Динамическое программирование

Дискретный анализ 2012/13

Андрей Калинин, Татьяна Романова

2 марта 2013 г.

Литература

- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.. Алгоритмы: построение и анализ, 2-е издание, М.:Вильямс, 2005, стр. 386-441, глава 15, «Динамическое программирование».

Раздел

Метод декомпозиции

«Разделяй и властвуй»:

1. Разделение задачи на несколько подзадач.
2. Покорение — рекурсивное решение этих подзадач.
3. Комбинирование решения исходной задачи из решения вспомогательных задач.

Метод декомпозиции

«Разделяй и властвуй»:

1. Разделение задачи на несколько подзадач.
2. Покорение — рекурсивное решение этих подзадач.
3. Комбинирование решения исходной задачи из решения вспомогательных задач.

Пример — алгоритм сортировки слиянием. Важное условие: подзадачи должны быть независимыми!

Метод динамического программирования

- Не алгоритм, а метод создания алгоритмов.
- Программирование — табличный метод. Был разработан тогда, когда компьютеров не существовало.
- Используется для решения задач оптимизации:
  - Найти решение с оптимальным значением.
  - Минимизация или максимизация.

Четыре этапа

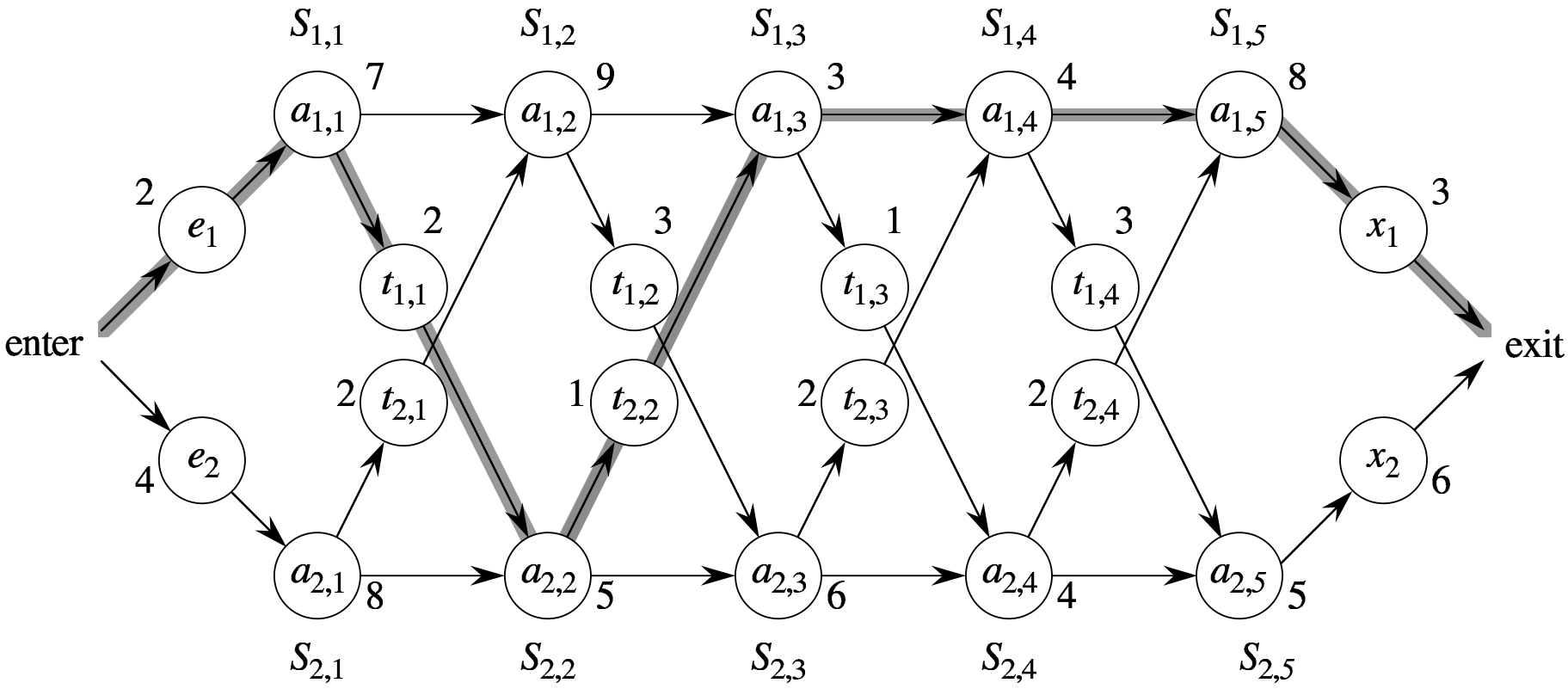
1. Описать структуру оптимального решения.
2. Составить рекурсивное решение для нахождения оптимального решения.
3. Вычисление значения, соответствующего оптимальному решению, методом восходящего анализа.
4. Непосредственное нахождение оптимального решения из полученной на предыдущих этапах информации.

Раздел

Постановка задачи

1. Есть два конвеера, по n рабочих мест.
2. $S_{i,j}$ — рабочее место под номером j на i-ом конвеере.
3. Время выполнения операций разное: $a_{i,j}$ на месте $S_{i,j}$.
4. Постановка на конвеер занимает время e_i, снятие — x_i.
5. Время на перемещение шасси с конвеера i на соседний с места $S_{i,j}$ равно $t_{i,j}$.

Определение оптимального способа сборки



Постановка задачи

Задача: При всех заданных значениях $e_i, a_{i,j}, t_{i,j}, x_i$ определить самый быстрой способ сборки одного шасси используя оба конвеера.

Постановка задачи

Задача: При всех заданных значениях $e_i, a_{i,j}, t_{i,j}, x_i$ определить самый быстрой способ сборки одного шасси используя оба конвеера.

Перебором решить нельзя:

- Сущуествует 2^n возможных комбинаций.
- Неприемлимо при больших n.

Первый этап: структура оптимального решения

Подумаем о самом быстром пути от входа до «за $S_{1,j}$»:

- $j = 1$, просто: нужно только определить сколько времени займёт прохождение через $S_{1,1}$.
- $j \geq 2$, есть два варианта попасть к $S_{1,j}$:
  - Через $S_{1,j-1}$, затем прямо в $S_{1,j}$.
  - Через $S_{2,j-1}$, затем переместиться на другой конвеер и попасть в $S_{1,j}$.

Первый этап: структура оптимального решения

Подумаем о самом быстром пути от входа до «за $S_{1,j}$»:

- $j = 1$, просто: нужно только определить сколько времени займёт прохождение через $S_{1,1}$.
- $j \geq 2$, есть два варианта попасть к $S_{1,j}$:
  - Через $S_{1,j-1}$, затем прямо в $S_{1,j}$.
  - Через $S_{2,j-1}$, затем переместиться на другой конвеер и попасть в $S_{1,j}$.

Оптимальное решение задачи (самый быстрый путь через $S_{1,j}$) содержит в себе оптимальное решение подзадач (самый быстрый путь через $S_{1,j-1}$ или $S_{2,j-1}$). Это свойство назовём *оптимальной подструктурой*.

Второй этап: рекурсивное решение

- $f_i[j]$ — самое быстрое время пройти этап $S_{i,j}$.
- Цель: найти самое быстрое время пройти все этапы конвеера, f^*:

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

- В начале:

$$f_1[1] = e_1 + a_{1,1}$$
$$f_2[1] = e_2 + a_{2,1}$$

- Для остальных $j = 2, \ldots n$:

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

Сохранение пути

$f_i[j]$ сохраняет только оптимальное значение, восстановить по нему путь сложно. Для упрощения введём дополнительный массив:

- $l_i[j]$ — номер конвеера, откуда была пришёл на j-ю позицию объект.
- l^* — конвеер, на котором объект побывал в n-ой позиции.

Третий этап: вычисление минимальных промежутков времени

Можно было бы реализовать рекурсию и вычислить «сверху вниз», однако это будет слишком долго. Введём значения $r_i(j)$ — количество обращений к величине $f_i[j]$ в рекурсивном алгоритме. Тогда:

- $r_1(n) = r_2(n) = 1$.
- $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$ для $j = 1, \ldots, n-1$. Покажем, что $r_i(j) = 2^{n-j}$:
  1. $j = n$, тогда $2^{n-j} = 2^0 = 1 = r_i(n)$.
  2. Предположим, что $r_i(j+1) = 2^{n-(j+1)}$, тогда:

$$\begin{aligned} r_i j &= r_i(j+1) + r_2(j+1) \\ &= 2^{n-(j+1)} + 2^{n-(j+1)} \\ &= 2^{n-(j+1)_1} = 2^{n-j} \end{aligned}$$

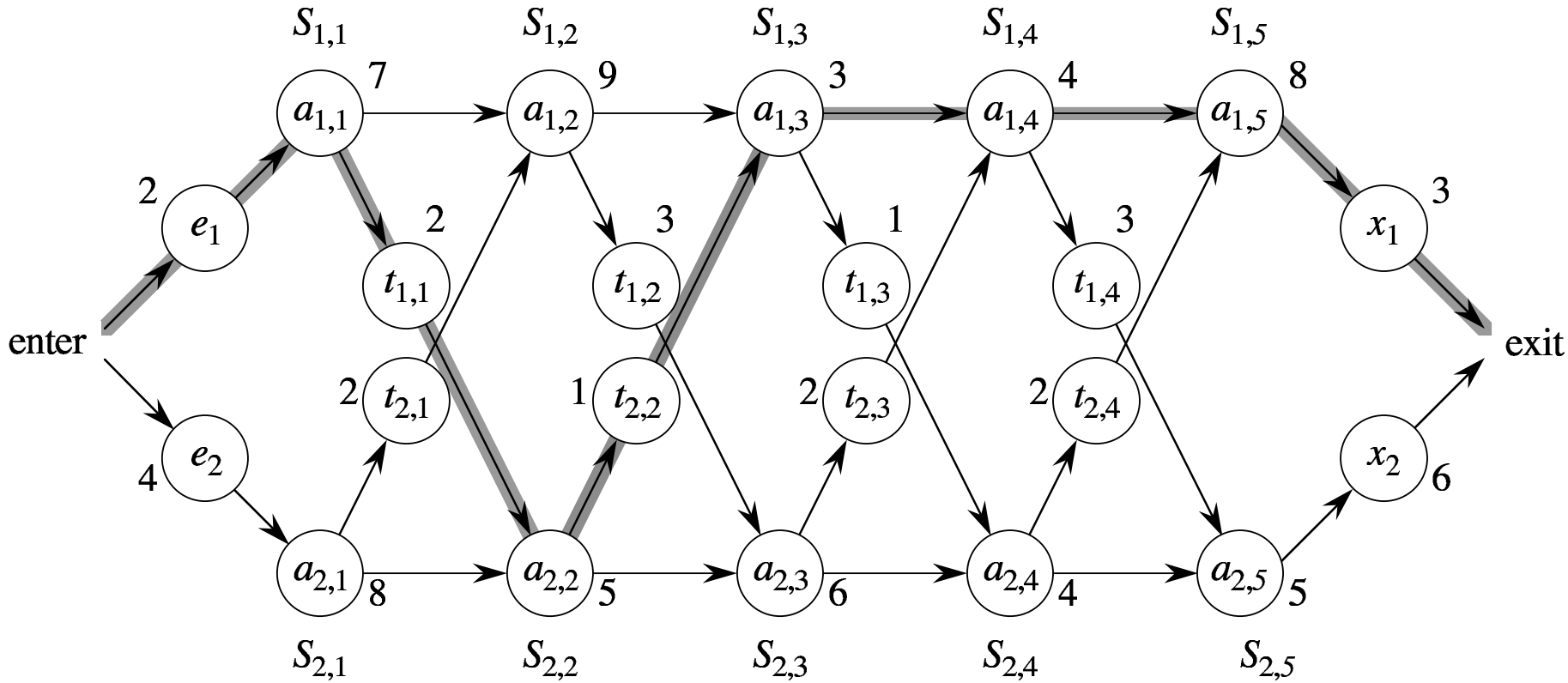
- Т.е., значение $f_1[1]$ будет вычислено 2^{n-1} раз!

Вычисление «снизу вверх»

- $f_i[j]$ зависит только от $f_1[j-1]$ и $f_2[j-1]$ для $j \geq 2$.
- Нужно вычислять в порядке возрастания j.

Алгоритм вычисления минимального времени

1 $f_1[1] \leftarrow e_1 + a_{1,1}$, $f_2[1] \leftarrow e_2 + a_{2,1}$
2 **for** $j \leftarrow 2$ **to** n
3 **if** $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
4 $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$, $l_1[j] \leftarrow 1$
5 **else** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$, $l_1[j] \leftarrow 2$
6 **if** $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$
7 $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$, $l_2[j] \leftarrow 2$
8 **else** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$, $l_2[j] \leftarrow 1$
9 **if** $f_1[n] + x_1 \leq f_2[n] + x_2$
10 $f^* \leftarrow f_1[n] + x_1$, $l^* \leftarrow 1$
11 **else** $f^* \leftarrow f_2[n] + x_2$, $l^* \leftarrow 2$



j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$$f^* = 35$$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$$l^* = 1$$

Четвёртый этап: построение самого быстрого пути

```
1  i ← l*
2  print «Конвейер i, рабочее место n»
3  for j ← n downto 2
4        i ← l_i[j]
5        print «Конвейер i, рабочее место j − 1»
```

Раздел

Задача

Имеется последовательность (цепочка), состоящая из n матриц, нужно вычислить их произведение:

$$A_1 A_2 \cdots A_n.$$

Для $\langle A_1, A_2, A_3, A_4 \rangle$ есть пять вариантов вычисления произведения:

$$(A_1(A_2(A_3 A_4))),$$
$$(A_1((A_2 A_3)A_4)),$$
$$((A_1 A_2)(A_3 A_4)),$$
$$((A_1(A_2 A_3))A_4),$$
$$(((A_1 A_2)A_3)A_4).$$

Перемножение матриц

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

- $A_{p\times q} \times B_{q\times r} = C_{p\times r}$.
- Время вычислений определяется количеством произведений скаляров, pqr.

Стоимость перемножения матриц

- Три матрицы $\langle A_1, A_2, A_3 \rangle$ размерностями 10×100, 100×5 и 5×50.
- Перемножение $((A_1 A_2) A_3)$ потребует сначала $10 \cdot 100 \cdot 500 = 2500$ скалярных произведений, а затем ещё $10 \cdot 5 \cdot 50 = 2500$, т.е. всего 7500 произведений.
- Для вычисления $(A_1 (A_2 A_3))$ потребуется $100 \cdot 5 \cdot 50 = 25000$ скалярных умножений и ещё $10 \cdot 100 \cdot 50 = 50000$ скалярных умножений, т.е. всего 75000.

Постановка задачи

Для заданной последовательности матриц $\langle A_1, A_2, \ldots, A_n \rangle$, в которой матрица A_i 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 сведётся к минимуму.

Перебор?

$P(n)$ — количество альтернативных способов расстановки скобок в последовательности матриц:

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

$P(n) = \Omega(4^n/n^{3/2})$, т.е. количество вариантов расстановки скобок экспоненциально увеличивается с ростом n и метод прямого перебора не подходит.

Первый этап: структура решения

- $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$, где $i \leq j$.
- Если $i < j$, то $i \leq k < j$ соответствует расстановка скобок:

$$A_{i\ldots j} = (A_{i\ldots k}) \cdot (A_{k+1\ldots j})$$

- Стоимость расстановки скобок будет равна сумме *оптимальной* стоимости перемножения $A_{i\ldots k}$, *оптимальной* стоимости $A_{k+1\ldots j}$ и стоимости вычисления их произведения.
- Задача разбивается на две подзадачи и строится из оптимальных решений подзадач.

Второй этап: рекурсивное решение

$m[i, j]$ — минимальное количество скалярных умножений, необходимых для вычисления матрицы $A_{i\ldots j}$. Тогда:

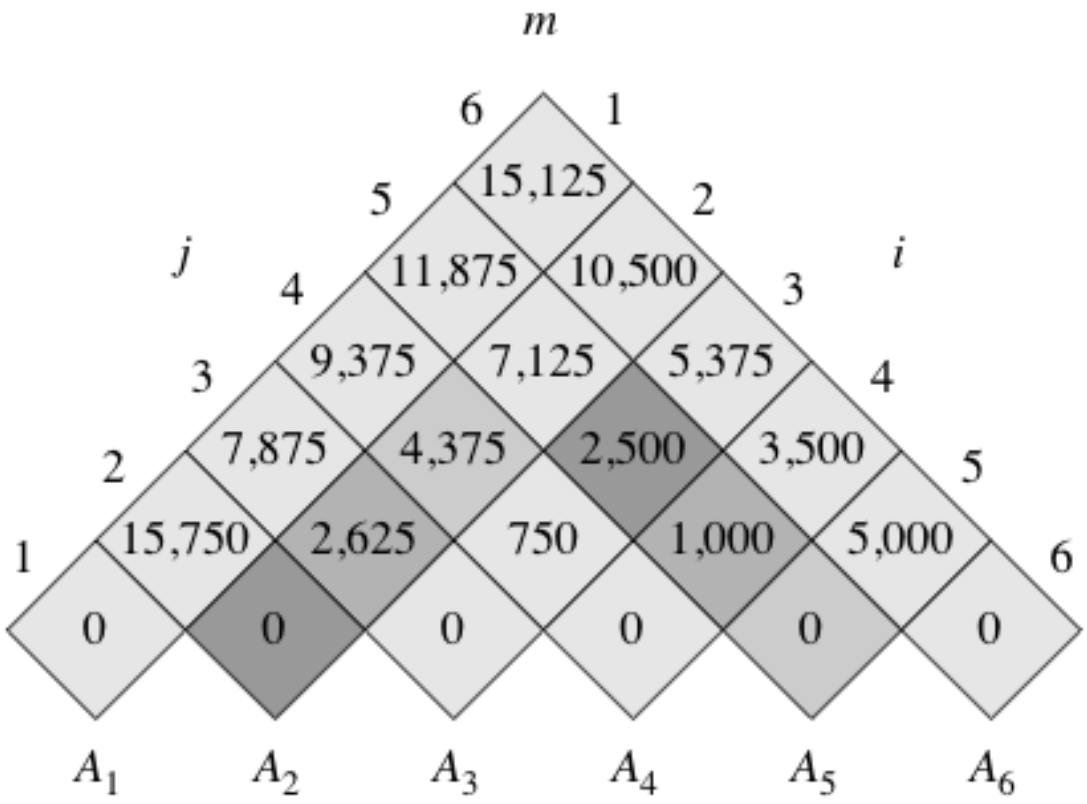
$$m[i,j] = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{при } i < j. \end{cases}$$

Третий этап: вычисление оптимальной стоимости

MATRIX-CHAIN-ORDER(p)

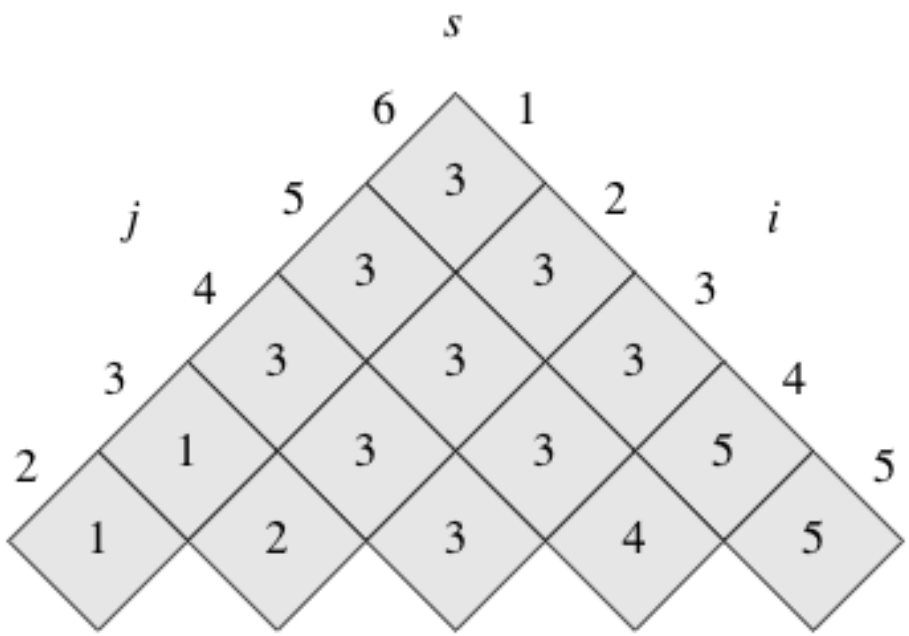
```
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3        m[i, i] ← 0
 4  for l ← 2 to n
 5        for i ← 1 to n − l + 1
 6              j ← i + l − 1, m[i, j] ← ∞
 7              for k ← i to j − 1
 8                    q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
 9                    if q < m[i, j]
10                          m[i, j] ← q
11                    else m[i, j] ← q, s[i, j] ← k
12  return ⟨m, s⟩
```

Пример вычислений



$$
\begin{array}{ll}
A_1 & 30 \times 35 \\
A_2 & 35 \times 15 \\
A_3 & 15 \times 5 \\
A_4 & 5 \times 10 \\
A_5 & 10 \times 20 \\
A_6 & 20 \times 25
\end{array}
$$

Массив s



Четвёртый этап: получение решения

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2        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 «)»
```

Раздел

Применимость

Когда применим метод динамического программирования?

- Оптимальная подструктура.
- Перекрытие вспомогательных подзадач.

Раздел

Оптимальная подструктура

- Показывает, что решение задачи состоит из некоторого выбора, оставляющего одну или больше подзадач для решения.
- Представим, что мы знаем последний выбор, который привёл к оптимальному решению.
- Зная его, определим, какие подзадачи были возможны и выявим характеристики полученного пространства подзадач.
- Покажем, что решения подзадач для получения общего оптимального решения должны быть так же оптимальными. (обычно, от противного)

Пространство подзадач

Нужно убедиться, что были рассмотрены все варианты выбора подзадач. Однако:

- Нужно держать пространство подзадач как можно более простым.
- Расширять его при необходимости.

Время работы алгоритма

Говоря просто, время работы зависит от общего количества подзадач умноженного на количество вариантов выбора. Например:

- Для конвеера: $\Theta(n)$ подзадач, 2 варианта выбора на каждой задаче, т.е. $\Theta(n)$ времени.
- Расстановка скобок: $\Theta(n^2)$ подзадач, $O(n)$ вариантов выбора внутри каждой подзадачи, $O(n^3)$ времени.

Решение снизу вверх

- Сначала найти оптимальные решения для подзадач.
- Затем выбрать те, которые нужно использовать в оптимальном решении для всей задачи.

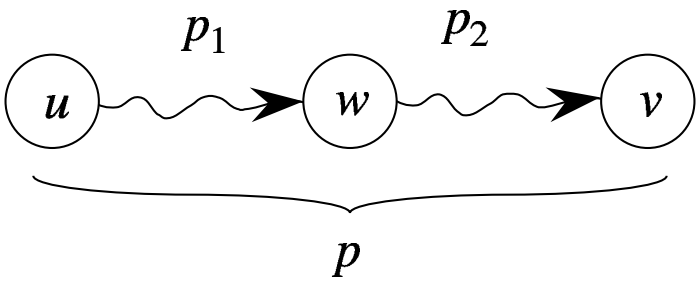
С жадными алгоритмами это не так.

Наличие оптимальной структуры

Дан ориентированный граф $G = (V, E)$ и вершины $u, v \in V$.
Две похожие задачи:

- Задача о кратчайшем невзвешенном пути. Нужно найти путь от вершины u к вершине v, состоящий из минимального количества рёбер.
- Задача о самом длинном невзвешенном пути. Определить простой (без циклов) путь от вершины u к вершине v, состоящий из максимального количества рёбер, $u \rightsquigarrow v$.

Оптимальная подструктура для кратчайшего пути

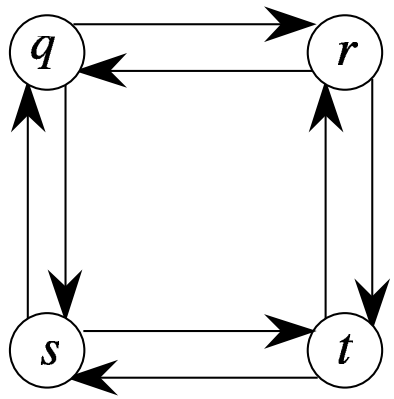


- Допустим p — кратчайший путь $u \rightsquigarrow v$.
- w — любая вершина на p.
- p_1 — подпуть p, $u \rightsquigarrow w$.
- Тогда p_1 — кратчайший путь $u \rightsquigarrow w$.
- Допустим, существует более короткий путь, p_1', $u \rightsquigarrow w$.

  Заменим в p путь p_1 на p_1', получим путь $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ короче, чем p.

Оптимальная подструктура для самого длинного пути

Здача выглядит похоже, вероятно так же можно найти оптимальную подструктуру. Однако, это не так.



- $q \to r \to t$ — самый длинный простой путь $q \rightsquigarrow t$. Но его подпути не являются самыми длинными путями!
- Подпуть для $q \rightsquigarrow r$: $q \to r$. Самый длинный путь $q \to s \to t \to r$.
- Подпуть для $r \rightsquigarrow t$: $r \to t$. Самый длинный путь $r \to q \to s \to t$.
- Комбинация самых длинных путей не является простым путём: $q \to s \to t \to r \to q \to s \to t$

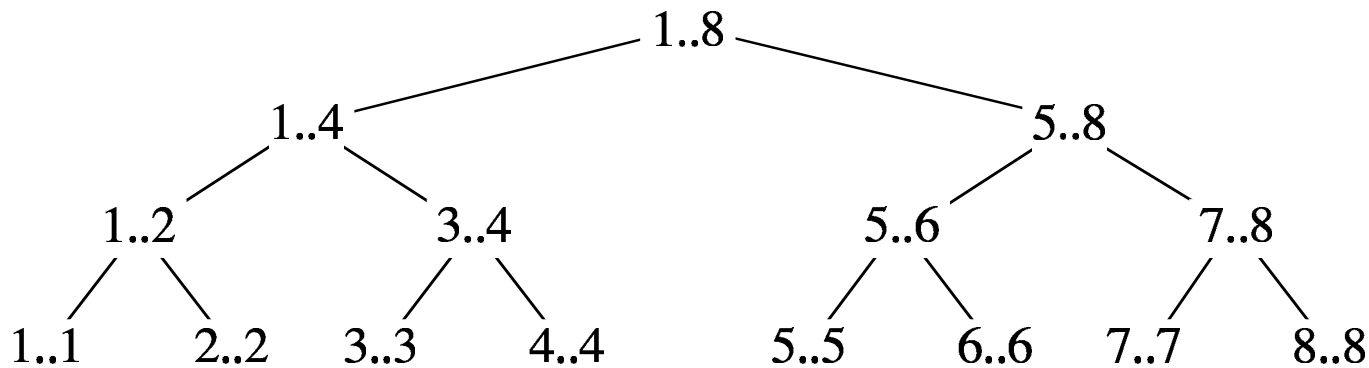
В чём отличие?

- Задача о коротком пути имеет независимые подзадачи, т.е. решение одной позадачи не сказывается на решении другой подзадачи.
- В задаче о длинном пути подзадачи зависимы.

Раздел

Неперекрывающиеся подзадачи

Хороший алгоритм вида «разделяй-и-властвуй» должен на каждом шаге решать совершенно новую подзадачу. Например, сортировка слиянием:



Перекрывающиеся подзадачи

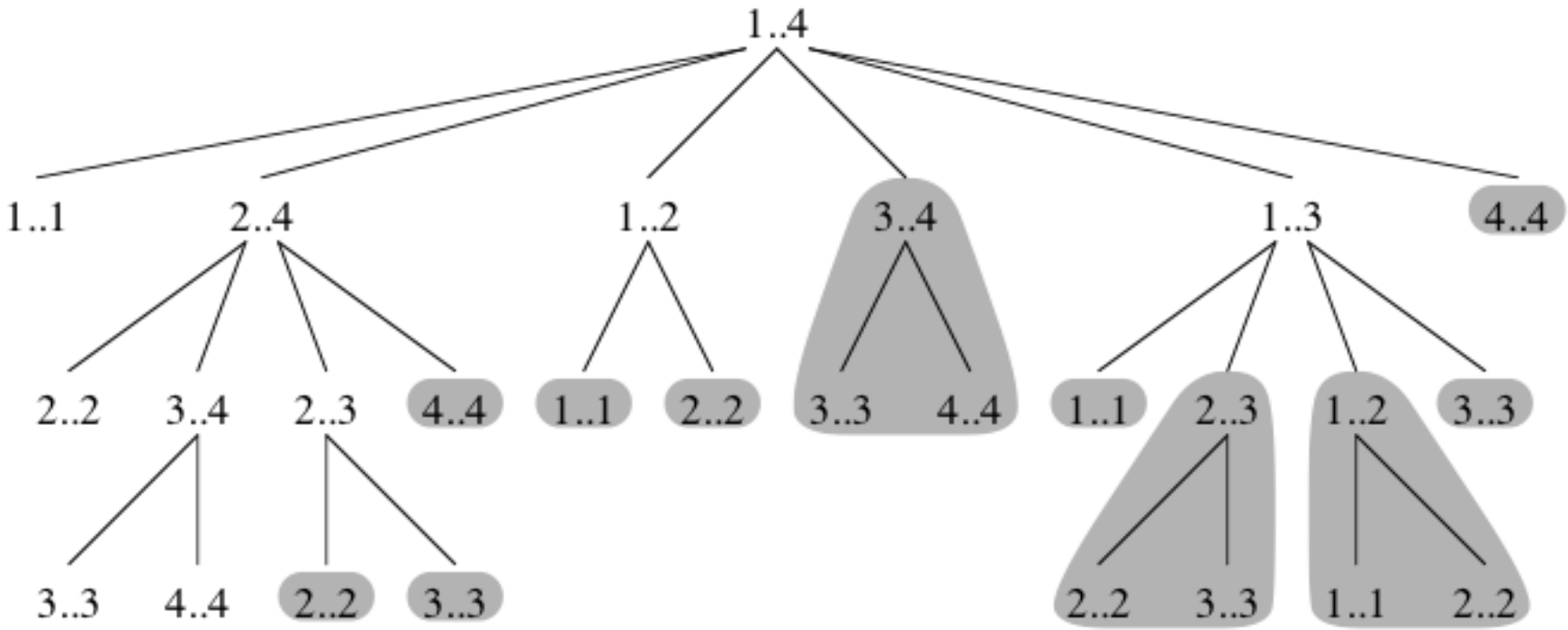
- Эффективность алгоритмов динамического программирования зависит от того, насколько часто одна и та же подзадача рассматривается в рекурсивном решении.
- Перекрытие подазадач и независимость решения подзадач — разные понятия!
- Если есть перекрытие, то можно либо выстроить эффективную последовательность решения подзадач, или модифицировать рекурсивный алгоритм, чтобы он запоминал промежуточные результаты.

Перекрытие в рекурсивном алгоритме

RECURSIVE-MATRIX-CHAIN(p, i, j)

```
1  if i = j
2        return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        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              m[i, j] ← q
8  return m[i, j]
```

Дерево рекурсии для (1,4)



Запоминание

MEMOIZED-MATRIX-CHAIN(p)

```
1  n ← length[p] − 1
2  for i ← 1 to n
3        for j ← i to n
4              m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)
```

LOOKUP-CHAIN(p, i, j)

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

Раздел

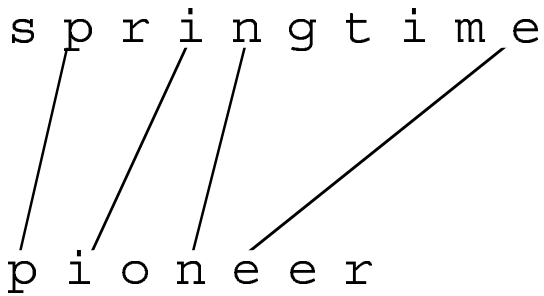
Раздел

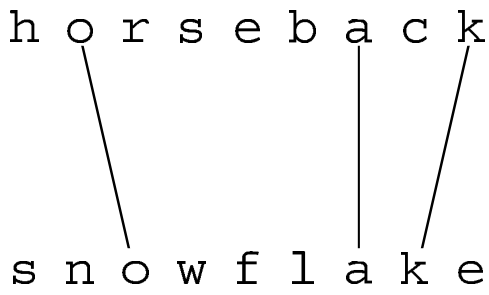
Постановка задачи

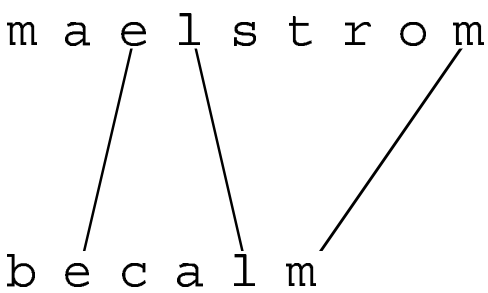
Даны две последовательности, $X = \langle x_1, \ldots, x_m \rangle$ и $Y = \langle y_1, \ldots, y_n \rangle$. Нужно найти общую для обеих последовательностей подпоследовательность, чья длина будет наибольшей.

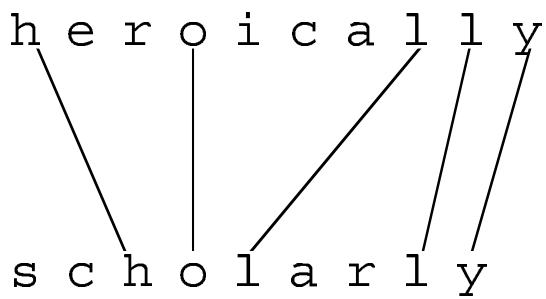
Задача тесно связана с вычислением расстояния Левенштейна.

Примеры


springtime
pioneer


horseback
snowflake


maelstrom
becalm


heroically
scholarly

Перебор

Время на перебор $\Theta(n2^m)$:

- Нужно проверить 2^m подпоследовательностей X.
- Каждая проверка занимает $\Theta(n)$ времени.

Первый этап: оптимальная подструктура

Будем считать, что X_i — префикс $\langle x_1, \ldots, x_i \rangle$ и Y_i — префикс $\langle y_1, \ldots y_i \rangle$

Теорема

Пусть $Z = \langle z_1, \ldots z_k \rangle$ *какая-то* $LCS(X, Y)$. *Тогда:*

1. $x_m = y_n$, *тогда* $z_k = x_m = y_n$ *и* $Z_{k-1} = LCS(X_{m-1}, Y_{m-1})$.
2. $x_m \neq y_n$ *и* $z_k \neq x_m$, *тогда* $Z = LCS(X_{m-1}, Y)$.
3. $x_m \neq y_n$ *и* $z_k \neq y_n$, *тогда* $Z = LCS(X, Y_{n-1})$.

Доказательство.

- $x_m = y_n$, допустим $z_k \neq x_m$. Тогда возьмём $Z' = \langle z_1, \ldots, z_k, x_m \rangle$, которая будет являться подпоследовательностью X и Y длиной $k+1$, что невозможно.
- $x_m \neq y_n$ и $z_k \neq x_m$. Тогда Z — общая подпоследовательность для X_{m-1} и Y. И она наибольшая, потому что в противном случае должна была бы существовать $W = LCS(X_{m-1}, Y)$ длиной большей k, которая была бы $LCS(X, Y)$ длиной большей k, что невозможно.
- $x_m \neq y_n$ и $z_k \neq y_n$ — аналогично.

Второй этап: рекурсивное определние

$c[i,j] = |LCS(X_i, Y_j)|$, нам нужно вычислить $c[m,n]$. Тогда:

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

Третий этап: алгоритм

LCS-LENGTH(X, Y, m, n)

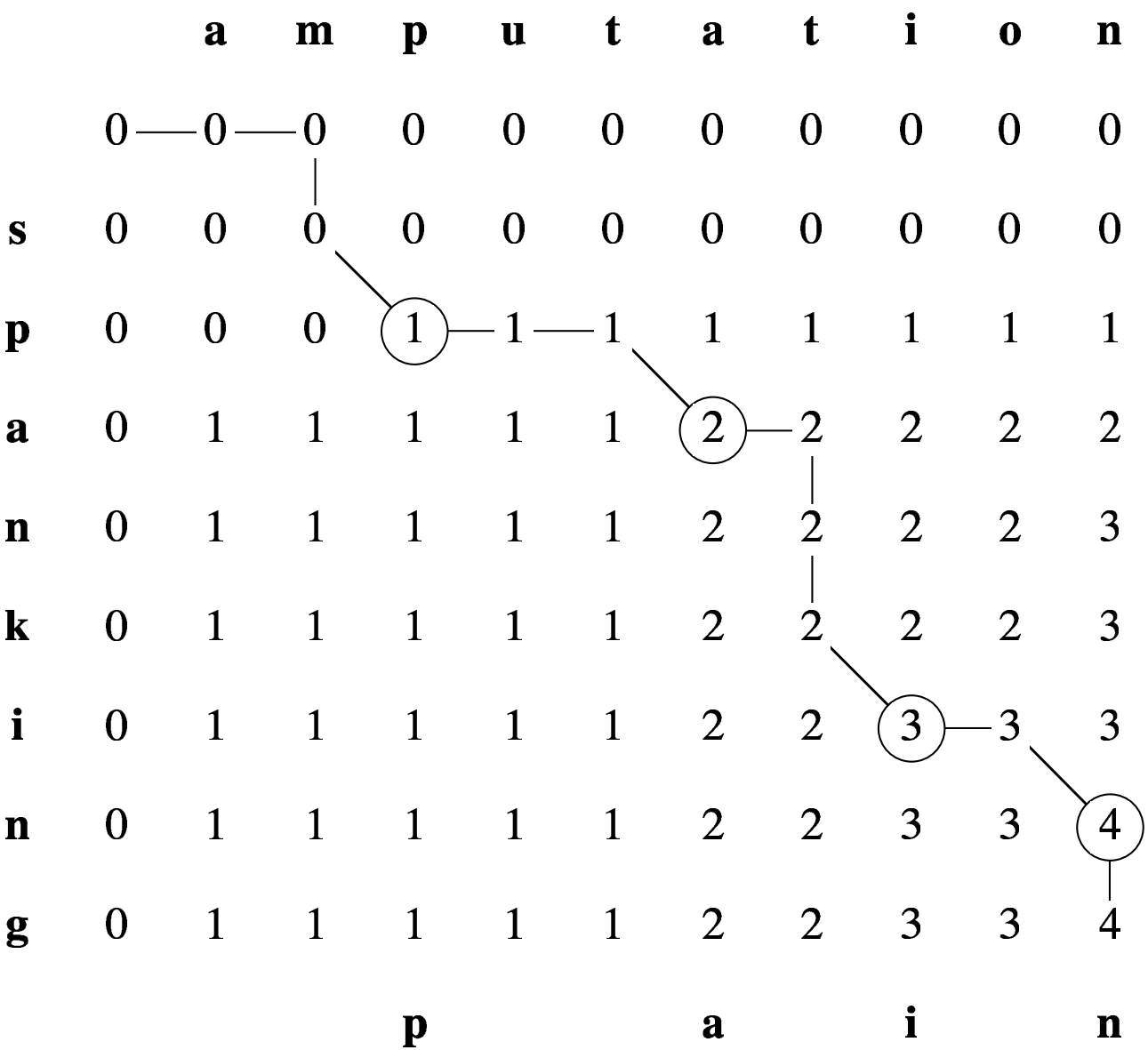
for $i \leftarrow 1$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  do for $j \leftarrow 1$ **to** n
    do if $x_i = y_j$
      then $c[i, j] \leftarrow c[i-1, j-1] + 1$
        $b[i, j] \leftarrow$ "$\nwarrow$"
      else if $c[i-1, j] \geq c[i, j-1]$
        then $c[i, j] \leftarrow c[i-1, j]$
          $b[i, j] \leftarrow$ "$\uparrow$"
        else $c[i, j] \leftarrow c[i, j-1]$
          $b[i, j] \leftarrow$ "$\leftarrow$"

return c and b

Четвёртый этап: распечатка решения

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

Пример


a m p u t a t i o n
0 0 0 0 0 0 0 0 0 0 0
s 0 0 0 0 0 0 0 0 0 0 0
p 0 0 0 1 1 1 1 1 1 1 1
a 0 1 1 1 1 1 2 2 2 2 2
n 0 1 1 1 1 1 2 2 2 2 3
k 0 1 1 1 1 1 2 2 2 2 3
i 0 1 1 1 1 1 2 2 3 3 3
n 0 1 1 1 1 1 2 2 3 3 4
g 0 1 1 1 1 1 2 2 3 3 4
p a i n

Раздел

Постановка задачи

- n отсортированных различных ключей: $K = \langle k_1, k_2, \ldots, k_n \rangle$.
- p_i — вероятность поиска ключа k_i.
- Вероятность поиска отсутствующих значений:
  - $n + 1$ фиктивных ключей $\langle d_0, d_1, \ldots, d_n \rangle$, $d_0 < k_1$, $k_i < d_i < k_{i+1}$ $(1 \leq i < n)$, $k_n < d_n$.
  - q_i — вероятность поиска ключей, соответствующих d_i.
- Поиск может быть успешным или неуспешным, т.е.:

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

Стоимость поиска

$$\begin{aligned}\mathrm{E}[\mathrm{COST}(T)] &= \sum_{i=1}^{n}(depth_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(depth_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i)\cdot p_i + \sum_{i=0}^{n} depth_T(d_i)\cdot q_i\end{aligned}$$

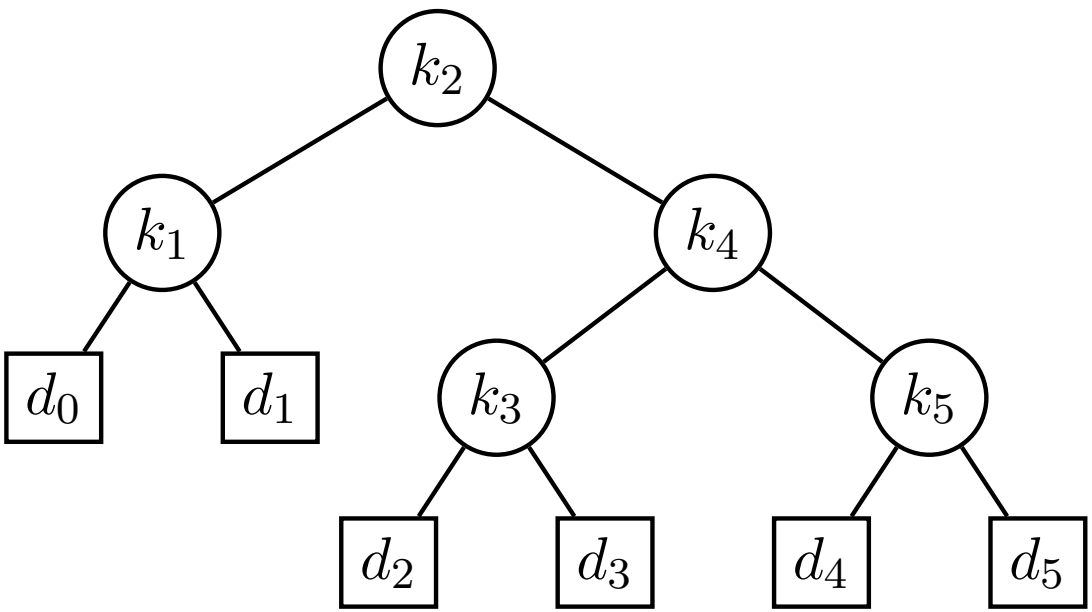
Стоимость поиска

$$\mathrm{E}[\mathrm{COST}(T)] = \sum_{i=1}^{n}(depth_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(depth_T(d_i)+1)\cdot q_i$$
$$= 1 + \sum_{i=1}^{n} depth_T(k_i)\cdot p_i + \sum_{i=0}^{n} depth_T(d_i)\cdot q_i$$

Нужно для данного набора вероятностей построить бинарное дерево поиска, математическое ожиданеие стоимости поиска для которого будет минимальным. Такое дерево — оптимальное бинарное дерево поиска.

Сбалансированное дерево поиска

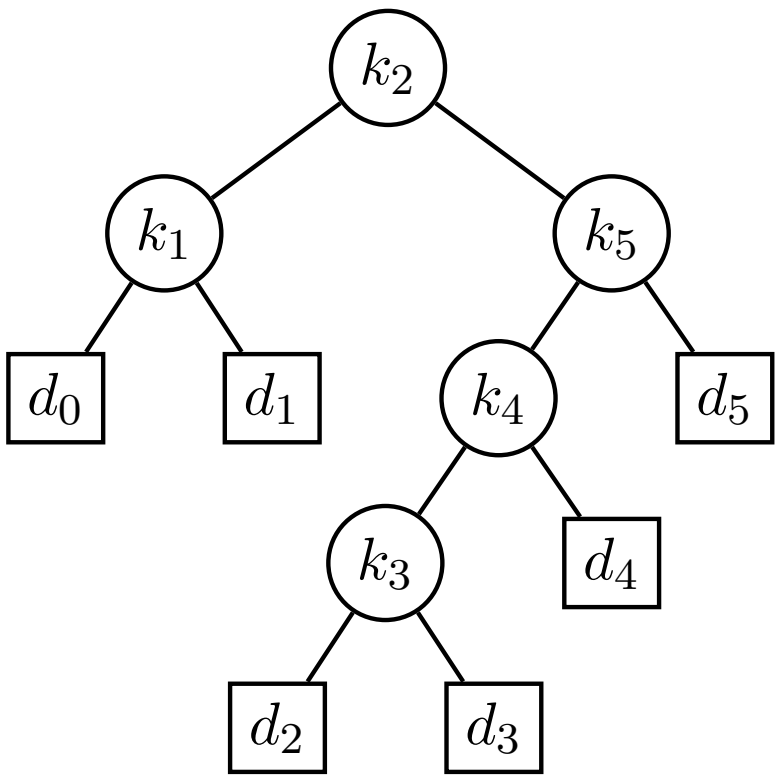
i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10



Стоимость дерева 2.80.

Оптимальное дерево поиска

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



1. Стоимость дерева 2.75.
2. Оптимальное дерево может не быть идеально сбалансированным.
3. Ключ с максимальной вероятностью не обязан быть в корне.

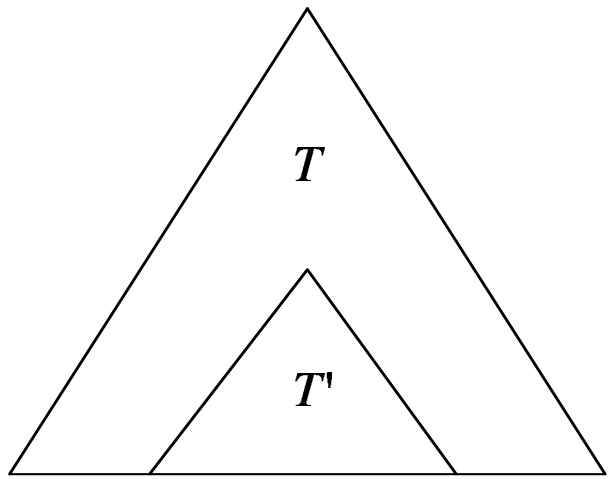
Перебор?

1. Построить все деревья с n узлами.
2. Рассчитать для каждого стоимость.
3. Выбрать оптимальное.

$\Omega(4^n/n^{3/2})$ различных бинарных деревьев с n узлами, поэтому перебор не подходит.

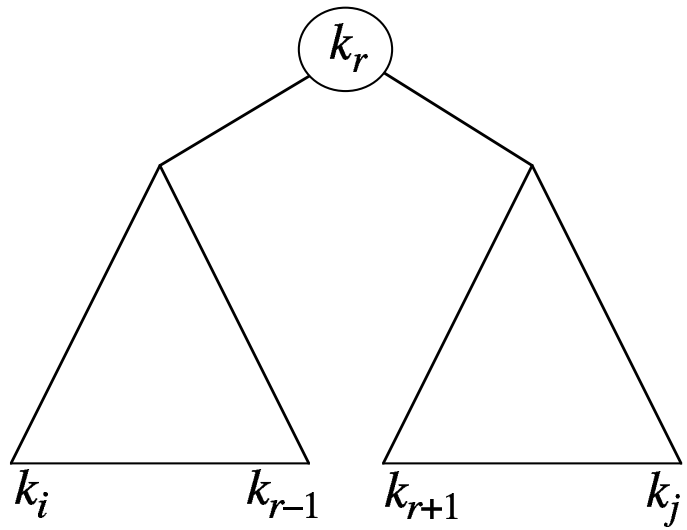
Оптимальная подструктура

- Поддерево бинарного дерева поиска содержит ключи из непрерывного диапазона $k_i, \ldots, k_j$ для $1 \leq i \leq j \leq n$.
- Если T — оптимальное дерево поиска и T' — его поддерево с ключами $k_i, \ldots, k_j$, то T' должно быть оптимальным деревом с ключами $k_i, \ldots, k_j$ и фиктивными ключами $d_{i-1}, \ldots, d_j$.



Выбор корня

- Даны ключи $k_i, \ldots, k_j$ и $d_{i-1}, \ldots, d_j$.
- Один из них, k_r, должен стать корнем.
- Тогда левое поддерево состоит из $k_i, \ldots, k_{r-1}$ и $d_{i-1}, \ldots, d_{r-1}$.
- Правое поддерво — из $k_{r+1}, \ldots, k_j$ и $d_r, \ldots, d_j$.
- Дерево из ключей $k_i, \ldots, k_{i-1}$ состоит из одного фиктивного ключа d_{i-1}.



Рекурсивное решение

- Задача поиска оптимального дерева, содержащего ключи $k_i, \ldots, k_j$, $i \geq 1$, $j \leq n$ и $j \geq i - 1$ (если $j = i - 1$, то дерево состоит только из d_{i-1}.
- $e[i, j]$ — мат. ожидание стоимости поиска в дереве с ключами $k_i, \ldots, k_j$.
- В конечном итоге, нужно вычислить $e[1, n]$
- В случае $j = i - 1$ есть только d_{i-1} и $e[i, i - 1] = q_{i-1}$.
- Если $j \geq i$, то среди ключей $k_i, \ldots, k_j$ нужно выбрать корень k_r и составить левое и правое оптимальные деревья поиска из $k_i, \ldots, k_{r-1}$ и $k_{r+1}, \ldots, k_j$.

Когда оптимальное дерево становится поддеревом, его стоимость увеличивается на

$$w(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k.$$

Тогда:

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

Из

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

следует что:

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

Точная запись решения

$$e[i,j] = \begin{cases} q_{i-1} & j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & i \ge j \end{cases}$$

Алгоритм

```
1   for i ← 1 to n + 1
2         e[i, i − 1] ← q_{i−1}, w[, i − 1] ← q_{i−1}
3   for k ← 1 to n
4         for j ← 1 to n − k + 1
5               j ← i + k − 1, e[i, j] ← ∞
6               w[i, j] ← w[i, j − 1] + p_j + q_j
7               for r ← i to j
8                     t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
9                     if t < e[i, j]
10                          e[i, j] ← t, root[i, j] ← r
11  return ⟨e, root⟩
```