Задача нахождения **наибольшей общей подпоследовательности** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *longest common subsequence*, LCS) — это задача поиска [последовательности](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C), которая является подпоследовательностью нескольких последовательностей (обычно двух). Часто задача определяется как поиск *всех* наибольших подпоследовательностей. Это классическая задача [информатики](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), которая имеет приложения, в частности, в задаче сравнения текстовых файлов (утилита [diff](http://ru.wikipedia.org/wiki/Diff)), а также в [биоинформатике](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%BE%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0).

Подпоследовательность можно получить из некоторой конечной последовательности, если удалить из последней некоторое множество её элементов (возможно пустое). Например, BCDB является подпоследовательностью последовательности ABCDBAB. Будем говорить, что последовательность Z является общей подпоследовательностью последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Требуется для двух последовательностей X и Y найти общую подпоследовательность наибольшей длины. Заметим, что НОП может быть несколько.

|  |
| --- |
| **Содержание*** [1 Решение задачи](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#.D0.A0.D0.B5.D1.88.D0.B5.D0.BD.D0.B8.D0.B5_.D0.B7.D0.B0.D0.B4.D0.B0.D1.87.D0.B8)
	+ [1.1 Полный перебор](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#.D0.9F.D0.BE.D0.BB.D0.BD.D1.8B.D0.B9_.D0.BF.D0.B5.D1.80.D0.B5.D0.B1.D0.BE.D1.80)
	+ [1.2 Метод динамического программирования](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#.D0.9C.D0.B5.D1.82.D0.BE.D0.B4_.D0.B4.D0.B8.D0.BD.D0.B0.D0.BC.D0.B8.D1.87.D0.B5.D1.81.D0.BA.D0.BE.D0.B3.D0.BE_.D0.BF.D1.80.D0.BE.D0.B3.D1.80.D0.B0.D0.BC.D0.BC.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.B8.D1.8F)
* [2 Реализация в языках программирования](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#.D0.A0.D0.B5.D0.B0.D0.BB.D0.B8.D0.B7.D0.B0.D1.86.D0.B8.D1.8F_.D0.B2_.D1.8F.D0.B7.D1.8B.D0.BA.D0.B0.D1.85_.D0.BF.D1.80.D0.BE.D0.B3.D1.80.D0.B0.D0.BC.D0.BC.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.B8.D1.8F)
	+ [2.1 C++](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#C.2B.2B)
	+ [2.2 Ruby](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#Ruby)
* [3 См. также](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C#.D0.A1.D0.BC._.D1.82.D0.B0.D0.BA.D0.B6.D0.B5)
 |

**Решение задачи**

Сравним два метода решения: [полный перебор](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%BD%D1%8B%D0%B9_%D0%BF%D0%B5%D1%80%D0%B5%D0%B1%D0%BE%D1%80) и динамическое программирование.

**Полный перебор**

Существуют разные подходы при решении данной задачи при полном переборе — можно перебирать варианты подпоследовательности, варианты вычеркивания из данных последовательностей и т. д. Однако в любом случае, время работы программы будет [экспонентой](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%B0) от длины строки.

**Метод динамического программирования**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **A** | **B** | **C** | **B** |
|  | 0 | 0 | 0 | 0 | 0 |
| **D** |   0 | **←** 0 | **←** 0 | **←** 0 | **←** 0 |
| **C** |   0 | **←** 0 | **←** 0 | **↖** 1 | **←** 1 |
| **B** |   0 | **←** 0 | **↖** 1 | **←** 1 | **↖** 2 |
| **A** |   0 | **↖** 1 | **←** 1 | **←** 1 | **↑** 2 |

Вначале найдём длину наибольшей подпоследовательности. Допустим, мы ищем решение для случая (n1, n2), где n1, n2 — длины первой и второй строк. Пусть уже существуют решения для всех подзадач (m1, m2), меньших заданной. Тогда задача (n1, n2) сводится к меньшим подзадачам следующим образом:

![ f(n_1, n_2) = \left \{  \begin{array}{ll}  0, & n_1 = 0 \or n_2 = 0 \\  f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\  max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2]  \end{array}  \right.]()

Теперь вернемся к задаче построения подпоследовательности. Для этого в существующий алгоритм добавим запоминание для каждой задачи той подзадачи, через которую она решается. Следующим действием, начиная с последнего элемента, поднимаемся к началу по направлениям, заданным первым алгоритмом, и записываем символы в каждой позиции. Это и будет ответом в данной задаче.

Очевидно, что время работы алгоритма будет .

**Реализация в языках программирования**

 string getLongestCommonSubsequence(const string& a, const string& b)

 {

 vector<vector<int> > max\_len;

 max\_len.resize(a.size() + 1);

 for(int i = 0; i <= static\_cast<int>(a.size()); i++)

 max\_len[i].resize(b.size() + 1);

 for(int i = static\_cast<int>(a.size()) - 1; i >= 0; i--)

 {

 for(int j = static\_cast<int>(b.size()) - 1; j >= 0; j--)

 {

 if(a[i] == b[j])

 {

 max\_len[i][j] = 1 + max\_len[i+1][j+1];

 }

 else

 {

 max\_len[i][j] = max(max\_len[i+1][j], max\_len[i][j+1]);

 }

 }

 }

 string res;

 for(int i = 0, j = 0; max\_len[i][j] != 0 && i < static\_cast<int>(a.size()) && j < static\_cast<int>(b.size()); )

 {

 if(a[i] == b[j])

 {

 res.push\_back(a[i]);

 i++;

 j++;

 }

 else

 {

 if(max\_len[i][j] == max\_len[i+1][j])

 i++;

 else

 j++;

 }

 }

 return res;

 }

[**Ruby**](http://ru.wikipedia.org/wiki/Ruby)

#>> a = "aaaaabbbb34354354345"

#>> b = "abbb34aaabbbb"

#>> longest\_common\_subsequence(a, b)

#=> "aaaabbbb"

 def longest\_common\_subsequence(a, b)

 max\_len = Array.new(a.size + 1, 0)

 max\_len.map! {Array.new(b.size + 1, 0)}

 (a.size - 1).downto(0) do |i|

 (b.size - 1).downto(0) do |j|

 if a[i] == b[j]

 max\_len[i][j] = 1 + max\_len[i+1][j+1]

 else

 max\_len[i][j] = [max\_len[i+1][j], max\_len[i][j+1]].max

 end

 end

 end

 res = ""

 i = 0

 j = 0

 while max\_len[i][j] != 0 && i < a.size && j < b.size

 if a[i] == b[j]

 res << a[i]

 i += 1

 j += 1

 else

 if max\_len[i][j] == max\_len[i+1][j]

 i += 1

 else

 j += 1

 end

 end

 end

 res

 end

№2

Отметим, подпоследовательность может и не являться подстрокой (то есть, её элементы не обязательно идут подряд в исходной последовательности). Формально, для строки *x* длины *n* необходимо найти максимальное число *l* и соответствующую ему возрастающую [последовательность](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C) индексов *i1*..*il*, таких что ![x[i_{k}] < x[i_{k+1}]\,\forall {k}\,\mathcal{2}{\,1\,..\,l-1} ](). Наибольшая увеличивающая подпоследовательность имеет применения в физике, математике, теории представления групп, теории случайных матриц. В общем случае известно решение этой задачи за время *n log n*[[1]](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_%D0%BD%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B5%D0%B9_%D1%83%D0%B2%D0%B5%D0%BB%D0%B8%D1%87%D0%B8%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9%D1%81%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8%22%20%5Cl%20%22cite_note-0) в худшем случае.

**Родственные алгоритмы**

* Задача наибольшей увеличивающейся подпоследовательности схожа с задачей [поиска наибольшей общей подпоследовательности](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B0%D1%8F_%D0%BE%D0%B1%D1%89%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C), имеющей квадратичное динамическое решение.
* В частном случае, если строка является перестановкой 1..n, задача имеет решение за *n log log n*[[2]](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_%D0%BD%D0%B0%D0%B8%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%B5%D0%B9_%D1%83%D0%B2%D0%B5%D0%BB%D0%B8%D1%87%D0%B8%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9%D1%81%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8%22%20%5Cl%20%22cite_note-doi%3A10.1145.2F359581.359603-1) с использованием [деревьев ван Эмд Босса](http://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%B2%D0%B0%D0%BD_%D0%AD%D0%BC%D0%B4_%D0%91%D0%BE%D1%81%D1%81%D0%B0&action=edit&redlink=1).
* При использовании дерева, построенного для элементов алфавита, возможно решение задачи за O( n log A ), где A - мощность алфавита, определяемая заранее. При реализации [сбалансированными деревьями](http://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0), необязательно задавать A наперёд. По очевидным причинам A ограничивается длиной строки.
* Возможно также свести задачу к поиску длиннейшего пути в ориентированном ациклическом графе, задавая рёбра между возрастающими элементами. Хотя подсчёт длиннейшего пути будет занимать линейное время от числа рёбер, в худшем случае оно может быть квадратично от длины строки.

**Пример эффективного алгоритма**

Для строки *x* будем хранить массивы *M* и *P* длины *n*. *M[i]* содержит наименьший по величине из последних элементов возрастающих подпоследовательностей *xnj* длины *i*, ![(\mathcal{8}j\,\mathcal{2}\,1\,..\,i:x[n_{j}] \le\,M[i])](), найденных на данном шаге. *P[i]* хранит индекс предшествующего символа для наидлиннейшей возрастающей подпоследовательности, оканчивающейся в i-й позиции. На каждом шаге будем хранить текущий максимум длины подпоследовательности и соответствующий индекс конечного символа, не забывая поддерживать свойства массивов. Шаг представляет собой переход к следующему элементу строки, для каждого перехода потребуется не более логарифма времени (бинарный поиск по массиву *M*).

L = index = M[0] = 0

for i = 1 to n

 бинарный поиск наибольшего индекса j ≤ L, удовлетворяющего X[M[j]] < X[i]

 P[i] = M[j]

 if j == L or X[i] < X[M[j+1]] // нашли более оптимальную подпоследовательность

 M[j+1] = i

 L = max{L, j+1}