* [Двумерное динамическое программирование: таблицы](https://foxford.ru/lessons/32272/conspects/1)
* [Одномерное динамическое программирование: наилучший способ](https://foxford.ru/lessons/32272/conspects/2)
* [Одномерное динамическое программирование: количество способов](https://foxford.ru/lessons/32272/conspects/3)
* [Наибольшая общая подпоследовательность](https://foxford.ru/lessons/32272/conspects/4)
* [Вычисление расстояния Левенштейна](https://foxford.ru/lessons/32272/conspects/5)
* [Наибольшая возрастающая подпоследовательность](https://foxford.ru/lessons/32272/conspects/6)
* [Алгоритм "укладки рюкзака"](https://foxford.ru/lessons/32272/conspects/7)
* [Двумерное динамическое программирование: игры](https://foxford.ru/lessons/32272/conspects/8)
* [Динамическое программирование на подотрезках](https://foxford.ru/lessons/32272/conspects/9)

**Двумерное динамическое программирование: таблицы**

В этом разделе будут рассматриваться задачи, похожие на задачи из раздела Одномерное динамическое программирование, но вместо одномерных задач на движение по прямой будут рассматриваться перемещения в двумерном пространстве — например, перемещения на шахматной доске или на клетчатом листе бумаги

**Подсчет числа маршрутов**

Рассмотрим шахматную доску в левом верхнем углу которой находится король. Король может перемещаться только вправо, вниз или по диагонали вправо-вниз на одну клетку. Необходимо определить количество различных маршрутов короля, приводящих его в правый нижний угол.

Сопоставим каждой клетке ее координаты (i,j), где i будет обозначать номер строки на доске, j — номер столбца. Нумеровать строки будем сверху вниз, столбцы — слева направо, нумерация начинается с 0. Тогда начальное положение короля будет клетка (0,0).

Обозначим через F(i,j) количество способов прийти из клетки (0,0) в клетку (i,j). В клетку (i,j) можно прийти из трех клеток — слева из (i,j−1), сверху из (i−1,j) и по диагонали из (i−1,j−1). Поэтому число маршрутов ведущих в клетку равно числу маршрутов из всех ее предшественников, а именно:

F(i,j)=F(i,j−1)+F(i−1,j)+F(i−1,j−1)

Отдельно нужно задать значения для граничных клеток, то есть когда i=0 или j=0. В результате получится таблица заполненная следующим образом:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 3 | 5 | 7 | 9 |
| 1 | 5 | 13 | 25 | 41 |
| 1 | 7 | 25 | 63 | 129 |
| 1 | 9 | 41 | 129 | 321 |

Для заполнения этой таблицы и подсчета числа маршрутов можно использовать следующую программу, в которой сначала создается двумерный список, затем заполняются крайние клетки (первый столбец и первая строка), затем заполняются остальные элементы таблицы при помощи приведенного выше рекуррентного соотношения. В данном примере n - число строк, m - число столбцов на доске.

**F = [[0] \* m for i in range(n)]**

**for i in range(n):**

**F[i][0] = 1**

**for j in range(m):**

**F[0][j] = 1**

**for i in range(1, n):**

**for j in range(1, m):**

**F[i][j] = F[i][j - 1] + F[i - 1][j] + F[i - 1][j - 1]**

На этом примере можно составить общий план решения задачи методом динамического программирования. Этот план можно использовать для решения любых задач при помощи динамического программирования:

1. Записать то, что требуется найти в задаче, как целевую функцию от некоторого набора аргументов (числовых, строковых или еще каких-либо).
2. Свести решение задачи для произвольного набора параметров к решению аналогичных подзадач для других наборов параметров (как правило, с меньшими значениями параметров). Если задача несложная, то полезно бывает выписать явное рекуррентное соотношение, задающее значение функции для данного набора параметров.
3. Задать начальные значения функции, то есть те наборы аргументов, при которых задача тривиальна и можно явно указать значение функции.
4. Создать массив (или другую структуру данных) для хранения значений функции. Как правило, если функция зависит от одного целочисленного параметра, то используется одномерный массив, для функции от двух целочисленных параметров — двумерный массив и т. д.
5. Организовать заполнение массива с начальных значений, определяя очередной элемент массива при помощи выписанного на шаге 2 рекуррентного соотношения или алгоритма.

Для заполнения первой строки и первого столбца таблицы мы использовали «специальную» формулу, отличающуюся от общего случая. Но в некоторых задачах удобней бывает все значения вычислять по одной и той же формуле, а для граничных значений функции ввести специальные «фиктивные» элементы. В данной задаче тоже можно так поступить — введем специальную «каемочку» из одного фиктивного столбца слева и одной фиктивной строки сверху таблицы.

Для того, чтобы значения в остальной таблице вычислялись по общим формулам, во все клетки каемочки нужно записать число 0, кроме клетки (0,0), в которую будет записано значение 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 3 | 5 | 7 | 9 |
| 0 | 1 | 5 | 13 | 25 | 41 |
| 0 | 1 | 7 | 25 | 63 | 129 |
| 0 | 1 | 9 | 41 | 129 | 321 |

Теперь во всех остальных клетках таблицы значения могут быть вычислены по общей формуле: F(i,j)=F(i,j−1)+F(i−1,j)+F(i−1,j−1), а программа может выглядеть так:

**F = [[0] \* (m + 1) for i in range(n + 1)]**

**F[0][0] = 1**

**for i in range(1, n + 1):**

**for j in range(1, m + 1):**

**F[i][j] = F[i][j - 1] + F[i - 1][j] + F[i - 1][j — 1]**

**Маршрут наименьшей стоимости**

Теперь решим задачу о нахождении маршрута минимальной стоимости из левого верхнего угла в правый нижний, считая что для каждой клетке указана стоимость прохода через эту клетку. Сразу же будем считать, что таблица снабжена «каемочкой», поэтому начальная клетка будет иметь индексы (1,1), конечная клетка — (n,m), а строка и столбец с индексом 0 будут относится к фиктивной каемочке.

Если считать, что стоимость прохода через клетку (i,j) записана в отдельном списке Price[i,j], то обозначив через СС(i,j) стоимость кратчайшего пути из начальной клетки (1,1) в клетку (i,j) получим рекуррентное соотношение: C(i,j)=min(C(i,j−1),C(i−1,j),C(i−1,j−1))+Price[i,j]

При этом при вычислении граничных значений (в первом столбце и первой строке) необходимо учитывать только клетки из этого столбца и этой строки (но не из предыдущего столбца и предыдущей строки). Это удобно реализовать, если заполнить предыдущую строк и предыдущий столбец «каемочкой», записав для клеток каемочки значения функции C равными некоторому очень большому числу. Это число мы будем обозначать как INF, в качестве значения INF следует взять число, заведомо больше, чем максимальное число, которое может быть записано в таблице C. А в угол «каемочки» нужно записать число 0: C[0][0]=0.

**INF = 10 \*\* 20**

**C = [[0] \* (m + 1) for i in range(n + 1)]**

**C[0][0] = 0**

**for i in range(1, n + 1):**

**C[i][0] = INF**

**for j in range(1, m + 1):**

**C[0][j] = INF**

**for i in range(1, n + 1):**

**for j in range(1, m + 1):**

**C[i][j] = min(C[i][j - 1], C[i - 1][j], C[i - 1][j - 1]) + Price[i][j]**

**Восстановление ответа**

Теперь напишем восстановление ответа. Точно так же, как и в одномерном случае будем идти на конечной клетке в начальную, на каждом шаге выбирая ту из возможных предшествующих клеток, для которой значение функции C меньше.

**Answer = []**

**i = n**

**j = m**

**while (i, j) != (0, 0):**

**Answer.append((i, j))**

**prev\_C = min(C[i][j - 1], C[i - 1][j], C[i - 1][j - 1])**

**if C[i][j - 1] == prev\_C:**

**i, j = i, j - 1**

**elif C[i - 1][j] == prev\_C:**

**i, j = i - 1, j**

**else: i, j = i - 1, j - 1**

**Answer = Answer[::-1]**

#### Одномерное динамическое программирование: наилучший способ

### ЗАДАЧА О КУЗНЕЧИКЕ СО СТОИМОСТЯМИ

Пусть кузнечик прыгает на одну или две точки вперед, а за прыжок в каждую точку необходимо заплатить определенную стоимость, различную для различных точек. Стоимость прыжка в точку i задается значением Price[i] списка Price. Необходимо найти минимальную стоимость маршрута кузнечика из точки 0 в точку n.

На этот раз нам необходимо модифицировать определение целевой функции. Пусть C(n) — минимальная стоимость пути из 0 в n. Выведем рекуррентное соотношение для C(n). Чтобы попасть в точку n мы должны попасть в точку n−1 или n−2. Минимальные стоимости этих маршрутов будут равны C(n−1) и C(n−2) соответственно, к ним придется добавить значение Price[n] за прыжок в клетку n. Но из двух клеток n−1 и n−2 нужно выбрать тот маршрут, который имеет наименьшую стоимость.

Получили рекуррентное соотношение:

C(n)=min(C(n−1),C(n−2))+Price(n).

Вычислить значение целевой функции также лучше при помощи динамического программирования, а не рекурсии:

**Пример на языке Python**

**ПРимер на языке Pascal**

 После выполнения этого цикла в списке С будет записана минимальная стоимость маршрута для всех точек от 0 до n.

### ВОССТАНОВЛЕНИЕ ОТВЕТА

Но помимо нахождения наименьшей стоимости маршрута, разумеется, хотелось бы найти и сам маршрут минимальной  стоимости. Такая задача называется задачей «восстановления ответа».

Для восстановления ответа будем для каждой точки i запоминать номер точки Prev[i], из которой кузнечик попал в точку i, если он будет передвигаться по пути минимальной стоимости. То есть Prev[i] — это точка, предшествующая точке с номером i на пути минимальной стоимости (также говорят, что Prev — это массив предшественников). Как определить Prev[i]? Если C[i−1]<C[i−2], то кузнечик попал в точку i из точки i -1, поэтому Prev[i] = i - 1, иначе Prev[i] = i - 2. Модифицируем алгоритмы вычисления значений целевой функции, одновременно вычисляя значения Prev[i].

**Пример программы на языке Python:**

**Пример программы на языке Pascal:**

 Теперь для восстановления пути необходимо начать с точки n и переходить от каждой точки к ее предшественнику, пока путь не дойдет до начальной точки с номером 0. Номера всех вершин будем добавлять в список (массив) Path.

**Пример программы на языке Python:**

**Пример программы на языке Pascal:**

 В конце в список Path добавляется начальная вершина номер 0, которая не была обработана в основном цикле, а затем весь список Path разворачивается в обратном порядке (т. к. вершины добавляются в обратном порядке, от конечной к начальной).

Но можно обойтись и без списка Prev. Мы в любой момент можем определить, из какой точки кузнечик пришел в точку i, если сравним C[i-1] и C[i-2]. Поэтому решение о том, к какой вершине переходить при восстановлении ответа можно принимать непосредственно при восстановлении ответа, сравнив C[i-1] и C[i-2]. Путь восстанавливается через ту вершину, для которой значение C будет меньше.

**ПРимер программы на языке Python**

**Пример программы на языке Pascal:**

#### Одномерное динамическое программирование: количество способов

### Задача о кузнечике

Рассмотрим следующую задачу. На числовой прямой сидит кузнечик, который может прыгать вправо на одну или на две единицы. Первоначально кузнечик находится в точке с координатой 0. Определите количество различных маршрутов кузнечика, приводящих его в точку с координатой n.

Обозначим количество маршрутов кузнечика, ведущих в точку с координатой n, как F(n). Теперь научимся вычислять функцию F(n). Прежде всего заметим, что F(0)=1 (это вырожденный случай, существует ровно один маршрут из точки 0 в точку 0 — он не содержит ни одного прыжка), F(1)=1, F(2)=2. Как вычислить F(n)? В точку n кузнечик может попасть двумя способами — из точки n−2 при помощи прыжка длиной 2 и из точки n−1 прыжком длины 1. То есть число способов попасть в точку n равно сумме числа способов попасть в точку n−1 и n−2, что позволяет выписать рекуррентное соотношение: F(n)=F(n−2)+F(n−1), верное для всех n⩾2.

### Рекурсивное решение

Теперь мы можем оформить решение этой задачи в виде рекурсивной функции:

**Пример на языке Python**

**Пример на языке Pascal**

 Но при попытке вычислить решение этой функции для уже не очень больших n, например, для n=40, окажется, что эта функция работает крайне медленно. И при этом время работы функции с увеличением n растет экспоненциально, то есть такое решение неприемлемо по сложности. Причина этого заключается в том, что при вычислении рекурсивной функции подзадачи, для которых вычисляется решение, «перекрываются». То есть для того, чтобы вычислить F(n) нам нужно вызвать F(n−1) и F(n−2). В свою очередь F(n−1) вызовет F(n−2) и F(n−3). То есть функция F(n−2) будет вызвана два раза — один раз это будет сделано при вычислении F(n−1) и один раз — при вычислении F(n−2). Значение F(n−3) будет вычислено уже три раза, а значение F(n−4) будет вычисляться уже пять раз. При увеличении глубины рекурсии количество «перекрывающихся» вызовов функций будет расти экспоненциально. То есть одна из причин неэффективности рекурсивного решения — одно и то же значение функции вычисляется несколько раз, так как оно используется для вычисления нескольких других значений функции.

### Нерекурсивное решение

На самом деле несложно видеть, что значения рекурсивной функции в данном случае будут совпадать с числами Фибоначчи, так как вычисляются по тем же рекуррентным соотношениям. А для вычисления чисел Фибоначчи можно использовать цикл, а не рекурсию — следующее число Фибоначчи определяется, как сумма двух предыдущих.

**Пример на языке Python**

**Пример на языке Pascal**

Сложность такого решения будет O(n). Сложность вычисления уменьшается за счет того, что для каждого промежуточного i значение F(i) вычисляется один раз и сохраняется в списке, чтобы впоследствии использовать это значение несколько раз для вычисления F(i+1)  и F(i+2).

Такой прием называется**динамическим программированием.** Динамическое программирование использует те же рекуррентные соотношения, что и рекурсивное решение, но в отличии от рекурсии в динамическом программировании значения вычисляются в цикле и сохраняются в списке. При этом заполнение списка идет от меньших значений к большим, в то время как в рекурсии — наоборот, рекурсивная функция вызывается для больших значений, а затем вызывает сама себя для меньших значений.

### Модификации задачи о кузнечике

Модифицируем задачу. Пусть кузнечик прыгает на одну, две или три единицы, необходимо также вычислить количество способов попасть в точку n. В рекуррентном соотношении добавится еще одно слагаемое: F(n)=F(n−1)+F(n−2)+F(n−3). И начальные значения для вычисления теперь должны состоять из трех чисел: F(0), F(1), F(2). Решение изменится не сильно:

**Пример на языке Python**

**Пример на языке Pascal**

Еще раз модифицируем задачу. Пусть некоторые точки являются «запретными» для кузнечика, он не может прыгать в эти точки. «Карта» запрещенных точек задается при помощи списка Map: если Map[i] == 0 (для языка Pascal — массива Map), то в точку номер i кузнечик не может прыгать, а если Map[i] == 1, то данная точка является разрешенной для кузнечика. Как и в предыдущей задаче, необходимо найти количество маршрутов в точку n.

В данном случае также придется модифицировать вид рекуррентного соотношения: если Map[i] == 0, то F[i] = 0, то есть если точка — «запрещенная», то количество способов попасть в эту точку равно 0, так как нет ни одного допустимого маршрута, заканчивающегося в этой точке. Если же Map[i] == 1, то значение F[i] вычисляется по тем же рекуррентным соотношениям, что и ранее. Получаем следующее решение:

**Пример на языке Python**

**Пример на языке Pascal**

Здесь используется немного другой код для вычисления суммы F[i - 3] + F[i - 2] + F[i - 1] для того, чтобы крайние значения F[1] и F[2] также можно было вычислить при помощи этого кода в основном цикле, а не перед ним.

#### Наибольшая общая подпоследовательность

Рассмотрим две строки (или числовые последовательности) — A и B. Пусть первая строка состоит из n символов a0…an−1, вторая строка состоит из m символов b0…bm−1. Подпоследовательностью данной строки (последовательности) называется некоторое подмножество символов исходной строки, следующих в том же порядке, в котором они идут в исходной строке, но не обязательно подряд. Если в строке n символов, то у нее 2n различных подпоследовательностей: каждый из n символов строки может либо входить, либо не входить в любую выбранную подпоследовательность. Пустая подпоследовательность не содержит ни одного элемента и также является подпоследовательностью любой строки.

Рассмотрим задачу — для двух данных строк найти такую строку наибольшей длины, которая была бы подпоследовательностью каждой из них. Например, если A=«abcabaac», B=«baccbca» то у строк A и B них есть общая подпоследовательность длины 4, например, «acba» или «acbc».

Данную задачу можно решить перебором — например, перебрав все 2n подпоследовательностей первой строки и для каждой их них проверив, является ли она подпоследовательностью второй строки. Но при помощи динамического программирования эту же задачу можно решить за сложность O(nm).

Рассмотрим последние символы данных строк an−1 и bm−1. Если эти символы совпадают, то они обязательно войдут последними символами и в наибольшую общую подпоследовательность данных строк. Тогда можно свести задачу нахождения наибольшей общей подпоследовательности для строк A=a0…an−1 и B=b0…bm−1 к задаче нахождения наибольшей общей подпоследовательности для строк, полученных отбрасыванием от данных строк последнего символа, то есть для a0…an−2 и b0…bm−2. Затем к ответу для «укороченных» строк добавим последние (равные) символы исходных строк (an−1 или bm−1) и получим ответ для исходных строк.

Если же последние символы исходных строк не совпадают, то эти символы (an−1 и bm−1) не могут одновременно входить в наибольшую общую подпоследовательность, поэтому можно один из них отбросить. Тогда задача сводится к нахождению наибольшей общей подпоследовательности для одного из двух случаев - для строк a0…an−2 и b0…bm−1 или для строк a0…an−1 и b0…bm−2.

Мы научились сводить задачу нахождения наибольшей общей подпоследовательности двух строк к меньшей задаче - нахождения наибольшей общей подпоследовательности для строк, полученных отбрасыванием последних символов от исходных строк, то есть для префиксов исходных строк. Для дальнейшего решения задачи будем следовать принципу построения решения при помощи динамического программирования.

Рассмотрим префикс A′ первой строки из i символов: A′=a0…ai−1 и префикс B′ второй строки из j символов: B′=b0…bj−1. В терминах срезов языка Питон можно считать, что A′=A[:i] и B′=B[:j]. Обозначим через F(i,j) длину наибольшей общей подпоследовательности для A′ и B′.

Теперь выпишем рекуррентные соотношения. Они зависят от того, совпадают ли последние символы рассматриваемых строк A′ и B′. Если ai−1=bj−1, то тогда F(i,j)=F(i−1,j−1)+1 - нужно решить задачу для строк, полученных отбрасыванием последних символов рассматриваемых строк и добавить 1 символ к ответу. В противном случае нужно рассмотреть два случая: F(i−1,j) и F(i,j−1), которые соответствуют отбрасыванию по одному символу от конца каждой из рассматриваемых строк. В этом случае F(i,j)=max(F(i−1,j),(i,j−1)).

Начальные значения функции F задаются просто: если одна из строк - пустая, то общая подпоследовательность также пустая, то есть имеет длину 0: F(0,j)=F(i,0)=0.

Далее необходимо завести двумерный массив размером (n+1)×(m+1) и заполнить его значениями по указанным рекуррентным соотношениям. Сначала весь массив заполним нулями (что задаст граничные значения), затем двумя вложенными циклами по i и по j заполним оставшуюся часть массива:

**n = len(A)**

**m = len(B)**

**F = [[0] \* (m + 1) for i in range(n + 1)]**

**for i in range(1, n + 1):**

**for j in range(1, m + 1):**

**if A[i - 1] == B[j - 1]:**

**F[i][j] = F[i - 1][j - 1] + 1**

**else:**

**F[i][j] = max(F[i - 1][j], F[i][j - 1])**

**print(F[n][m])**

В таблице ниже приведен пример заполнения массива для строки «abcabaac» и «baccbca». Длина наибольшей общей подпоследовательности для данных строк равна 4.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **b** | **a** | **c** | **c** | **b** | **c** | **a** |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **a** | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| **b** | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| **c** | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| **a** | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 4 |
| **b** | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 4 |
| **a** | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 4 |
| **a** | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 4 |
| **c** | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 4 |

Этот код находит длину наибольшей общей подпоследовательности. Для нахождения самой общей подпоследовательности необходимо восстановить ответ. Для этого выполним «обратный проход» по массиву F начиная с последнего элемента. В каждой рассматриваемой ячейке F[i][j] выясним, как было получено значение в этой ячейке. Это зависит от последних символов рассматриваемых префиксов. Если ai−1=bj−1, то тогда ответ для элемента F[i][j] получен из F[i−1][j−1] добавлением 1, поэтому перейдем к элементу F[i−1][j−1], а к ответу добавим символ ai−1=bj−1. Иначе нужно перейти к тому элементу F[i−1][j] или F[i][j−1], значение в котором совпадает со значением F[i][j].

Алгоритм восстановления ответа записан ниже:

**Ans = []**

**i = n**

**j = m**

**while i > 0 and j > 0:**

**if A[i - 1] == B[j - 1]:**

**Ans.append(A[i - 1])**

**i -= 1**

**j -= 1**

**elif F[i - 1][j] == F[i][j]:**

**i -= 1**

**else:**

**j -= 1**

**Ans = Ans[::-1]**

**Вычисление расстояния Левенштейна**

Пусть через канал связи передается какой-либо сигнал, состоящий из последовательности символов. При передаче сигнала возможны искажения - некоторые символы заменяются на другие. Число различий между двумя последовательностями сигналов (между двумя строками), то есть количество соответствующих символов в последовательностях, которые различаются, называется расстоянием Хемминга между строками. Расстояние Хемминга (оно определено только для строк равной длины) - это количество символов, которые нужно изменить в одной строке, чтобы получить другую строку, то есть это мера «близости» двух строк.

Если разрешать не только менять символы, но также удалять и вставлять символы, то минимальное количество изменений (замена символа, удаление символа, вставка символа), которое необходимо сделать с одной строкой, чтобы получить другую строку, называется расстоянием Левенштейна (в честь российского математика Владимира Левенштейна) или расстоянием редактирования. Такое расстояние появляется, например, в системах автоматической проверки орфографии. Если при наборе текста человек допустил одну или несколько ошибок, то набранное им слово будет отсутствовать в словаре. При этом нужно понять, какое слово наборщик имел в виду. Если считать, что наиболее типичные ошибки при наборе текста - это пропуск символа, вставка лишнего символа или замена одного символа на другой, то необходимо неправильное слово заменить на слово из словаря с минимальным расстоянием Левенштейна.

Алгоритм вычисления расстояния Левенштейна между двумя словами аналогичен алгоритму нахождения наибольшей общей подпоследовательности. Пусть F(i,j) есть расстояние редактирования между префиксами данных строк A′=A[:i] и B′=B[:j]. Рассмотрим последние символы этих префиксов ai−1 и bj−1. Посмотрим, как префикс B′=B[:j] мог быть получен из префикcа A′=A[:i] при помощи разрешенных операций редактирования. Здесь возможны следующие варианты:

1. Если последние символы префиксов совпадают (ai−1=bj−1), то в этом случае можно не менять эти последние символы. Тогда F(i,j)=F(i−1,j−1).
2. Если ai−1≠bj−1, то тогда можно потратить 1 операцию на замену символа ai−1 на bj−1 и также потратить F(i−1,j−1) операцию на превращение префикса A[:i−1] в B[j−1]. Тогда F(i,j)=F(i−1,j−1)+1.
3. Символ ai−1 был удален при редактировании, тогда необходимо префикс A[:i−1] превратить в B[:j], на что необходимо F(i−1,j) операций редактирования. В этом случае F(i,j)=F(i−1,j)+1.
4. Символ bj−1 был добавлен при редактировании, тогда необходимо префикс A[:i] превратить в B[:j−1], на что необходимо F(i,j−1) операций редактирования. В этом случае F(i,j)=F(i,j−1)+1.

Далее при вычислении F(i,j) необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 1 или 2 рассматривается только один, в зависимости от условия ai−1=bj−1).

Начальные значения: F(i,0)=i, F(0,j)=j.

#### Наибольшая возрастающая подпоследовательность

Рассмотрим числовую последовательность из n элеметов a0…an−1. Рассмотрим задачу нахождения среди всех возможных подпоследовательностей данной последовательности монотонно возрастающей подпоследовательности наибольшей длины.

У данной задачи есть несколько способов решения.

**Первый способ** — отсортируем последовательность в порядке неубывания, удалим из нее повторяющиеся элементы (то есть получим строго возрастающую последовательность B из элементов b0,…,bm−1. Теперь для последовательностей A и B найдем наибольшую общую подпоследовательность. Понятно, что эта подпоследовательность будет подпоследовательностью A, будет монотонно возрастать и будет иметь наибольшую длину из всех таких последовательностей. Сложность такого алгоритма будет O(n2).

Возможен и **другой способ** решения задачи при помощи динамического программирования. Обозначим через F(i) длину наибольшей возрастающей подпоследовательности, последним элементом которой будет элемент ai. Тогда для вычисления значения F(i) рассмотрим предпоследний элемент этой последовательности. Пусть это элемент aj, тогда j<i и aj<ai. Длина наибольшей возрастающей подпоследовательности, заканчивающейся aj есть F(i), значит, необходимо найти такое подходящее j, что F(j) будет наибольшим. Итак, F(i)=1+minj<i,aj<aiF(j). Если же ни одного такого подходящего j нет (то есть все aj⩾ai при j<i), то F(i)=1.

Соответствующая программа вычисления значений функции F будет выглядеть так:

**F = [0] \* len(A)**

**for i in range(len(A)):**

**for j in range(i):**

**if A[j] < A[i] and F[j] > F[i]:**

**F[i] = F[j]**

**F[i] += 1**

Ответом (длиной наибольшей возрастающей подпоследовательности) будет наибольшее значение F(i).

Для восстановления ответа найдем такой элемент последовательности ai, что F(i) будет максимальным. Это будет последний элемент наибольшей возрастающей подпоследовательности. Теперь найдем предыдущий элемент. Это такой элемент aj, что j<i и F(j)=F(i)−1. Будем повторять поиск предыдущего элемента до тех пор, пока не дойдем до такого j, что F(j)=, это будет первым элементом наибольшей возрастающей подпоследовательности.

Легко видеть, что сложность такого алгоритма O(n2). Тем не менее, можно придумать решение, которое будет иметь сложность O(nlog⁡n).

Рассмотрим следующую функцию: F(i) - наименьший элемент последовательности, которым может заканчиваться наибольшая возрастающая подпоследовательность длины i. то есть будем для каждой возможной длины i возрастающей подпоследовательности пытаться выбрать подпоследовательность из i элементов, при этом минимизировать последний элемент последовательности.

Также добавим фиктивные элементы: F(0)=−inf, а также F(n+1)=+inf, что имеет следующий смысл --последовательность длины n+1 не существует, то есть считаем, что последний элемент бесконечной большой (к ней ничего больше нельзя добавить). А последовательность длины 0 наоборот имеет последним элементом «минус бесконечность», то есть к ней можно добавить что угодно.

Теперь будем по одному рассматривать элементы исходной последовательности, начиная с самого первого. При этом начальная инициализация такая - F(0)=−inf, F(i)=+inf при i>0, то есть в самом начале не известно ни одной возрастающей подпоследовательности, даже длины 1. Теперь рассмотрим очередной элемент ai. Его можно добавить в конец любой возрастающей подпоследовательности, которая заканчивается числом, меньшим, чем ai. Если есть возрастающая подпоследовательность длины k, которая заканчивается каким-то значением x, то есть F(k)=x, при этом x<ai, то возможно построить последовательность длины k+1 с последним элементом, равным ai, добавив его в конец подпоследовательности длины k. При этом это должно приводить к улучшению значения F(k+1), то есть F(k+1) должно быть не меньше, чем ai. Это означает, что необходимо среди элементов списка F найти такое значение k, что F(k)<ai, F(k+1)⩾ai, после чего необходимо установить F(k+1)=ai. Это можно сделать при помощи функции двоичного поиска LowerBound, которая позволяет в списке F найти первое значение, не меньшее, чем ai, после чего ему необходимо присвоить ai. Заметим, что при этом монотонность значений в списке F сохраняется, что делает оправданным двоичный поиск.

Соответствующий алгоритм может выглядеть следующим образом:

**INF = 10 \*\* 10**

**F = [INF] \* (len(A) + 1)**

**F[0] = -INF**

**for i in range(len(A)):**

**left = 0**

**right = len(A)**

**while right - left > 1:**

**middle = (left + right) // 2**

**if F[middle] >= A[i]:**

**right = middle**

**else:**

**left = middle**

**F[right] = A[i]**

Ответом (длиной наибольшей возрастающей подпоследовательности) является такое наибольшее i, что F(i)≠+inf.

Восстановление ответа для данной задаче оставим в качестве самостоятельного упражнения. Подскажем, что для восстановления ответа необходимо хранить для каждого элемента предыдущий элемент, а также для каждой возможной длины i необходимо хранить номер последнего элемента возрастающей подпоследовательности длины i в исходной последовательности.

#### Алгоритм "укладки рюкзака"

Пусть имеется набор предметов, каждый из которых имеет два параметра — вес и ценность. И есть рюкзак, определенной вместимости. Задача заключается в том, чтобы собрать рюкзак с максимальной ценностью предметов внутри, соблюдая при этом весовое ограничение рюкзака.

#### Задача «Банкомат»

Рассмотрим следующую задачу. В банкомате имеется банкноты n различных номиналов a1, a2, …, an. Клиент хочет получить сумму в K денежных единиц. Необходимо определить, при помощи какого минимального числа банкнот можно выдать эту сумму (а при необходимости восстановления ответа - определить способ выдачи, использующий минимальное число банкнот).

Очевидно приходящий в голову «жадный алгоритм» - выдавать банкноты наиболее крупного номинала, пока это возможно, затем переходить к более мелкому номиналу, в общем случае неверен. Например, пусть номиналы банкнот a1=1, a2=50, a3=90, а сумма для выдачи K=100. Тогда «жадный» алгоритм выдаст банкноту в 90 и 10 банкнот по 1, в то время как существует решение, использующее всего лишь две банкноты по 50.

На помощь придет динамическое программирование. Пусть F(k) - минимальное число банкнот, при помощи которых можно выдать сумму в k рублей. Выберем одну из банкнот, входящую в оптимальный способ выдачи. Пусть это банкнота ai. Тогда необходимо выдать оставшуюся сумму k−ai, что можно сделать при помощи F(k−ai) банкнот. То есть F(k)=1+F(k−ai). Далее необходимо взять минимум по всем возможным значениями i: F(k)=1+miniF(k−ai).

Начальные значения удобно сделать такими: F(0)=0, F(k)=+∞ при k>0. Значение +∞ будет означать невозможность выдачи суммы вообще, то есть «очень плохой вариант», когда банкнот потребуется бесконечно много. Можно добавить к списку значений функции F(k) «каемочку» для отрицательных значений k, но лучше просто рассматривать только такие значения k и i, когда k−ai≥0.

В данном примере программы будем считать, что номиналы банкнот хранятся в спискe A и нумерация банкнот начинается с числа 0.

**INF = 10 \*\* 10**

**F = [INF] \* (K + 1)**

**F[0] = 0**

**for k in range(1, K + 1):**

**for i in range(len(A)):**

**if k - A[i] >= 0 and F[k - A[i]] < F[k]:**

**F[k] = F[k - A[i]]**

**F[k] += 1**

Для восстановления ответа будем опять идти «к началу» списка, уменьшая сумму k, выбирая такую банкноту ai, что F(k)=F(k−ai)+1. Номиналы банкнот, которые будут при этом использоваться для восстановления ответа, будут записаны в список Ans.

**Ans = []**

**k = K**

**while k != 0:**

**for i in range(len(A)):**

**if k - A[i] >= 0 and F[k] == F[k - A[i]] + 1:**

**Ans.append(A[i])**

**k -= A[i]**

#### Задача «Золотые слитки»

Предыдущий алгоритм также решал задачу проверки возможности выдачи любой суммы при помощи заданного номинала банкнот - сумму k возможно выдать, если F(k)<+∞.

Также в предыдущей задаче банкнот каждого номинала было неограниченно много. Теперь рассмотрим задачу, в которой существует ровно одна банкнота каждого номинала (но номиналы банкнот могут повторяться), необходимо проверить, можно ли заданную сумму K выдать при помощи данных банкнот.

Этой задаче можно придать и такой смысл: есть n золотых слитков массами a1, a2, …, an. Какую максимальную массу золота можно унести, если она не может превышать K (то есть «грузоподъемность» не превосходит K).

Задачу также можно решать при помощи динамического программирования. Пусть F(k) - признак того, можно ли набрать слитков на массу в точности k, то есть одно из двух значений True (или 1) или False (или 0).

Будем по очереди рассматривать все слитки, обновляя значения F(k). При рассмотрении слитка ai необходимо пометить F(k)=1, если F(k−ai)=1, то есть если массу в точности k можно набрать, если ранее было возможно набрать массу в точности k−ai.

**F = [0] \* (K + 1)**

**F[0] = 1**

**for i in range(len(A)):**

**F\_new = F[:] # копия списка F для обновления**

**for k in range(A[i], K + 1):**

**if F[k - A[i]] == 1:**

**F\_new[k] = 1**

**F = F\_new**

Данный алгоритм нуждается в пояснении. В самом начале список F заполняется значением 0, кроме F(0)=1, что означает, что нулевую массу можно набрать не используя ни одного слитка, а любую другую массу не используя ни одного слитка, набрать нельзя.

Внутренний цикл начинается со значения ai, так как массу k можно набрать, если ранее было возможно набрать массу k−ai, то есть минимальное значение для k равно ai.

Кроме того, нельзя вносить исправления сразу же в список F, потому что в этом случае один предмет будет учтен более одного раза, то есть будут помечены единицами F(ai), затем - F(2ai), так как F(ai)=1, затем F(3ai) и т. д. Чтобы избежать этой ошибки, в этом алгоритме создается копия списка F и в него вносятся изменения, тем самым не будут учитываться изменения в списке, сделанные при добавлении слитка ai.

Другой способ избежать этой проблемы - обходить список F с конца - от большего значения к меньшему.

**F = [0] \* (K + 1)**

**F[0] = 1**

**for i in range(len(A)):**

**for k in range(K, A[i] - 1, -1):**

**if F[k - A[i]] == 1:**

**F[k] = 1**

Тогда ответом в задаче будет являться индекс самой правой единички в списке F.

Для восстановления ответа для каждого значения k будем хранить массу слитка, при помощи которого он был получен, будем хранить его в списке Prev. Значение списка Prev будет обновляться при записи F(k)=1:

**F = [0] \* (K + 1)**

**F[0] = 1**

**Prev = [0] \* (K + 1)**

**for i in range(len(A)):**

**for k in range(K, A[i] - 1, -1):**

**if F[k - A[i]] == 1:**

**F[k] = 1**

**Prev[k] = A[i]**

**i = K**

**while F[i] == 0:**

**i -= 1**

**#i - максимальная масса, которую можно набрать**

Теперь для восстановления ответа будем уменьшать значение k на Prev(k) пока не получим k=0.

**Ans = []**

**k = i**

**while k > 0:**

**Ans.append(Prev[k])**

**k -= Prev[k]**

### ДИСКРЕТНАЯ ЗАДАЧА ОБ УКЛАДКЕ РЮКЗАКА

Следующим обобщением задачи про золотые слитки является «задача об укладке рюкзака». В этой задаче также имеется несколько предметов, для каждого предмета заданы две характеристики: вес wi>0 и стоимость («полезность») предмета pi>0. Необходимо выбрать множество предметов суммарной максимальной стоимости, при этом суммарная масса выбранных предметов должна быть ограничена значением K.

Одним из способов решения этой задачи является полный перебор всех подмножеств из n предметов, которых будет 2n и выбор среди них наилучшего подмножества, удовлетворяющего условиям задачи. Такой алгоритм будет иметь сложность O(2n).

Если же массы всех предметов являются целыми числами (так называемая «дискретная» задача), то в данном случае возможно придумать решение сложности O(nK) при помощи динамического программирования.

Как и в задаче про золотые слитки, будем для каждой возможной массы k хранить информацию о способе набора этой массы, но в отличии от задачи про слитки будем хранить не возможность набора данной массы (0 или 1), а наилучшее решение для данной массы, то есть наибольшую стоимость предметов, которые можно набрать в рюкзак данной массы.

Формально определим так: F(i,k) - максимальная стоимость предметов, которые можно уложить в рюкзак массы k, если можно использовать только первые i предметов.

Выведем рекуррентное соотношение для F(i,k) уменьшив значение i. Есть две возможности собрать рюкзак, используя первые i предметов - взять предмет с номером i или не брать.

Если не брать предмет с номером i, то в этом случае F(i,k)=F(i−1,k), так как рюкзак массы k будет собран только с использованием первых i−1 предмета. В этом случае F(i,k)=F(i−1,k).

Если же предмет номер i войдет в рюкзак (это можно сделать только при k⩾wi), то останется свободная вместимость рюкзака k−wi, которую можно будет заполнить первыми i−1 предметом, максимальная стоимость рюкзака в этом случае будет F(i−1,k−wi). Но поскольку предмет номер i был включен в рюкзак, то стоимость рюкзака увеличится на pi. То есть в этом случае F(i,k)=F(i−1,k−wi)+pi.

Из двух возможных вариантов нужно выбрать вариант наибольшей стоимости, то есть F(i,k)=max(F(i−1,k),F(i−1,k−wi)+pi).

Для хранения значения функции F будем использовать двумерный список. При этом массы предметов хранятся в списке W, их стоимости - в списке P. Будем считать (для простоты записи программы), что предметы пронумерованы от 1 до n.

**F = [ [0] \* (K + 1) for i in range(n + 1)]**

**for i in range(1, n + 1):**

**for k in range(1, K + 1):**

**if k >= W[i]:**

**F[i][k] = max(F[i - 1][k], F[i - 1][k - W[i]] + P[i])**

**else:**

**F[i][k] = F[i - 1][k]**

Для восстановления ответа будем перебирать все предметы «с конца» от n до 1. В переменной k будет храниться текущая вместимость рюкзака. Рассматривая предмет номер i определим, как было получено значение F(i,k). Если F(i,k)=F(i−1,k), то можно не включать предмет i в рюкзак и перейти к предмету i−1 не меняя значения k. Иначе предмет i нужно включить в рюкзак, при этом значение k уменьшается на wi.

**k = K**

**for i in range(n, 0, -1):**

**if F[i][k] != F[i - 1][k]:**

**Ans.append(i)**

**k -= W[i]**

В этой реализации случай, когда F(i,k)=F(i−1,k) просто пропускается, и рассматривается только случай F(i,k)≠F(i−1,k). После окончания алгоритма в списке Ans будут храниться номера предметов, входящих в рюкзак.

#### Двумерное динамическое программирование: игры

Рассмотрим игру «Ферзя в угол» для двух игроков. В левом верхнем углу доски размером n×m находится ферзь, который может двигаться только вправо-вниз. Игроки по очереди двигают ферзя, то есть за один ход игрок может переместить ферзя либо по вертикали вниз, либо по горизонтали вправо, либо во диагонали вправо-вниз. Игрок, который не может сделать ход — проигрывает, иными словами, выигрывает игрок, который поставит ферзя в правый нижний угол. Необходимо определить, какой из игроков может выиграть в этой игре независимо от ходов другого игрока.

Эту задачу также можно решить при помощи динамического программирования. Будем заполнять доску знаками «+» и «--». Знак «+» будет означать, что данная клетка является выигрышной для ходящего с неё игрока (то есть если ферзь стоит в этой клетке, то игрок, который делает ход, может всегда выиграть), а знак «--» означает, что он проигрывает. Клетки последней строки, последнего столбца и диагонали, ведущей из правого нижнего угла необходимо отметить, как «+», так как если ферзь стоит в этой клетке, то ходящий игрок может выиграть одним ходом:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + |  |  |  |  | + |
|  | + |  |  |  | + |
|  |  | + |  |  | + |
|  |  |  | + |  | + |
|  |  |  |  | + | + |
| + | + | + | + | + | -- |

Но в правом нижнем углу необходимо поставить знак «--» — если ферзь стоит в углу, то тот игрок, которых должен делать ход, уже проиграл.

Теперь рассмотрим две клетки, из которых можно пойти только в те клетки, в которых записан знак «+». В этих клетках нужно записать знак «--» — если ферзь стоит в этих клетках, то какой бы ход не сделал ходящий игрок, ферзь окажется в клетке, в которой стоит знак «+», то есть выигрывает ходящий игрок. Значит, тот, кто сейчас ходит — всегда проигрывает.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + |  |  |  |  | + |
|  | + |  |  |  | + |
|  |  | + |  |  | + |
|  |  |  | + | - | + |
|  |  |  | - | + | + |
| + | + | + | + | + | - |

Но теперь в те клетки, из которых можно попасть в клетку, в которой стоит знак «-» за один ход, необходимо записать знак «+» — если ферзь стоит в этой клетке, то игрок, который делает ход, может выиграть, если передвинет ферзя в клетку, в которой стоит знак «--»:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | + |  | + | + | + |
| + | + | + | + | + | + |
|  | + | + | + | + | + |
| + | + | + | + | -- | + |
| + | + | + | -- | + | + |
| + | + | + | + | + | -- |

Дальше таблица заполняется аналогично. В клетке ставиться знак «+», если есть ход, который ведет в клетку, в  которой стоит знак «--». В клетке ставится знак «-», если все ходы из этой клетки ведут в клетки, в которых записан знак «+».

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | + | -- | + | + | + |
| + | + | + | + | + | + |
| -- | + | + | + | + | + |
| + | + | + | + | -- | + |
| + | + | + | -- | + | + |
| + | + | + | + | + | -- |

Продолжая таким образом, можно определить выигрывающего игрока для любой начальной клетки. Реализацию данного алгоритма оставим читателям.

#### Динамическое программирование на подотрезках

Рассмотрим следующую задачу. Пусть есть некоторая строка, нужно вычеркнуть из нее наименьшее число символов, чтобы невычеркнутые символы образовывали палиндром.

Иными словами, из данной строки нужно выбрать подпоследовательность наибольшей длины (т.е. некоторое число элементов, сохраняя их порядок, но не обязательно идущих подряд), образующую палиндром.

Пусть длина исходной строки равна n, символы строки индексируются начиная с нуля. Рассмотрим следующую функцию динамического программирования:

f(i,j) — длина максимальной подпоследовательности-палиндрома, которую можно получить из строки s[i..j], то есть из подстроки из символов с индексами от i до j (включительно). При этом 0⩽i⩽j<n, то есть функция определена для каждой подстроки s[i..j] исходной строки s. Такой тип динамического программирования называется динамическим программированием на подотрезках.

При этом отметим, что f(i,i)=1 (подстрока из одного символа уже палиндром), а вот f(i,i+1) может быть равно 1 (если s[i]≠s[i+1]) или 2 (если s[i]=s[i+1]).

Также удобно считать, что в вырожденных случаях i>j значение f(i,j)=0, т.к. рассматриваемая подстрока пустая.

Выведем рекуррентное соотношение для f(i,j). Если символы s[i] и s[j] равны, то включим их оба в выбранную подпоследовательность, а между ними вставим наибольшую подпоследовательность-палиндром между ними, ее длина равна f(i+1,j−1). То есть в этом случае f(i,j)=2+f(i+1,j−1).

Если же s[i]≠s[j], то символы s[i] и s[j] не могут одновременно входить в подпоследовательность-палиндром, значит, один из них обязательно нужно выбросить.

В этом случае задача сводится либо к подстроке s[i+1…j], либо s[i…j−1]. Выберем из них наибольший вариант, то есть в этом случае f(i,j)=max(f(i+1,j),f(i,j−1)).

Теперь для нахождения ответа необходимо организовать заполнение двумерного массива f значениями так, чтобы при рассмотрении отрезка [i,j] все значения на вложенных в него подотрезках уже были бы вычислены. Это удобно делать при помощи двух вложенных циклов - первый цикл по возрастанию правой границы j, второй вложенный цикл - по убыванию левой границы i от j до нуля.

**vector<vector<int> > f(n, vector<short int>(n));**

**for (int j = 0; j < n; ++j)**

**{**

**f[j][j] = 1;**

**for (int i = j - 1; i >= 0; --i)**

**if (s[i] == s[j])**

**f[i][j] = 2 + f[i + 1][j - 1];**

**else**

**f[i][j] = max(f[i + 1][j], f[i][j - 1]);**

**}**

Ответом (длиной максимальной подпоследовательности-палиндрома) является значение f[0][n−1].

Необходимо также уметь восстанавливать ответ - то есть находить не только длину подпоследовательности-палиндрома, но и сам палиндром.

Сделаем это при помощи рекурсивной функции, у которой также два параметра - первый и последний символ подстроки, для которой восстанавливается ответ.

**string ans(int i, int j)**

**{**

**if (i > j)**

**return "";**

**else if (i == j)**

**return s[i];**

**else if (s[i] == s[j])**

**return s[i] + ans(i + 1, j - 1) + s[j];**

**else if (f[i + 1][j] > f[i][j - 1])**

**return ans(i + 1, j);**

**else**

**return ans(i, j - 1);**

**}**

Для построения ответа необходимо вызвать функцию ans(0, n - 1).