

# Алгоритмизация и программирование. Язык Python

§ 35. Целочисленные алгоритмы

§ 36. Структуры

§ 37. Словари

§ 38. Стек, очередь, дек

§ 39. Деревья

§ 40. Графы

§ 41. Динамическое программирование

# Алгоритмизация и программирование. Язык Python

## **§ 35. Целочисленные алгоритмы**

# Решето Эратосфена



Эратосфен Киренский  
(Eratosthenes, Ερατοσθένης)  
(ок. 275-194 до н.э.)

## Алгоритм:

- 1) начать с  $k = 2$
- 2) «выколоть» все числа через  $k$ , начиная с  $k \cdot k$
- 3) перейти к следующему «невыколотому»  $k$
- 4) если  $k \cdot k \leq N$ , то перейти к шагу 2
- 5) напечатать все числа, оставшиеся «невыколотыми»

Новая версия – [решето Аткина](#).

**?** Как улучшить?

**+** высокая скорость, количество операций

$$O((N \cdot \log N) \cdot \log \log N)$$

**-** нужно хранить в памяти все числа от 1 до  $N$

# Решето Эратосфена

Задача. Вывести все простые числа от 2 до  $N$ .

Массив (сначала все не вычеркнуты):

```
N = 100
```

```
A = [True] * (N+1)
```

выделяем на 1  
элемент больше,  
чтобы начать с A[1]

Вычёркивание непростых:

```
k = 2
```

```
while k*k <= N:
```

```
    if A[k]: # если k не вычеркнуто
```

```
        i = k*k # начать с k*k
```

```
        while i <= N: # вычеркнуть кратные k
```

```
            A[i] = False
```

```
            i += k
```

```
        k += 1
```

# Решето Эратосфена

или так:

```
from math import sqrt

for k in range(2, int(sqrt(N))+1):
    if A[k]: # если k не вычеркнуто
        for i in range(k*k, N, k):
            A[i] = False
```

все кратные k

# Решето Эратосфена

Вывод результата:

```
for i in range(2, N+1):  
    if A[i]:  
        print ( i )
```

или так:

```
P = [ i for i in range(2, N+1)  
      if A[i] ]  
print ( P )
```

выбираем те, которые  
не вычеркнуты

# «Длинные» числа

Ключи для шифрования:  $\geq 256$  битов.

Целочисленные типы данных *обычно*  $\leq 64$  битов.



Как хранить?

**Длинное число** – это число, которое не помещается в переменную одного из стандартных типов данных языка программирования.

«**Длинная арифметика**» – алгоритмы для работы с длинными числами.

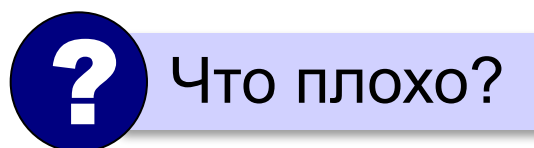


В Python длинная арифметика встроенная!

# «Длинные» числа

**A = 12345678**

	0	1	2	3	4	5	6	7
<b>A</b>	1	2	3	4	5	6	7	8



- неудобно вычислять (с младшего разряда!)
- неэкономное расходование памяти (одна цифра в ячейке)

**Обратный порядок элементов:**

	7	6	5	4	3	2	1	0
<b>A</b>	1	2	3	4	5	6	7	8



# «Длинные» числа

Упаковка элементов:  $A = 12345678$

	2	1	0
A	12	345	678

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0$$



На что похоже?

система счисления с  
основанием 1000!

Если 4 байтовая ячейка:

от  $-2^{31} = -2\,147\,483\,648$  до  $2^{31} - 1 = 2\,147\,483\,647$ .



Какие основания можно использовать?

должны помещаться все  
промежуточные результаты!

# Вычисление факториала

Задача 1. Вычислить точно значение факториала

$$100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100$$



Как оценить количество цифр?

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100 <$$

201 цифра

основание 10000000

6 цифр в ячейке  $\Rightarrow$  34 ячейки

Основной алгоритм:

длинное  
число

```
{A} = 1
for k in range(2, 101):
    {A} *= k
```

# Вычисление факториала

основание  $d = 1\,000\,000$

$[A] = 12345678901734567$

	3	2	1	0	
A	0	12345	678901	734567	*3

$734\ 567 \cdot 3 = 2\ 203\ 701$

остаётся в  $A[0]$

$r = \text{перенос в } A[1]$

**?** Как найти перенос?

```
s = A[0] * k
A[0] = s % d
r = s // d
```

**?** Что изменится для  $A[1]$ ?

$s = A[1] * k + r$

# Вычисление факториала

Умножение «длинного» числа на  $k$ :

```
r = 0
for i in range(len(A)) :
    s = A[i]*k + r
    A[i] = s % d
    r = s // d
if r > 0:
    A.append ( r )
```

все разряды

число  
удлиняется

Вычисление  $100!$ :

```
for k in range(2, 101) :
    {A} *= k
```

# Вывод длинного числа

	3	2	1	0
A	0	1	2	3



Какое число?

A = 1000002000003

- вывести старший ненулевой разряд

```
h = len ( A ) - 1  
print ( A[h] , end = "" )
```

- вывести все следующие разряды, добавляя лидирующие нули до 6 цифр

```
for i in range (h-1, -1, -1) :  
    print ( "{:06d}".format(A[i]) , end = "" )
```

ДОПОЛНИТЬ  
НУЛЯМИ

В 6  
ПОЗИЦИЯХ

# Квадратный корень

**Задача.** Извлечь квадратный корень из «длинного» целого числа; если это не целое число, найти корень с округлением «вниз» (к меньшему значению).

**Метод Герона:**  $x^* = \sqrt{a}$

$$x_0 = a \longrightarrow x_i = \frac{1}{2} \left( x_{i-1} + \frac{a}{x_{i-1}} \right)$$



Начальное приближение – любое  $> 0$ !

$$a = 16$$

$$x_0 = 16 \longrightarrow x_1 = 0,5 \cdot (16 + 16/16) = 8,5$$

$$x_2 = 0,5 \cdot (8,5 + 16/8,5) \approx 5,19$$

$$x_3 = 0,5 \cdot (5,19 + 16/5,19) \approx 4,14$$

...

# Квадратный корень

Метод Герона в целых числах:

```
x = (x + a // x) // 2
```

или так:

```
x = (x*x + a) // (2*x)
```

$$x_i = \frac{1}{2} \left( x_{i-1} + \frac{a}{x_{i-1}} \right)$$

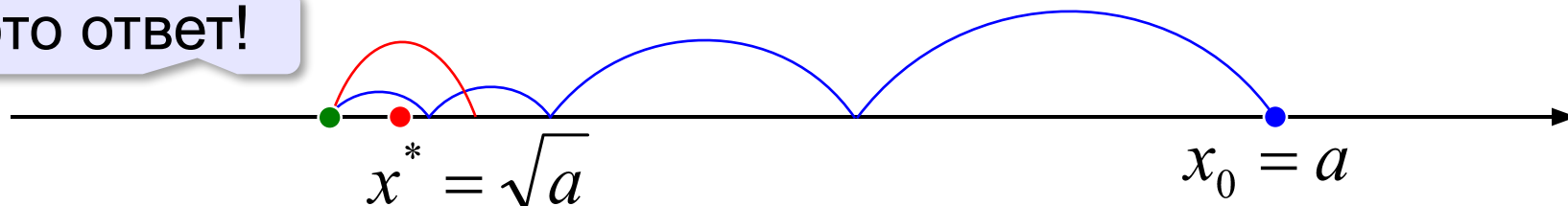
$$x_i = \frac{x_{i-1}^2 + a}{2 \cdot x_{i-1}}$$

только целые  
числа!



Когда остановиться?

это ответ!



Если следующее приближение больше  
предыдущего, то стоп!

# Квадратный корень

## Метод Герона в целых числах:

```
def isqrt(a):  
    x = a    # начальное приближение  
    while True:  
        x1 = (x*x + a) // (2*x)  
        if x1 >= x:  
            return x  
        x = x1
```

следующее  
приближение

вернуть  
результат



# Алгоритмизация и программирование. Язык Python

## **§ 36. Структуры**

# Зачем нужны структуры?

## Книги в библиотеках:

- автор
- название
- количество экземпляров

символьные строки

целое число



Как хранить данные?

## Несколько массивов:

```
N = 100
```

```
authors = [""]*N
```

```
titles = [""]*N
```

```
count = [0]*N
```

```
...
```

неудобно работать  
(сортировать и т.д.),  
ошибки

**Задача:** объединить разнотипные данные в один блок.

# Структуры

**Структура** – это тип данных, который может включать в себя несколько **полей** – элементов разных типов (в том числе и другие структуры).

НОВЫЙ ТИП (КЛАСС)  
ДАННЫХ

```
class TBook:  
    pass
```

название типа  
данных

«пустой»  
оператор

```
class TBook:  
    author = ""  
    title = ""  
    count = 0
```

эти поля будут у  
всех структур  
класса TBook

# Работа со структурами

## Создание:

```
В = TBook ( )
```

## Заполнение:

```
В.author = "Пушкин А.С."
```

```
В.title = "Полтава"
```

```
В.count = 1
```

создание полей

точечная запись



В Python не нужно заранее объявлять поля!

## Ввод с клавиатуры:

```
В.author = input ( )
```

```
В.title = input ( )
```

```
В.count = int ( input ( ) )
```

# Работа со структурами

## Обработка:

```
B.author = "Пушкин А.С."  
fam = B.author.split()[0]    # фамилия  
print ( fam )  
B.count -= 1                 # одну книгу взяли  
if B.count == 0:  
    print ( "Этих книг больше нет!" )
```

## Вывод на экран:

```
print ( B.author, B.title + ". ",  
        B.count, "шт." )
```

# Массив структур

## Создание:

~~Books = [TBook()] \* 100~~



Что плохо?

```
Books = []  
for i in range(100):  
    Books.append(TBook())
```

## Изменение полей:

```
Books[5].author = "Пушкин А.С."  
Books[5].title = "Полтава"  
Books[5].count = 1
```

# Запись структур в CSV-файлы

**Текстовые файлы** – это файлы, в которых данные разбиты на строки и содержат только символы, присутствующие в тексте (английском, русском).

**CSV** = Comma Separated Values

Пушкин А.С. , Полтава , 12  
Лермонтов М.Ю. , Мцыри , 8

разделитель

```
F = open( "library.csv", "w" )  
for b in Books:  
    F.write( b.author + "," )  
    F.write( b.title + "," )  
    F.write( str(b.count) + "\n" )  
F.close()
```

"w" – запись  
"r" – чтение  
"a" – добавление

# Чтение структур из CSV-файлов

Пушкин А.С. , Полтава , 12  
Лермонтов М.Ю. , Мцыри , 8

```
Books = []  
F = open( "library.csv" )  
for line in F:  
    line = line.rstrip()  
    data = line.split( "," )  
    b = TBook()  
    b.author = data[0]  
    b.title = data[1]  
    b.count = int(data[2])  
    Books.append( b )  
F.close()
```

по умолчанию  
"r" – чтение

убрать "\n" в  
конце строки

[ "Пушкин А.С." , "Полтава" , "12" ]

data[0]

data[1]

data[2]



# Сортировка структур

**Ключ** — это поле, по которому выполняется сортировка.

**Ключ** – фамилия автора:

```
# В – массив структур TBook
N = len(B)
for i in range(0, N-1):
    for j in range(N-2, i-1, -1):
        if B[j].author > B[j+1].author:
            B[j], B[j+1] = B[j+1], B[j]
```



Какой метод?

# Сортировка структур (в стиле Python)

Ключ – фамилия автора:

```
def getAuthor ( B ) :  
    return B.author
```

```
Books.sort ( key = getAuthor )
```

или так:

```
Books.sort ( key =  
    lambda x: x.author )
```

лямбда-функция

не меняя Books:

```
sortedBooks = sorted ( Book ,  
    key = lambda x: x.author )
```

# Сортировка структур

---

По убыванию (обратный порядок) :

```
Books.sort ( reverse = True ,  
             key = lambda x: x.author )
```

Сначала по автору, потом – по названию:

```
Books.sort ( key =  
            lambda x: (x.author, x.title) )
```

Сначала по количеству (по убыванию), потом – по автору:

```
Books.sort ( key =  
            lambda x: (-x.count, x.author) )
```

# Запись структуры в двоичный файл

```
import pickle
В = TBook ()
В.author = "Тургенев И.С."
В.title = "Муму"
В.count = 2

F = open ( "books.dat", "wb" )
pickle.dump ( В, F )
F.close ()
```

*binary*, ДВОИЧНЫЙ

"wb" – запись  
"rb" – чтение  
"ab" – добавление

# Запись массива структур

---

```
Books = []  
for i in range(10):  
    Books.append( TBook() )
```

Сразу все:

```
pickle.dump( Books, F )
```

По одной структуре:

```
for B in Books:  
    pickle.dump( B, F )
```

файл, открытый  
на запись

# Чтение структур из файла

---

## Одна структура:

```
F = open ( "books.dat", "rb" )
B = pickle.load ( F )
print ( B.author, B.title,
        B.count, sep = ", " )
F.close ()
```

## Массив структур целиком:

```
Books = pickle.load ( F )
```

## Массив из N структур по одной:

```
for i in range (N) :
    Books[i] = pickle.load ( F )
```

# Чтение структур из файла

Число структур неизвестно:

```
Books = []  
while True:  
    try:  
        B = pickle.load ( F )  
        Books.append ( B )  
    except:  
        break
```

**try** – попытаться выполнить следующие операторы

**except** – что делать в случае ошибки (*исключения, аварийной ситуации*)

**Исключение** — это аварийная ситуация, которая делает дальнейшие вычисления по основному алгоритму невозможными или бессмысленными.

# Алгоритмизация и программирование. Язык Python

## **§ 37. Словари**



# Что такое словарь?

**Задача.** В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Построить **алфавитно-частотный словарь**: список слов в алфавитном порядке, справа от каждого слова должно быть указано, сколько раз оно встречается в исходном файле.



Какая структура данных нужна?

```
print ( D["бегемот"] )
```

ключ → значение  
(отображение, *map*)

поиск не по индексу,  
а по слову (*ключу*)

**Словарь** – это неупорядоченный набор элементов, в котором доступ к элементу выполняется по ключу.

# Алгоритм (псевдокод)

ключ → значение

СЛОВО

СЧЁТЧИК



Как выбрать ключ и значение?

```
создать пустой словарь
while есть слова в файле:
    прочитать очередное слово
    if слово есть в словаре:
        увеличить на 1 счётчик
        для этого слова
    else:
        добавить слово в словарь
        записать 1 в счётчик слова
```

# Работа со словарями в Python

## Создание:

```
D = { }    # пустой словарь
```

```
D = { "бегемот" : 0, "пароход" : 2 }
```

## Добавление (изменение) элемента:

```
D [ "самолёт" ] = 1
```



Создаётся новый элемент!

```
D [ "самолёт" ] += 1
```

ошибка, если  
ключа нет



Нужно проверить, есть ли элемент!

# Работа со словарями в Python

## Изменение с проверкой:

```
if "самолёт" in D:  
    D["самолёт"] += 1  
else:  
    D["самолёт"] = 1
```

## или так:

```
D["самолёт"] = D.get( "самолёт", 0 ) + 1
```

получить значение  
по ключу

значение по  
умолчанию (если  
ключа нет)

# Основной цикл

создать пустой  
словарь

```
D = { }
```

```
F = open ( "input.txt" )
```

```
while True:
```

```
    word = F.readline() .strip()
```

```
    if not word:
```

```
        break
```

```
    D[word] = D.get ( word, 0 ) + 1
```

```
F.close()
```

прочитать  
строку, убрать  
"**\n**" в конце

кончились данные – выход

увеличить  
счётчик слова

# Вывод результата

Получить массив всех ключей:

```
allKeys = D.keys()
```

отсортировать ключи:

```
sortKeys = sorted(D.keys())
```

или так:

```
sortKeys = sorted(D)
```

Вывод результата:

```
F = open ( "output.txt", "w" )  
for k in sorted(D):  
    F.write ( "{}: {}".format(k, D[k]) )  
F.close()
```

пароход: 12

самолёт: 20

# Ещё о словарях

---

## Перебор значений:

```
for i in D.values():  
    print ( i )
```

## Перебор ключей и значений:

```
for k, v in D.items():  
    print ( k, "->", v )
```

список пар  
(ключ, значение)

# Словарь и массив пар

## Массив (список) пар «ключ-значение»:

```
A = list(D.items())
```

список пар  
(ключ, значение)

```
D = {"бам": 2, "там": 3}
```

A[0]    **кортеж**    A[1]

```
A = [ ("бам", 2), ("там", 3) ]
```

A[0][0]    A[0][1]

## Сортировка:

```
for i in range(N-1):  
    for j in range(N-2, i-1, -1):  
        if A[j][1] < A[j+1][1]:  
            A[j], A[j+1] = A[j+1], A[j]
```

по значению!



# Сортировка по одному значению

```
A.sort()
```

по ключам, если ключи  
равны – по значениям

```
A.sort( key = lambda x: x[0] )
```

по ключам

по значениям

```
A.sort( key = lambda x: x[1] )
```

В обратном порядке:

только для чисел!

```
A.sort( key = lambda x: -x[1] )
```

```
A.sort( key = lambda x: x[1],  
        reverse = True )
```

работает всегда!

# Сортировка по сложному критерию

```
A.sort( key = lambda x: (x[1], x[0]) )
```

по значениям, если они  
равны – по ключам

```
A.sort( key = lambda x: (-x[1], x[0]) )
```

по убыванию значений, если  
они равны – по ключам

# Словарь и массив пар

---

## Вывод массива пар

```
for x in A:  
    print( x[0], ":", x[1], sep="")
```

или так

```
for x in A:  
    print( "{}: {}".format(x[0], x[1]) )
```

# Алгоритмизация и программирование. Язык Python

## **§ 38. Стек, дек, очередь**

# Что такое стек?

**Стек** (англ. *stack* – стопка) – это линейный список, в котором элементы добавляются и удаляются только с одного конца («последним пришел – первым ушел»).

**LIFO** = *Last In – First Out*.



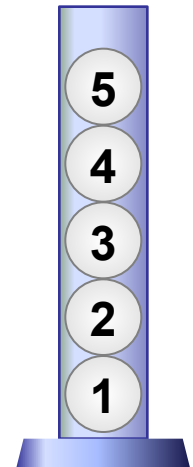
## Системный стек:

- адреса возврата из подпрограмм
- передача аргументов подпрограмм
- хранение локальных переменных

# Реверс массива

**Задача.** В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.

```
while файл не пуст:  
    прочитать x  
    добавить x в стек
```



```
while стек не пуст:  
    вытолкнуть число из стека в x  
    записать x в файл
```

# Использование списка



Конец списка – вершина стека!

Создать стек:

```
stack = []
```

«Втолкнуть» **x** в стек:

```
stack.append ( x )
```

Снять элемент с вершины стек в **x**:

```
x = stack.pop ()
```

- удалить последний элемент
- вернуть удалённый элемент как результат функции

# Инверсия массива неизвестной длины

Чтение из файла:

```
F = open ( "input.txt" )
stack = []
while True:
    s = F.readline()
    if not s: break
    stack.append( int(s) )
F.close()
```

или так:

```
stack = []
for s in open( "input_arr.dat" ) :
    stack.append ( int(s) )
```



# Инверсия массива неизвестной длины

Запись в файл (в обратном порядке):

```
F = open ( "output.txt" , "w" )  
while len(stack) > 0:      while stack:  
    x = stack.pop()          
    F.write ( str(x) + "\n" )  
F.close()
```

без переменной x:

```
F = open ( "output.txt" , "w" )  
while stack:  
    F.write ( str(stack.pop()) + "\n" )  
F.close()
```

# Вычисление арифметических выражений



Как компьютер вычисляет арифметические выражения?

$(5+15) / (4+7-1)$  **инфиксная форма** (знак операции между данными)

1920 (Я. Лукашевич): **префиксная форма**  
(знак операции перед данными)

/ + 5 15 - + 4 7 1

/ 20 - + 4 7 1

/ 20 - 11 1

/ 20 10

2



не нужны скобки



первой стоит последняя операция (вычисляем с конца)

# Вычисление арифметических выражений

$$(5+15) / (4+7-1)$$

1950-е: **постфиксная форма**

(знак операции после данных)

5 15 + 4 7 + 1 - /

20 4 7 + 1 - /

20 11 1 - /

20 10 /

2



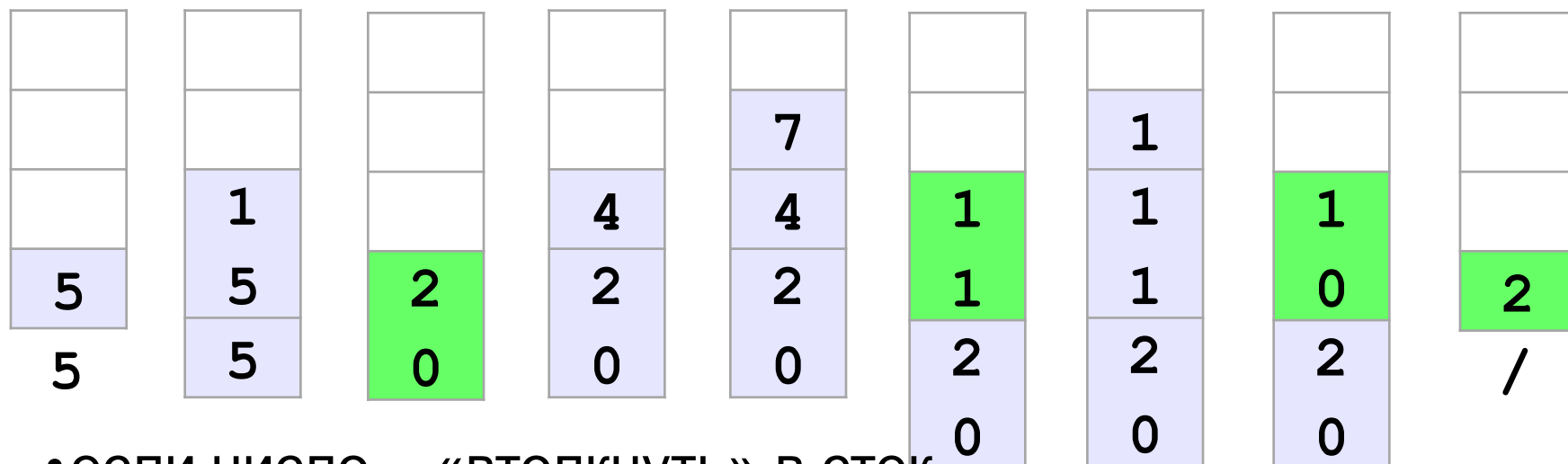
- не нужны скобки
- вычисляем с начала



Вычисляем с помощью стека!

# Использование стека

5 15 + 4 7 + 1 - /



- если число – «втолкнуть» в стек
- если операция – выполнить с верхними элементами стека



В стеке остается результат!

# Вычисление постфиксной формы

```
data = input().split()
stack = []
for x in data:
    if x in "+-*/": # если операция
        op2 = int(stack.pop())
        op1 = int(stack.pop())
        if x == "+": res = op1 + op2
        elif x == "-": res = op1 - op2
        elif x == "*": res = op1 * op2
        else: res = op1 // op2
        stack.append(res)
    else:
        stack.append(x)
print(stack[0]) # результат
```

разбить строку  
на элементы

взять 2 значения  
со стека

выполнить  
операцию

результат в стек

данные в стек

# Скобочные выражения

**Задача.** Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов: **()**, **[]** и **{}**.

Проверить, правильно ли расставлены скобки.

**()** **[{ () []}]** ✓ **[ ()]** ✗ **[ () }** ✗ **) (** ✗ **( [ ) ]** ✗

**Для одного типа скобок:**

		(	)	(	(	)	(	(	)	)	)
счётчик	0	1	0	1	2	1	2	3	2	1	0



Когда выражение правильное?

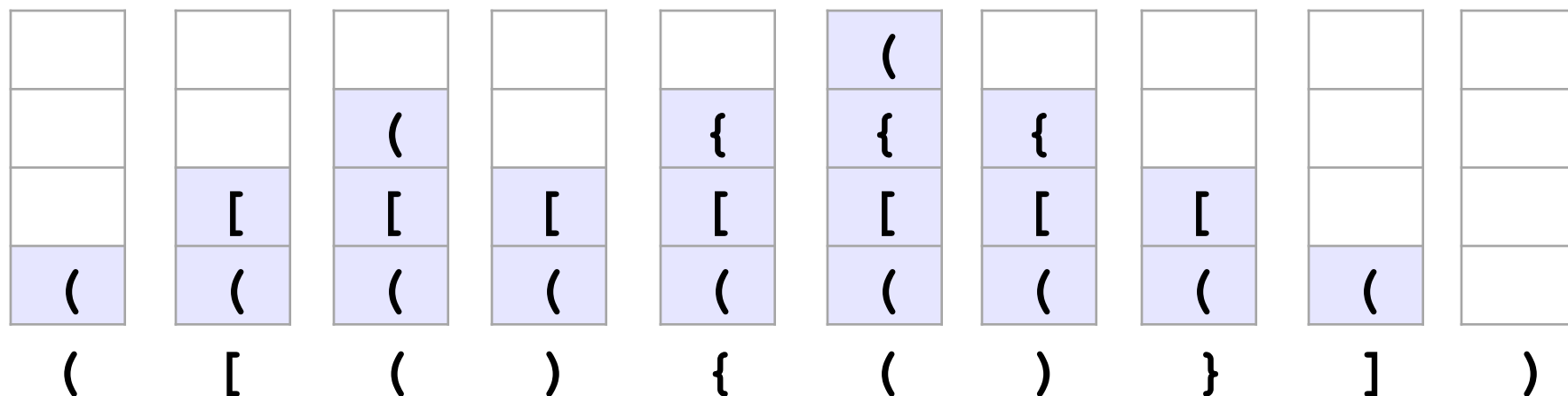
- счётчик всегда  $\geq 0$
- в конце счётчик = 0

**( { [ ] } )**



Для разных скобок не работает!

# Скобочные выражения (стек)



- если открывающая скобка – «втолкнуть» в стек
- если закрывающая скобка – снять парную со стека



Когда выражение правильное?

- когда встретили закрывающую скобку, на вершине стека лежит соответствующая открывающая
- в конце работы стек пуст

# Скобочные выражения (стек)

## Подготовка:

```
pairs = {"(": ")", "[": "]", "{": "}" }  
stack = []      # пустой стек  
err = False     # признак ошибки
```

## Вывод результата:

```
if not err:  
    print ( "Выражение правильное." )  
else:  
    print ( "Выражение неправильное." )
```



# Скобочные выражения (стек)

для всех СИМВОЛОВ  
строки *s*

```
for c in s:
    if c in pairs:
        stack.append( pairs[c] )
    elif c in pairs.values():
        if not stack or c != stack.pop():
            err = True
            break
```

парную скобку  
в стек

если закрывающая  
скобка...

стек  
пуст

не та  
скобка

```
if stack: err = True
```

если не все скобки  
закрыты



Что ещё?

# Что такое очередь?

**Очередь** — это линейный список, для которого введены две операции:

- добавление элемента в конец
- удаление первого элемента

**FIFO** = *Fist In – First Out*.

## Применение:

- очереди сообщений в операционных системах
- очереди запросов ввода и вывода
- очереди пакетов данных в маршрутизаторах
- ...



# Заливка области

**Задача.** Рисунок задан в виде матрицы  $A$ , в которой элемент  $A[y][x]$  определяет цвет пикселя на пересечении строки  $y$  и столбца  $x$ . Перекрасить в цвет **2** одноцветную область, начиная с пикселя  $(x_0, y_0)$ .

	0	1	2	3	4			0	1	2	3	4
0	0	1	0	1	1	$(1, 2)$ →	0	0	2	0	1	1
1	1	1	1	2	2		1	2	2	2	2	2
2	0	1	0	2	2		2	0	2	0	2	2
3	3	3	1	2	2		3	3	1	2	2	2
4	0	1	1	0	0		4	0	1	1	0	0

## Заливка: использование очереди

```
добавить в очередь точку  $(x_0, y_0)$   
color = цвет начальной точки  
  
while очередь не пуста:  
    взять из очереди точку  $(x, y)$   
    if  $A[y][x] == \text{color}$ :  
         $A[y][x] = \text{цвет заливки}$   
        добавить в очередь точку  $(x-1, y)$   
        добавить в очередь точку  $(x+1, y)$   
        добавить в очередь точку  $(x, y-1)$   
        добавить в очередь точку  $(x, y+1)$ 
```

# Заливка

## Подготовка:

```
YMAX = len (A)  
XMAX = len (A [0] )  
NEW_COLOR = 2
```

```
y0 = 0  
x0 = 1      # начать заливку отсюда  
color = A[y0][x0]  # цвет начальной точки
```

## Элементы очереди – координаты:

(x, y)  
кортеж

```
Q = []  
Q.append ( (x0, y0) )
```

добавить  
начальную точку



**Кортеж** – неизменяемая последовательность элементов (как список, но нельзя изменять)!

# Заливка (основной цикл)

пока очередь не пуста

```
while len(Q) > 0:
    x, y = Q.pop(0)
    if A[y][x] == color:
        A[y][x] = NEW_COLOR
        if x > 0: Q.append( (x-1, y) )
        if x < XMAX-1: Q.append( (x+1, y) )
        if y > 0: Q.append( (x, y-1) )
        if y < YMAX-1: Q.append( (x, y+1) )
```

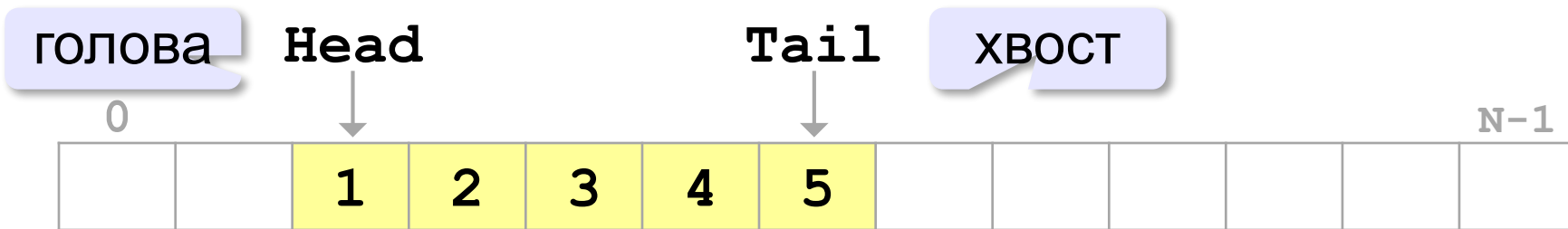
перекрасить

с проверкой  
выхода за  
границы

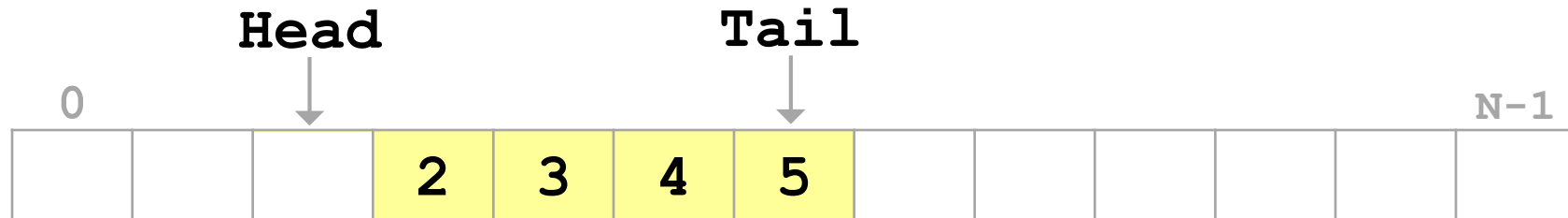


Что можно улучшить?

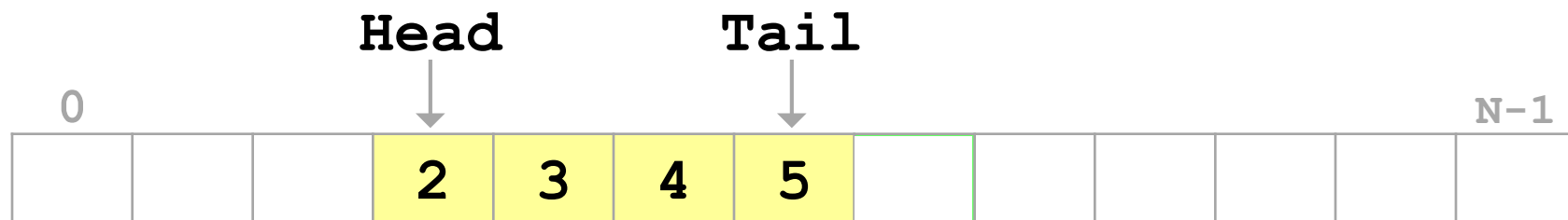
# Очередь: статический массив



## Удаление элемента:



## Добавление элемента:



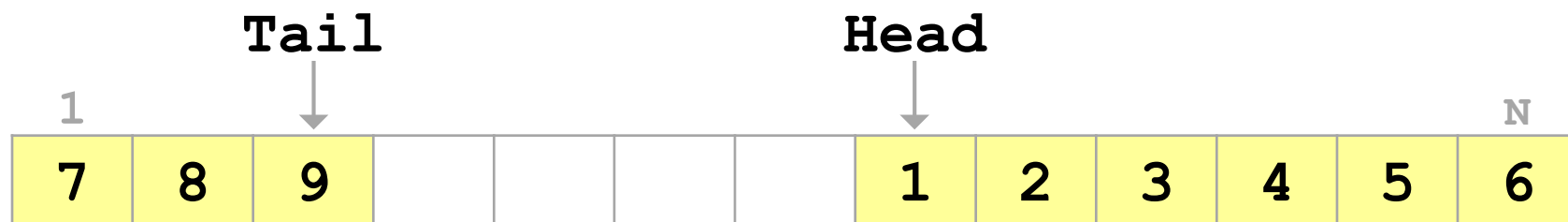
■ не двигаем элементы



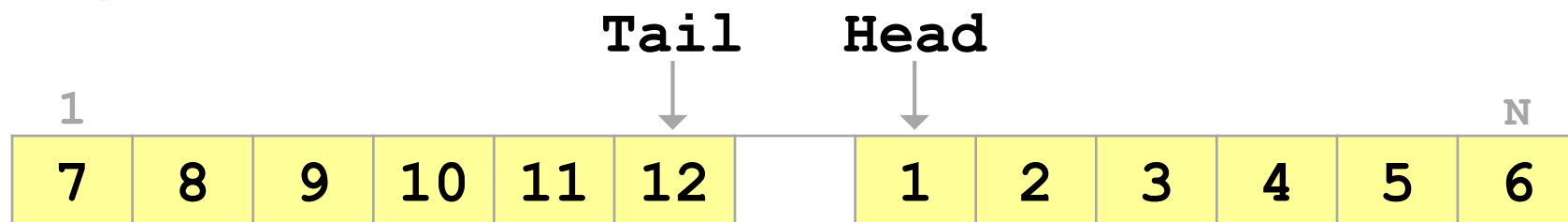
■ нужно знать размер

# Очередь: статический массив

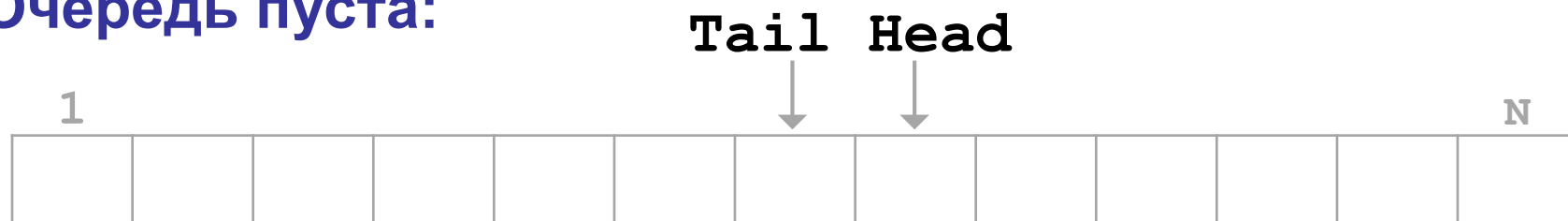
## Замыкание в кольцо:



## Очередь заполнена:



## Очередь пуста:

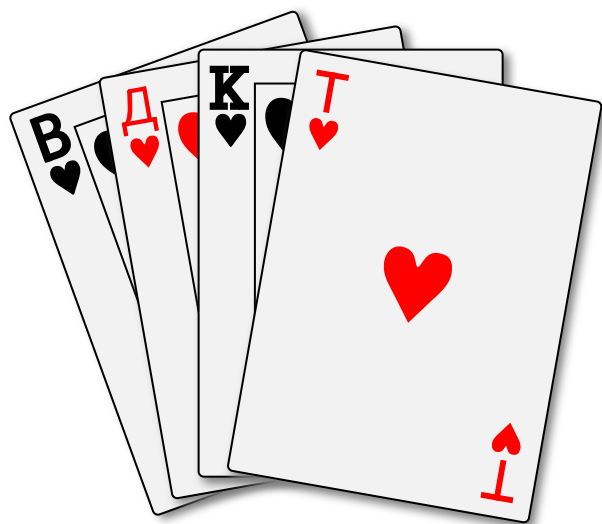


Вариант: хранить размер очереди в переменной!



# Что такое дек?

**Дек** – это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.



добавить в начало

удалить с начала

## Моделирование:

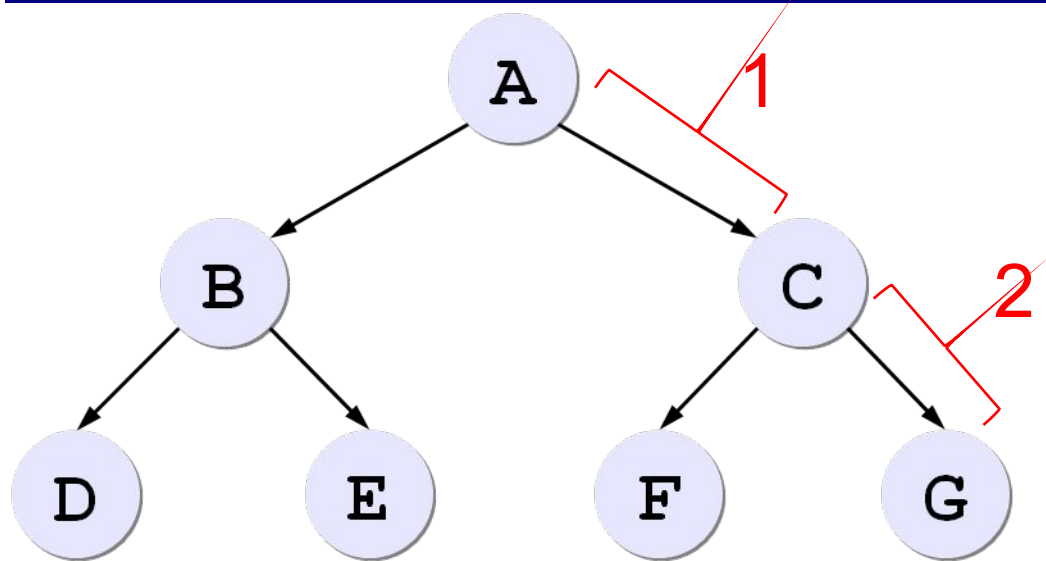
- массив (список) изменяющегося размера
- `collections.deque`

```
import collections
d = collections.deque()
d.append(1)
d.appendleft(0)
d.pop()
d.popleft()
```

# Алгоритмизация и программирование. Язык Python

## **§ 39. Деревья**

# Что такое дерево?



**Высота дерева** —  
наибольшее  
расстояние от  
корня до листа

**«Сыновья» A:** B, C.

**«Родитель» B:** A.

**«Потомки» A:** B, C, D, E, F, G. **«Предки» F:** A, C.

**Корень** — узел, не имеющий предков (A).

**Лист** — узел, не имеющий потомков (D, E, F, G).

# Рекурсивные определения

- 1) пустая структура – это **дерево**
- 2) дерево – это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев

## Двоичное (бинарное) дерево:

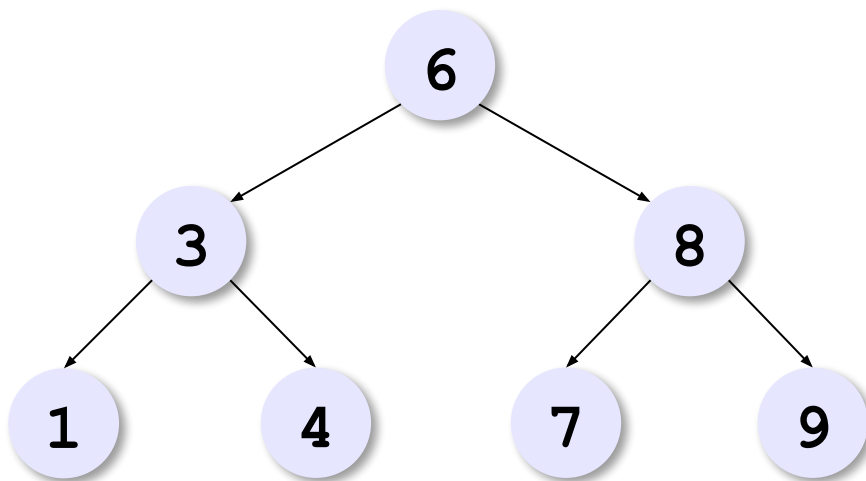
- 1) пустая структура – это **двоичное дерево**
- 2) двоичное дерево – это корень и **два** связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья)

## Применение:

- поиск в большом массиве неменяющихся данных
- сортировка данных
- вычисление арифметических выражений
- оптимальное сжатие данных (метод Хаффмана)

# Деревья поиска

**Ключ** – это значение, связанное с узлом дерева, по которому выполняется поиск.



- **слева** от узла – узлы с *меньшими* или равными ключами
- **справа** от узла – узлы с *большими* или равными ключами

$O(\log N)$



Сложность поиска?

Двоичный поиск  $O(\log N)$   
Линейный поиск  $O(N)$

# Обход дерева

Обойти дерево  $\Leftrightarrow$  «посетить» все узлы по одному разу.

$\Rightarrow$  список узлов

**КЛП – «корень-левый-правый»** (в прямом порядке):

посетить корень  
обойти левое поддерево  
обойти правое поддерево

**ЛКП – «левый-корень-правый»** (симметричный):

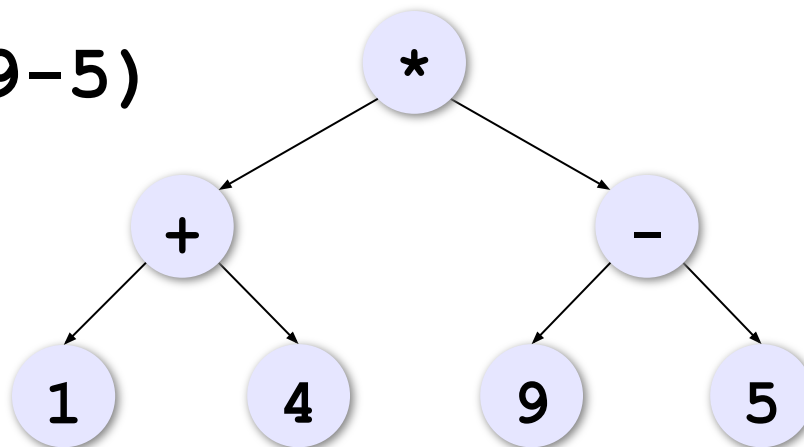
обойти левое поддерево  
посетить корень  
обойти правое поддерево

**ЛПК – «левый-правый-корень»** (в обратном порядке):

обойти левое поддерево  
обойти правое поддерево  
посетить корень

# Обход дерева

$(1+4) * (9-5)$



«в глубину»

КЛП: \* + 1 4 - 9 5    префиксная форма

ЛКП: 1 + 4 \* 9 - 5    инфиксная форма

ЛПК: 1 4 + 9 5 - \*    постфиксная форма

Обход «в ширину»: «сыновья», потом «внуки», ...

\* + - 1 4 9 5

# Обход КЛП – обход «в глубину»

используется вспомогательный стек

записать в стек корень дерева

**while** стек не пуст:

выбрать узел  $V$  с вершины стека

посетить узел  $V$

**если** у узла  $V$  есть правый сын то

добавить в стек правого сына  $V$

**если** у узла  $V$  есть левый сын то

добавить в стек левого сына  $V$

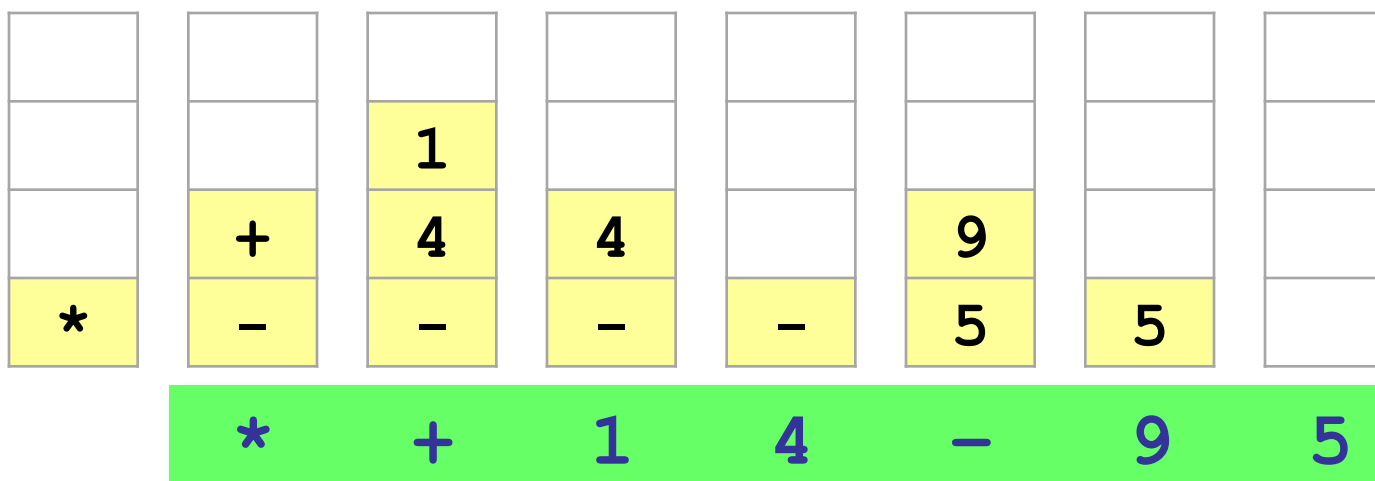
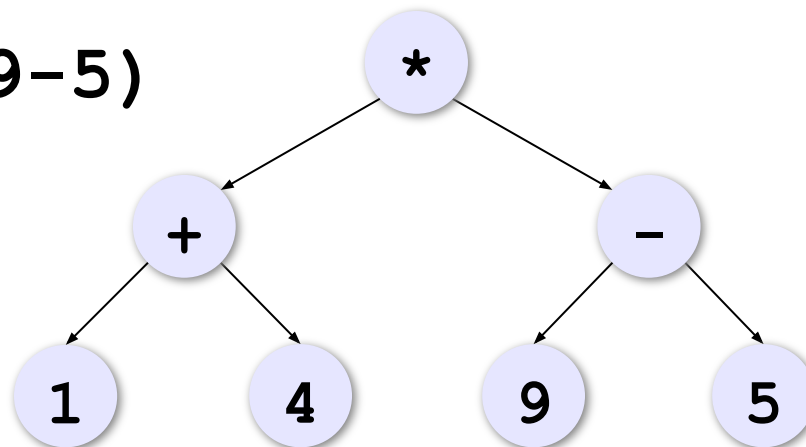


Почему сначала добавить правого сына?



# Обход КЛП – обход «в глубину»

$(1+4) * (9-5)$



# Обход «в ширину»

используется вспомогательная очередь

записать в очередь корень дерева

пока очередь не пуста:

выбрать узел  $V$  из очереди

посетить узел  $V$

если у узла  $V$  есть левый сын то

добавить в очередь левого сына  $V$

если у узла  $V$  есть правый сын то

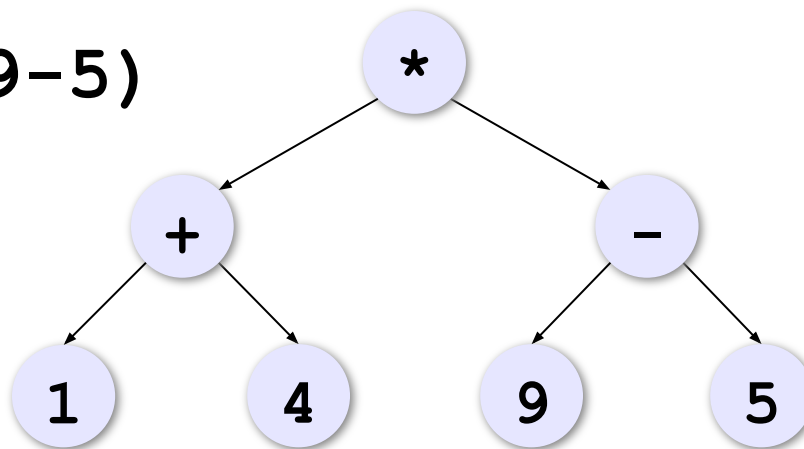
добавить в очередь правого сына  $V$



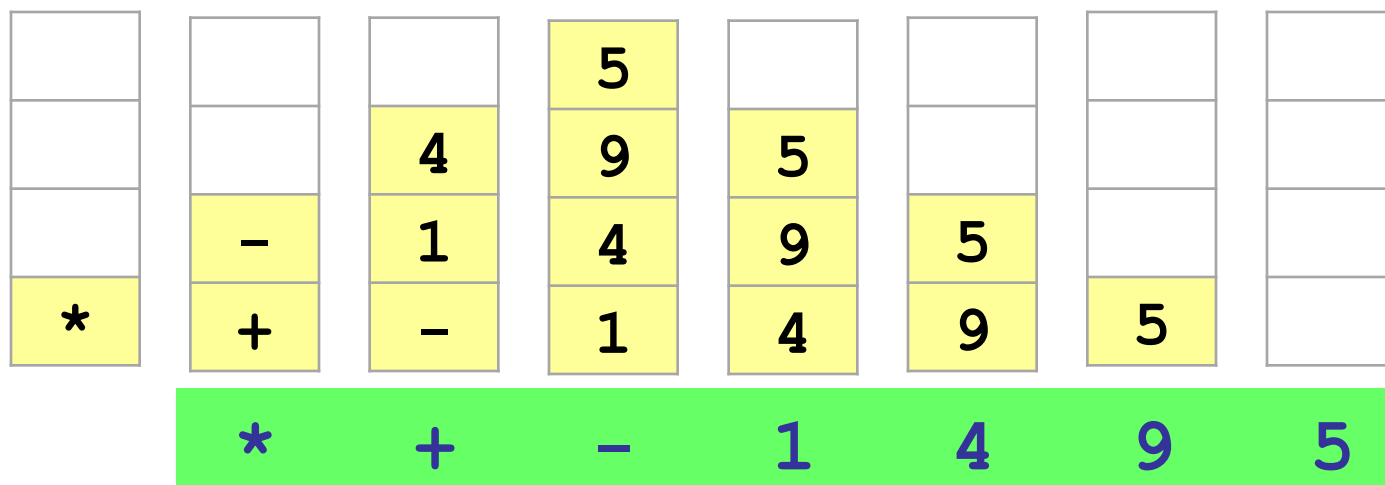
Почему сначала добавить левого сына?

# Обход «в ширину»

$(1+4) * (9-5)$

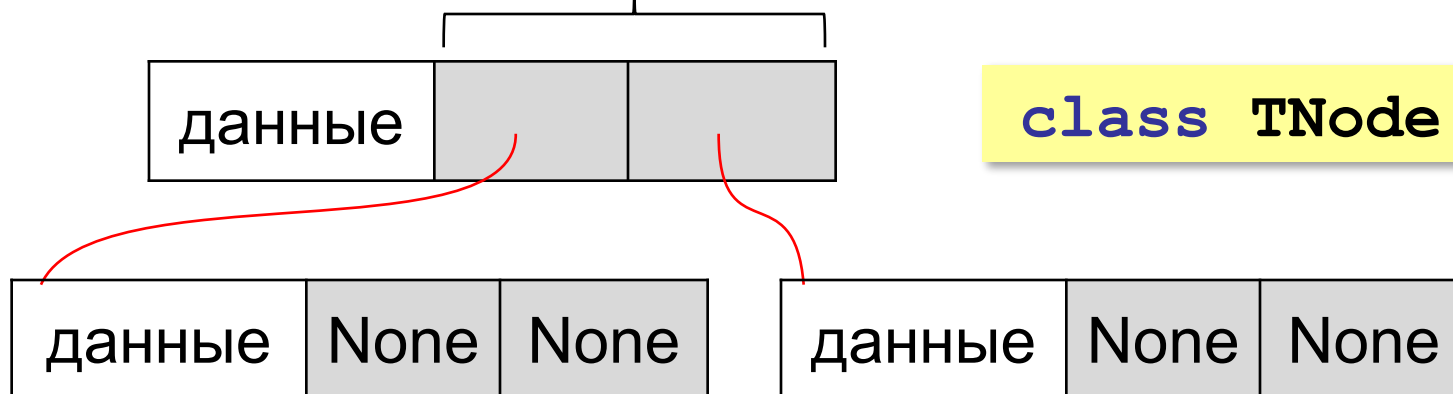


голова  
очереди



# Вспомогательные структуры

ссылки на сыновей



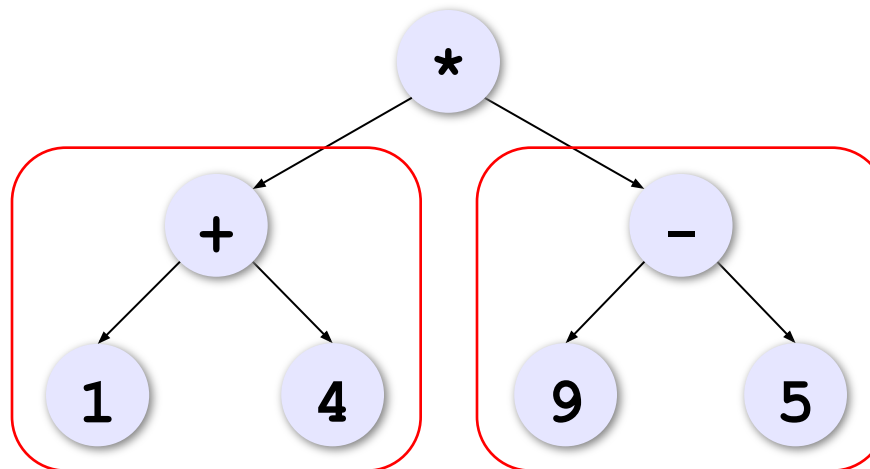
```
class TNode: pass
```

Создание узла:

значения по умолчанию

```
def node(d, L = None, R = None):  
    newNode = TNode() # создаём новый узел  
    newNode.data = d  
    newNode.left = L  
    newNode.right = R  
    return newNode
```

# Создание дерева



```
T = node ("*",  
        node ("+", node ("1"), node ("4")),  
        node ("-", node ("9"), node ("5"))  
        )
```

L = None, R = None

# Обход дерева в глубину

---

## Рекурсивный:

```
def DFS( T ) :  
    if not T: return  
    print(T.data, end=" ")  
    DFS(T.left)  
    DFS(T.right)
```

## Без рекурсии (стек):

```
def DFS_stack( T ) :  
    stack = [T]  
    while stack:  
        V = stack.pop()  
        print(V.data, end=" ")  
        if V.right: stack.append(V.right)  
        if V.left: stack.append(V.left)
```

# Обход дерева в ширину

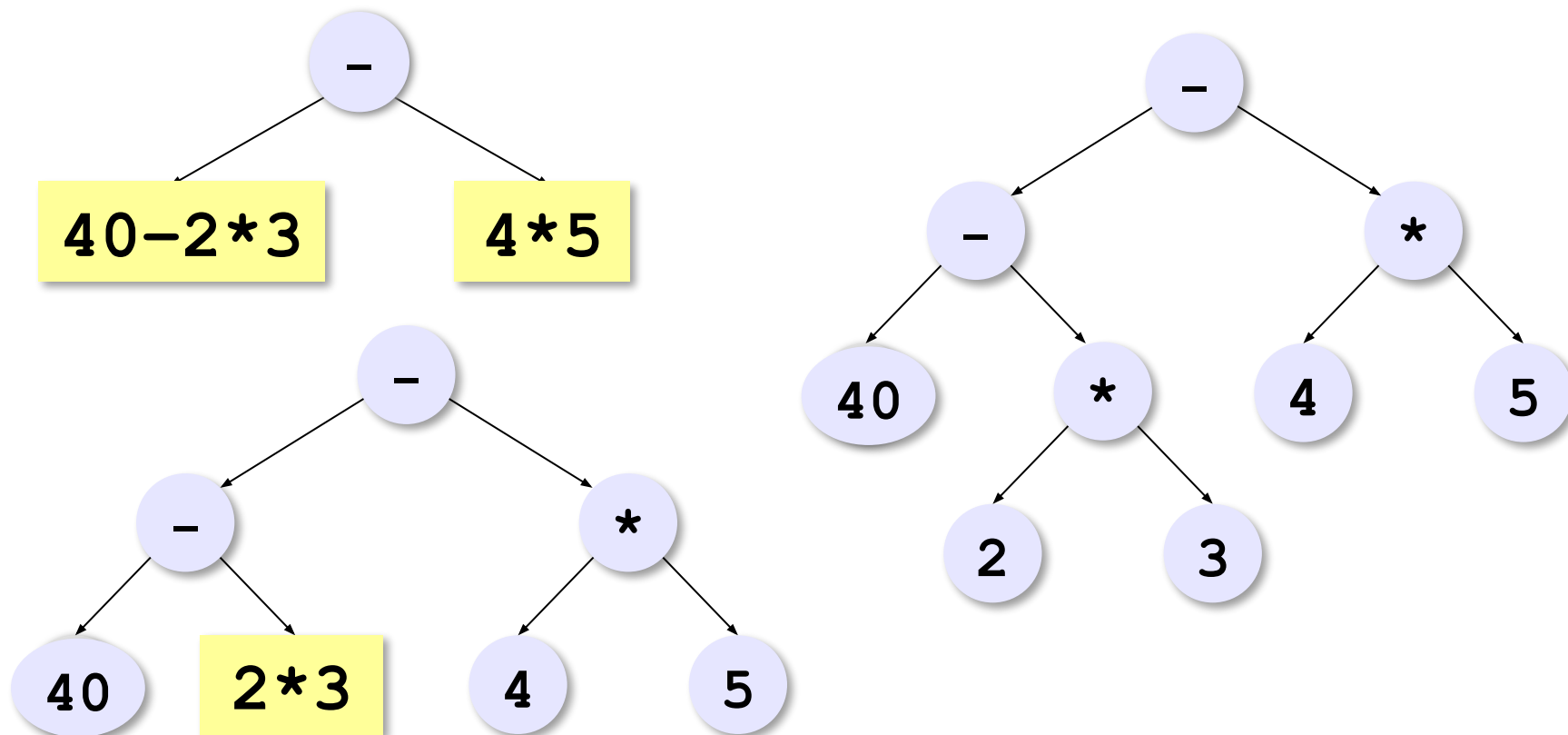
```
def BFS ( T ) :  
    queue = [T]  
    while queue:  
        V = queue.pop(0)    # первый из очереди  
        print(V.data, end=" ")  
        if V.left:    queue.append(V.left)  
        if V.right:    queue.append(V.right)
```

# Вычисление арифметических выражений

 $40 - 2 * 3 - 4 * 5$ 

Что будет в корне дерева?

В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.





# Вычисление арифметических выражений

## Построение дерева:

найти последнюю выполняемую операцию

if операций нет:

создать узел-лист

return

поместить операцию в корень дерева

построить левое поддерево

построить правое поддерево



Рекурсия!

# Вычисление арифметических выражений

## Вычисление по дереву:

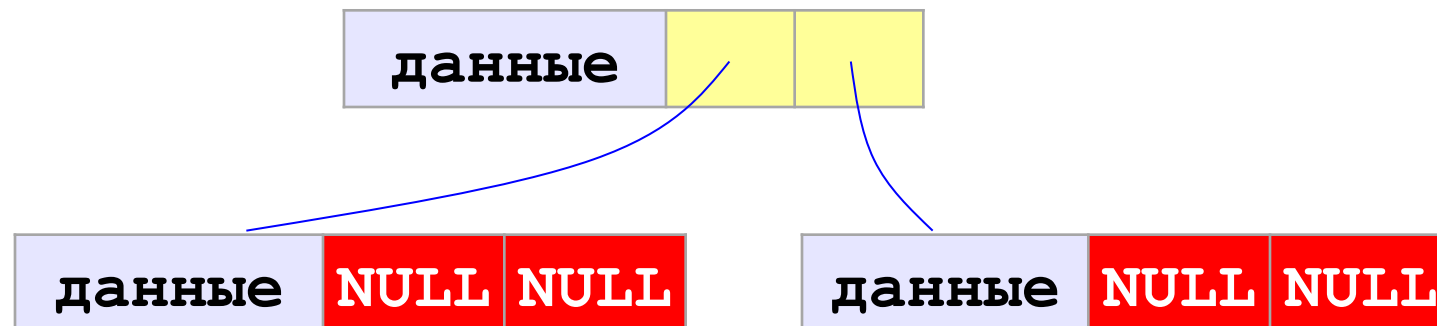
```
n1 = значение левого поддерева  
n2 = значение правого поддерева  
результат = операция(n1, n2)
```



Рекурсия!

# Использование связанных структур

Дерево – **нелинейная** структура  $\Rightarrow$  динамический массив неудобен!



```
class TNode:  
    pass
```

НОВЫЙ ТИП:  
узел

Создание узла:

```
def newNode ( d ) :  
    node = TNode ()  
    node.data = d  
    node.left = None  
    node.right = None  
    return node
```

# Основная программа

---

```
s = input ( "Введите выражение: " )  
T = makeTree ( s )  
print ( "Результат: ", calcTree(T) )
```



Нужно построить `makeTree` и `calcTree`!

# Построение дерева

```
def makeTree ( s ) :  
    k = lastOp (s) # номер последней операции  
    if k < 0:      # создать лист  
        Tree = newNode ( s )  
    else: # создать узел-операцию  
        Tree = newNode ( s[k] )  
        Tree.left = makeTree ( s[:k] )  
        Tree.right = makeTree ( s[k+1:] )  
    return Tree
```

вернёт ссылку на  
корень нового  
дерева



Рекурсия!

# Вычисление по дереву

```
def calcTree ( Tree ) :  
    if Tree.left == None :  
        return int(Tree.data)  
    else :  
        n1 = calcTree ( Tree.left )  
        n2 = calcTree ( Tree.right )  
        if Tree.data == "+" :    res = n1 + n2  
        elif Tree.data == "-" : res = n1 - n2  
        elif Tree.data == "*" : res = n1 * n2  
        else : res = n1 // n2  
        return res
```

ЭТО ЧИСЛО  
(ЛИСТ)



Рекурсия!

# Вспомогательные функции

## Приоритет операции:

```
def priority ( op ) :  
    if op in "+-": return 1  
    if op in "* /": return 2  
    return 100
```

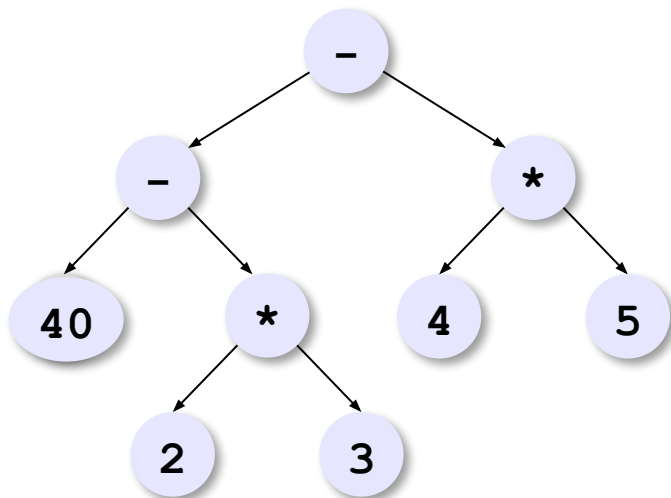
## Номер последней операции:

```
def lastOp ( s ) :  
    minPrt = 50      # любое между 2 и 100  
    k = -1  
    for i in range(len(s)) :  
        if priority(s[i]) <= minPrt :  
            minPrt = priority(s[i])  
            k = i  
    return k
```



Почему <=?

# Двоичное дерево в массиве



$A[0]$   $\swarrow$   $A[1]$   
 $\searrow$   $A[2]$

$A[1]$   $\swarrow$   $A[3]$   
 $\searrow$   $A[4]$

$A[2]$   $\swarrow$   $A[5]$   
 $\searrow$   $A[6]$

$A[4]$   $\swarrow$   $A[9]$   
 $\searrow$   $A[10]$

0	1	2	3	4	5	6	7	8	9	10
-	-	*								

0	1	2	3	4	5	6	7	8	9	10
-	-	*	40	*						

0	1	2	3	4	5	6	7	8	9	10
-	-	*	40	*	4	5				

0	1	2	3	4	5	6	7	8	9	10
-	-	*	40	*	4	5			2	3

$A[i]$   $\swarrow$   $A[?]$   
 $\searrow$   $A[?]$

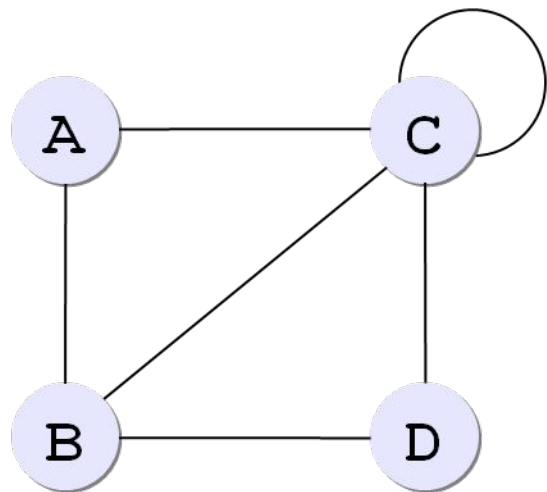


# Алгоритмизация и программирование. Язык Python

## § 40. Графы

# Что такое граф?

**Граф** – это набор вершин и связей между ними (рёбер).



## Матрица смежности:

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	1	1
D	0	1	1	0

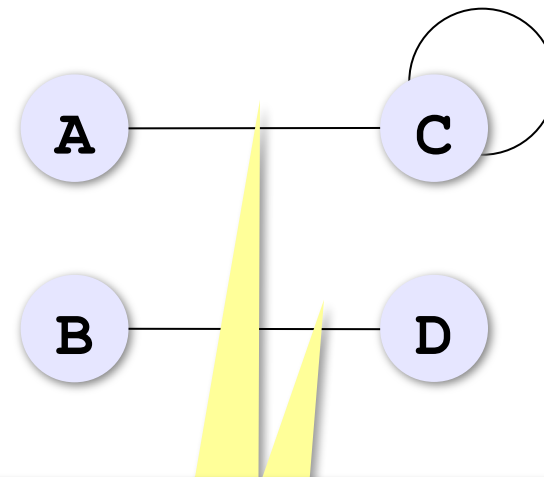
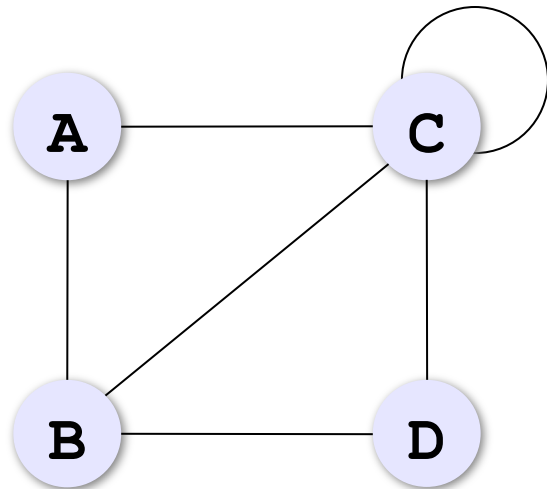
петля

## Список смежности:

( **A** (B, C) ,  
  **B** (A, C, D) ,  
  **C** (A, B, C, D) ,  
  **D** (B, C) )

# Связность графа

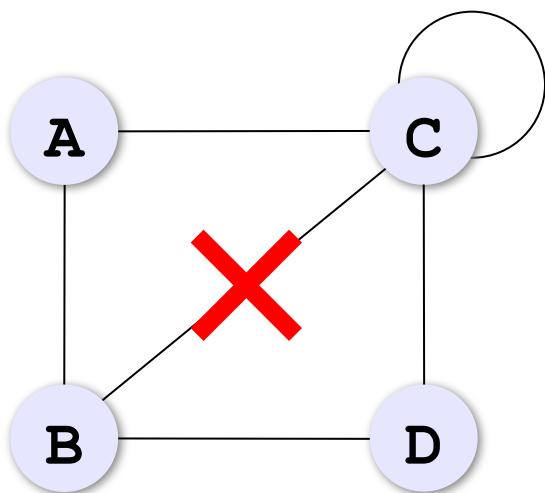
**Связный граф** – это граф, между любыми вершинами которого существует путь.



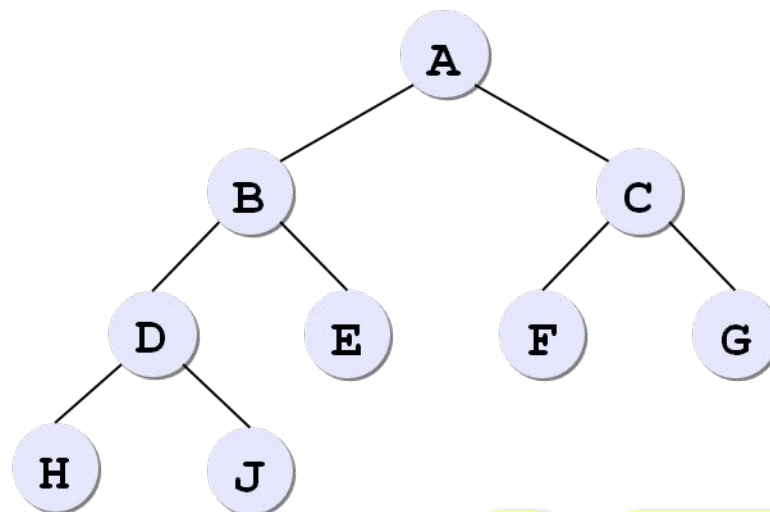
КОМПОНЕНТЫ СВЯЗНОСТИ

# Дерево – это граф?

**Дерево** – это связный граф без циклов (замкнутых путей).

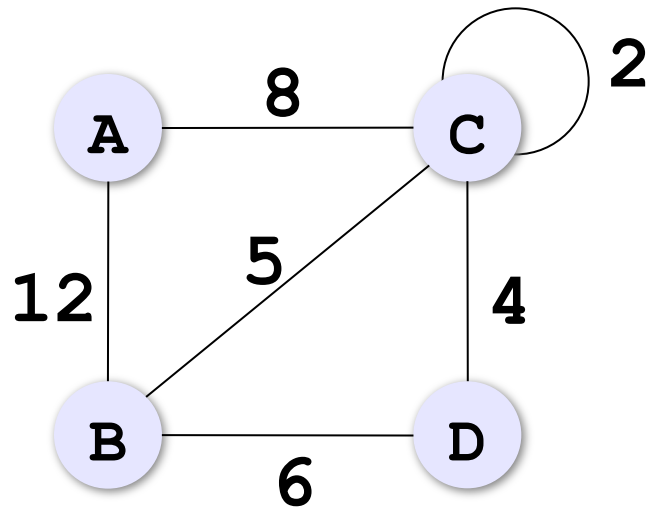


ABC ABDC  
BCD CCC...



дерево

# Взвешенные графы



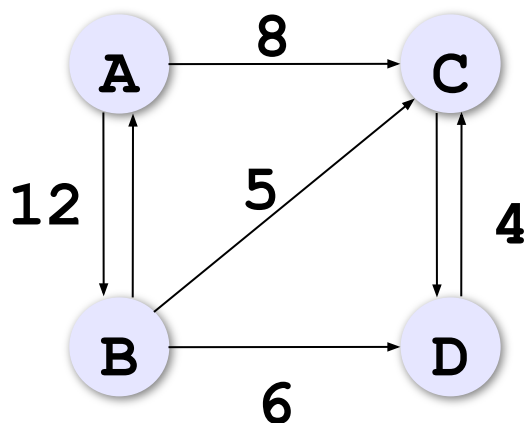
вес ребра

Весовая матрица:

	A	B	C	D
A		12	8	
B	12		5	6
C	8	5	2	4
D		6	4	

# Оrientированные графы (орграфы)

Рёбра имеют направление (начало и конец), рёбра называю **дугами**.



	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	



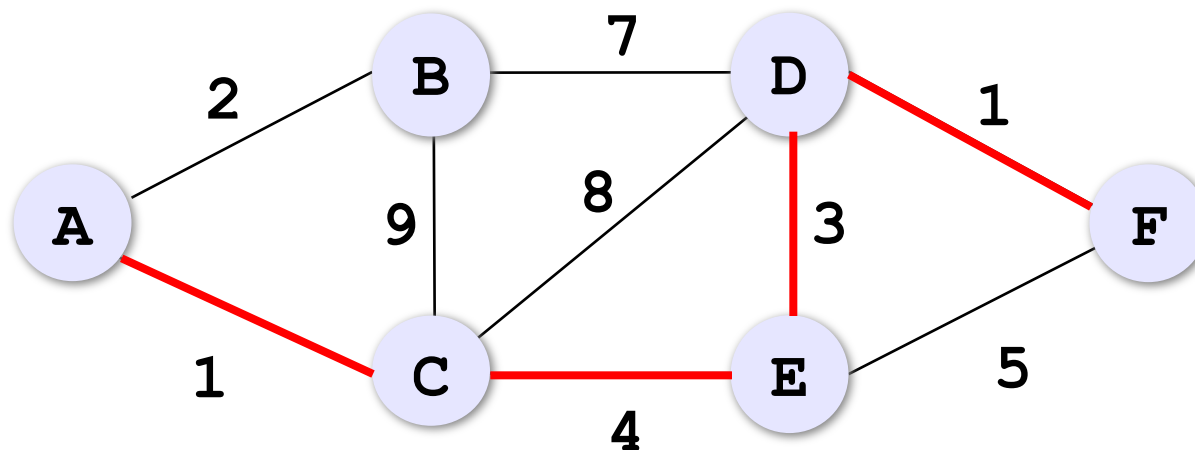
Весовая матрица может быть несимметрична!

# Жадные алгоритмы



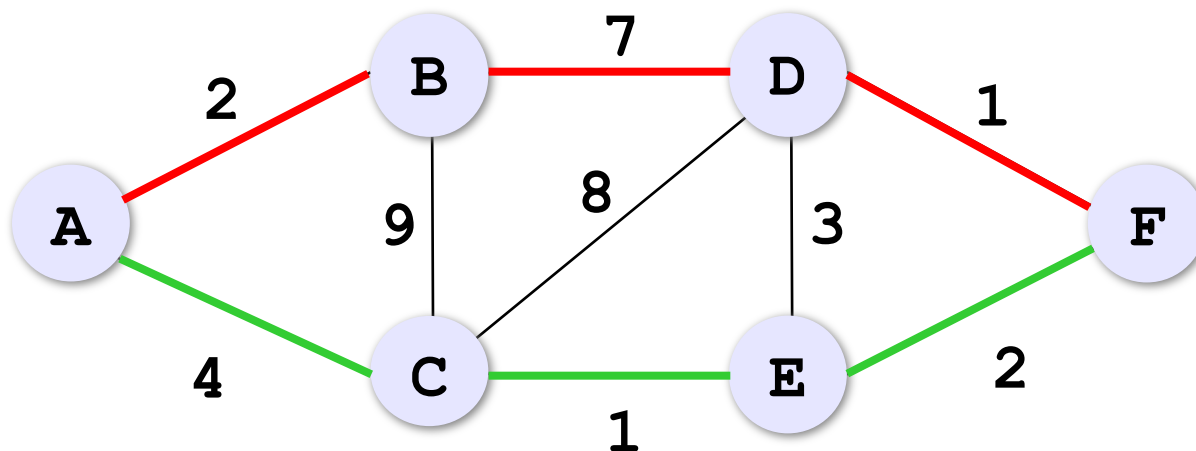
**Жадный алгоритм** – это многошаговый алгоритм, в котором на каждом шаге принимается решение, лучшее в данный момент.

**Задача.** Найти кратчайший маршрут из **A** в **F**.



# Жадные алгоритмы

Задача. Найти кратчайший маршрут из **A** в **F**.



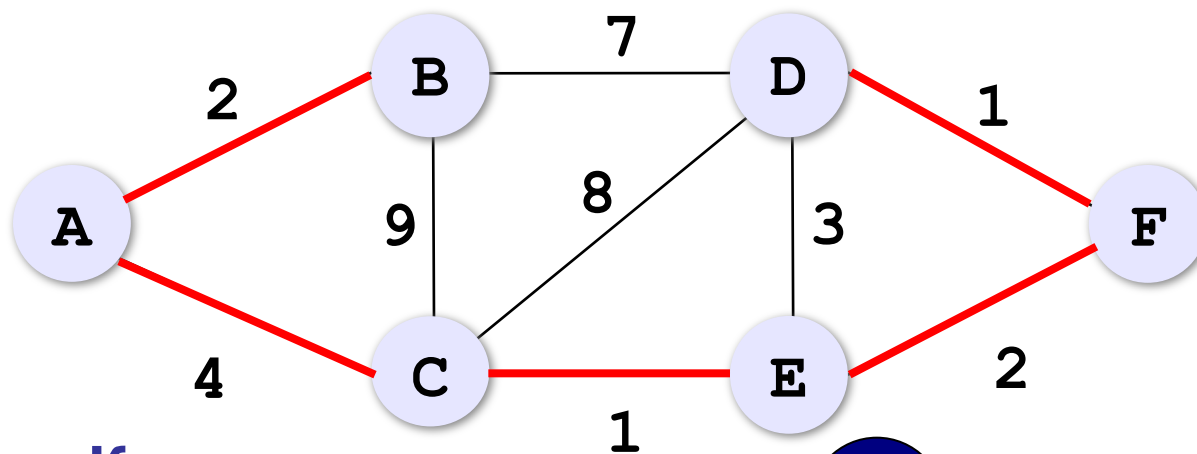
**?** Это лучший маршрут?

**!** Жадный алгоритм не всегда даёт наилучшее решение!



# Задача Прима-Крускала

**Задача.** Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?  
(**минимальное остовное дерево**)



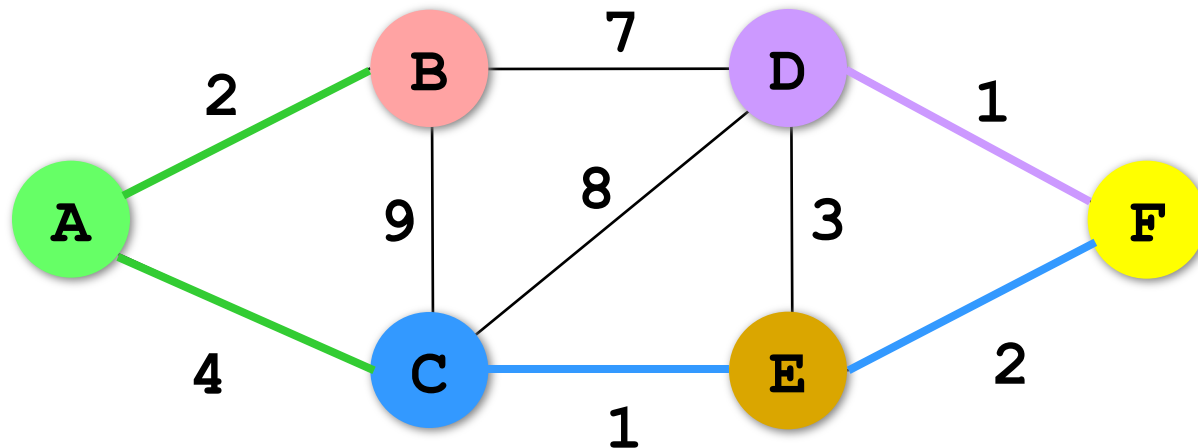
## Алгоритм Крускала:

- начальное дерево — пустое
- на каждом шаге добавляется ребро минимального веса, которое ещё не входит в дерево и не приводит к появлению цикла



Лучшее решение!

# Раскраска вершин



```
col = [i for i in range(N)]
```

каждой  
вершине  
свой цвет

## Сделать N-1 раз:

- ищем ребро минимальной длины среди всех рёбер, **концы которых окрашены в разные цвета**;
- найденное ребро **(iMin, jMin)** добавляется в список выбранных, и все вершины, имеющие цвет **col[jMin]**, перекрашиваются в цвет **col[iMin]**.

# Раскраска вершин

## Весовая матрица:

```
N = 6
INF = 30000 # очень большое число
W = []
for i in range(N):
    W.append( [0]*N )
# заполнить матрицу
```

## Вывод результата:

```
for edge in ostov:
    print( "(" , edge[0] , " , " ,
           edge[1] , ")" )
```

# Раскраска вершин

```
ostov = [] # список выбранных рёбер
for k in range(N-1):
    minDist = INF # очень большое число
    for i in range(N):
        for j in range(N):
            if col[i] != col[j] \
                and W[i][j] < minDist:
                iMin = i
                jMin = j
                minDist = W[i][j]
    ostov.append( (iMin, jMin) )
    c = col[jMin]
    for i in range(N):
        if col[i] == c:
            col[i] = col[iMin]
```

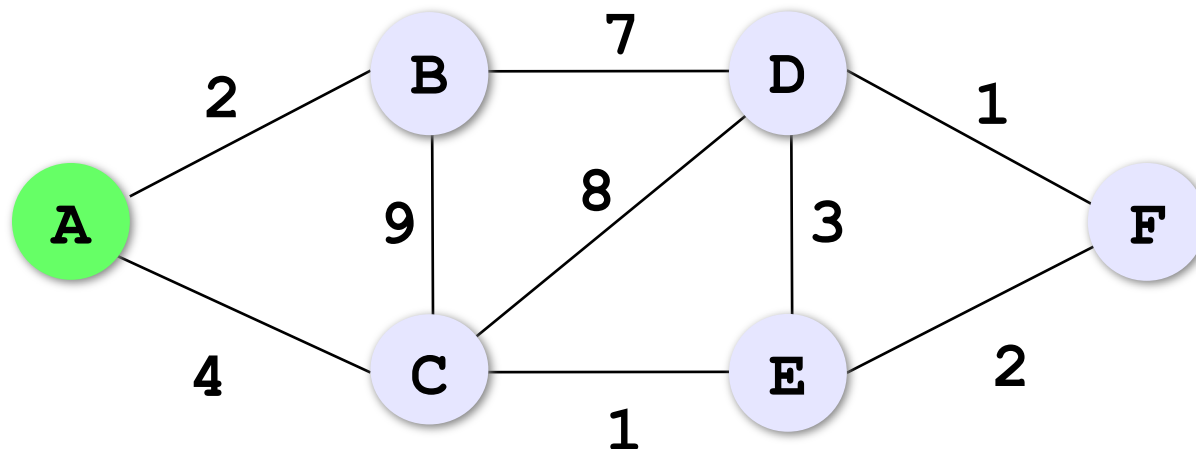
ищем ребро с  
минимальным  
весом

добавить  
новое ребро

перекраска  
вершин

# Кратчайший маршрут

## Алгоритм Дейкстры (1960):



Э.В.  
Дейкстра

	A	B	C	D	E	F
dist	0	2	4	$\infty$	$\infty$	$\infty$
selected	+					

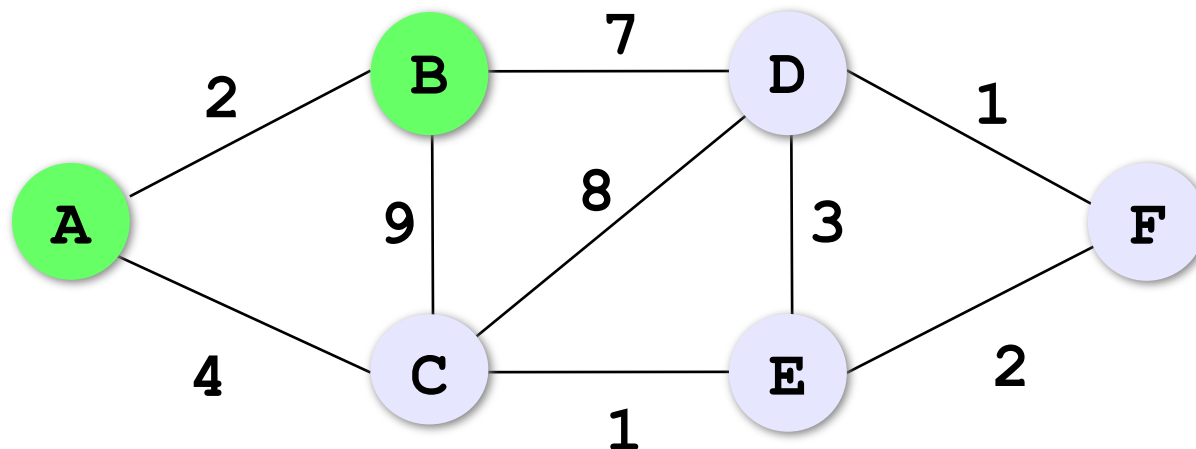
кратчайшее  
расстояние от A

выбрана?

ближайшая от A  
невыбранная вершина

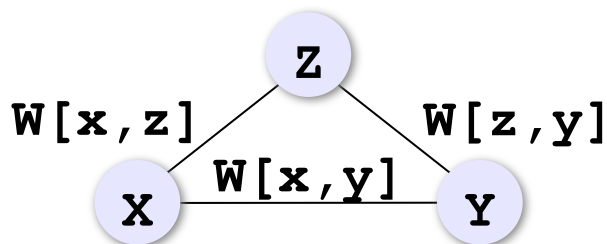
# Кратчайший маршрут

## Алгоритм Дейкстры (1960):



Э.В.  
Дейкстра

	A	B	C	D	E	F
dist	0	2	4	9	$\infty$	$\infty$
selected	+	+				

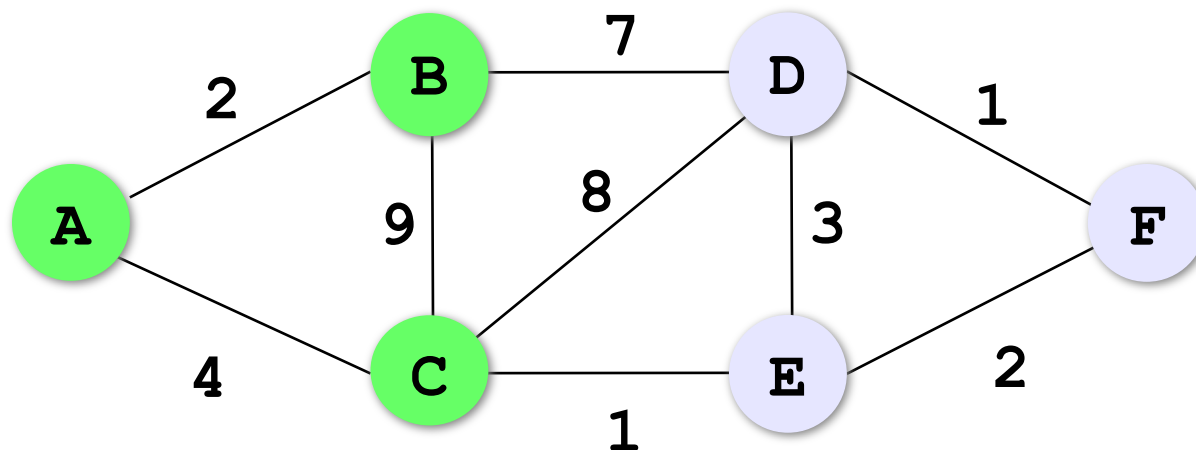


может быть так, что

$$W[x, z] + W[z, y] < W[x, y]$$

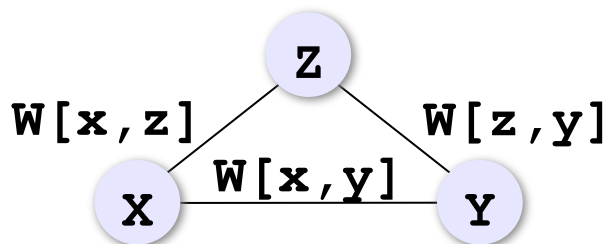
# Кратчайший маршрут

## Алгоритм Дейкстры (1960):



Э.В.  
Дейкстра

	A	B	C	D	E	F
dist	0	2	4	9	5	$\infty$
selected	+	+	+			

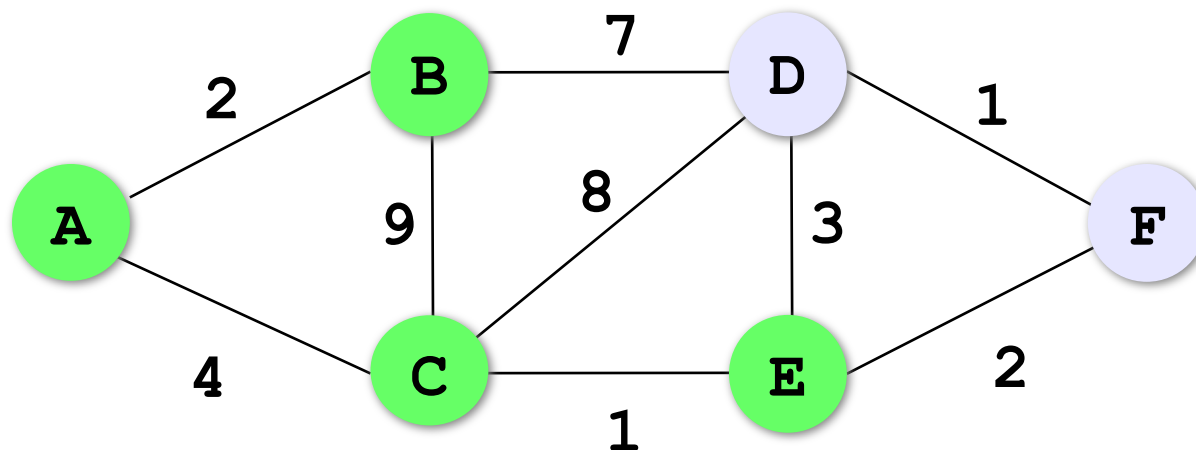


может быть так, что

$$W[x, z] + W[z, y] < W[x, y]$$

# Кратчайший маршрут

## Алгоритм Дейкстры (1960):



Э.В.  
Дейкстра

	A	B	C	D	E	F
dist	0	2	4	8	5	7
selected	+	+	+		+	

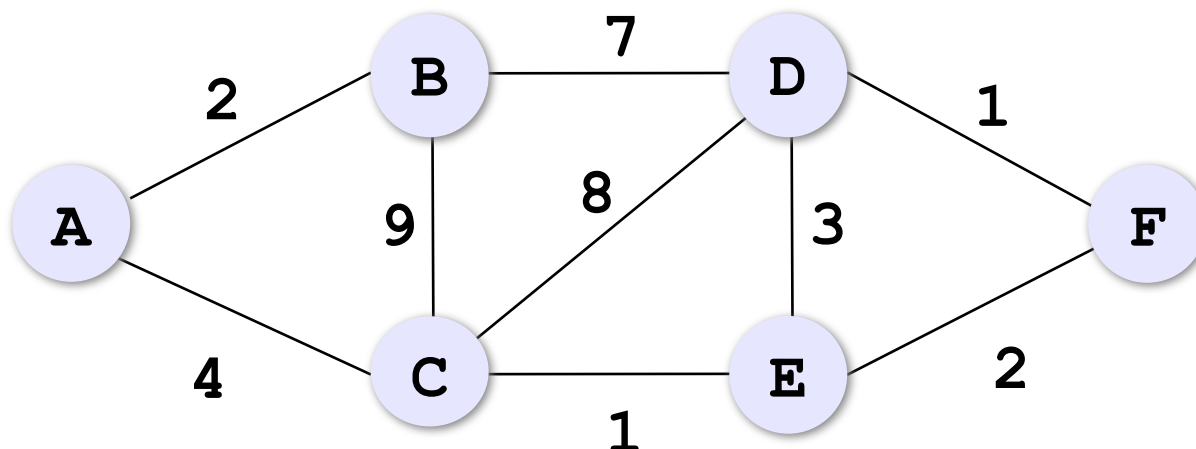
длины кратчайших  
маршрутов из A в  
другие вершины



При рассмотрении вершин **F** и **D**  
таблица не меняется!



# Как найти самый короткий маршрут?



	A	B	C	D	E	F
dist	0	2	4	8	5	7

$$\text{dist}[D] + DF = 8 + 1 = 9$$

$$\text{dist}[E] + EF = 5 + 2 = 7$$

E → F

откуда  
приехали в  
F?

# Алгоритм Дейкстры

## Весовая матрица:

```
N = 6
W = []
for i in range(N):
    W.append( [0]*N )
# заполнить матрицу
```

## Вспомогательные массивы:

```
INF = 30000          # очень большое число
selected = [False]*N # все не выбраны
dist = [INF]*N       # расстояния неизвестны
```

## Стартовая вершина:

```
start = 0
dist[start] = 0
V = start          # выбранная вершина
```

вершина A  
просмотрена

# Алгоритм Дейкстры

## Основной цикл:

```
minDist = 0
while minDist < INF:
    selected[V] = True
    # проверка маршрутов через вершину V
    for i in range(N):
        if dist[V] + W[V][i] < dist[i]:
            dist[i] = dist[V] + W[V][i]
    # поиск новой вершины dist[i] -> min
    minDist = INF # большое число
    for i in range(N):
        if not selected[i] and \
            dist[i] < minDist:
            minDist = dist[i]
            V = i
```



Какова сложность?

$O(N^2)$

# Алгоритм Дейкстры

Вывод результата (маршрут  $0 \rightarrow N-1$ ):

```
V = N - 1
print(V)
while V != start:
    for i in range(N):
        if i != V and \
            dist[i] + W[i][V] == dist[V]:
            V = i
            break
    print(V)
```

пока не пришли в начальную вершину обратным ходом

нашли вершину  $i$ , из которой могли прийти в вершину  $V$

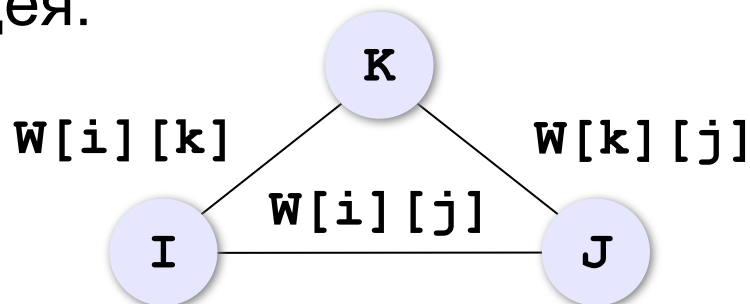


Она единственная?

# Алгоритм Флойда

Все кратчайшие пути (из любой вершины в любую)

Идея:



Какова сложность?

$O(N^3)$

проверить все  
вершины

для всех  
пар вершин

без  
вершины k

```
for i in range(N):  
    for j in range(N):  
        if W[i][k] + W[k][j] < W[i][j]:  
            W[i][j] = W[i][k] + W[k][j]
```

путь через  
вершину k



Как найти сам маршрут?

# Списки смежности

## Список смежности:

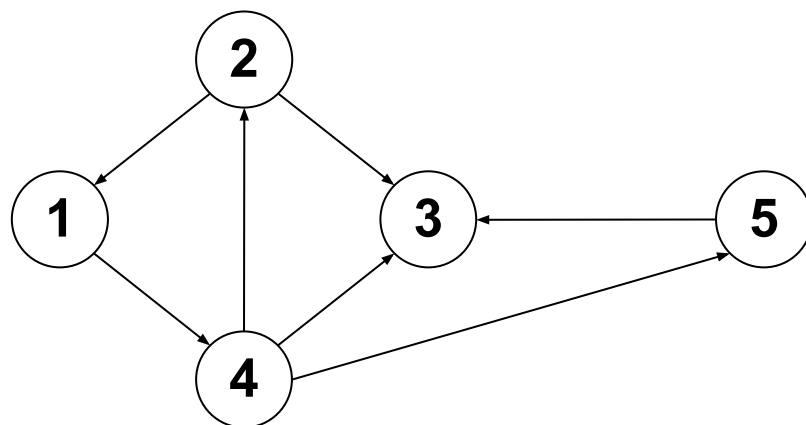
вершина 1: ( **4** )

вершина 2: ( **1**, **3** )

вершина 3: ( )

вершина 4: ( **2**, **3**, **5** )

вершина 5: ( **3** )



```
Graph = [ [],          # фиктивный элемент
          [4],         # для вершины 1
          [1, 3],      # ... для вершины 2
          [],          # ... для вершины 3
          [2, 3, 5],   # ... для вершины 4
          [3] ]        # ... для вершины 5
```

# Списки смежности

**Задача.** Составить функцию, которая находит число путей из одной вершины в другую.

**Основная программа:**

```
Graph = [[], [4], [1, 3], [],  
         [2, 3, 5], [3]]  
print ( pathCount (Graph, 1, 3, []) )
```

СПИСОК  
посещённых  
вершин

откуда

куда



Зачем список посещенных вершин?

# Число путей: функция

СПИСКИ  
СМЕЖНОСТИ

СПИСОК ПОСЕЩЁННЫХ  
ВЕРШИН

```
def pathCount ( graph, vStart, vEnd,
                visited ) :

    if vStart == vEnd:
        return 1
    visited.append ( vStart )
    count = 0
    for v in graph[vStart]:
        if not v in visited:
            count += pathCount ( graph, v, vEnd,
                                visited )

    visited.pop()
    return count
```

если уже пришли,  
ВЫХОД

суммируем пути  
через соседние



# Задача коммивояжера

Коммивояжер (бродячий торговец) должен выйти из города 1 и, посетив по разу в неизвестном порядке города  $2, 3, \dots, N$ , вернуться обратно в город 1. В каком порядке надо обходить города, чтобы путь коммивояжера был кратчайшим?



Это NP-полная задача, которая строго решается только перебором вариантов (пока)!

## Точные методы:

- 1) простой перебор;
- 2) метод ветвей и границ;
- 3) метод Литтла;
- 4) ...



большое время счета для больших  $N$

$O(N!)$

## Приближенные методы:

- 5) метод случайных перестановок (*Matlab*)
- 6) генетические алгоритмы
- 7) метод муравьиных колоний
- 8) ...



не гарантируется оптимальное решение

# Некоторые задачи

---

**Задача на минимум суммы.** Имеется  $N$  домов, в каждом из которых живет  $p_i$  жителей ( $i=1, \dots, N$ ). Надо разместить магазин в одном из них так, чтобы общее расстояние, проходимое всеми жителями по дороге в магазин, было минимальным.

**Задача о наибольшем потоке.** Есть система труб, которые имеют соединения в  $N$  узлах. Один узел  $S$  является источником, еще один – стоком  $T$ . Известны пропускные способности каждой трубы. Надо найти наибольший поток от источника к стоку.

**Задача о наибольшем паросочетании.** Есть  $M$  мужчин и  $N$  женщин. Каждый мужчина указывает несколько (от 0 до  $N$ ) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до  $M$ ), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

# Алгоритмизация и программирование. Язык Python

## **§ 41. Динамическое программирование**

# Что такое динамическое программирование?

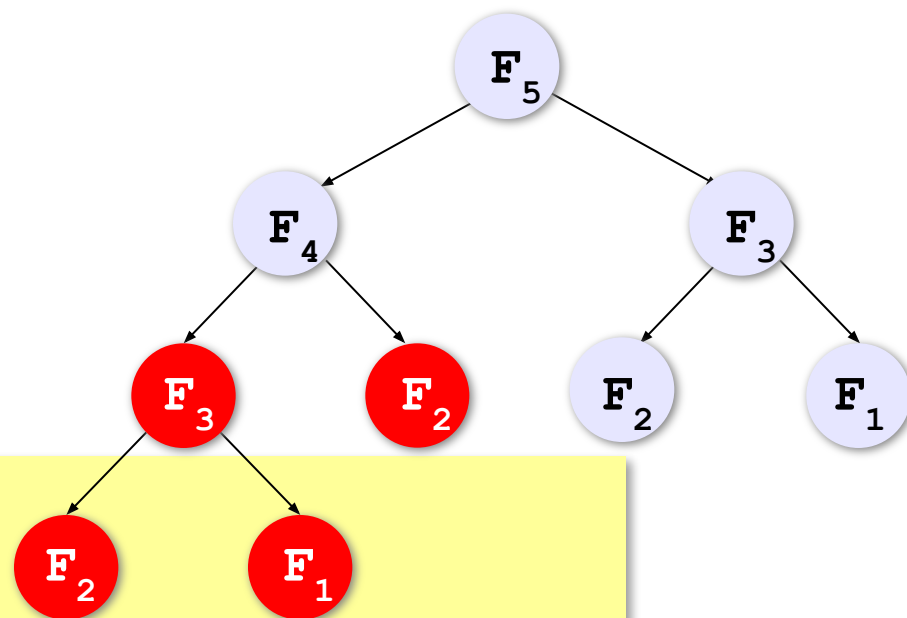
## Числа Фибоначчи:

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \text{ при } n > 2$$

**!** Рекурсия!

```
def Fib ( n ) :  
    if n < 3 :  
        return 1  
    return Fib (n-1) + Fib (n-2)
```



**—** повторное вычисление тех же значений

**!** Запоминать то, что вычислено!

# Динамическое программирование

$F_1$	$F_2$	$F_3$	$F_4$	$F_5$
1	1			

$$F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}, \text{ при } n > 2$$

Создание массива:

```
F = [1] * (N+1) # чтобы начать с 1
```

Заполнение массива:

```
for i in range(3, N+1):  
    F[i] = F[i-1] + F[i-2]
```

$F_{35}$ : рекурсия: **58 с**

дин. программирование: **< 0,001 с**

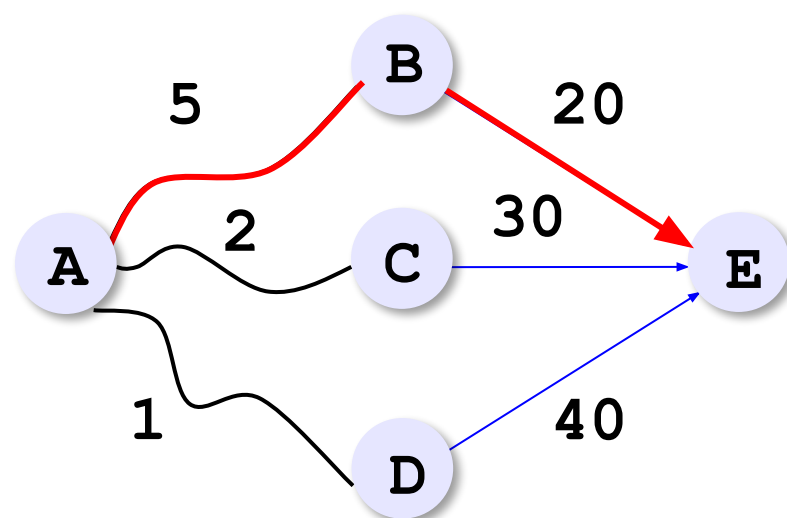


Можно ли обойтись без массива?

нужны только  
два последних!

# Динамическое программирование

**Динамическое программирование** – это способ решения сложных задач путем сведения их к более простым задачам того же типа.



$$ABE: 5 + 20 = 25$$

$$ACE: 2 + 30 = 32$$

$$ADE: 1 + 40 = 41$$



увеличение скорости



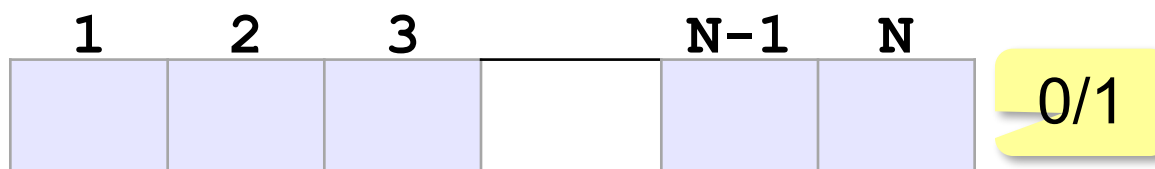
дополнительный расход памяти

# Количество вариантов

**Задача.** Найти количество  $K_N$  цепочек, состоящих из  $N$  нулей и единиц, в которых нет двух стоящих подряд единиц.

**Решение «в лоб»:**

битовые цепочки



- построить все возможные цепочки
- проверить каждую на «правильность»



Сколько возможных цепочек?

$2^N$

Сложность  
алгоритма  $O(2^N)$

# Количество вариантов

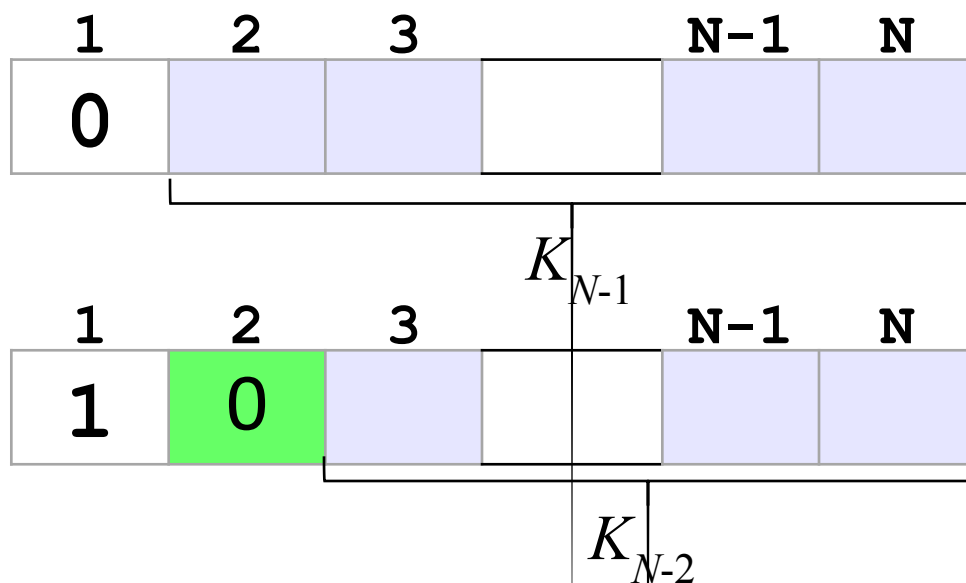
**Задача.** Найти количество  $K_N$  цепочек, состоящих из  $N$  нулей и единиц, в которых нет двух стоящих подряд единиц.

**Простые случаи:**

$N = 1:$     **0**    **1**             $K_1 = 2$

$N = 2:$     **00**    **01**    **10**     $K_2 = 3$

**Общий случай:**



$$K_N = K_{N-1} + K_{N-2} = F_{N+2}$$

$K_{N-1}$  «правильных»  
цепочек начинаются  
с нуля!

$K_{N-2}$  «правильных»  
цепочек начинаются  
с единицы!



# Оптимальное решение

**Задача.** В цистерне  $N$  литров молока. Есть бидоны объемом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все бидоны были заполнены и количество используемых бидонов было **минимальным**.

**Перебор?**

при больших  $N$  – очень долго!

**«Жадный алгоритм»?**

$$N = 10: \quad 10 = 6 + 1 + 1 + 1 + 1 \quad K = 5$$

$$10 = 5 + 5 \quad K = 2$$



Не даёт оптимального решения!

# Оптимальное решение

$K_N$  – минимальное число бидонов для  $N$  литров

Сначала выбрали бидон...

$$\left. \begin{array}{l} 1 \text{ л: } K_N = 1 + K_{N-1} \\ 5 \text{ л: } K_N = 1 + K_{N-5} \\ 6 \text{ л: } K_N = 1 + K_{N-6} \end{array} \right\} \text{min}$$

Рекуррентная формула:

$$K_N = 1 + \text{min} (K_{N-1}, K_{N-5}, K_{N-6}) \quad \text{при } N \geq 6$$

$$K_N = 1 + \text{min} (K_{N-1}, K_{N-5}) \quad \text{при } N = 5$$

$$K_N = 1 + K_{N-1} \quad \text{при } N < 5$$

# Оптимальное решение (бидоны)

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6})$$

N	0	1	2	3	4	5	6	7	8	9	10
$K_N$	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

объём бидона, взятого последним

N	0	1	2	3	4	5	6	7	8	9	10
$K_N$	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

2 бидона

5 + 5



Похоже на алгоритм Дейкстры!

# Задача о куче

**Задача.** Из камней весом  $p_i$  ( $i=1, \dots, N$ ) набрать кучу весом ровно  $W$  или, если это невозможно, максимально близкую к  $W$  (но меньшую, чем  $W$ ).

**Решение «в лоб»:**

1	2	3		N-1	N
1	0	0		1	0



камень  
взят

камень  
не взят



Выбрать лучшую цепочку!



Сколько возможных цепочек?

$2^N$

Сложность  
алгоритма  $O(2^N)$

# Задача о куче

**Задача.** Из камней весом  $p_i$  ( $i=1, \dots, N$ ) набрать кучу весом ровно  $W$  или, если это невозможно, максимально близкую к  $W$  (но меньшую, чем  $W$ ).

**Идея:** сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней  $N$  и меньшем весе  $W$ ).

**Пример:**  $W = 8$ , камни 2, 4, 5 и 7

		0	1	2	3	4	5	6	7	8	<b>w</b>
1	2	0	0	2	2	2	2	2	2	2	
2	4	0									
3	5	0									
4	7	0									

базовые случаи

**i**

**$p_i$**

$T[i][w]$  – оптимальный вес, полученный для кучи весом  $w$  из  $i$  первых по счёту камней.

# Задача о куче

		0	1	2	3	4	5	6	7	8
1	2	0	0	2	2	2	2	2	2	2
2	4	0	0	2	2	4	4	6	6	6
3	5	0								
4	7	0								

Добавляем камень с весом **4**:

для  $w < 4$  ничего не меняется!

для  $w \geq 4$ :

если его не брать:  $T[2][w] = T[1][w]$

если его взять:  $T[2][w] = 4 + T[1][w-4]$



Какой вариант выбрать?

**max**

# Задача о куче

		0	1	2	3	4	5	6	7	8
1	2	0	0	2	2	2	2	2	2	2
2	4	0	0	2	2	4	4	6	6	6
3	5	0	0	2	2	4	5	6	7	7
4	7	0								

Добавляем камень с весом **5**:

для  $w < 5$  ничего не меняется!

для  $w \geq 5$ :

если его не брать:

если его взять:

$$T[3][w] = T[2][w]$$

$$T[3][w] = 5 + T[2][w-5]$$

max

# Задача о куче

		0	1	2	3	4	5	6	7	8
1	2	0	0	2	2	2	2	2	2	2
2	4	0	0	2	2	4	4	6	6	6
3	5	0	0	2	2	4	5	6	7	7
4	7	0	0	2	2	4	5	6	7	7

Добавляем камень с весом 7:

для  $w < 7$  ничего не меняется!

для  $w \geq 7$ :

если его не брать:  $T[4][w] = T[3][w]$

если его взять:  $T[4][w] = 7 + T[3][w-7]$

max



# Задача о куче

Добавляем камень с весом  $p_i$ :

для  $w < p_i$  ничего не меняется!

для  $w \geq p_i$ :

если его не брать:  $T[i][w] = T[i-1][w]$

если его взять:  $T[i][w] = p_i + T[i-1][w-p_i]$

max

Рекуррентная формула:

при  $w < p_i$ :  $T[i][w] = T[i-1][w]$

при  $w \geq p_i$ :  $T[i][w] = \max ( T[i-1][w] ,$   
 $p_i + T[i-1][w-p_i] )$

# Задача о куче

**?** Какие камни нужно взять?

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Diagram illustrating the optimal selection of stones (2, 4, 5, 7) based on the table. The values 2, 4, 5, and 7 are circled in green in the first column. The values 2, 2, 7, and 7 are circled in red in the last column. Blue arrows indicate the selection path: from the first 2 to the first 2, from the first 2 to the first 2, from the first 2 to the first 7, and from the first 7 to the first 7.

Оптимальный вес 7    5 + 2

# Задача о куче

**?** Какова сложность алгоритма?

Заполнение таблицы:

$W+1$

		$W+1$								
		0	1	2	3	4	5	6	7	8
$N$	2	0	0	2	2	2	2	2	2	2
	4	0	0	2	2	4	4	6	6	6
	5	0	0	2	2	4	5	6	7	7
	7	0	0	2	2	4	5	6	7	7

**!** Сложность  $O(N \cdot W)$  !

псевдополиномиальный

# Количество программ

---

**Задача.** У исполнителя Утроитель есть команды:

1. прибавь 1
2. умножь на 3

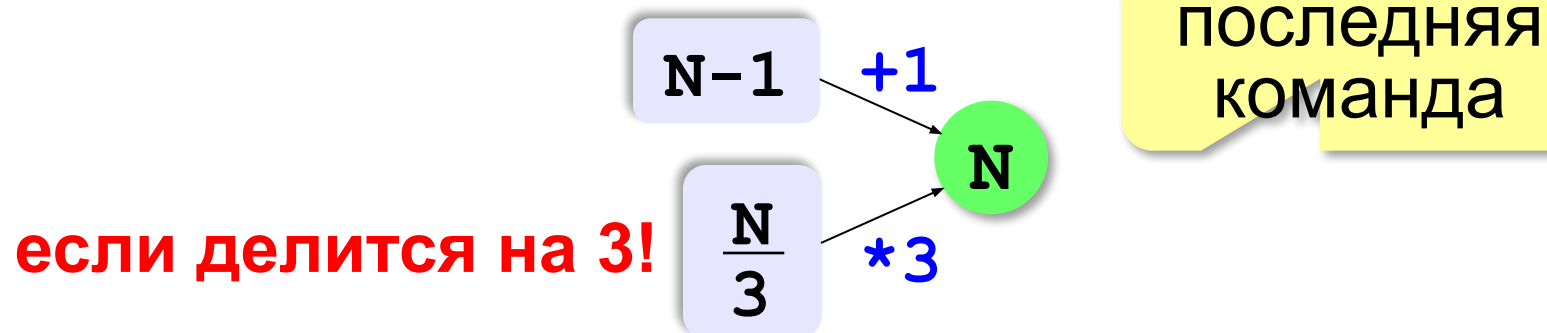
Сколько есть разных программ, с помощью которых можно из числа **1** получить число **20**?



Как решать, не выписывая все программы?

# Количество программ

Как получить число N:



Рекуррентная формула:

$$K_N = K_{N-1}$$

если N **не** делится на 3

$$K_N = K_{N-1} + K_{N/3}$$

если N делится на 3

# Количество программ

Рекуррентная формула:

$$K_N = K_{N-1} \quad \text{если } N \text{ не делится на } 3$$

$$K_N = K_{N-1} + K_{N/3} \quad \text{если } N \text{ делится на } 3$$

Заполнение таблицы:

N	1	2	3	4	5	6	7	8	9	10
$K_N$	1	1	2	2	2	3	3	3	5	5

одна пустая!

$$K_2 + K_1$$

$$K_5 + K_2$$

$$K_8 + K_3$$

N	11	12	13	14	15	16	17	18	19	20
$K_N$	5	7	7	7	9	9	9	12	12	12

# Количество программ

Только точки изменения:

20

N	1	3	6	9	12	15	18	21
$K_N$	1	2	3	5	7	9	12	15

Программа:

```
K = [0] * (N+1)
K[1] = 1
for i in range(2, N+1):
    K[i] = K[i-1]
    if i % 3 == 0:
        K[i] += K[i//3]
```



# Размен монет

**Задача.** Сколькими различными способами можно выдать сдачу размером  $W$  рублей, если есть монеты достоинством  $p_i$  ( $i=1, \dots, N$ )? В наборе есть монета достоинством 1 рубль ( $p_1 = 1$ ).

## Перебор?

при больших  $N$  и  $W$  – очень долго!

## Динамическое программирование:

запоминаем решения всех задач меньшей размерности: для меньших значений  $W$  и меньшего числа монет  $N$ .





# Размен монет

**Пример:**  $W = 10$ , монеты 1, 2, 5 и 10

		0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	1										
3	5	1										
4	10	1										

**w**

базовые случаи

**i**

**$p_i$**

$T[i][w]$  – количество вариантов для суммы  $w$  с использованием  $i$  первых по счёту монет.

**Рекуррентная формула** (добавили монету  $p_i$ ):

при  $w < p_i$ :  $T[i][w] = T[i-1][w]$

без этой монеты

при  $w \geq p_i$ :  $T[i][w] = T[i-1][w] + T[i][w-p_i]$

все варианты размена остатка

# Размен монет

**Пример:**  $W = 10$ , монеты 1, 2, 5 и 10

		0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	1	1	2	2	3	3	4	4	5	5	6
3	5	1	1	2	2	3	4	5	6	7	8	10
4	10	1	1	2	2	3	4	5	6	7	8	11

**Рекуррентная формула** (добавили монету  $p_i$ ):

при  $w < p_i$ :  $T[i, w] = T[i-1][w]$

при  $w \geq p_i$ :  $T[i, w] = T[i-1][w] + T[i][w-p_i]$

# Конец фильма

---

**ПОЛЯКОВ Константин Юрьевич**

д.т.н., учитель информатики

ГБОУ СОШ № 163, г. Санкт-Петербург

[kpolyakov@mail.ru](mailto:kpolyakov@mail.ru)

**ЕРЕМИН Евгений Александрович**

к.ф.-м.н., доцент кафедры мультимедийной

дидактики и ИТО ПГГПУ, г. Пермь

[eremin@pspu.ac.ru](mailto:eremin@pspu.ac.ru)

# Источники иллюстраций

---

1. [wallpaperscraft.com](http://wallpaperscraft.com)
2. [www.mujerhoy.com](http://www.mujerhoy.com)
3. [www.pinterest.com](http://www.pinterest.com)
4. [www.wayfair.com](http://www.wayfair.com)
5. [www.zchocolat.com](http://www.zchocolat.com)
6. [www.russiantable.com](http://www.russiantable.com)
7. [www.kursachworks.ru](http://www.kursachworks.ru)
8. [ebay.com](http://ebay.com)
9. [centrgk.ru](http://centrgk.ru)
10. [www.riverstonellc.com](http://www.riverstonellc.com)
11. [53news.ru](http://53news.ru)
12. [10hobby.ru](http://10hobby.ru)
13. [ru.wikipedia.org](http://ru.wikipedia.org)
14. иллюстрации художников издательства «Бином»
15. авторские материалы