**Двумерные массивы**

**Хранение матриц**

Часто в задачах приходится хранить прямоугольные таблицы с данными. Такие таблицы называются **матрицами** или **двумерными массивами**. В языке программирования Python таблицу можно представить в виде списка, каждый элемент которого тоже является списком, например, списком чисел.

**Пример:**

Если требуется создать числовую таблицу из двух строк и трёх столбцов, то это можно сделать следующим образом: a = [[2, 3, 4], [5, 6, 7]].

Здесь первый элемент списка a[0] является списком из чисел [2, 3, 4]. То есть a[0][0] равно 22, a[0][1] равно 33, a[0][2] равно 44. Таким образом, чтобы обратиться к элементу, расположенному в ii-й строке и jj-м столбце, то надо написать a[i][j]. Также можно использовать отрицательные индексы, например, элемент a[-1][-1] нашей таблицы — это элемент из последней строки и последнего столбца, и он равен числу 77.

**Обработка и вывод списка**

Для обработки и вывода списка, как правило, используются два вложенных цикла. Первый цикл перебирает все строки, второй цикл — элементы внутри строки. Например, вывести двумерный числовой список на экран построчно, разделяя числа пробелами внутри одной строки, можно так:

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

 **for** j **in** range(len(a[i])):

 **print**(a[i][j], end=' ')

 **print**()

Обратим внимание на то, что пробелы будут выведены не только между числами в строке, но и после последнего элемента. При сдаче задач эти пробелы не мешают, поэтому можно об этом не беспокоиться.

Можно перебирать элементы списка не по индексу, а по значению. Тогда код вывода двумерного массива будет иметь вот такой вид:

**for** row **in** a:

 **for** elem **in** row:

 **print**(elem, end=' ')

 **print**()

Кроме того, для вывода одной строки можно воспользоваться методом join:

**for** row **in** a:

 **print**(' '.join(list(map(str, row))))

Наконец, можно использовать и упрощенный способ вывода списков:

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

 **print**(\*a[i])

**Пример:**

Используем два вложенных цикла для подсчета суммы всех чисел в таблице и нахождения минимального элемента в таблице:

S = 0

M = a[0][0]

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

 **for** j **in** range(len(a[i])):

 S += a[i][j]

 M = min(M, a[i][j])

**print**(S, M)

Для решения этой задачи можно использовать циклы не по индексам, а по элементам:

S = 0

M = a[0][0]

**for** row **in** a:

 **for** elem **in** row:

 S += elem

 M = min(M, elem)

**print**(S, M)

Заметим, что если мы хотим параллельно изменять значения элементов в нашей таблице, то придётся использовать первый вариант. Действительно, переменная elem является дополнительной переменной, и если мы ее меняем, то элемент таблицы при этом не меняется. Таким образом, если мы хотим не только найти сумму чисел и минимум исходной таблицы, но и увеличить все элементы таблицы на 11, то для этого надо использовать первый вариант, и код будет выглядеть следующим образом:

S = 0

M = a[0][0]

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

 **for** j **in** range(len(a[i])):

 S += a[i][j]

 M = min(M, a[i][j])

 a[i][j] += 1

**print**(S, M)

**Создание вложенных списков**

**Копирование списков**

Научимся копировать списки в языке Python. Пусть у нас есть список a = [2, 3, 4]. Оказывается, что операция b = a не создаёт новый список. Такая операция делает имя b ссылкой на тот же список, на который ссылается имя a. Таким образом, a и b — это два разных имени, ссылающиеся на один список. Поэтому если мы модифицируем список a, например, добавив число 55 в конец списка: a.append(5), то модифицируется и список b.

Нам же нужно выполнить такую операцию, чтобы имя b ссылалось на список, который является копией списка a, а не на сам список a. Для этого можно использовать присваивание с использованием среза: b = a[:]. Также можно использовать генератор: b = [elem **for** elem **in** a].

Если есть необходимость создать копию двумерного списка, то возникают дополнительные сложности. Чтобы их избежать, рекомендуется выполнять копирование с использованием специальных функций из модуля copy. Подключить модуль copy можно, написав **from**copy **import**\* в начале программы.

**Создание двумерных списков**

Пусть теперь нам нужно создать двумерный список из 33 строк и 44 столбцов, заполненный нулями. Кажется, что такой двумерный список можно создать следующим образом: a = [[0] \* 4] \* 3. Но тут возникает проблема. При таком способе a[0], a[1] и a[2] являются ссылками на один и тот же список [0] \* 4. Поэтому после операции a[0][0] = 1, окажется, что элементы a[1][0] и a[2][0] тоже стали равны числу 11.

Чтобы правильно создать двумерный список, заполненный нулями и состоящий из nn строк и mm столбцов, необходимо, чтобы каждая строка списка создавалась заново. Есть несколько способов сделать это.

Например, можно создать список из nn нулей, а затем каждый элемент этого списка заменить на список из mm нулей. Во время замены список из mm нулей будет создаваться заново, поэтому каждая из nn строк окажется ссылкой на свой независимый список из mm нулей:

a = [0] \* n

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

 a[i] = [0] \* m

Можно создать изначально пустой список, а потом nn раз добавить в конец этого списка новый элемент, который является списком из mm нулей:

a = []

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

 a.append([0] \* m)

Также для создания двумерного списка можно использовать генератор, который создает список из nn элементов, каждый из которых будет списком, состоящим из mm нулей (подробнее о генераторах списков будет рассказано в следующем модуле):

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

В каждом из этих трёх способов очередная строка создается независимо от остальных: заново конструируется список [0] \* m, а не копируются ссылки на один и тот же список.

**Считывание**

**Считывание двумерных массивов**

Пусть программа получает на вход двумерный массив, состоящий из nn строк и mm столбцов. Научимся считывать такой массив.

Обычно в задачах нам сначала дают два числа nn и mm — размеры массива, а затем дают nn строк, в каждой из которых записаны по mm чисел, разделённых пробелами. Пример входных данных:

3 4

1 2 3 4

2 3 4 5

3 4 5 6

Сначала считаем два числа nn и mm, задающие размеры двумерного массива. После этого создадим пустой список a и повторим nn раз следующую последовательность действий: считаем очередную строку из чисел, превратим строку в список из mm чисел, разбив её по пробелам и преобразовав элементы к типу int, добавим полученный список в конец списка a.

n, m = map(int, input().split())

a = []

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

 a.append(list(map(int, input().split())))

Отметим, что для считывания нам не понадобилось значение переменной m, равное количеству столбцов массива. Но в других языках программирования это число используется при считывании, поэтому в задачах дают оба размера таблицы.

Можно организовать считывание при помощи генератора следующим образом:

n, m = map(int, input().split())

a = [list(map(int, input().split())) **for** i **in** range(n)]

**Вложенные генераторы**

**Вложенные генераторы**

Научимся использовать вложенные генераторы для заполнения двумерных массивов по формулам, которые зависят от номера строки и номера столбца элемента.

Мы уже встречались с генератором, который позволяет заполнить двумерный массив нулями:

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

Можно модифицировать этот генератор таким образом, чтобы внутренний список, состоящий из mm нулей, тоже создавался через генератор:

a = [[0 **for** j **in** range(m)] **for** i **in** range(n)]

Таким образом, мы получили два генератора, один из которых вложен в другой. Теперь, если в теле вложенного генератора вместо числа 00 записать какую-то формулу, зависящую от индексов ii и jj, то мы получим способ нетривиально заполнить наш двумерный массив. Ниже мы рассмотрим несколько примеров такого заполнения.

**Пример 1**

a = [[j **for** j **in** range(m)] **for** i **in** range(n)]

Для n=3n=3, m=4m=4 этот генератор заполнит двумерный массив следующим образом:

0 1 2 3

0 1 2 3

0 1 2 3

**Пример 2**

a = [[i **for** j **in** range(m)] **for** i **in** range(n)]

Для n=3n=3, m=4m=4 этот генератор заполнит двумерный массив так:

0 0 0 0

1 1 1 1

2 2 2 2

**Пример 3**

a = [[i + j **for** j **in** range(m)] **for** i **in** range(n)]

Для n=3n=3, m=4m=4 этот генератор заполнит двумерный массив следующим образом:

0 1 2 3

1 2 3 4

2 3 4 5

**Пример 4**

a = [[(int)(i == j) **for** j **in** range(m)] **for** i **in** range(n)]

В этом генераторе выражение (i == j) равно True, если элемент расположен на диагонали, и равно False в противном случае. Если преобразовать это выражение к целочисленному типу (int)(i == j), то в сгенерированной таблице на диагонали будут стоять единицы, а в остальных ячейках будут стоять нули.

Для n=3n=3, m=4m=4 этот генератор заполнит двумерный массив следующим образом:

1 0 0 0

0 1 0 0

0 0 1 0

**Маршруты на клетчатом поле**

**Маршруты на клетчатом поле**

Рассмотрим следующую задачу. Дана прямоугольная доска из nn строк и mm столбцов. В левом верхнем углу этой доски находится фишка, которую необходимо переместить в правый нижний угол. За один ход фишку разрешается передвинуть на одну клетку вниз или одну клетку вправо. Необходимо определить, сколько существует различных маршрутов фишки, ведущих из левого верхнего в правый нижний угол.

Будем считать, что положение фишки задаётся парой чисел (i,j)(i,j), где ii — номер строки, а jj — номер столбца. Строки нумеруются сверху вниз от 00 до n−1n−1, а столбцы — слева направо от 00 до m−1m−1. Таким образом, первоначальное положение фишки — клетка (0,0)(0,0), а конечное — клетка (n−1,m−1)(n−1,m−1).

Пусть w(i,j)w(i,j) — количество маршрутов, ведущих в клетку (i,j)(i,j) из начальной клетки. Запишем рекуррентное соотношение. В клетку (i,j)(i,j) можно прийти двумя способами: из клетки (i,j−1)(i,j−1), расположенной слева, и из клетки (i−1,j)(i−1,j), расположенной сверху от данной. Поэтому количество маршрутов, ведущих в клетку (i,j)(i,j), равно сумме количеств маршрутов, ведущих в клетку слева и сверху от неё. Получили рекуррентное соотношение:

w(i,j)=w(i,j−1)+w(i−1,j)w(i,j)=w(i,j−1)+w(i−1,j)

Это соотношение верно при i>0i>0 и j>0j>0. Зададим начальные значения: если i=0i=0, то клетка расположена на верхнем краю доски и прийти в неё можно единственным способом — двигаясь только вправо, поэтому w(0,j)=1w(0,j)=1 для всех 0⩽j<m0⩽j<m. Аналогично, w(i,0)=1w(i,0)=1 для всех 0⩽i<n0⩽i<n.

Создадим массив ww для хранения значений функции, заполним первую строку и первый столбец единицами, а затем заполним все остальные элементы массива, используя рекуррентную формулу. Поскольку каждый элемент равен сумме значений, стоящих слева и сверху, заполнять массив ww будем по строкам сверху вниз, а каждую строку — слева направо.

В результате такого заполнения получим следующий массив (пример для n=4n=4, m=5m=5):

1 1 1 1 1

1 2 3 4 5

1 3 6 10 15

1 4 10 20 35

Код на языке Python, решающий эту задачу, будет выглядеть следующим образом:

n, m = map(int, input().split())

w = [[1] \* m **for** i **in** range(n)]

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

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

 w[i][j] = w[i][j - 1] + w[i - 1][j]

**print**(w[-1][-1])

**Маршруты на клетчатом поле с дополнительными ограничениями**

Теперь решим эту задачу с дополнительным ограничением — некоторые клетки таблицы запрещены для посещения фишкой. В этой задаче нам дополнительно даётся таблица tt размера nn на mm состоящая из 00 и 11. Если t(i,j)=0t(i,j)=0, то в клетку (i,j)(i,j) перемещать фишку запрещено. Гарантируется, что t(0,0)=1t(0,0)=1.

Будем решать эту задачу аналогично предыдущей. Изменение состоит в том, что есть клетки, в которые мы не можем перемещать фишку. Формально можно записать, что если t(i,j)=0t(i,j)=0, то w(i,j)=0w(i,j)=0. Если же t(i,j)=1t(i,j)=1, то w(i,j)=w(i,j−1)+w(i−1,j)w(i,j)=w(i,j−1)+w(i−1,j). Можно объединить эти два случая в одну формулу следующим образом:

w(i,j)=t(i,j)⋅(w(i,j−1)+w(i−1,j))w(i,j)=t(i,j)⋅(w(i,j−1)+w(i−1,j))

Также следует более внимательно заполнить первый столбец и первую строку таблицы ww, так как теперь не во всех клетках первого столбца и первой строки обязательно будут стоять единицы.

Код, решающий эту задачу, будет выглядеть следующим образом:

n, m = map(int, input().split())

t = [list(map(int, input().split())) **for** i **in** range(n)]

w = [[1] \* m **for** i **in** range(n)]

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

 w[i][0] = t[i][0] \* w[i - 1][0]

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

 w[0][j] = t[0][j] \* w[0][j - 1]

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

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

 w[i][j] = t[i][j] \* (w[i][j - 1] + w[i - 1][j])

**print**(w[-1][-1])