МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра вычислительной математики и математической физики

Курс «Основы алгоритмизации и программирования»

Андрей Петрович Мелехов

Лекция 3. Динамическое программирование

Ростов-на-Дону

2020

У многих рекурсивных функций есть недостаток: они требуют выполнения слишком большого числа операций и слишком большую вложенность рекурсивных вызовов. Часто решение можно найти за конечное время лишь для небольших размерностей. Или происходит переполнение стека. Можно оптимизировать.

Во многих таких задачах альтернативой становится другой метод решения: применение «динамического программирования». Обычно решение этим методом требует на порядки меньше времени и не использует стек.

Динамическое программирование – это раздел математики, который занимается решением оптимизационных задач, нахождения оптимальных решений, например, минимального расстояния, максимальной прибыли и т. п.

Термин «Динамическое программирование» и сам метод впервые в 40-х годах 20-го века ввел Ричард Беллман. Подробнее об этом методе можно узнать из его книг.

Общей характеристикой таких задач является то, что решения подзадач могут использоваться при решении всей задачи. Этот метод рассчитан на применение компьютера при решении задач.

Мы не будем здесь изучать этот математический раздел, мы на примерах рассмотрим применение идеи разбиения задачи на подзадачи. Основной технический прием – разбивать на подзадачи меньшей размерности, запоминать их решения и постепенно увеличивать размерность до требуемой в задаче.

Замечание. Рекурсия – это тоже разбиение задачи на подзадачи, т. е. переход от большей задачи к меньшим подзадачам. Но в динамическом программировании направление решения обратное: сначала рассматриваются простейшие задачи, а потом увеличиваем размерность задачи.

Рассмотрим простую задачу.

Пример 1. Числа Фибоначчи

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

Мы рассматривали рекурсивный алгоритм вычисления чисел Фибоначчи. Я говорил, что на самом деле их не надо вычислять рекурсивно – это простой пример демонстрирующий рекурсию. Если мы запустим программу при $n = 40$,

она будет долго решать эту простую задачу, а для $n = 50$ можем и не дождаться ответа.

Дело в том, что при рекурсии мы много раз вычисляем одни и те же значения. Если каждое значение F_n вычислять только один раз, то программа будет работать гораздо быстрее.

Введем список, в котором будем хранить значения чисел Фибоначчи и перепишем рекурсию так:

```
# Список хранящий числа Фибоначчи:
n = 20
F = [0] * (n + 1)
```

```
def fibonacci(n):
   if F[n] == 0: # если F[n] еще не вычислено
      if n == 0 or n == 1: # конечная ветвь
         F[n] = 1
      else: # рекурсивная ветвь
         F[n] = fibonacci(n-1) + fibonacci(n-2)
   return F[n]
# вызов функции:
print("F = ", fibonacci(n))
print(F)
```

Результат:

F = 10946

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946]

Такая рекурсивная функция будет работать намного быстрее. Каждое число Фибоначчи вычисляется ровно один раз. В остальных случаях оно берется из списка. Здесь уже используется идея динамического программирования – хранить промежуточные данные. Такое решение можно назвать нисходящее динамическое программирование.

Но есть лучший алгоритм, использующий рекуррентное соотношение:

```
n = 20
F = [0] * (n + 1)
F[0] = 1;   F[1] = 1
for i in range(2, n + 1):
    F[i] = F[i - 1] + F[i - 2]
print(F)
```

Этот алгоритм относится к (восходящему) динамическому программированию: идем от простых задач к более сложным и запоминаем промежуточные данные.

Это очень простая задача. Если бы Вам сказали написать программу, вычисляющую числа Фибоначчи, скорее всего Вы бы сразу написали последний вариант.

Рассмотрим более сложную задачу.

Пример 2. Треугольник

Дан треугольник из чисел:

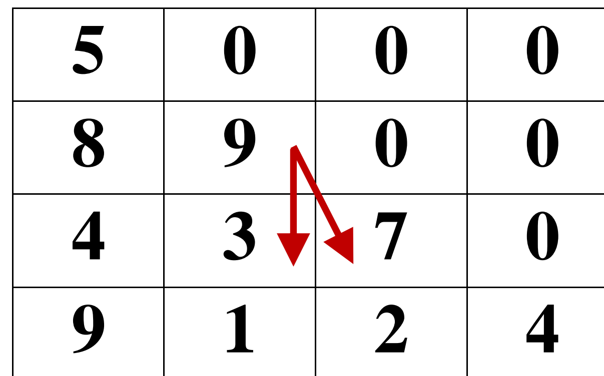
```
      5        – строка 1
    8   9      – строка 2
  4   3   7
9   1   2   4
     …
```

Строк может быть от 2 до 100. числа от 0 до 99.

Написать программу, которая находит наибольшую сумму на пути от вершины треугольника к его основанию. Каждый шаг на пути вниз может идти по диагонали влево или вправо.

Введем обозначения:

В квадратной матрице A размерности n на n будем хранить сами числа:



5	0	0	0
8	9	0	0
4	3	7	0
9	1	2	4

Будем использовать только нижнюю половину матрицы. Из $A[i][j]$ элемента путь может идти только в $A[i+1][j]$ и $A[i+1][j+1]$.

Введем еще вспомогательную матрицу S такой же размерности. В ячейке $S[i][j]$ будем хранить наибольшую сумму чисел на пути от вершины треугольника до позиции $A[i][j]$:

5	0	0	0
13	14	0	0
17	17	21	0
26	18	23	25

Метод будет состоять в постепенном решении: сначала для 1-й строки, потом для 2-х строк и т. д. увеличивая количество строк. Промежуточные результаты будем сохранять в матрице S.

Для 1-й строки: Очевидно $S[1][1] = A[1][1] = 5$.

Для 2-й строки: $S[2][1] = S[1][1] + A[2][1] = 5 + 8 = 13$,

$S[2][2] = S[1][1] + A[2][2] = 5 + 9 = 14$.

Матрица *A*:

5	0	0	0
8	9	0	0
4	3	7	0
9	1	2	4

Матрица *S*:

5	0	0	0
13	14	0	0
17	17	21	0
26	18	23	25

Для 3-й строки:

$S[3][1] = S[2][1] + A[3][1] = 13 + 4 = 17,$

$S[3][2] = \max(S[2][1], S[2][2]) + A[3][2] = 14 + 3 = 17,$

$S[3][3] = S[3][2] + A[3][3] = 14 + 7 = 21$

И т. д. Видно, что формула для крайних элементов и средних различна.

Если Мы расширим матрицу *S* нулевым столбцом и справа по диагонали добавим нули, то можно записать общую формулу:

Матрица *S*:

0	5	0		
0	13	14	0	
0	17	17	21	0
0	26	18	23	25

$$\begin{cases} S[1][1] = A[1][1], \\ S[i][j] = \max(S[i-1][j-1], S[i-1][j]) + A[i][j], \end{cases}$$

для строк i от 2 до n и столбцов j от 1 до i.

Ответ всей задачи – это максимум в последней строке: 26.

Программа

```
import random
random.seed()
n = 5 # количество строк в треугольнике
# Создадим случайную матрицу A размерности n:
A = []
for i in range(n):
  L = []
  for j in range(i + 1):
    L.append(random.randint(0, 10))
  A.append(L)
print(A) # Печать A
```

В Python создаем двумерный список:

[[1], [7, 10], [3, 3, 4], [2, 8, 3, 7], [7, 10, 3, 6, 5]]

Матрицу *S* будем заполнять в процессе решения.

```
# Алгоритм:
# Заполняем первую строку (с двумя нулями):
S = [[0, A[0][0], 0]]
for i in range(1, n):
 # Заполняем нулями i-ю строку:
  S.append([0] * (i + 3))
  for j in range(0, i + 1):
   # Индексы пришлось сместить из-за списков:
    S[i][j+1] = max(S[i-1][j], S[i-1][j+1])
                + A[i][j]
# Выводим список S:
print(S)
# Находим max в последней строке:
print('Ответ:', max(S[n - 1]))
```

Замечание. **Из-за того что используем списки, а не матрицы, нумерацию пришлось сместить в программе.**

Матрицы A и S:

A	S
[1]	[0, 1, 0]
[7, 10]	[0, 8, 11, 0]
[3, 3, 4]	[0, 11, 14, 15, 0]
[2, 8, 3, 7]	[0, 13, 22, 18, 22, 0]
[7, 10, 3, 6, 5]	[0, 20, 32, 25, 28, 27, 0]

Ответ: 32

Задания

1. Как найти min?

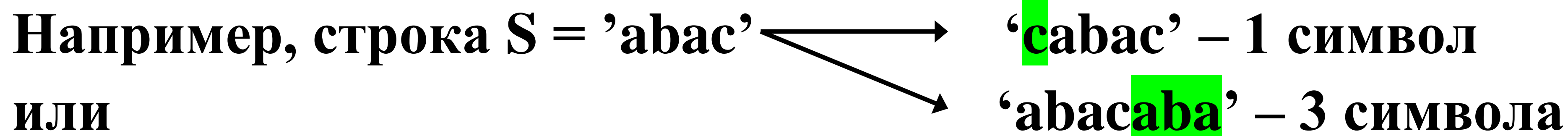
2. Вывести еще и оптимальный путь.

3. Что в этой программе можно оптимизировать?

4*. Решить эту задачу используя рекурсию. Если использовать просто рекурсию, то даже для небольших n будет долго решаться. Лучше рекурсия плюс матрица S.

5*. Черепашка. Дана квадратная доска. В каждой клетке расставлены натуральные числа. Черепашка должна проползти из левого верхнего угла в правый нижний угол и набрать max-ю (или min-ю) сумму. Причем она может передвигаться только на клетку вниз или вправо.

<u>Пример 3. Палиндром.</u> Написать программу, которая по заданной строке определяет минимальное количество символов, которые необходимо вставить в строку для образования палиндрома.



Решать тоже будем переходом от маленьких строк к большим, вплоть до строки S.

Введем матрицу R размерности n × n, где n – длина строки S. R[i][j] – минимальное количество букв, которые необходимо вставить в подстроку S(i, j) – с i-го символа по j-й символ строки S, чтобы получить из нее палиндром.

<u>Замечание.</u> В Python нумерация строк начинается с 0 до n – 1. И вместо матрицы тоже будем использовать двумерный список с нумерацией от 0 до n – 1.

R =

0			
0	0		
0	0	0	
0	0	0	0

Для $i > j$ элементы матрицы $R[i][j] = 0$ – нет таких строк. Для $i = j$ также $R[i][j] = 0$ – один символ является палиндромом.

Заполнять будем, увеличивая количество символов – по диагоналям матрицы R. Результат – это число $R[0][n-1]$.

Как перейти от меньшей строки к большей?

Пусть есть строка S(i, j) и все ее меньшие строки уже рассмотрены.

Например (1) S = ’abaa’ или (2) S = ‘abac’.

Если, как в (1), у нас крайние символы равны S[i] = S[j], то мы от этой строки переходим к меньшей: S = ’abaa’. Т. е.

$$\boldsymbol{R[\,i\,][\,j\,] \;=\; R[\,i\;+\;1][j - \;1]}$$

Если, как в (2), крайние символы разные S[i] ≠ S[j], то к этой строке надо добавить один символ

1) либо слева: S = ‘abac’ → ‘cabac’

2) либо справа: S = ‘abac’ → ‘abaca’

Т. е. тоже переходим к рассмотрению меньших строк и из них выберем с минимальным значением.

Т. е.

$$R[i][j] = \min(R[i][j-1], R[i+1][j]) + 1.$$

Для примера S = ‘abac’ получим:

1) либо слева: S = ‘abac’ → ‘cabac’, и $R[0][2] = 0$;

2) либо справа: S = ‘abac’ → ‘abaca’ и $R[1][3] = 2$;

$$R[0][3] = \min(R[0][2], R[1][3]) + 1 = \min(0, 2) + 1 = 1.$$

Рекуррентная формула:

$$R[i][j] = \begin{cases} 0, & \text{при } i \geq j \\ R[i+1][j-1], & \text{при } S[i] == S[j], (i > j) \\ \min(R[i][j-1], R[i+1][j]) + 1, & \text{при } S[i] \mathrel{!=} S[j], (i > j) \end{cases}$$

Замечание. Используется идея, что если крайние символы равны, то мы можем рассмотреть меньшую строку без них. А если они не равны, то либо слева, либо справа придется добавить один символ и перейти к рассмотрению меньшей строки.

Для нашего примера S = ‘abac’.

R =

0	1	0	1
0	0	1	2
0	0	0	1
0	0	0	0

S(0, 1) = ‘ab’, R[0][1] = 1;

S(1, 2) = ‘ba’, R[1][2] = 1;

S(2, 3) = ‘ac’, R[2][3] = 1;

S(0, 2) = ‘aba’, R[0][2] = R[1][1] = 0;

S(1, 3) = ‘bac’, R[1][3] = min(R[1][2], R[2][3]) + 1 = 2;

S(0, 3) = ‘abac’, R[0][3] = min(R[0][2], R[1][3]) + 1 = 1.

Ответ: 1. Добавить нужно 1 символ: S = ‘cabac’.

Программа

```
S = input('S = ')
n = len(S) # длина строки (размер матрицы)
# Создадим двумерный список n*n из 0-й:
R = [[0] * n  for i in range(n)]
# Цикл по диагоналям (d - номер диагонали):
for d in range(1, n):
  # Цикл по строкам (i - номер строки):
  for i in range(n - d):
  # j - номер столбца:
  j = i + d
  # заполняем R[i, j] элемент:
    if S[i] == S[j]: R[i][j] = R[i + 1][j - 1]
    else: R[i][j] = min(R[i][j - 1],
                         R[i + 1][j]) + 1
```

```
# Печатаем матрицу R:
for i in range(n):
  for j in range(n):
    print('%2d' %R[i][j], end = '  ')
  print()

# Печатаем ответ:
print('Ответ: ', R[0][n - 1])
```

Результат работы программы:

```
S = abac
0 1 0 1
0 0 1 2
0 0 0 1
0 0 0 0
Ответ: 1
```

Результат работы программы:

```
S = abccba
0 1 2 2 1 0
0 0 1 1 0 1
0 0 0 0 1 2
0 0 0 0 1 2
0 0 0 0 0 1
0 0 0 0 0 0
Ответ: 0
```

Результат работы программы:

```
S = ababc
0 1 0 1 2
0 0 1 0 1
0 0 0 1 2
0 0 0 0 1
0 0 0 0 0
Ответ: 2
```

Задания

1*. Выдавать еще какие символы и куда нужно вставить?

2*. Решить эту задачу используя рекурсию. Лучше рекурсия плюс матрица *R*.

3*. Скобки. Строка состоит из символов-скобок: ( ) [ ] { }. Определить, какое минимальное количество скобок нужно добавить к ней, чтобы она стала правильным скобочным выражением.