# Глава 4. Программирование (язык Python)

## Условные обозначения

 – задание выполняется на компьютере

 – задание выполняется в тетради

 – задание выполняется устно

 – задание выполняется с использованием дополнительных источников (литературы, Интернета)

Материал, который изучается только на углублённом уровне, выделен знаками

…

## § 19. Символьные строки

**Ключевые слова**:

- символьная строка
- длина строки
- сцепление строк
- подстрока
- удаление символов
- вставка символов
- поиск подстроки

### Что такое символьная строка?

Если в середине XX века первые компьютеры создавались, прежде всего, для выполнения сложных математических расчётов, то сейчас они чаще всего обрабатывают текстовую (символьную) информацию.

**Символьная строка** – это последовательность символов.

В языке Python работы со строками используется специальный тип данных *str* (от английского слова *string*), который позволяет

- работать с целой символьной строкой как с единым объектом;
- использовать строки переменной длины.

Переменная типа *str* называется *строковой* или *символьной*.

*Используя дополнительные источники, выясните значение английского слова string.*

Новое значение присваивается строковой переменной с помощью оператора присваивания:

```
s = "Вася пошёл гулять"
```

Проверить её тип можно следующей командой:

```
print( type(s) )
```

которая выведет

```
<class 'str'>
```

Для ввода значения строковой переменной с клавиатуры этого используется уже знакомая нам функция **input**:

```
s = input()
```

Встроенная функция **len** определяет длину строки – количество символов в ней. Вот так в переменную *n* записывается длина строки *s*:

```
n = len(s)
```

*Напишите полную программу, которая вводит строку с клавиатуры и выводит на экран её длину. Проверьте, как эта программа реагирует на строку с пробелами.*

## Сравнение строк

Строки можно сравнивать между собой так же, как числа. Например, можно проверить равенство двух строк:

```
password = input( "Введите пароль:" )
if password == "sEzAm":
  print( "Слушаюсь и повинуюсь!" )
else:
  print( "No pasaran!" )
```

Можно также определить, какая из двух строк больше, какая – меньше. Если строки состоят только из русских или только из латинских букв, то меньше будет та строка, которая идет раньше в алфавитном списке. Например, слово «*паровоз*» будет «меньше», чем слово «*пароход*»: они отличаются в пятой букве и «*в*» < «*х*». Это можно проверить экспериментально, например, с помощью такой программы:

```
s1 = "паровоз"
s2 = "пароход"
if s1 < s2:
  print( s1, "<", s2 )
elif s1 == s2:
  print( s1, "=", s2 )
else:
  print( s1, ">", s2 )
```

Но откуда компьютер знает, что такое алфавитный порядок? Оказывается, при сравнении используются коды символов (вспомните материал 8 класса). В современных кодировках[1] и русские, и английские буквы расположены в алфавитном порядке, то есть код буквы «*в*» меньше, чем код буквы «*х*».

*С помощью программы сравните пары слов и сделайте выводы:*

```
пар – парк          Пар – пар
steam – Пар         Steam – steam
5Steam – Steam
```

*Не используя программу, сравните пары слов:*

```
парта – парк        ПАрта – Парк
СПАМ – Spam         ПОЧТА – spam
ПО4та – ПОЧта       почТА – Post
55 – 66             9 - 128
```

## Сложение и умножение

Оператор «+» используется для «сложения» (объединения, сцепления) строк. Эта операция иногда называется *конкатенация*. Например:

```
s1 = "Привет"
s2 = "Вася"
s = s1 + ", " + s2 + "!"
```

[1] Раньше использовалась кодировка KOI8-R, в которой русские буквы стояли не по алфавиту.

*Запишите в тетради, какое значение будет иметь переменная s после выполнения этого фрагмента программы. Проверьте ответ с помощью компьютера.*

В язык Python введена операция умножения строки на число: она заменяет многократное сложение. Например,

```
s = "Уа! Уа! Уа! Уа! Уа! Уа! Уа! Уа! Уа! Уа! "
```

можно заменить на

```
s = "Уа! "*10
```

или

```
s = 10*"Уа! "
```

## Обращение к символам

В Python каждый символ строки имеет свой номер (индекс), причём нумерация, как и во многих других языках программирования (C, C++, Java), всегда начинается с нуля.

Индекс можно понимать как смещение символа от начала строки. Первый по счёту символ имеет нулевое смещение (находится в самом начале строки), поэтому его индекс – 0:

| | индексы | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|---|---|
| **hello = "** | | П | р | и | в | е | т | ! | " |

К любому символу можно обратиться по индексу, записав индекс в квадратных скобках после имени строки:

```
print( hello[1] )                    # р
print( hello[5] + hello[2] + "к" ) # тик
```

В языке Python можно указывать отрицательные индексы. Это значит, что отсчёт ведётся от конца строки, так что символ **hello[-1]** – это последний символ строки **hello**:

| | индексы | –7 | –6 | –5 | –4 | –3 | –2 | –1 | |
|---|---|---|---|---|---|---|---|---|---|
| **hello = "** | | П | р | и | в | е | т | ! | " |

Чтобы рассчитать «обычную» позицию символа в строке, к отрицательному индексу нужно добавить длину строки. Например,

```
hello[-1] = hello[len(hello)-1] = hello[6]
```

Предыдущую программу можно было переписать, используя отрицательные индексы:

```
print( hello[-6] )                    # p
print( hello[-2] + hello[-5] + "к" ) # тик
```

Если указать неправильный индекс, произойдёт ошибка – выход за границы строки, и программа завершится аварийно. Для строки из семи символов правильные индексы – от «–7» до 6.

## Перебор всех символов

Поскольку к символу можно обращаться по индексу, для перебора всех символов можно использовать цикл по переменной, которая будет принимать все возможные значения индексов.

Пусть нужно вывести в столбик коды всех символов строки с именем *s*. Длину строки можно найти с помощью функции **len**, индекс первого символа равен 0, а индекс последнего равен **len(s)-1**. Таким образом, все допустимые индексы – это последовательность, которую строит вызов функции **range(len(s))**. Перебор можно выполнить так:

```
for i in range(len(s)):
  print( s[i], ord(s[i]) )
```

В каждой строке сначала выводится сам символ, а потом – его код, который возвращает встроенная функция **ord**.

В языке Python существует еще один удобный способ перебора всех элементов строки:

```
for c in s:
  print( c, ord(c) )
```

Заголовок такого цикла можно «прочитать» так: «для всех *c*, входящих в состав *s*». Это означает, что все символы строки *s*, с первого до последнего, по очереди оказываются в переменной *c*, и в теле цикла нам остаётся вывести код символа *c*.

В отличие от большинства современных языков программирования, в Python нельзя изменить символьную строку, поскольку строка – это неизменяемый объект. Это значит, что оператор присваивания **s[5]="a"** не сработает – будет выдано сообщение об ошибке.

Тем не менее, можно составить из символов существующей строки новую строку, и внести в неё нужные изменения. Приведём полную программу, которая вводит строку с клавиатуры, заменяет в ней все буквы «*э*» на буквы «*е*» и выводит полученную строку на экран[2].

```
s = input( "Введите строку: " )
sNew = ""
for c in s:
  if c == "э": c = "е"
  sNew += c
print( sNew )
```

Здесь в цикле

```
for c in s:
  ...
```

перебираются все символы, входящие в строку *s*. В теле цикла проверяем значение переменной *c* (это очередной символ исходной строки): если оно совпадает с буквой «*э*», то заменяем его на букву «*е*»

```
if c == "э": c = "е"
```

и добавляем в конец новой строки *sNew* с помощью оператора сложения:

```
sNew += c
```

*Вспомните, чем отличается запись* `c == "э"` *от записи* `c = "э"`.

*Запишите решение этой задачи, используя цикл «пока» (while).*

## Срезы

Для того, чтобы выделить часть строки (*подстроку*), в языке Python применяется операция получения среза (англ. *slicing*). Например, `s[3:8]` означает «символы строки *s* с 3-го по 7-й» (то

[2] И. Ильф и Е. Петров в романе «Золотой теленок» описывают пишущую машинку «с турецким акцентом», в которой не хватало буквы «е», и её пришлось заменить буквой «э».

есть до 8-го, не включая его). Следующий фрагмент копирует в строку *s1* символы строки s с 3-го по 7-й (всего 5 символов):

```
s = "0123456789"
s1 = s[3:8]
```

*Запишите в тетради, какое значение будет иметь переменная s1 после выполнения этого фрагмента программы. Проверьте ответ с помощью компьютера.*

Можно использовать и отрицательные индексы – в этом случае отсчёт выполняется с конца строки:

```
s1 = s[-7:-2]  # s1 = "34567"
```

Для получения «обычной» позиции символа в строке к отрицательному значению добавляется длина строки. Например, второй индекс «–2» можно заменить на **`len(s1)-2`**.Это означает, что последние два символа не входят в срез.

Если первый индекс не указан, считается, что он равен нулю (берём начало строки), а если не указан второй индекс, то в срез включаются все символы до конца строки. Например:

```
s = "0123456789"
sFirst = s[:4]     # sFirst = "0123"
sLast = s[-4:]     # sLast = "6789"
```

Срезы позволяют легко выполнить *реверс строки* (переставить её символы в обратном порядке):

```
sReversed = s[::-1]
```

Так как начальный и конечный индексы элементов строки не указаны, задействована вся строка. Число «–1» означает шаг изменения индекса и говорит о том, что все символы перебираются в обратном порядке.

*Используя только операции среза и «сложения» строк, постройте из строки*

```
s = 'информатика'
```

*как можно больше слов русского языка. Постарайтесь использовать наименьшее возможное число операций. Проверьте ваши решения с помощью программы.*

*Приведите несколько способов построения строки* **"А. Семёнов"** *из строки*

```
s = "Семёнов Андрей"
```

*Какой из них лучше? Как вы сравнивали эти способы?*

## Встроенные методы

В Python существует множество встроенных алгоритмов для работы с символьными строками. Многие из них вызываются с помощью точечной записи, они называются *методами* обработки строк. Например, методы **upper** и **lower** позволяют перевести строку соответственно в верхний и нижний регистр:

```
s = "aAbBcC"
sUp = s.upper()    # sUp = "AABBCC"
sLow = s.lower()   # sLow = "aabbcc"
```

Слева от точки записывается имя строки (или сама строка в кавычках), к которой нужно применить метод, а справа от точки – название метода. Например, возможна такая запись:

```
sWow = "Wow!".upper()     # "WOW!"
```

Здесь метод **upper** применяется к строке «Wow!».

Методы строк мы уже использовали, когда выводили данные на экран с помощью метода **format**:

```
a = 5
b = 4
print( "{}+{}={}".format(a,b,a+b) )
```

Ещё один метод, **isdigit**, проверяет, все ли символы строки – цифры, и возвращает логическое значение:

```
s = "ab1c"
print( s.isdigit() )  # False
s = "123"
print( s.isdigit() )  # True
```

Полезный метод **strip** (по-английски – лишать) удаляет пробелы в начале и в конце строки:

```
sRaw = "   Python & C++    "
sClear = sRaw.strip()  # sClear = "Python & C++"
```

Это метод удобно использовать для обработки ввода пользователя, когда пробелы в начале и в конце строки не нужны.

О других встроенных методах обработки строк вы можете узнать в литературе или в Интернете.

### Удаление и вставка

Для удаления части строки нужно составить новую строку, объединив части исходной строки до и после удаляемого участка:

```
s = "0123456789"
s = s[:3] + s[9:]
```

*Запишите в тетради, какое значение будет иметь переменная s после выполнения этого фрагмента программы. Проверьте ответ с помощью компьютера.*

Срез **s[:3]** означает «от начала строки до символа **s[3]**, не включая его», а запись **s[9:]** – «все символы, начиная с **s[9]** до конца строки». Таким образом, в переменной *s* остаётся значение «0129».

С помощью срезов и сцепления строк можно также вставить новый фрагмент внутрь строки:

```
s = "0123456789"
s = s[:3] + "ABC" + s[3:]
```

*Запишите в тетради, какое значение будет иметь переменная s после выполнения этого фрагмента программы.*

### Поиск в символьных строках

В Python существует метод для поиска подстроки (и отдельного символа) в символьной строке, он называется **find** (по-английски – найти). В скобках нужно указать образец для поиска, это может быть один символ или символьная строка:

```
s = "Здесь был Вася."
n = s.find( "с" )        # n = 3
if n >= 0:
```

```
    print( "Номер символа", n )
else:
    print( "Символ не найден." )
```

Метод **find** возвращает целое число – индекс символа, с которого начинается образец (буква «*c*») в строке *s*. Если образец в строке встречается несколько раз, функция находит первый из них. В рассмотренном примере в переменную *n* будет записано число 3.

*Выясните экспериментально, какое значение возвращает метод* **find**, *если образец для поиска не найден в строке.*

Аналогичный метод **rfind** (от англ. *reverse find* – искать в обратную сторону) ищет последнее вхождение образца в строку. Для той же строки *s*, что и в предыдущем примере, метод **rfind** вернёт 12 (индекс последней буквы «*c*»):

```
s = "Здесь был Вася."
n = s.rfind( "c" )      # n = 12
```

*Как можно найти вторую букву «с» с начала строки?*

*Вводится строка, в которой сначала записана фамилия человека, а затем через пробел – его имя, например,* **"Семёнов Андрей"**. *Запишите операторы, которые позволяют*:

- *найти номер пробела, разделяющего фамилию и имя, и записать его в переменную p;*
- *выделить из строки фамилию и записать её в переменную fam;*
- *выделить из строки имя и записать его в переменную name;*
- *приписать перед фамилией первую букву имени, точку и пробел.*

## Преобразования «строка↔число»

Пусть символьная строка содержит запись числа. С таким значением нельзя выполнять арифметические операции, потому что это символы, а не число.

*Чему будут равны значения переменных n и s после выполнения этих команд?*

```
n = 12 + 34
s = '12' + '34'
```

Для того чтобы с числовыми данными можно было выполнять вычисления, нужно цепочку символов в числовое значение. В языке Python есть встроенные функции для преобразования типов данных, некоторые из них мы уже использовали:

**int** – переводит строку в целое число;

**float** – переводит строку в вещественное число;

**str** – переводит целое или вещественное число в строку.

Приведём пример преобразования строк в числовые значения:

```
s = "123"
n = int( s )      # n = 123
s = "123.456"
x = float( s )    # x = 123.456
```

Если строку не удалось преобразовать в число (например, если в ней содержатся буквы), возникает ошибка и выполнение программы завершается аварийно.

*Какие из этих строк можно преобразовать в целое число, какие – в вещественное:*

```
"45"        "5 p."        "14.5"      "14;5"
"tu154"     "543.0"       "(30)"      "1E3"
```

Теперь покажем примеры обратного преобразования:

```
n = 123
s = str( n )    # s = "123"
x = 123.456
s = str( x )    # s = "123.456"
```

Эти операции всегда завершаются успешно, ошибки быть не может.

Функция **str** использует правила форматирования, установленные по умолчанию. При необходимости можно использовать собственное форматирование, например:

```
n = 123
s = "{:5}".format(n)    # s = "◦◦123"
```

Здесь значение переменной *n* записано в 5 позициях, то есть в начале строки будут добавлены два пробела.

Для вещественных чисел можно использовать форматы *f* (с фиксированной запятой) и *e* (научный формат, с плавающей запятой):

```
x = 123.456
s = "{:7.2f}".format(x)    # s = "◦123.46"
s = "{:10.2e}".format(x)   # s = "◦◦1.23e+02"
```

Формат «**7.2f**» обозначает «вывести в 7 позициях с двумя знаками в дробной части», а формат «**10.2e**» – «в научном формате в 10 позициях с двумя знаками в дробной части».

**Практическая работа №12. Посимвольная обработка строк**

**Практическая работа №13. Обработка строк. Функции**

**Практическая работа №14. Преобразования «строка↔число»**

**Выводы:**

- Символьная строка – это последовательность символов.
- Длина строки – это количество символов в строке.
- Подстрока – это часть символьной строки.
- При обращении к отдельному символу строки его номер записывают в квадратных скобках. Нумерация символов в строке в языке Python начинается с нуля.
- Знак «+» при работе со строками означает объединение (конкатенацию) строк.

- Для обработки символьных строк используют встроенные функции стандартной библиотеки.
- Функция поиска подстроки возвращает номер символа, с которого начинается подстрока, или –1 в случае неудачи.
- Строку можно преобразовать в число для того, чтобы затем выполнять с ним вычисления. Число можно преобразовать в символьную строку.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Во многих языках (в том числе и в Python) можно использовать массивы символов, то есть массивы, каждый элемент которых – один символ. Чем отличается строка от массива символов?
2. Чем отличается действие оператора «+» для чисел и для символьных строк?
3. Как определить, что при поиске в строке образец не найден?
4. Как бы вы искали первый символ «с» с конца строки, если бы не было метода *rfind*?

## Задачи

1. Напишите программу, которая заменяет в символьной строке все точки на нули и все буквы **X** на единицы. Например, из строки '**..XX..X.**' должна получиться строка '00110010'.
2. Напишите программу, которая выполняет инверсию битов в символьной строке: заменяет в ней все нули на единицы и наоборот. Например, из строки «00110010» должна получиться строка «11001101».
3. Введите битовую строку и дополните её последним битом, который должен быть равен 0, если в исходной строке чётное число единиц, и равен 1, если нечётное (в получившейся строке должно всегда быть чётное число единиц). Например, из строки «00110010» должна получиться строка «001100101».

4. Напишите программу, которая принимает символьную строку, содержащую фамилию и имя (они разделены одним пробелом). Нужно построить новую строку, в которой записан инициал (первая буква имени с точкой) и через пробел – фамилия.
5. Напишите программу, которая принимает строку, содержащую фамилию, имя и отчество человека (каждая пара слов разделена одним пробелом). Нужно построить новую строку, в которой записаны инициалы (первые буквы имени и отчества с точками после них) и через пробел – фамилия.
6. Напишите программу, которая вводит адрес файла и «разбирает» его на части, разделенные знаком «**/**». Каждую часть нужно вывести в отдельной строке.

7. Напишите программу, которая определяет количество слов в строке. Слова разделены пробелами, причём в начале строки, в конце строки и между словами может быть сколько угодно пробелов.
8. Напишите программу, которая принимает символьную строку и проверяет, является ли она перевёртышем (слово-перевёртыш или *палиндром* читается одинаково в обоих направлениях, например, слово «казак»).
9. Напишите программу, которая «переворачивает» введённое слово, то есть переставляет буквы так, чтобы первая буква стала последней, вторая – предпоследней и т.д.
10. Напишите программу, которая вычисляет сумму двух натуральных чисел, записанную в виде символьной строки, например, «1+25».
11. Напишите программу, которая вычисляет сумму трёх натуральных чисел, записанную в виде символьной строки, например, «1+25+56».
12. *Напишите программу, которая вычисляет сумму неизвестного количества натуральных чисел, записанную в виде символьной строки, например, «1+25+12+34+89».

# § 20. Обработка массивов

**Ключевые слова**:

- массив
- перестановка элементов
- выход за границы массива
- линейный поиск
- сортировка

Современные компьютеры работают с огромными массивами данных. При этом размер программы не зависит от размера массива: можно написать очень короткую программу, которая обрабатывает миллиарды чисел.

Из курса 8 класса вы уже знаете, как найти сумму элементов массива и его максимальный (минимальный) элемент, определить количество элементов, удовлетворяющих условию. В этом параграфе мы изучим некоторые более сложные алгоритмы.

*Вспомните и запишите в тетрадь, как*

– *выделить место в памяти под массив из N элементов;*
– *заполнить массив нулями;*
– *заполнить массив натуральными числами от 1 до N;*
– *заполнить массив случайными числами из отрезка* [50,100]*;*
– *найти сумму всех элементов массива;*
– *найти сумму чётных элементов массива;*
– *найти количество отрицательных элементов массива;*
– *найти максимальный элемент массива.*

## Перестановка элементов массива

Во многих задачах нужно переставлять элементы массива, то есть требуется поменять местами значения двух ячеек памяти.

*Представьте себе, что в кофейной чашке налит сок, а в стакане – кофе, и вы хотите, чтобы было наоборот. Что вы сделаете?*

Вернёмся к программированию. Чтобы поменять местами значения двух переменных, $a$ и $b$, нужно использовать третью переменную того же типа:

```
c = a
a = b
b = c
```

Перестановка двух элементов массива, например, $A[i]$ и $A[k]$, выполняется так же:

```
c = A[i]
A[i] = A[k]
A[k] = c
```

Кроме того, в языке Python есть красивый приём перестановки без использования вспомогательной переменной:

```
A[i], A[k] = A[k], A[i]
```

Рассмотрим несколько задач, в которых требуется переставлять элементы массива.

**Задача 1.** Массив $A$ содержит чётное количество элементов $N$. Нужно поменять местами пары соседних элементов: первый со вторым, третий – с четвёртым и т.д. (Рис. 4.1).

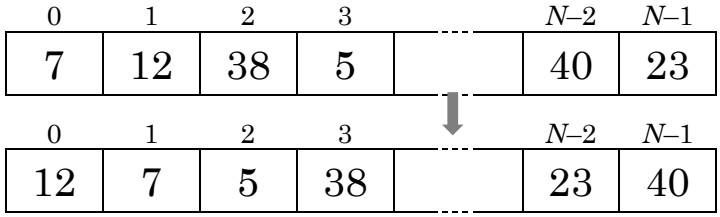



Рис. 4.1.

*С каким элементом нужно поменять местами элемент $A[i]$? Зависит ли ваш ответ от свойств числа $i$?*

*Сергей написал такой алгоритм для решения задачи* 1*:*

```
for i in range(N):
    # поменять местами A[i] и A[i+1]
```

*Выполните его вручную этот алгоритм для массива* [1, 2, 3, 4] *($N$ = 4). Объясните результат. Найдите ошибки в алгоритме Сергея.*

Обратите внимание, что при выполнении алгоритма Сергея на последнем шаге цикла мы будем менять местами эле-

мент $A[N–1]$ с несуществующим элементом $A[N]$. Это вызовет ошибку, которая называется «выход за границы массива». Но даже если использовать диапазон **`range(N-1)`**, программа все равно будет работать неправильно, то есть приведёт к неверному решению задачи 1. Получится, что первый элемент просто «переедет» на место последнего.

Для того чтобы исправить программу, нужно изменять переменную $i$ с шагом 2:

```
N = len(A)
for i in range(0,N-1,2):
  A[i], A[i+1] = A[i+1], A[i]
print( *A )
```

*Предложите другое решение этой задачи, записав нужные операторы в теле цикла while.*

```
i = 0
while i < N-1:
  ...
```

## Реверс массива

**Задача 2.** Требуется выполнить *реверс массива*, то есть переставить элементы массива в обратном порядке, так чтобы первый элемент стал последним, а последний – первым:

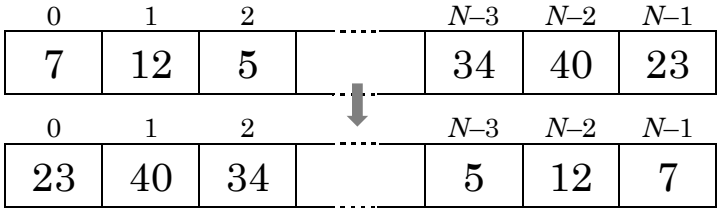



Рис. 4.2.

*С каким элементом нужно поменять местами элемент $A[0]$? элемент $A[1]$? элемент $A[i]$?*

*Если индекс первого элемента в паре увеличивается на 1, а индекс второго элемента уменьшится на 1, что можно сказать о сумме индексов этих элементов?*

В этой задаче сумма индексов элементов, участвующих в обмене, для всех пар равна $N - 1$, поэтому элемент с номером $i$ должен меняться местами с $(N - 1 - i)$-м элементом.

*Выполните вручную следующий алгоритм для массива* [1, 2, 3, 4] ($N = 4$). *Объясните результат. Найдите ошибки в этом алгоритме.*

```
for i in range(N):
  # поменять местами A[i] и A[N-1-i]
```

Для того, чтобы не выполнять реверс дважды, нужно остановить цикл на середине массива:

```
for i in range( N // 2 ):
  # поменять местами A[i] и A[N-1-i]
```

*Запишите в тетради операторы, которые нужно добавить в тело последнего цикла.*

*Запишите в тетради другое решение задачи реверса массива, которое использует цикл с условием (while).*

## Линейный поиск в массиве

Очень часто требуется найти в массиве заданное значение или определить, что его там нет. Для этого нужно просмотреть все элементы массива с первого до последнего. Как только найден элемент, равный заданному значению $X$, надо завершить поиск и вывести результат. Такой алгоритм поиска называется *линейным поиском*.

*Сколько операций сравнения придётся выполнить, если в массиве $N$ элементов? В каком случае количество сравнений будет наименьшим? наибольшим?*

*Катя торопилась и написала такой алгоритм линейного поиска:*

```
i = 0
while A[i] != X:
  i += 1
print( "A[{}]={}".format(i,X) )
```

*Проверьте, правильно ли сработает алгоритм, если искать в массиве* [1, 2, 3] *число* 2*? число* 4*?*

Этот алгоритм хорошо работает, если нужный элемент в массиве есть, однако приведет к ошибке, если такого элемента нет – получится зацикливание и выход за границы массива. Чтобы этого не произошло, в условие после **while** нужно добавить еще одно ограничение: $i < N$. Если после окончания цикла это условие нарушено, значит, поиск был неудачным – элемента нет:

```
i = 0
while i < N and A[i] != X:
  i += 1
if i < N:
  print( "A[{}]={}".format(i,X) )
else:
  print( "Не нашли!" )
```

Отметим одну тонкость. При $i \ge N$ элемента $A[i]$ в массиве нет. В этом случае проверка условия $A[i]$ != $X$ приведёт к *выходу за границы массива*, и программа завершится аварийно.

К счастью, этого можно избежать. Если первая часть сложного условия с операцией **and** ложна, то вторая часть не проверяется[3] – уже понятно, что всё условие тоже ложно. Поэтому в сложном условии

```
i < N and A[i]!= X
```

сначала нужно записать именно отношение $i < N$. Если это условие ложно, цикл сразу завершится.

Возможен ещё один поход к решению этой задачи. Используя цикл с переменной, можно перебрать все элементы массива и досрочно завершить цикл, если найдено требуемое значение.

```
nX = -1
for i in range(N):
  if A[i] == X:
```

[3] Во многих современных языках (например, в Python, C++, Javascript, PHP) такое поведение гарантировано стандартом.

```
      nX = i
      break
if nX >= 0:
  print( "A[{}]={}".format(nX,X) )
else:
  print( "Не нашли!" )
```

Для выхода из цикла используется оператор **break**, индекс найденного элемента сохраняется в переменной *nX*. Если её значение осталось равным «–1» (не изменилось в ходе выполнения цикла), то в массиве нет элемента, равного *X*.

Последний пример можно упростить, используя особые возможности цикла **for** в языке Python:

```
for i in range(N):
  if A[i] == X:
    print( "A[{}]={}".format(i,X) )
    break
else:
  print( "Не нашли!" )
```

Здесь мы выводим результат сразу, как только нашли нужный элемент, а не после цикла. Слово **else** после цикла **for** начинает блок, который выполняется при нормальном завершении цикла (без применения **break**). Таким образом, сообщение «Не нашли!» будет выведено только тогда, когда условие в операторе **if** ни разу не выполнилось.

В языке Python возможен и другой способ решения этой задачи, использующий встроенные методы для массивов (списков). Оператор **in** определяет, есть ли нужный элемент в массиве. Он возвращает логическое значение (*True* или *False*):

```
if X in A:
  print( "Нашли!" )
else:
  print( "Не нашли!" )
```

А метод **index** возвращает индекс первого найденного элемента, равного *X*:

```
nX = A.index(X)
```

Поэтому поиск элемента, равного *X*, и определение его индекса можно записать так:

```
if X in A:
  nX = A.index(X)
  print( "A[{}]={}".format(i,X) )
else:
  print( "Не нашли!" )
```

## Сортировка массивов

**Сортировка** – это расстановка элементов списка (массива) в заданном порядке.

Возникает естественный вопрос: «зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту облегчает поиск нужной информации.

Для чисел обычно используют сортировку *по возрастанию* (каждый следующий элемент больше предыдущего) или *убыванию* (следующий элемент меньше предыдущего). Если в массиве есть одинаковые элементы, их можно расставить по неубыванию (каждый следующий элемент не меньше предыдущего) или невозрастанию.

Математики и программисты изобрели множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие (на больших массивах) и 2) сложные, но быстрые.

Мы изучим один из простых методов, который называется *методом выбора*. Для примера будем рассматривать сортировку по возрастанию (неубыванию).

*Предположим, что мы нашли минимальный элемент массива. Где он должен размещаться в отсортированном массиве?*

*Запишите в тетради фрагмент программы для поиска номера минимального элемента массива.*

Для того чтобы поставить на место минимальный элемент, мы просто поменяем его местами с первым элементом массива.

*Запишите в тетради фрагмент программы, который меняет местами элементы A[0] и A[nMin].*

После того как мы установили на место первый (минимальный) элемент, находим минимальный из оставшихся и ставим его на второе место, и т.д. Полный алгоритм записывается в виде вложенного цикла:

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

Фоном выделен поиск номера минимального элемента в той части массива, которая начинается с элемента $A[i]$.

На первом шаге внешнего цикла значение $i$ равно 0, и мы ставим на место первый элемент, $A[0]$. На следующем шаге $i = 1$, то есть мы ищем минимальный элемент среди всех, кроме самого первого, и т.д.

Обратите внимание, что внешний цикл выполняется $N - 1$ раз (а не $N$). Этого достаточно, потому что если $N - 1$ элементов стоят на своих местах, то последний тоже стоит на своём месте (другого свободного места для него нет).

*Какой результат можно гарантировать, если внешний цикл (по переменной i) выполнить только 1 раз? только 3 раза?*

**Практическая работа №15. Перестановка элементов массива**

**Практическая работа №16. Линейный поиск в массиве**

**Практическая работа №17. Сортировка**

### Выводы:

- Выход за границы массива – это обращение к элементу массива с несуществующим индексом.
- Линейный поиск – это перебор всех элементов массива до тех пор, пока не будет найден нужный элемент или не закончится массив.
- Сортировка – это расстановка элементов списка (массива) в заданном порядке. Данные сортируют для того, чтобы ускорить последующий поиск.
- Для чисел обычно используют сортировку по возрастанию (неубыванию) или убыванию (невозрастанию).

*Нарисуйте интеллект-карту этого параграфа.*

### Вопросы и задания

1. Что такое выход за границы массива? Что опаснее – чтение или запись данных за границами массива?
2. На какой идее основан метод сортировки выбором?
3. Объясните, зачем нужен вложенный цикл в алгоритме сортировки.
4. Как нужно изменить программу сортировки, чтобы элементы массива были отсортированы по убыванию?

### Задачи

1. В переменных записаны значения $a = 1$, $b = 2$ и $c = 3$. Как изменятся значения переменных после выполнении алгоритма:

```
c = a
b = a
a = c
```

   Исправьте один символ так, чтобы получился правильный алгоритм обмена значений переменных $a$ и $b$.
2. Что произойдет с массивом [1, 2, 3, 4] ($N = 4$) при выполнении следующего фрагмента программы:

```
for i in range(N-1):
  A[i] = A[i+1]
```

3. Что произойдет с массивом [1, 2, 3, 4] ($N$ = 4) при выполнении следующего фрагмента программы:

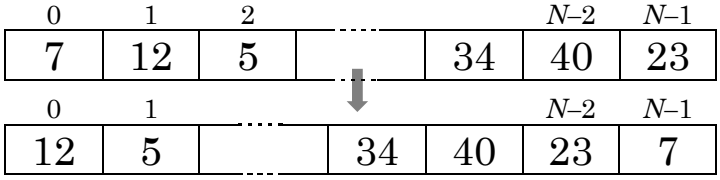
```
for i in range(N-1):
  A[i+1] = A[i]
```

4. Напишите программу, которая меняет местами пары соседних элементов, кроме первого и последнего.
5. Напишите программу, которая выполняет реверс отдельно первой и второй половин массива. При изменении значения $N$ на другое чётное натуральное число программа должна работать правильно без дополнительных изменений.
6. Напишите программу, которая сортирует массив в порядке убывания.
7. Напишите программу, которая сортирует массив по возрастанию последней цифры числа.
8. Напишите программу сортировки массива по возрастанию, в которой на каждом шаге ищется максимальный элемент.

9. Что произойдет с массивом [1, 2, 3, 4, 5, 6] ($N$ = 6) при выполнении следующего фрагмента программы:

```
i = 0
while i < N-1:
  c = A[i]
  A[i] = A[i+1]
  A[i+1] = A[i+2]
  A[i+2] = c
  i = i + 3
```

10. Напишите программу, которая выполняет циклический сдвиг массива влево. При этом элемент $A[1]$ переходит на место $A[0]$, $A[2]$ – на место $A[1]$ и т.д., а элемент $A[0]$ «переезжает» на место последнего:



11. Напишите программу, которая выполняет циклический сдвиг массива вправо.
12. Заполните массив случайными числами и выполните реверс для части массива между элементами с индексами *K* и *M* (включая эти элементы). Значения *K* и *M* вводятся с клавиатуры.
13. Напишите программу, которая находит максимальный и минимальный из чётных положительных элементов массива. Если в массиве нет чётных положительных элементов, нужно вывести сообщение об этом.
14. Введите массив с клавиатуры и найдите количество элементов, имеющих максимальное значение.
15. Напишите программу, которая находит индекс первого элемента массива, равного введённому числу *X*. Программа должна вывести ответ «не найден», если в массиве таких элементов нет.
16. Напишите программу, которая находит индекс *последнего* элемента массива, равного введённому числу *X*.
17. Напишите программу, которая находит индексы всех элементов массива, равных введённому числу *X*.
18. Напишите программу, которая выполняет неполную сортировку массива: ставит в начало массива *K* самых меньших по величине элементов в порядке возрастания (неубывания). Число *K* вводится с клавиатуры.
19. *Как, используя операции сложения и вычитания, поменять местами значения двух ячеек памяти без использования третьей переменной?
20. *Напишите программу, которая сортирует массив по убыванию суммы цифр числа.
21. *Напишите программу, которая сортирует первую половину массива по возрастанию, а вторую – по убыванию (элементы из первой половины не должны попадать во вторую и наоборот).

22. *Напишите программу, которая сортирует массив, а затем находит количество различных чисел в нём.
23. *Напишите программу, которая сортирует массив, а затем находит максимальное из чисел, встречающихся в массиве несколько раз.

### Темы сообщений:

а) «Сортировка методом пузырька»
б) «Сортировка методом вставки»

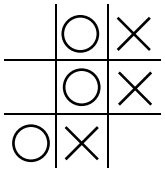
## § 21. Матрицы (двумерные массивы)

**Ключевые слова**:

- матрица
- строка
- столбец
- главная диагональ
- побочная диагональ

### Что такое матрицы?

Многие программы работают с данными, организованными в виде таблиц. Например, при составлении программы для игры в крестики-нолики нужно запоминать состояние каждой клетки квадратной доски. Можно поступить так: пустым клеткам присвоить код «–1», клетке, где стоит нолик – код 0, а клетке с крестиком – код 1. Тогда информация о состоянии поля может быть записана в виде таблицы:



| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | -1 | 0 | 1 |
| 1 | -1 | 0 | 1 |
| 2 | 0 | 1 | -1 |

Рис. 4.3.

Такие таблицы называются *матрицами* или двухмерными массивами.

**Матрица** — это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.).

Каждый элемент матрицы, в отличие от обычного (линейного) массива, имеет два индекса – номер *строки* и номер *столбца*. Это похоже на координаты пикселя растрового рисунка или точки на плоскости. На Рис. 4.3 фоном выделен элемент, находящийся на пересечении строки 1 и столбца 2, он обозначается в программе как *A*[1][2]. Нумерация строк и столбцов, как и индексов одномерных массивов, в языке Python начинается с нуля.

Матрицу часто называют двумерным массивом, потому что каждый элемент матрицы имеет два индекса – номера строки и столбца.

*Определите значения элементов A*[0][1]*, A*[1][0] *и A*[2][3] *на Рис. 4.3.*

## Создание матрицы

Поскольку в Python нет массивов, то нет и матриц в классическом понимании. Для того, чтобы работать с таблицами, используют списки. Двухмерная таблица хранится как «список списков» – список, каждый элемент которого тоже представляет собой список. Например, таблицу, показанную на Рис. 4.3, можно записать так:

```
A = [[-1, 0, 1],
     [-1, 0, 1],
     [0, 1, -1]]
```

или в одну строчку:

```
A = [[-1, 0, 1], [-1, 0, 1], [0, 1, -1]]
```

Конечно, первый способ более нагляден.

Иногда нужно создать в памяти матрицу заданного размера, заполненную некоторыми начальными значениями, например, нулями. Первая мысль – применить такой алгоритм, использующий операцию повторения «*»:

```
N = 3
```

```
M = 2
# неверное создание матрицы!
row = [0]*M  # создаём список-строку длиной M
A = [row]*N  # создаём массив (список) из N строк
```

Однако этот способ работает неверно из-за особенностей языка Python. Например, если после этого выполнить присваивание

```
A[0][0] = 1
```

мы увидим, что *все* элементы столбца 0, то есть **A[0][0]**, **A[1][0]** и т.д. стали равны 1. Дело в том, что матрица – это список ссылок на списки-строки (список адресов строк). При выполнении оператора

```
row = [0]*M
```

транслятор создаёт в памяти одну единственную строку, а затем следующий оператор

```
A = [row]*N
```

устанавливает на эту единственную строку все ссылки в массиве *A* (Рис. 4.4).

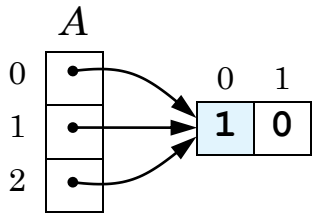



Рис. 4.4.

Естественно, что когда мы меняет элемент с индексом 0 в строке 0 (он выделен фоном на Рис. 4.4), меняются и все элементы с индексом 0 во всех строках.

Для создания полноценной матрицы нам нужно как-то заставить транслятор создать все строки в памяти как разные объекты. Для этого сначала построим пустой список, а потом будем в цикле добавлять к нему (с помощью метода **append**) новые строки, состоящие из нулей:

```
A = []
for i in range(N):
  A.append( [0]*M )
```

или так, с помощью генератора:

```
A = [ [0]*M for i in range(N) ]
```

Теперь все строки расположены в разных областях памяти (Рис. 4.5).

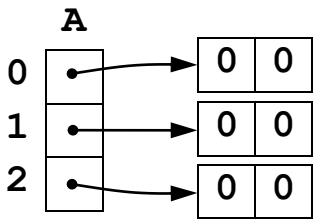



Рис. 4.5.

Каждому элементу матрицы можно присвоить любое значение. Поскольку индексов два, для заполнения матрицы или обработки всех её элементов нужно использовать вложенный цикл. В одном (обычно – во внешнем) цикле изменяется индекс строки, а во втором – индекс столбца.

Далее будем считать, что существует матрица $A$, состоящая из $N$ строк и $M$ столбцов, а $i$ и $j$ – целочисленные переменные, обозначающие индексы строки и столбца. В этом примере матрица заполняется случайными числами из отрезка [20; 80]:

```
from random import randint
A = [ [0]*M for i in range(N) ]
for i in range(N):
  for j in range(M):
    A[i][j] = randint( 20, 80 )
```

*Запишите в тетради фрагмент программы, который вычисляет сумму всех элементов матрицы в переменной* s.

*Запишите в тетради фрагмент программы, который вычисляет количество ненулевых элементов матрицы в переменной* k.

## Вывод матрицы на экран

Простейший способ вывода матрицы – с помощью одного вызова функции **print**:

```
print ( A )
```

В этом случае матрица выводится в одну строку, что не очень удобно. Поскольку человек воспринимает матрицу как таблицу, лучше и на экран выводить её в виде таблицы. Для этого можно написать такую процедуру (в классическом стиле):

```
def printMatrix( A ):
  for i in range(len(A)):
    for j in range(len(A[i])):
      print( "{:4d}".format(A[i][j]), end="" )
    print()
```

Здесь *i* – это номер строки, а *j* – номер столбца; **len(A)** – это число строк в матрице, а **len(A[i])** – число элементов в строке *i* (совпадает с числом столбцов).

После вывода значения очередного мы не делаем перевод строки на экране (**end=""**), а после вывода всей строки – делаем с помощью вызова функции **print** без аргументов.

Можно написать такую функцию и в стиле Python:

```
def printMatrix ( A ):
  for row in A:
    for x in row:
      print( "{:4}".format(x), end="" )
    print()
```

Первый цикл перебирает все строки в матрице; каждая из них по очереди попадает в переменную-массив *row*. Затем внутренний цикл перебирает все элементы этого массива (строки) и выводит их на экран.

## Перебор элементов матрицы

Каждый элемент матрицы имеет два индекса, поэтому для перебора всех элементов нужно использовать вложенный цикл. Обычно внешний цикл перебирает индексы строк, а внутренний – индексы столбцов. Вот так можно найти сумму всех элементов матрицы:

```
summa = 0
for i in range(N):
  for j in range(M):
```

```
        summa += A[i][j]
    print(summa)
```

Эту задачу можно красиво решить в стиле Python:

```
summa = 0
for row in A:
  summa += sum(row)
print(summa)
```

Здесь в цикле перебираются все строки матрицы *A*, каждая из них по очереди записывается в переменную *row*. В теле цикла сумма элементов очередной строки прибавляется к значению переменной *summa*.

## Квадратные матрицы

На практике (например, при решении шахматных задач) часто приходится работать с *квадратными матрицами*, у которых количество строк и количество столбцов одинаковые.

Для квадратной матрицы используют понятия «*главная диагональ*» (серые клетки на Рис. 4.6, *а*) и «*побочная диагональ*» (Рис. 4.6, *б*). На Рис. 4.6, *в* выделена главная диагональ и все элементы под ней.

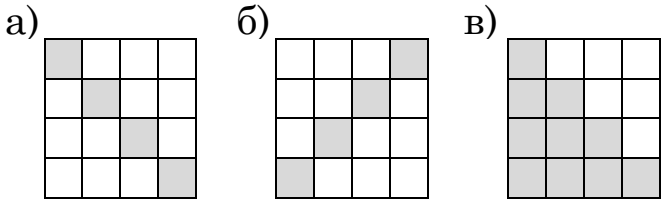



Рис. 4.6.

Главная диагональ – это элементы *A*[0][0], *A*[1][1], …, *A*[*N*–1][*N*–1], то есть элементы, у которых номер строки равен номеру столбца. Для перебора этих элементов достаточно одного цикла:

```
for i in range(N):
  # работаем с A[i][i]
```

Элементы побочной диагонали – это *A*[0][*N*–1], *A*[1][*N*–2], …, A[*N*–1][0]. Заметим, что сумма номеров строки и столбца для

каждого элемента равна $N-1$, поэтому получаем такой цикл перебора:

```
for i in range(N):
  # работаем с  A[i][N-1-i]
```

С помощью генератора легко выделить элементы главной диагонали в отдельный массив:

```
D = [ A[i][i] for i in range(N) ]
```

*Запишите в тетради фрагмент программы, который вычисляет сумму всех элементов главной диагонали квадратной матрицы в переменной s.*

*Запишите в тетради фрагмент программы, который вычисляет количество ненулевых элементов побочной диагонали матрицы в переменной k.*

Для обработки всех элементов на главной диагонали и под ней (Рис. 4.6, *в*) нужен вложенный цикл: номер строки будет меняться от 0 до $N-1$, а номер столбца для каждой строки $i$ – от 0 до $i$:

```
for i in range(N):
  for j in range(i+1):
    # работаем с  A[i][j]
```

*Запишите в тетради фрагмент программы, который вычисляет среднее арифметическое элементов матрицы, находящихся на главной диагонали и под ней.*

Чтобы переставить столбцы матрицы, достаточно одного цикла. Например, переставим столбцы с индексами 2 и 4, используя вспомогательную переменную *temp*:

```
for i in range(N):
  temp = A[i][2]
  A[i][2] = A[i][4]
  A[i][4] = temp
```

или, используя множественно присваивание в Python:

```
for i in range(N):
  A[i][2], A[i][4] = A[i][4], A[i][2]
```

Переставить две строки можно вообще без цикла, учитывая, что $A[i]$ – это ссылка на список элементов строки $i$. Поэтому достаточно просто переставить ссылки. Оператор

```
A[0], A[3] = A[3], A[0]
```

переставляет строки матрицы с индексами 0 и 3.

Для того чтобы создать копию строки с индексом $i$, нельзя делать так:

```
R = A[i] # это неверно!
```

потому что при этом мы получим новую ссылку на существующую строку. Вместо этого нужно создать копию в памяти с помощью среза, включающего все элементы строки:

```
R = A[i][:]
```

Построить копию столбца с индексом $j$ несколько сложнее, так как матрица расположена в памяти по строкам. В этой задаче удобно использовать генератор:

```
C = [ row[j] for row in A ]
```

В цикле перебираются все строки матрицы $A$, они по очереди попадают в переменную-массив *row*. Генератор выбирает из каждой строки элемент с индексом $j$ и составляет список из этих значений.

### Практическая работа №18. Матрицы

## Выводы:

- Матрица — это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.).
- Каждый элемент матрицы имеет два индекса – номера строки и столбца.
- Главная диагональ квадратной матрицы – это элементы, у которых индекс строки равен индексу столбца.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Сравните понятия «массив» и «матрица».

2. Как вы думаете, можно ли считать, что первый индекс элемента матрицы – это номер столбца, а второй – номер строки?
3. Почему суммирование элементов главной диагонали требует одиночного цикла, а суммирование элементов под главной диагональю – вложенного?

## Задачи

1. Напишите программу, которая находит максимальный элемент на главной диагонали квадратной матрицы.
2. Напишите программу, которая находит максимальный элемент матрицы и его индексы (номера строки и столбца).
3. *Напишите программу, которая находит минимальный из чётных положительных элементов матрицы. Учтите, что таких элементов в матрице может и не быть.
4. *Напишите программу, которая выводит на экран строку матрицы, сумма элементов которой наибольшая.
5. *Напишите программу, которая выводит на экран столбец матрицы, сумма элементов которого наименьшая.
6. Напишите программу, которая заполняет матрицу из $N$ строк и $M$ столбцов нулями и единицами в шахматном порядке.
7. Напишите программу, которая заполняет матрицу из $N$ строк и $N$ столбцов нулями и единицами так, что все элементы выше главной диагонали равны нулю, а остальные – единице.
8. Напишите программу, которая заполняет матрицу из $N$ строк и $N$ столбцов нулями и единицами так, что все элементы выше побочной диагонали равны нулю, а остальные – единице.
9. *Заполните матрицу, содержащую $N$ строк и $M$ столбцов, натуральными числами по спирали и змейкой, как на рисунках:

а)

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 10 | 11 | 12 | 5 |
| 9 | 8 | 7 | 6 |

б)

| | | | |
|---|---|---|---|
| 1 | 3 | 4 | 9 |
| 2 | 5 | 8 | 10 |
| 6 | 7 | 11 | 12 |

в)

| | | | |
|---|---|---|---|
| 1 | 6 | 7 | 12 |
| 2 | 5 | 8 | 11 |
| 3 | 4 | 9 | 10 |

10. **Напишите программу, которая играет с человеком в крестики-нолики на поле 3×3.

**Темы сообщений:**

«Игра "Жизнь"»

# § 22. Сложность алгоритмов

**Ключевые слова**:

- временная сложность
- пространственная сложность
- время работы алгоритма
- асимптотическая сложность
- линейная сложность
- квадратичная сложность

## Как сравнивать алгоритмы?

Чаще всего для решения задачи известно несколько алгоритмов. Поэтому возникает вопрос – как выбрать лучший? И ещё один важный вопрос – способен ли современный компьютер за приемлемое время найти решение задачи? Например, в игре в шахматы возможно лишь конечное количество позиций и, значит, только конечное количество различных партий. Значит, теоретически можно перебрать все возможные партии и выяснить, кто побеждает при правильной игре – белые или черные. Однако количество вариантов настолько велико, что современные компьютеры не могут выполнить такой перебор за приемлемое время.

Что мы хотим от алгоритма? Во-первых, чтобы он работал как можно быстрее. Во-вторых, чтобы объем необходимой памяти был как можно меньше. В-третьих, чтобы он был как можно более прост и понятен, это упрощает отладку программы. К сожалению, эти требования противоречивы, и в серьезных задачах редко удается найти алгоритм, который был бы лучше остальных по всем показателям.

Часто говорят о *временно́й сложности* алгоритма (быстродействии) и *пространственной сложности*, которая определяется объёмом необходимой памяти.

**Временем работы алгоритма** называется количество элементарных операций $T$, выполненных исполнителем.

Такой подход позволяет оценивать именно качество алгоритма, а не свойства исполнителя (например, быстродействие компью-

тера, на котором выполняется алгоритм). При этом мы считаем, что время выполнения всех элементарных операций одинаково.

Как правило, величина $T$ будет существенно зависеть от объема исходных данных: поиск в списке из 10 элементов завершится гораздо быстрее, чем в списке из 10000 элементов. Поэтому сложность алгоритма обычно связывают с размером входных данных $N$ и определяют как функцию $T(N)$. Например, для алгоритмов обработки массивов в качестве размера $N$ используют длину массива. Функция $T(N)$ называется *временно́й сложностью алгоритма.*

*Временная сложность алгоритма определяется функцией $T(N) = 2N^3$. Во сколько раз увеличивается время работы алгоритма, если размер данных $N$ увеличится в 10 раз?*

**Пространственная сложность** – это зависимость объёма занимаемой памяти от размера данных $N$. Для ускорения работы некоторых алгоритмов нужно использовать дополнительную память, которая может быть намного больше, чем память для хранения исходных данных.

*Для быстрой сортировки массива из $N$ элементов с помощью алгоритма Семёна требуется $N$ вспомогательных массивов, каждый из которых содержит $N$ элементов. Как изменится объём нужной дополнительной памяти, если $N$ увеличится в 10 раз?*

Поскольку память постоянно дешевеет, а быстродействие компьютеров растет медленно, более важна временна́я сложность алгоритмов. Пространственную сложность мы дальше рассматривать не будем.

## Примеры

Рассмотрим алгоритмы выполнения различных операций с массивом $A$ длины $N$.

**Пример 1**. Вычислить сумму первых трёх элементов массива (при $N \ge 3$).

Решение этой задачи содержит всего один оператор:

```
Sum = A[0] + A[1] + A[2]
```

Этот алгоритм включает две операции сложения и одну операцию записи значения в память, поэтому его сложность $T(N) = 3$ не зависит от размера массива вообще.

*Вычислите количество операций (считая сравнения и присваивание значений переменным) при выполнении фрагмента программы:*

```
a = 8
b = 15
if a < b:
  c = 2*a + b
else:
  c = 2*b + a
```

**Пример 2**. Вычислить сумму всех элементов массива.

В этой задаче уже не обойтись без цикла:

```
Sum = 0
for i in range(N):
  Sum = Sum + A[i]
```

Здесь выполняется $N$ операций сложения и $N$+1 операций записи в память, поэтому его сложность $T(N) = 2N + 1$ возрастает линейно с увеличением длины массива.

*Вычислите количество операций при выполнении фрагмента программы:*

```
a = 0; b = 1
for i in range(N):
  a = a + b*b
  b = b + 1
```

**Пример 3**. Отсортировать все элементы массива по возрастанию методом выбора.

Напомним, что метод выбора предполагает поиск на каждом шаге минимального из оставшихся неупорядоченных значений:

```
for i in range(N-1):
  nMin = i
  for j in range(i+1,N):
```

```
      if A[j] < A[nMin]: nMin = j
  c = A[i]
  A[i] = A[nMin]
  A[nMin] = c
```

Подсчитаем отдельно количество сравнений и количество перестановок. Количество сравнений элементов массива не зависит от данных и определяется числом шагов внутреннего цикла:

$$T_c(N) = (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N-1)}{2} = \frac{1}{2}N^2 - \frac{1}{2}N .$$

Здесь использована формула для суммы первых членов арифметической прогрессии.

На каждом шаге внешнего цикла происходит перестановка двух элементов, общее количество перестановок равно $T_n(N) = N - 1$, то есть сложность по перестановкам – линейная.

*Определите количество операций при вычислении суммы элементов квадратной матрицы A размером N×N:*

```
Sum = 0
for i in range(N):
  for j in range(N):
    Sum = Sum + A[i][j]
```

По результатам этих примеров можно сделать выводы:

- простой цикл, в котором количество шагов пропорционально $N$, – это алгоритм линейной сложности;
- вложенный цикл, в котором количество шагов внешнего и внутреннего цикла пропорционально $N$, – это алгоритм с квадратичной функцией сложности.

## Что такое асимптотическая сложность?

Допустим, что нужно сделать выбор между несколькими алгоритмами, которые имеют разную сложность. Какой из них лучше (работает быстрее)? Оказывается, для этого необходимо знать размер массива данных, которые нужно обрабатывать. Сравним, например, три алгоритма, сложность которых

$$T_1(N) = 10000 \cdot N,\ T_2(N) = 100 \cdot N^2,\ T_3(N) = N^3.$$

Построим эти зависимости на графике (см. Рис. 4.7). При $N \le 100$ получаем $T_3(N) < T_2(N) < T_1(N)$, при $N = 100$ количество операций для всех трёх алгоритмов совпадает, а при больших $N$ имеем $T_3(N) > T_2(N) > T_1(N)$.

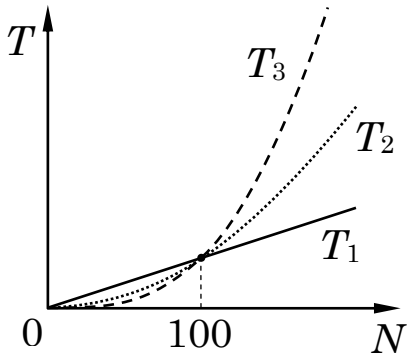



Рис. 4.7.

Обычно в теоретической информатике при сравнении алгоритмов используется их *асимптотическая сложность*, то есть скорость роста количества операций при больших значениях $N$.

Запись $O(N)$ (читается «О большое от $N$») обозначает *линейную сложность* алгоритма. Это значит, что, начиная с некоторого значения $N = N_0$, количество операций будет меньше, чем $c \cdot N$, где $c$ – некоторая постоянная:

$$T(N) \le c \cdot N \text{ для } N \ge N_0.$$

При увеличении размера данных в 10 раз объем вычислений алгоритма с линейной сложностью увеличивается тоже примерно в 10 раз.

Пусть, например, $T(N) \le 2N - 1$, как в алгоритме поиска суммы элементов массива. Очевидно, что при этом $T(N) \le 2N$ для всех $N \ge 1$, поэтому алгоритм имеет линейную сложность.

*Определите любые подходящие значения $c$ и $N_0$, такие что $T(N) \le c \cdot N$ для $N \ge N_0$, для алгоритмов с линейной асимптотической сложностью:*

*а)* $T(N) = 2N - 8$ б) $T(N) = 7N + 5$

Многие известные алгоритмы имеют *квадратичную сложность*, $O(N^2)$. Это значит, что количество операций при больших $N$ не больше, чем $c \cdot N^2$:

$$T(N) \le c \cdot N^2 \text{ для } N \ge N_0.$$

Если размер данных увеличивается в 10 раз, то количество операций (и время выполнения такого алгоритма) увеличивается примерно в 100 раз. Пример алгоритма с квадратичной сложностью – сортировка методом выбора, для которой число сравнений

$$T_c(N) = \frac{1}{2}N^2 - \frac{1}{2}N \le \frac{1}{2}N^2 \text{ для всех } N \ge 0.$$

*Определите любые подходящие значения $c$ и $N_0$, такие что $T(N) \le c \cdot N^2$ для $N \ge N_0$, для алгоритмов с квадратичной асимптотической сложностью:*

*а)* $T(N) = 5N^2 - 3N - 2$ б) $T(N) = 7N^2 + 5N$

Алгоритм относится к классу $O(f(N))$, если найдется такая постоянная $c$, что, начиная с некоторого $N = N_0$ выполняется условие $T(N) \le c \cdot f(N)$.

Это значит, что график функции $c \cdot f(N)$ проходит выше, чем график функции $T(N)$, по крайней мере, при $N \ge N_0$ (см. Рис. 4.8).

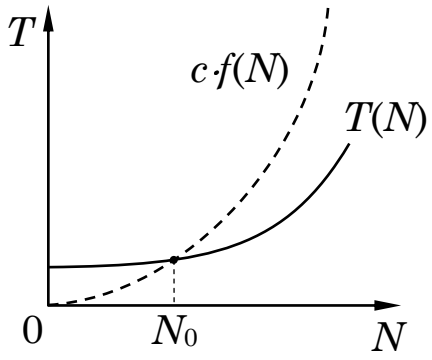



Рис. 4.8.

Строго говоря, обозначение $O(f(N))$ определяет множество всех алгоритмов, для которых количество операций $T(N)$ растёт не быстрее, чем $f(N)$. Тогда, например, алгоритм с количеством операций $T(N) = N + 5$ относится к классам $O(N)$, $O(N^2)$ , $O(N^3)$ и даже $O(2^N)$. Однако на практике выражение «алгоритм имеет сложность $O(N^2)$» чаще всего означает, что $O(N^2)$ – это наилучшая асимптотическая оценка роста количества операций $T(N)$,

то есть по крайней мере для некоторых данных $T(N)$ растёт быстрее, чем $O(N)$, $O(N^{1,5})$ и даже $O(N^{1,99999})$.

Если количество операций не зависит от размера данных, то говорят, что асимптотическая сложность алгоритма $O(1)$, то есть количество операций меньше некоторой постоянной при любых $N$.

Существует также немало алгоритмов с кубической сложностью, $O(N^3)$. При больших значениях $N$ алгоритм с кубической сложностью требует бóльшего количества вычислений, чем алгоритм со сложностью $O(N^2)$, а тот, в свою очередь, работает дольше, чем алгоритм с линейной сложностью. Однако при небольших значениях $N$ всё может быть наоборот, это зависит от постоянной $c$ для каждого из алгоритмов.

Известны и алгоритмы, для которых количество операций растет быстрее, чем любой многочлен, например, как $O(2^N)$. Они встречаются чаще всего в задачах поиска оптимального (наилучшего) варианта, которые решаются только методом полного перебора. Самая известная задача такого типа – это *задача коммивояжера* (бродячего торговца), который должен посетить по одному разу каждый из указанных городов и вернуться в начальную точку. Для него нужно выбрать маршрут, при котором стоимость поездки (или общая длина пути) будет минимальной.

В таблице на Рис. 4.9 показано примерное время работы алгоритмов, имеющих разную временную сложность, при $N = 100$ на компьютере с быстродействием 1 миллиард операций в секунду.

| $T(N)$ | время выполнения |
|---|---|
| $N$ | 100 нс |
| $N^2$ | 10 мс |
| $N^3$ | 0,001 с |
| $2^N$ | $10^{13}$ лет |

Рис. 4.9.

*Юный программист Григорий поспорил с учителем, что сможет с помощью компьютера решить сложную задачу*

*перебора вариантов к завтрашнему уроку. Дома он определил временную сложность алгоритма: $T(N) = 2^N$. Для какого наибольшего значения $N$ сможет Григорий решить задачу за сутки, если его компьютер выполняет 1 миллиард операций в секунду?*

## Выводы:

- Временем работы алгоритма называется количество элементарных операций $T$, выполненных исполнителем.
- Временная сложность алгоритма обычно зависит от объёма исходных данных $N$, например, от размера массива.
- Пространственная сложность – это объём памяти, необходимой для работы алгоритма.
- Простой цикл, в котором количество шагов пропорционально $N$, – это алгоритм линейной сложности;
- Вложенный цикл, в котором количество шагов внешнего и внутреннего цикла пропорционально $N$, – это алгоритм квадратичной сложности.
- Алгоритм относится к классу $O(f(N))$, если найдется такая постоянная $c$, что, начиная с некоторого $N = N_0$ выполняется условие $T(N) \le c \cdot f(N)$.
- Линейная сложность означает, что при увеличении размера массива в $K$ раз количество операций увеличивается примерно в $K$ раз.
- Квадратичная сложность означает, что при увеличении размера массива в $K$ раз количество операций увеличивается примерно в $K^2$ раз.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Какие критерии используются для оценки качества алгоритмов?
2. Почему скорость работы алгоритма оценивается не временем выполнения, а количеством элементарных операций?

3. Как учитывается размер данных при оценке быстродействия алгоритма?
4. В каких случаях алгоритм, имеющий асимптотическую сложность $O(N^2)$, может работать быстрее, чем алгоритм с асимптотической сложностью $O(N)$?

## Задачи

1. Оцените асимптотическую сложность для алгоритмов:
   а) вычисления произведения первого и последнего элементов массива;
   б) вычисления суммы элементов первой половины массива;
   в) нахождения минимального и максимального элементов массива;
   г) определения количества положительных элементов массива;
   д) определения количества нулевых элементов квадратной матрицы;
   е) поиска всех делителей числа $N$.

   Считайте, что массив содержит $N$ элементов, а матрица имеет размеры $N \times N$.
2. Определите асимптотическую сложность алгоритмов, для которых известно количество операций:
   а) $T(N) = 5 \cdot N + 6$
   б) $T(N) = 3 \cdot N^2 + 2 \cdot N + 19$
   в) $T(N) = 2 \cdot N^3 + 100$
   г) $T(N) = 4 \cdot 2^N + 25 \cdot N^{18} + 8$
3. Алгоритм обработки массива имеет асимптотическую сложность $O(N^2)$, где $N$ – длина массива. Во сколько раз увеличится время выполнения алгоритма, если длина массива увеличится в 5 раз?
4. *Алгоритм обработки массива имеет асимптотическую сложность $O(2^N)$, где $N$ – длина массива. Как увеличится время выполнения алгоритма, если длина массива увеличится на 5 элементов? в 5 раз?

## Темы сообщений:

«Задача коммивояжёра»

# § 23. Как разрабатывают программы?

**Ключевые слова**:

- постановка задачи
- построение модели
- разработка алгоритма и способа представления данных
- кодирование
- отладка
- тестирование
- документирование
- внедрение и сопровождение

Как вы знаете, новые программы для компьютеров пишут программисты. Но это не совсем точно: любая достаточно сложная программа проходит несколько этапов от рождения идеи до выпуска готового продукта, и в этом участвует множество специалистов.

## Этапы разработки программ

**1. Постановка задачи**. Сначала определяют задачи, которые должна решать программа, и записывают все требования к ней в виде документа – *технического задания*. Это очень важный этап, потому что ошибка в самом начале разработки приведёт к тому, что будет решена совершенно другая задача.

**2. Построение модели.** Когда задача поставлена, нужно выполнить формализацию – записать все требования на формальном языке, например, на языке математических формул. В результате строится модель исходной задачи, в которой чётко определяются все связи между исходными данными и желаемым результатом.

**3. Разработка алгоритма и способа представления данных.** Любая компьютерная программа служит для обработки данных. Поэтому очень важно определить, как будут представлены данные в памяти компьютера (например, в виде отдельных переменных или массивов).

Способ хранения данных определяет и алгоритмы работы с ними: если выбрана неудачная структура данных, очень сложно написать хороший алгоритм обработки. Известная книга швейцарского специалиста Никлауса Вирта, автора языка Паскаль,

так и называется «Алгоритмы + структуры данных = программы».

**4. Кодирование**. Только теперь, когда выбран способ хранения данных и готовы алгоритмы для работы с ними, программисты приступают к написанию программы. Эта работа называется *кодированием*, потому что программист кодирует алгоритм – записывает его на языке программирования. Результат его работы – текст программы – часто называют *программным кодом*.

**5. Отладка**. Ни один человек не может написать достаточно большую программу без ошибок. Поэтому программисту приходится искать и устранять ошибки в программах. Этот процесс называется *отладкой программы*.

Все ошибки можно разделить на две группы: синтаксические и логические. *Синтаксические ошибки* – несоответствия правилам языка программирования – обнаруживаются транслятором, поэтому найти и исправить их достаточно просто.

Сложнее исправлять *логические ошибки* – ошибки в составлении алгоритма. Из-за логических ошибок программа работает не так, как требуется. Чтобы исправить такую ошибку, программисту приходится внимательно изучить работу программы, иногда даже выполнить все вычисления вручную, без компьютера, и сравнить результаты каждого шага с теми результатами, которые даёт программа.

Логические ошибки могут привести к *отказу* – аварийной ситуации, например, к делению на ноль. Часто при отказе операционная система завершает работу программы, и данные могут быть потеряны. Отказы часто называют *ошибками времени выполнения* (англ. *runtime error*).

**5. Тестирование**. Когда программист исправил все обнаруженные им ошибки, он передаёт программу на тестирование – тщательную проверку в различных режимах. Обычно эту работу выполняют специально обученные люди – *тестировщики*.

Тестирование в компании, которая разрабатывает программу, называется альфа-тестированием. Когда оно завершено, начинается бета-тестирование (внешнее тестирование). Программа (так называемая «бета-версия») рассылается некоторым клиентам или даже распространяется свободно. Цель этого этапа – привлечь к тестированию множество людей, чтобы они смогли найти как можно больше ошибок в программе.

**6. Документирование** – это разработка документации на программу. Этим занимаются *технические писатели*. Техническая документация описывает, как работает программа, а руководство пользователя содержит инструкцию по использованию программы.

**7. Внедрение и сопровождение**. Когда программа отлажена и документация по ней готова, её нужно передать заказчику. Компания берёт на себя *сопровождение* программы – обучение пользователей, исправление найденных ими ошибок, техническую поддержку (ответы на вопросы). Часто компании выпускают новые версии программ, в которых исправляются ошибки и добавляются новые возможности.

## Методы проектирования программ

Современные программы очень сложны, они могут состоять из сотен тысяч и миллионов строк. Написать такую программу в одиночку невозможно, поэтому над проектом работают большие команды программистов.

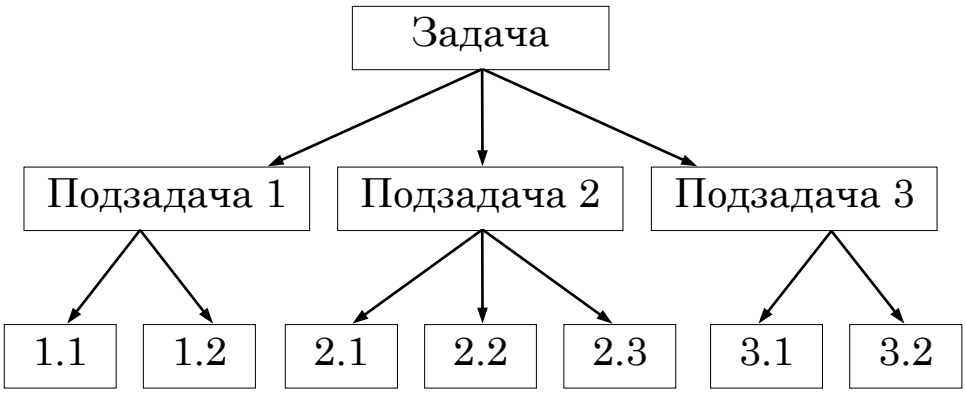



Рис. 4.10.

Всю работу нужно как-то разделить между программистами, чтобы каждый мог выполнять свою часть независимо от других. Для этого необходимо разбить задачу на подзадачи (Рис. 4.10).

Решение каждой подзадачи оформляется в виде подпрограммы – вспомогательного алгоритма (вспомните материал 7 класса). Программист получает персональное задание – написать одну или несколько подпрограмм. Он может работать независимо от других, важно только соблюдать правила обмена данными между «его» подпрограммой и остальными.

Если нужно, подзадачи разбиваются на более мелкие подзадачи (см. третий уровень дерева на Рис. 4.10) и так далее, так чтобы каждая подпрограмма была не длиннее, чем 30-40 строчек.

Такой приём называется **последовательным уточнением** или **проектированием «сверху вниз»**, от основной задачи к мелким подзадачам.

Существует и другой подход (**проектирование «снизу вверх»**) – сначала разработать подпрограммы для решения самых простых задач, а потом собирать из них подпрограммы для более крупных задач, как из кубиков. При этом мы строим дерево, показанное на Рис. 4.10, снизу вверх, с нижнего уровня. На практике программисты обычно сочетают оба подхода.

## Отладка программы

Простейший метод отладки программы – это *вывод отладочной информации*. Рассмотрим этот способ на примере.

Программисту нужно было написать программу, которая вычисляет корни квадратного уравнения $ax^2 + bx + c = 0$. Он поспешил и написал программу так:

```
from math import sqrt
print( "Введите a, b, c:" )
a = float( input() )
b = float( input() )
c = float( input() )
```

```
D = b*b - 4*a*a
x1 = (-b + sqrt(D))/ 2*a;
x2 = (-b - sqrt(D))/ 2*a;
print( "x1={:.3}, x2={:.3}".format(x1, x2) )
```

Для вычисления квадратного корня здесь используется встроенная функция *sqrt* из модуля *math*.

Оказалось, что программа в некоторых случаях работает верно (например, при $a = 1$, $b = 2$ и $c = 1$), а в других случаях – неверно (например, при $a = 1$, $b = –5$ и $c = 6$).

Для того чтобы найти ошибку, нужно определить её **возможные причины**. В нашем случае есть три варианта:

1) неверно вводятся данные;
2) неверно вычисляется дискриминант $D = b^2 – 4ac$;
3) неверно вычисляются корни $x_1 = \frac{-b+\sqrt{D}}{2a}$, $x_2 = \frac{-b-\sqrt{D}}{2a}$.

Добавим в программу две дополнительные команды для вывода отладочной информации (они выделены фоном):

```
a = float( input() )
b = float( input() )
c = float( input() )
print( a, b, c )
D = b*b - 4*a*a
print( "D={}".format(D) )
```

С помощью этих команд мы

1) выведем значения коэффициентов $a$, $b$ и $c$ сразу после ввода;
2) выведем вычисленное значение дискриминанта.

Значения корней уравнения уже и так выводятся в конце работы программы.

При вводе коэффициентов 1, –5 и 6 программа выводит

```
1.0 -5.0 6.0
D=21.0
x1=4.79, x2=0.209
```

По первой строчке видим, что ввод выполнен правильно – именно такие числа мы вводили. А вот значение дискриминан-

та, вычисленного программой, отличается от того, что мы ожидаем получить: $D = (-5)^2 - 4 \cdot 1 \cdot 6 = 1$. Поэтому нужно искать ошибку в выражении для вычисления $D$.

Если исправить эту ошибку (сделайте это самостоятельно), мы увидим, что дискриминант считается правильно, а корни уравнения – нет (при $a = 1$, $b = -5$ и $c = 6$ мы должны получить $x_1 = 3$ и $x_2 = 2$).

*Исправьте ошибки в тех строчках программы, где вычисляются корни уравнения.*

Современные среды программирования содержат встроенный отладчик, который позволяет:

- выполнять программу в пошаговом режиме;
- после выполнения очередной команды просматривать значения переменных в памяти;
- устанавливать *точки останова*, где программа должна остановиться и перейти в пошаговый режим.

*Доработайте программу так, чтобы учесть случай, когда уравнение не имеет вещественных корней.*

## Документирование программы

К выпуску программы компания-разработчик должна подготовить **документацию** на программу. Руководство пользователя (это наиболее важная часть документации) обычно содержит:

- назначение программы;
- формат входных данных;
- формат выходных данных;
- примеры использования программы.

Для примера составим документацию на простую программу, отладкой которой мы только что занимались.

**Назначение программы**: вычисление вещественных корней квадратного уравнения $ax^2 + bx + c = 0$.

**Формат входных данных**: значения коэффициентов $a$, $b$ и $c$ вводятся с клавиатуры по одному числу в строке.

**Формат выходных данных**: значения вещественных корней корни уравнения выводятся на экран через пробел в одной строке; перед значением первого корня выводится текст «**x1=**», перед значением второго корня – текст «**x2=**». Если вещественных корней нет, выводится слово «нет».

**Пример использования программы** (решение уравнения $x^2 - 5x + 6 = 0$):

```
Введите a, b, c: 1 -5 6
x1=3.0 x2=2.0
Введите a, b, c: 1 1 1
нет
```

### Практическая работа №19. Отладка программы

## Выводы:

- Этапы разработки программного обеспечения:
  - постановка задачи
  - построение модели
  - разработка алгоритма и способа представления данных
  - кодирование
  - отладка
  - тестирование
  - документирование
  - внедрение и сопровождение
- При использовании метода проектирования «сверху вниз» (метода последовательного уточнения) задача разбивается на подзадачи.
- При использовании метода проектирования «снизу вверх» разработка программы начинается с наиболее мелких подзадач, из которых основная программа затем собирается как из кубиков.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Почему способы хранения данных и алгоритмы их обработки обычно разрабатываются одновременно?
2. Чем отличается тестирование от отладки?
3. Можно ли считать, что программа, успешно прошедшая тестирование, не содержит ошибок?
4. Может ли произойти отказ в программе, в которой нет логических ошибок?
5. Если программа плохо документирована, к каким последствиям это может привести?
6. Как вы думаете, почему важно сопровождение программы после её сдачи заказчику?
7. Чем отличаются два подхода к проектированию программ: «сверху вниз» и «снизу вверх»?

## Задачи

1. Требуется написать программу, которая загружает массив из файла и выводит отсортированный массив в другой файл. Выделите подзадачи в этой задаче.
2. Требуется написать программу, которая загружает изображение из файла, выполняет его обрезку, преобразует из цветного формата в чёрно-белый и выводит полученное изображение в другой файл. Выделите подзадачи в этой задаче.
3. Требуется написать программу для игры с человеком в «крестики-нолики». Выделите подзадачи в этой задаче.
4. Отладьте программу для вычисления корней квадратного уравнения. Учтите, что уравнение может не иметь вещественных корней.
5. Используя образец, приведённый в тексте параграфа, составьте документацию на одну из написанных вами программ и предложите соседу ей воспользоваться. Вместе с ним исправьте описание программы, если это потребуется.

**Темы сообщений:**

а) «Структурное программирование»
б) «Парадигмы (стили) программирования»

# § 24. Процедуры

**Ключевые слова**:

- процедура
- параметр
- локальная переменная
- рекурсивная процедура

## Что такое подпрограмма?

Когда мы в 7 классе работали с исполнителем Робот, мы уже использовали вспомогательные алгоритмы (подпрограммы, процедуры). Каждая процедура решала одну подзадачу, из них строилась программа для решения основной задачи.

Подпрограммы полезны, в первую очередь, потому, что готовые алгоритмы можно использовать много раз при решении более сложных задач, не «изобретая велосипед». Из подпрограмм составляются библиотеки, некоторые из которых входят в состав языков программирования. Мы просто используем их, задумываясь только о том, как они работают. Это экономит время программистов, освобождая их от повторного выполнения работы, которая уже была кем-то сделана раньше.

Подпрограммы бывают двух типов – процедуры и функции. Подпрограммы-процедуры выполняют некоторые действия. Например, **`print`** в языке Python – это встроенная подпрограмма-процедура, которая выводит данные на экран. Подпрограммы-функции возвращают результат (число, строку). Подпрограмма **`sqrt`**, вычисляющая квадратный корень числа, – это функция.

В этом параграфе мы научимся писать свои подпрограммы-процедуры, а в следующем займёмся функциями.

*Определите тип подпрограммы (процедура или функция), которая*

*а) рисует окружность на экране;*

*б) определяет площадь круга;*

*в) вычисляет значение синуса угла;*

*г) изменяет режим работы программы;*

*д) возводит число x в степень y;*

*е) включает двигатель автомобиля;*

*ж) проверяет оставшееся количество бензина в баке;*

*з) измеряет высоту полёта самолёта.*

## Простая процедура

Предположим, что в нескольких местах программы требуется выводить на экран строчку из 10 знаков «–» (например, для того чтобы отделить два блока результатов друг от друга). Это можно сделать, например, так:

```
print( "----------" )
```

Конечно, можно вставить этот оператор вывода везде, где нужно вывести такую строчку. Но это решение имеет два недостатка. Во-первых, строка из минусов будет храниться в памяти много раз. Во-вторых, если мы задумаем как-то изменить эту строку (например, заменить знак «–» на «=»), нужно будет искать эти операторы вывода по всей программе.

Для таких случаев в языках программирования предусмотрены *процедуры*. Посмотрим на программу с процедурой:

```
def printLine():
  print( "----------" )
...
printLine()
...
printLine()
```

Многоточием в текстах программ будем обозначать некоторые операторы, которые нас пока не интересуют.

Сначала в программе расположена процедура, выделенная фоном. Она начинается со служебного слова **def** (от англ. *define* – определить). После имени процедуры записаны пустые скобки (чуть далее мы увидим, что они могут быть и непустые!) и двоеточие.

Все команды, входящие в тело процедуры, записываются с отступом (так же, как и команды, входящие в тело цикла или условного оператора).

Для того чтобы процедура заработала, в основной программе (или в другой процедуре) необходимо её *вызвать* по имени (не забыв скобки), причём таких вызовов может быть сколько угодно (в нашей программе – два).

Процедура должна быть определена к моменту её вызова, то есть должна быть выполнена инструкция **def**, которая создает объект-процедуру в памяти. Если процедура вызывается из основной программы, то нужно поместить её определение раньше точки вызова.

*Что произойдёт, если вызвать процедуру, но не включить её текст в программу? Проверьте этот вариант с помощью компьютера.*

*Что произойдёт, если включить текст процедуры в программу, но не вызывать её? Проверьте этот вариант с помощью компьютера.*

Использование процедур сокращает код, если какие-то операции должны выполняться несколько раз в разных местах программы. Кроме того, большую программу всегда разбивают на несколько подпрограмм для удобства, выделяя этапы сложного алгоритма. Такой подход делает всю программу более понятной и позволяет разделить работу между программистами.

Когда процедура написана и тщательно протестирована, можно передать её другим программистам для использования в этом же или другом проекте. Очень важно, чтобы автор процедуры подробно описал, что она делает, её входные и выходные данные. Тем, кто использует готовую процедуру, нужно убедиться, что она выполняет именно то, что требуется в данной задаче.

## Процедура с параметром

Теперь представьте себе, что нужно выводить строки из минусов разной длины (5, 10 и др). Конечно, можно сделать несколько процедур, например, так:

```
def printLine5():
  print( "-----" )
```

```
def printLine10():
  print( "----------" )
```

Но так делать не нужно. Дело в том, что обе процедуры выводят цепочки знаков «минус» (то есть, выполняют те же самые действия!), только разной длины. Поэтому хочется использовать всего одну процедуру, передавая ей нужную длину цепочки.

Процедуру **printLine10** можно переписать, применив «умножение» строки на число (заменяющее, как и в математике, многократное сложение):

```
def printLine10():
  print( "-"*10 )
```

Эта процедура делает то же самое, что и первый вариант – выводит 10 «минусов» подряд и переходит на новую строчку.

*Чем будет отличаться процедура, рисующая 5 знаков «минус», от последнего варианта процедуры* **printLine10***?*

Если мы хотим, чтобы длину строки можно было менять, в процедуре вместо числа нужно использовать переменную. И значение этой переменной необходимо как-то передать процедуре. Оформляется это так:

```
def printLine( n ):
  print( "-"*n )
```

Величина *n* называется *параметром* процедуры. Теперь в круглые скобки в заголовке процедуры не пустые. В них записано имя параметра – специальной переменной, с помощью которой можно управлять работой процедуры.

**Параметр** — это переменная, от значения которой зависит работа подпрограммы. Имена параметров перечисляются в заголовке подпрограммы.

Наша процедура **printLine** имеет один параметр, обозначенный именем *n* – это длина строки из «минусов».

При вызове такой процедуры в скобках нужно записать фактическое значение, которое присваивается переменной *n* внутри процедуры:

```
printLine( 10 )
```

Такое значение называется *аргументом* (или фактическим параметром).

**Аргумент** — это значение параметра, которое передаётся подпрограмме при её вызове.

Аргументом может быть не только постоянное значение (число, символ), но также и переменная, и даже арифметическое выражение.

*Что будет выведено на экран при выполнении фрагмента программы*

```
printLine( 7 );
printLine( 5 );
printLine( 3 );
```

*Для тестирования процедуры* **printLine** *Иван хочет написать небольшую программу, в которой длина линии вводится с клавиатуры. Где нужно поместить оператор ввода – в процедуре или в основной программе?*

## Локальные и глобальные переменные

Переменные, которые введены в основной программе, называются *глобальными* (общими). Их могут использовать все подпрограммы (процедуры и функции).

Часто бывает нужно ввести дополнительные переменные, которые будут использоваться только в подпрограмме. Такие переменные называются *локальными* (местными), к ним можно обращаться только внутри этой подпрограммы, и остальные подпрограммы (а также основная программа) про них ничего не «знают».

*Где вы уже встречались со словом «локальный» в курсе информатики? Вспомните, от какого иностранного слова оно произошло.*

Такой приём называется *инкапсуляция* (от лат. «помещение в капсулу») – мы ограничиваем *область видимости* (область действия) переменной только той подпрограммой, где она действительно нужна.

Составим процедуру, которая «рисует» на экране треугольник из символов «O» высотой $n$ (Рис. 4.11).

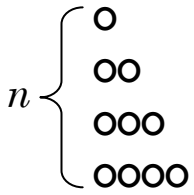



Рис. 4.11.

Как видно из Рис. 4.11, рисунок строится из $n$ строчек, причём в $n$-й по порядку строчке выводится ровно $n$ символов. Процедуру можно написать так:

```
def triangleO( n ):
  for i in range(1,n+1):
    print( "O"*i )
```

Мы специально построили цикл по переменной так, чтобы переменная $i$ (она принимает последовательно все целые значения от 1 до $n$) при каждом повторении цикла совпадала с длиной очередной цепочки символов «O».

Переменная $i$ – это *локальная* переменная, она введена и используется внутри процедуры. Другие процедуры и основная программа не могут обращаться к «чужой» локальной переменной. В других процедурах тоже можно создать локальные переменные с таким же именем $i$, но они будут связаны уже с другими областями памяти, то есть это будут другие переменные.

**Локальная переменная** — это переменная, введённая внутри подпрограммы. Другие подпрограммы и основная программа не могут к ней обращаться.

Локальная переменная создаётся только при вызове процедуры. Как только работа процедуры закончена, все локальные переменные удаляются из памяти.

Имена локальных переменных в каждой подпрограмме можно выбирать независимо от имён локальных переменных других подпрограмм. Это очень облегчает коллективную работу

программистов, каждый из которых может использовать любые имена локальных переменных в своих процедурах.

Посмотрим на такую программу:

```
def show():
  print( i )
```

В процедуре выводится значение *i*, но откуда его взять? Транслятор сначала ищёт локальную переменную с таким именем – её нет. Потом он начинает искать глобальную переменную: если такая переменная есть, на экран выводится её значение, если нет – будет выдано сообщение об ошибке.

Теперь посмотрим на другую процедуру:

```
def showLocal():
  i = 2
  print( i )
```

Даже если существует глобальная переменная *i*, в первой строчке этой процедуры будет создана новая локальная переменная *i*, и её значение (2) появится на экране.

Что же делать, если в процедуре необходимо изменить именно глобальную переменную? Пусть переменная *i* – глобальная, её значение задано в основной программе:

```
i = 15
```

Изменим её значение в процедуре. Для этого нужно явно сказать, что она глобальная, используя команду **global**:

```
def showGlobal():
  global i
  i = 2
  print( i )
```

Эта процедура работает с глобальной переменной *i*. Она присвоит ей новое значение 2 (это «увидят» все остальные подпрограммы!) и выведет его на экран.

Ещё раз подчеркнём, что если мы забудем написать инструкцию **global**, транслятор, не сомневаясь, создаст в памяти новую локальную переменную и будет с ней работать. Глобальная переменная при этом не изменится.

Нужно стараться писать процедуры, которые не обращаются к глобальным переменным (как говорят, не имеют побочных эффектов). Работа процедуры в идеале должна зависеть только от переданных значений параметров, тогда можно легко использовать её в другом проекте. К сожалению, в языке Python это не всегда выполнимо.

### Несколько параметров

Давайте немного улучшим процедуру: сделаем так, чтобы можно было изменять не только длину строки, но и символы, из которых она строится. Для этого добавим в процедуру ещё один параметр, который можно назвать *symbol*:

```
def printLine( symbol, n ):
  print( symbol*n )
```

Имена параметров в заголовке процедуры отделяются запятой.

Чем больше параметров у процедуры, тем больше разных задач она может решать, но тем сложнее её понимать и легче сделать ошибку. Поэтому не рекомендуется передавать в процедуру больше 3-4 параметров.

*Запишите в тетради полный текст процедуры* **printLine**, *которая использует цикл с условием вместо операции «умножения» символьных строк.*

*Что будет выведено на экран при выполнении фрагмента программы*

```
printLine( '-', 10 )
printLine( '=', 7 )
printLine( 'o', 5 )
```

### Процедуры в других языках программирования

Процедура **printLine** с одним параметром на языке Паскаль и может быть записана так:

```
procedure printLine( n: integer );
var i: integer;
```

```
begin
  for i:=1 to n do
    write('-');
  writeln
end;
```

Процедура начинается словом **procedure**, тело процедуры заключено между служебными словами **begin** и **end**. После имени параметра процедуры *n* через двоеточие указывают тип этой величины (англ. *integer* – целый). Процедура должна располагаться выше основной программы.

Вот вариант этой же процедуры на языке C++:

```
void printLine ( int n )
{
  int i;
  for( i=1; i<=n; i++ )
    putchar('-');
}
```

Признак процедуры – слово **void** в заголовке (вместо **procedure** на Паскале). Тело процедуры заключено в фигурные скобки. Встроенная функция **putchar** выводит на экран один символ, который заключён в апострофы («одиночные кавычки»).

## Рекурсия

Составим процедуру, которая выводит на экран двоичную запись натурального числа. Поскольку мы хотим использовать процедуру для разных чисел, это должна быть процедура с параметром:

```
def printBin( n ):
  ...
```

В теле процедуры должен быть алгоритм для перевода числа в двоичную систему. Один такой алгоритм мы знаем: нужно делить число на 2, каждый раз выписывая остаток от деления, пока не получится 0. На языке Python этот алгоритм можно записать так:

```
while n != 0:
  print( n % 2, end="" )
  n = n // 2
```

Напомним, что именованный аргумент **end**, равный пустой строке "", отключает переход на новую строку в конце работы функции **print** (иначе все цифры будут выведены в столбик).

*Проверьте вручную работу этого алгоритма для числа 6. Удалось ли вам получить правильный ответ? Почему?*

Проблема только в том, что первой мы получаем последнюю цифру двоичной записи, поэтому остатки выводятся в обратном порядке (не так, как нужно!).

Есть разные способы решения этой задачи, которые сводятся к тому, чтобы запоминать остатки от деления (например, в символьной строке) и затем, когда результат полностью получен, вывести его на экран. Однако можно применить ещё один красивый подход. Идея такова: *чтобы вывести двоичную запись числа n, нужно сначала вывести двоичную запись числа* $n$ // 2, а затем – его последнюю двоичную цифру, которая вычисляется как $n$ % 2.

Что же получилось? Прочитайте еще раз фразу, выделенную курсивом в предыдущем абзаце. Выходит, что для того, чтобы решить задачу для исходного числа, нужно предварительно решить *ту же самую задачу* для меньшего числа, $n$ // 2.

Такой алгоритм очень просто программируется:

```
def printBin( n ):
  printBin( n // 2 )
  print( n % 2, end="" )
```

У нас получилось, что процедура **printBin** вызывает сама себя! Такой приём в программировании называется *рекурсией*, а процедура – рекурсивной.

**Рекурсивная процедура** — это процедура, которая вызывает сама себя.

*Проверьте с помощью отладчика в пошаговом режиме, что произойдет при вызове этой процедуры.*

Приведённая процедура **printBin** ошибочна, вернее, она не доделана. Представим себе, что мы передали процедуре число 2. Сначала она вызывает сама себя для значения 2 // 2 = 1, затем – для значения 1 // 2 = 0, и потом ещё бесконечно много раз для нуля. Такие вызовы никогда не закончатся и программа зациклится. Чтобы этого не произошло, нужно выйти из процедуры (и закончить эти вложенные вызовы), когда значение параметра стало равно нулю:

```
def printBin( n ):
  if n == 0: return
  printBin( n // 2 )
  print( n % 2, end="" )
```

Убедимся, что теперь процедура остановится при любом заданном натуральном числе. Действительно, при каждом вложенном вызове значение параметра уменьшается (делится на 2). В результате когда-нибудь оно обязательно станет равно нулю, и вложенные вызовы закончатся.

*Сформулируйте алгоритм вывода цепочки из n одинаковых символов, использующий рекурсию. Напишите рекурсивную процедуру. Попробуйте придумать два варианта решения задачи.*

**Практическая работа №20. Процедуры**

**Практическая работа №21. Рекурсивные процедуры**

## Выводы:

- Процедура – это вспомогательный алгоритм (подпрограмма), решающий самостоятельную задачу, который может использоваться несколько раз.

- Локальная переменная — это переменная, объявленная внутри подпрограммы. Другие подпрограммы и основная программа не могут к ней обращаться.
- Параметр — это величина, от которой зависит работа подпрограммы. Параметр имеет имя, в теле подпрограммы с ним можно работать так же, как с локальной переменной.
- Аргумент — это значение параметра, которое передаётся подпрограмме.
- Рекурсивная процедура — это процедура, которая вызывает сама себя.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Зачем нужны процедуры?
2. Достаточно ли включить процедуру в текст программы, чтобы она «сработала»?
3. Какие возможности появляются, когда в процедуру добавляются параметры?
4. Как определить, что переменная – локальная?
5. Имеет ли смысл оформлять процедуру, если она вызывается в программе только один раз? Обсудите достоинства и недостатки такого решения.

## Задачи

1. Напишите процедуру, которая принимает два параметра: символ *c* и натуральное число *N*, и выводит на экран треугольник из символов *c* со стороной *N*. Например, при *c* = "o" и *N* = 5 мы должны получить

   o
   oo
   ooo
   oooo
   ooooo

2. Напишите процедуру, которая принимает два параметра – *W* и *H*, – и рисует на экране рамку из точек, ширина кото-

рой равна $W$, а высота – $H$. Например, для $W = 6$ и $H = 4$ программа должна вывести рисунок:

```
******
*    *
*    *
******
```

3. Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с последней.

4. Напишите процедуру, которая выводит на экран все делители переданного ей числа (в одну строчку через пробел).
5. *Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с первой.
6. *Напишите процедуру, которая выводит на экран запись переданного ей числа в римской системе счисления.
7. *Напишите процедуру, которая принимает числовой параметр – возраст человека в годах, и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «12 лет».
8. *Напишите процедуру, которая выводит переданное ей число прописью. Например,

   21 → «двадцать один».
9. Напишите рекурсивную процедуру для перевода числа в восьмеричную систему счисления.
10. Напишите рекурсивную процедуру для перевода числа в любую систему счисления с основанием от 2 до 9.
11. *Напишите рекурсивную процедуру для перевода числа в шестнадцатеричную систему счисления.

**Темы сообщений:**

а) «Рекурсия в природе и искусстве»

б) «Ханойские башни»

# § 25. Функции

**Ключевые слова**:

- функция
- вызов функции
- параметры
- рекурсивная функция

## Что такое функция?

Представьте себе, что вы заказываете товар с доставкой по телефону или в Интернет-магазине. Если говорить на языке программистов, вы вызываете вспомогательный алгоритм (подпрограмму). Но, в отличие от процедуры, исполнитель этого алгоритма не только выполняет какие-то действия, но и *возвращает результат* – товар, который вам привозит курьер. Это второй тип вспомогательных алгоритмов (подпрограмм). Такие подпрограммы называются *функциями*.

**Функция** — это вспомогательный алгоритм, который возвращает результат (число, строку символов и др.).

Построим функцию, которая возвращает среднее арифметическое двух целых чисел.

*Какой тип данных подходит для хранения среднего арифметического двух целых чисел?*

Функция принимает два параметра – исходные целые числа и возвращает результат, который может быть вещественным числом:

```
def Avg( a, b ):
  sred = (a+b)/2
  return sred
```

Заголовок функции ничем не отличается от заголовка процедуры. В языках Python и C++ (в отличие, например, от Паскаля) процедуры рассматриваются как функции особого типа, не возвращающие никакого результата.

Функция *возвращает* результат, записанный после специального оператора **`return`**. В нашем примере функция **`Avg`** возвращает результат – значение локальной переменной *sred*.

*Используя дополнительные источники, выясните, что означает английское слово average, от которого образовано название функции* **Avg**.

Если вызвать функцию так же, как и процедуру:

```
Avg(5, 9)
```

то значение, которое она вернула, потеряется. Но его можно сохранить в переменной:

```
sr = Avg(5, 9)
```

В операторе присваивания вместо вызова функции транслятор «подставляет» результат вызова – то значение, которое эта функция вернёт. Поэтому предыдущий оператор равносилен такому:

```
sr = 7
```

Результат функции можно сразу вывести на экран:

```
print( Avg(4, 8) )
```

*Что будет выведено на экран в результате работы этого фрагмента программы:*

```
sr = Avg(3, 5)
print( sr + Avg(7, 11) );
```

Функции можно передавать не только постоянные аргументы (числа, символы, строки и др.), но также значения переменных и арифметических выражений:

```
a = 5
b = 7
sr = Avg(a, b+8)
```

В этом случае первый аргумент функции будет равен 5, а второй – 15.

Наша функция **Avg** возвращает вещественное число, поэтому вызовы этой функции можно применять везде, где можно использовать вещественное число, в том числе в арифметических выражениях, условных операторах и циклах. Например,

```
c = 2*Avg(x, y) + z
if Avg(a, b) > 4:
  print( "Свистать всех наверх!" )
```

```
while Avg(a, b) < x:
  a += 1
```

*Найдите значения переменных a, b и x, при которых в результате работы этого фрагмента программы будет выведено сообщение «Да!»:*

```
if Avg(a, b) > x:
  print( "Да!" )
```

*Найдите начальные значения переменных a, b и x, при которых этот цикл выполнится ровно 4 раза:*

```
while Avg(a, b) < x-1:
  b += 1
```

### Функции в других языках программирования

Функция **Avg** на языке Паскаль может быть записана так:

```
function Avg(a, b: integer): real;
begin
  Avg:=(a+b)/2
end.
```

В языке Паскаль заголовок функции начинается словом **function**. В скобках перечисляются параметры (как у процедур), а после скобок через двоеточие записывают тип результата – **real** (вещественное число). Результат функции (возвращаемое значение) нужно сохранить в специальной переменной, имя которой совпадает с именем функции[4].

Вот вариант той же функции на языке C++:

```
float Avg( int a, int b )
{
  return (a+b)/2.0;
}
```

[4] В современных версиях языка Паскаль можно также использовать специальную переменную *Result*.

Как и в языке Python, результат работы функции – это значение, записанное после служебного слова **return** («вернуть»). Тело функции в заключается в фигурные скобки.

По умолчанию (если не указано иначе) в языке C++ при делении целого числа на целое получается целое число (остаток отбрасывается). Чтобы получить вещественный результат, мы разделили сумму $a+b$ (целое число) на вещественное число 2,0.

*Изучите текст программы на языке C++, сравните его с программой на Паскале и выясните, как в языке C++ указывается, что результат работы функции – вещественная величина.*

## Примеры функций

**Задача 1.** Составить функцию, которая определяет наибольшее из двух целых чисел.

Алгоритм определения наибольшего из двух чисел вы уже знаете из курса 8 класса. Остаётся только «завернуть» его в функцию. Например, так:

```
def Max( a, b ):
  if a > b:
    maximum = a
  else:
    maximum = b
  return maximum
```

Одна функция может вызывать другую. Например, можно составить функцию **Max3**, которая возвращает наибольшее из трёх чисел, используя готовую функцию **Max**:

```
def Max3( a, b, c ):
  return Max( Max(a,b), c )
```

*Постройте функцию **Max4**, которая вычисляет наибольшее из четырёх чисел, используя функцию **Max**. Приведите два варианта решения задачи.*

**Задача 2.** Составить функцию, которая вычисляет сумму цифр натурального числа.

Последняя цифра – это остаток от деления числа на 10 (результат операции %). Для того чтобы «удалить» последнюю цифру числа, можно разделить его на 10 без остатка (операция //). Чтобы получить нужный результат, на каждом шаге цикла «отрезаем» от числа последнюю цифру, добавляем её значение к сумме, и затем удаляем её из числа. Цикл заканчивается, когда все цифры удалены и осталось нулевое значение:

```
def sumDigits( n ):
  summa = 0
  while n != 0:
    digit = n % 10
    summa += digit
    n = n //10
  return summa
```

*Как нужно изменить функцию, чтобы она вычисляла количество цифр числа?*

*Как нужно изменить функцию, чтобы она вычисляла количество единиц в двоичной записи числа.*

**Задача 3.** Составить функцию, которая удаляет все двойные пробелы в символьной строке, заменяя их на одиночные.

*Ответьте на вопросы по условию задачи:*

- *какие исходные данные (параметры) принимает функция?*
- *какой тип данных нужно использовать для хранения исходных данных?*
- *что будет результатом работы этой функции?*
- *какой тип данных нужно использовать для хранения результата?*
- *нужно ли внутри функции использовать цикл?*

– *если цикл нужен, то какого типа должен быть цикл (с известным числом повторений или с условием)?*

Функция принимает один параметр – символьную строку, и возвращает тоже символьную строку, поэтому её заголовок выглядит так:

```
def noDSp( s ):
  ...
```

Поскольку двойных пробелов может быть много, в программе нужен цикл. Так как неизвестно, сколько двойных пробелов есть в строке, это будет цикл с условием (**while**). Он должен остановиться, когда двойных пробелов больше не осталось.

*С помощью какой функции можно определить, что в символьной строке больше нет двойных пробелов? Как её нужно использовать?*

*Запишите в тетради операторы языка программирования, с помощью которых:*

– *в переменную p записывается номер символа в строке s, с которого начинается двойной пробел;*

– *из строки s удаляется один символ в позиции p.*

Приведём полный текст функции:

```
def noDSp( s ):
  p = s.find( '  ' )
  while p >= 0:
    s = s[:p] + s[p+1:]
    p = s.find( '  ' )
  return s
```

*Изучите текст функции и ответьте на вопросы:*

– *что означает условие $p \geq 0$ в заголовке цикла?*

– *что произойдет, если удалить строчку с вызовом метода find перед циклом? а если удалить такую же строчку внутри цикла?*

## Логические функции

Программисты часто используют логические функции, возвращающие логические значения («да»/«нет», «истина»/«ложь», *True*/*False*). Такие функции полезны для того, чтобы определять, успешно ли выполнена задача или обладают ли данные каким-то свойством.

Мы напишем простую функцию, которая определяет чётность числа – возвращает значение «да» (в Python оно обозначается как *True*), если число-параметр чётное, и «нет» (*False*), если нечётное:

```
def Even( n ):
  return (n % 2 == 0)
```

В правой части оператора **return** записано условие: результатом функции будет «да» (*True*), если условие истинно, и «нет» (*False*), если оно ложно. Можно было записать то же самое иначе:

```
if n % 2 == 0:
  return True
else:
  return False;
```

но эта запись более длинная и опытные программисты так не делают.

*Запишите в развёрнутой форме возврат логического значения из функции:*

```
return (a > b + c)
```

*Запишите в краткой форме возврат логического значения из функции*:

```
if a + b > 10:
  return False
else:
  return True
```

Результат, который возвращает логическая функция, можно использовать во всех условиях как обычное логическое значение. Например, так:

```
if Even(a):
  half = a // 2
```

или так:

```
count = 0
while Even(x):
  x = x // 2
  count += 1
```

*Найдите значения переменных a и b, при которых в результате работы этого фрагмента программы будет выведено сообщение «Да!»:*

```
if Even( a+3*b ):
  print( "Да!" )
```

*Найдите значение переменной a, при котором этот цикл выполнится ровно 4 раза:*

```
while Even( a ) and a > 5:
  a = a // 2
```

## Рекурсия

Вы уже знакомы с рекурсивными процедурами, которые вызывают сами себя. Функции тоже могут быть рекурсивными, в некоторых случаях это позволяет записать решение задачи намного проще.

**Рекурсивная функция** — это функция, которая вызывает сама себя.

Вернёмся к задаче вычисления суммы цифр числа. Можно сформулировать алгоритм её решения так: *сумма цифр числа N равна значению последней цифры плюс сумма цифр числа, полученного отбрасыванием последней цифры.*

*Вход:* натуральное число $N$.

Шаг 1. $d = N$ % 10

Шаг 2. $M = N$ // 10

Шаг 3. $s$ = сумма цифр числа $M$

Шаг 4. $sum = s + d$

*Результат: sum.*

Итак, для того чтобы найти сумму цифр числа, нужно сложить его последнюю цифру и *сумму цифр другого числа*, то есть выполнить тот же самый алгоритм, только с другими исходными данными (эта строка в записи алгоритма выделена фоном). Получился *рекурсивный алгоритм*, в программе его можно записать в виде рекурсивной функции:

```
def sumDigRec( N ):
  if N == 0: return 0
  d = N % 10
  s = sumDigRec( N // 10 )
  return s + d
```

*Изучите текст функции и ответьте на вопросы:*

– *зачем добавлен условный оператор в начале функции?*

– *что произойдет, если удалить этот условный оператор?*

– *как можно доказать, что для любого целого числа рекурсия обязательно закончится?*

В рекурсивном варианте функции исчез цикл, поэтому можно сделать вывод: рекурсия может заменить цикл. Верно и обратное: любую рекурсивную функцию можно записать без рекурсии, с помощью циклов. Решение с помощью цикла (оно называется *итерационным*) обычно работает быстрее, чем рекурсивное, и требует меньше памяти. Однако рекурсивное решение очень часто короче и проще для понимания.

**Практическая работа №22. Функции**

**Практическая работа №23. Функции-2**

## Выводы:

- Функция — это подпрограмма, которая возвращает результат (число, строку символов и др.).
- Вызов функции можно использовать в арифметических выражениях и условиях так же, как и переменную того типа, который возвращает функция.

- В теле функции можно вызывать другие функции и процедуры.
- Логическая функция возвращает логическое значение «истина» (True) или «ложь» (False).
- Рекурсивная функция – это функция, которая вызывает сама себя.

*Нарисуйте интеллект-карту этого параграфа.*

## Вопросы и задания

1. Чем функция отличается от процедуры?
2. Определите, какие распоряжения начальника можно считать вызовом процедуры, а какие – вызовом функции:
   а) «Проводите Ивана Ивановича!»
   б) «Принесите, пожалуйста, кофе!»
   в) «Подготовьте годовой отчёт!»
   г) «Постройте конуру для собаки!»
3. Как по тексту программы определить, значение какого типа возвращает функция?
4. Сравните рекурсивное решение задачи о сумме цифр числа и решение с помощью цикла. Какое из них вам больше нравится? Обсудите этот вопрос в классе.

## Задачи

1. Напишите функцию, которая вычисляет среднее арифметическое пяти целых чисел.
2. Напишите функцию, которая находит количество цифр в десятичной записи числа.
3. Напишите функцию, которая находит количество единиц в двоичной записи числа.
4. Напишите функцию, которая удаляет из символьной строки все пробелы в начале строки и возвращает новую строку.
5. Напишите функцию, которая возвращает последнюю цифру десятичной записи числа.

6. Напишите функцию, которая определяет минимальное из пяти чисел.
7. Напишите функцию, которая «разворачивает» десятичную запись трёхзначного числа наоборот, например, из числа 123 получается 321, а из 210 – 12.
8. Напишите функцию, которая возвращает количество цифр в восьмеричной записи числа. Число вводится в десятичной системе счисления.
9. Напишите функцию, которая возвращает количество делителей натурального числа.
10. *На соревнованиях выступление спортсмена оценивают 5 экспертов, каждый из них выставляет оценку в баллах (целое число). Для получения итоговой оценки лучшая и худшая из оценок экспертов отбрасываются, а для оставшихся трёх находится среднее арифметическое. Напишите функцию, которая принимает 5 оценок экспертов и возвращает итоговую оценку спортсмена.
11. Напишите функцию, которая удаляет из символьной строки все пробелы в начале и в конце строки и возвращает новую строку.
12. Напишите функцию, которая возвращает первое слово из символьной строки (слева и справа от этого слова может быть сколько угодно пробелов).
13. Напишите функцию, которая заменяет в символьной строке все точки на запятые и возвращает новую строку.
14. *На веб-странице команды разметки (тэги) заключаются в угловые скобки <>. Напишите функцию, которая удаляет в символьной строке все тэги и возвращает новую строку.
15. *Напишите функцию, которая вычисляет значение арифметического выражения, содержащего только целые числа и знаки сложения и вычитания. Выражение записано в символьной строке.
16. Напишите логическую функцию, которая возвращает значение «истина», если переданное ей число помещается в 8-

битную ячейку памяти (вспомните, какое минимальное и максимальное число можно записать с помощью 8 битов).

17. *Напишите логическую функцию, которая возвращает значение «истина», если переданное ей число простое (делится только на само себя и на единицу).
18. *Напишите рекурсивную и нерекурсивную функции, которые вычисляют наибольший общий делитель (НОД) двух натуральных чисел с помощью алгоритма Евклида. *Алгоритм Евклида*: заменять наибольшее из двух чисел на их разность до тех пор, пока числа не станут равны. Полученное значение этих чисел и есть их НОД.
19. *Напишите рекурсивную и нерекурсивную функции, которые вычисляют наибольший общий делитель (НОД) двух натуральных чисел с помощью модифицированного (улучшенного) алгоритма Евклида. *Модифицированный алгоритм Евклида*: заменять наибольшее из двух чисел на остаток от деления большего на меньшее до тех пор, пока этот остаток не станет равен нулю. Тогда второе число и есть НОД.