом «разделяй и властвуй», так не бывает: для них рекурсивный алгоритм, как правило, на каждом шаге порождает совершенно новые подзадачи. Алгоритмы, основанные на динамическом программировании, используют перекрытие подзадач следующим образом: каждая из подзадач решается только один раз, и ответ заносится в специальную таблицу; когда эта же подзадача встречается снова, программа не тратит время на её решение, а берёт готовый ответ из таблицы.

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Подпоследовательность получается из данной последовательности, если удалить некоторые её элементы (сама последовательность также считается своей подпоследовательностью). Формально: последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ называется **подпоследовательностью** (subsequence) последовательности $X = \langle x_1, x_2, \ldots, x_n \rangle$, если существует строго возрастающая последовательность индексов $\langle i_1, i_2, \ldots, i_k \rangle$, для которой $z_j = x_{i_j}$ при всех $j = 1, 2, \ldots, k$. Например, $Z = \langle B, C, D, B \rangle$ является подпоследовательностью последовательности $X = \langle A, B, C, B, D, A, B \rangle$; соответствующая последовательность индексов есть $\langle 2, 3, 5, 7 \rangle$. (Отметим, что говоря о последовательностях, мы — в отличие от курсов математического анализа — имеем в виду конечные последовательности.)

Будем говорить, что последовательность Z является **общей подпоследовательностью** (common subsequence) последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Пример: $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$, $Z = \langle B, C, A \rangle$. Последовательность Z в этом примере — не самая длинная из общих подпоследовательностей X и Y (последовательность $\langle B, C, B, A \rangle$ длиннее). Последовательность $\langle B, C, B, A \rangle$ будет наибольшей общей подпоследовательностью для X и Y, поскольку общих подпоследовательностей длины 5 у них нет. Наибольших общих подпоследовательностей может быть несколько. Например, $\langle B, D, A, B \rangle$ — другая наибольшая общая подпоследовательность X и Y.

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Строение наибольшей общей подпоследовательности

Если решать задачу о НОП «в лоб», перебирая все подпоследовательности последовательности X и проверяя для каждой из них, не будет ли она подпоследовательностью последовательности Y, то алгоритм будет работать экспоненциальное время, поскольку последовательность длины m имеет 2^m подпоследовательностей (столько же, сколько подмножеств у множества $\{1, 2, \ldots, m\}$).

Однако задача о НОП обладает свойством оптимальности для подзадач, как показывает теорема 16.1 (см. ниже). Подходящее множество подзадач — множество пар префиксов двух данных последовательностей. Пусть $X = \langle x_1, x_2, \ldots, x_m \rangle$ — некоторая последовательность. Её **префикс** (prefix) длины i — это последовательность $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ (при i от 0 до m). 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.

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Теорема 1 (о строении НОП). *Пусть* $Z = \langle z_1, z_2, \ldots, z_k \rangle$ — *одна из наибольших общих подпоследовательностей для* $X = \langle x_1, x_2, \ldots, x_m \rangle$ *и* $Y = \langle y_1, y_2, \ldots, y_n \rangle$. *Тогда:*

1. *если* $x_m = y_n$, *то* $z_k = x_m = y_n$ *и* Z_{k-1} *является НОП для* X_{m-1} *и* Y_{n-1};
2. *если* $x_m \neq y_n$ *и* $z_k \neq x_m$, *то* Z *является НОП для* X_{m-1} *и* Y;
3. *если* $x_m \neq y_n$ *и* $z_k \neq y_n$, *то* Z *является НОП для* X_m *и* Y_{n-1}.

Доказательство. (1) Если $z_k \neq x_m$, то мы можем дописать $x_m = y_n$ в конец последовательности Z и получить общую подпоследовательность длины $k+1$, что противоречит условию. Стало быть, $z_k = x_m = y_n$. Если у последовательностей X_{m-1} и Y_{n-1} есть более длинная (чем Z_{k-1}) общая подпоследовательность, то мы можем дописать к ней $x_m = y_n$ и получить общую подпоследовательность для X и Y, более длинную, чем Z — противоречие.

(2) Коль скоро $z_k \neq x_m$, последовательность Z является общей подпоследовательностью для X_{m-1} и Y. Так как Z — НОП для X и Y, то она тем более является НОП для X_{m-1} и Y.

(3) Аналогично (2). □

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Рекуррентная формула

Теорема 16.1 показывает, что нахождение НОП последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$ сводится к решению либо одной, либо двух подзадач. Если $x_m = y_n$, то достаточно найти НОП последовательностей X_{m-1} и Y_{n-1} и дописать к ней в конце $x_m = y_n$. Если же $x_m \neq y_n$, то надо решить две подзадачи: найти НОП для X_{m-1} и Y, а затем найти НОП для X и Y_{n-1}. Более длинная из них и будет служить НОП для X и Y.

Теперь сразу видно, что возникает перекрытие подзадач. Действительно, чтобы найти НОП X и Y, нам может понадобиться найти НОП X_{m-1} и Y, а также НОП X и Y_{n-1}; каждая из этих задач содержит подзадачу нахождения НОП для X_{m-1} и Y_{n-1}. Аналогичные перекрытия будут встречаться и далее.

Как и в задаче перемножения последовательности матриц, мы начнём с рекуррентного соотношения для стоимости оптимального решения. Пусть $c[i,j]$ обозначает длину НОП для последовательностей X_i и Y_j. Если i или j равно нулю, то одна из двух последовательностей пуста, так что $c[i,j] = 0$. Сказанное выше можно записать так:

$$c[i,j] = \begin{cases} 0 & \text{если } i = 0 \text{ или } j = 0, \\ c[i-1,j-1]+1 & \text{если } i,j > 0 \text{ и } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]), & \text{если } i,j > 0 \text{ и } x_i \neq y_j. \end{cases} \quad (5)$$

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Вычисление длины НОП

Исходя из соотношения (5), легко написать рекурсивный алгоритм, работающий экспоненциальное время и вычисляющий длину НОП двух данных последовательностей. Но поскольку различных подзадач всего $\Theta(mn)$, лучше воспользоваться динамическим программированием.

Исходными данными для алгоритма LCS-LENGTH служат последовательности $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Числа $c[i,j]$ записываются в таблицу $c[0\,..\,m, 0\,..\,n]$ в таком порядке: сначала заполняется слева направо первая строка, затем вторая, и т.д. Кроме того, алгоритм запоминает в таблице $b[1\,..\,m, 1\,..\,n]$ «происхождение» $c[i,j]$: в клетку $b[i,j]$ заносится стрелка, указывающая на клетку с координатами $(i-1, j-1)$, $(i-1, j)$ или $(i, j-1)$, в зависимости от того, равно ли $c[i,j]$ числу $c[i-1, j-1]+1$, $c[i-1, j]$ или $c[i, j-1]$ (см. (5)). Результатами работы алгоритма являются таблицы c и b.

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

	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

Рис. 3 Таблицы c и b, созданные алгоритмом LCS-LENGTH при $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$. В клетке с координатами (i, j) записаны число $c[i, j]$ и стрелка $b[i, j]$. Число 4 в правой нижней клетке есть длина НОП. При $i, j > 0$ значение $c[i, j]$ определяется тем, равны ли x_i и y_j, и вычисленными ранее значениями $c[i - 1, j]$, $c[i, j - 1]$ и $c[i - 1, j - 1]$. Путь по стрелкам, ведущий из $c[7, 6]$, заштрихован. Каждая косая стрелка на этом пути соответствует элементу НОП (эти элементы выделены).

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

```
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, b
```

На рис. 3 показана работа LCS-LENGTH для $X = \langle A, B, C, B, D, A, B \rangle$ и $Y = \langle B, D, C, A, B, A \rangle$.

Алгоритм LCS-LENGTH требует времени $O(mn)$: на каждую клетку требуется $O(1)$ шагов.

Лекция 4-5. Динамическое программирование

Наибольшая общая подпоследовательность

Построение НОП

Таблица b, созданная процедурой LCS-LENGTH, позволяет быстро найти НОП последовательностей $X = \langle x_1, x_2, \ldots, x_m \rangle$ и $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Для этого надо пройти по пути, указанному стрелками, начиная с $b[m, n]$. Пройденная стрелка $\nwarrow$ в клетке (i, j) означает, что $x_i = y_j$ входит в наибольшую общую подпоследовательность. Вот как это реализовано в рекурсивной процедуре PRINT-LCS (НОП для X и Y печатается при вызове 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          напечатать 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)
```

Будучи применённой к таблице рис. 3, эта процедура напечатает $BCBA$. Время работы процедуры есть $O(m + n)$, поскольку на каждом шаге хотя бы одно из чисел m и n уменьшается.