Факультет вычислительной
математики и кибернетики

Решение сложных задач по программированию в экзаменах по информатике

Университетская суббота для школьников

А.В. Якушин, к.п.н., доцент, ВМК МГУ

Факультет ВМК МГУ имени М.В. Ломоносова

Базовые знания и навыки

Язык Python

1. знать основные принципы построения программы на языке программирования;
2. знать основные конструкции языка Python: объявление переменных, оператор присваивания, условный оператор, циклы
3. знать и уметь использовать основные структуры данных Python: строка, список, словарь, множество
4. уметь написать функцию и рекурсивную функцию на Python
5. уметь читать программу
6. уметь читать данные из файла: текстовые и строковые

Методы программирования

1. знать и уметь реализовывать числовые алгоритмы
2. знать и уметь реализовывать алгоритмы обработки массивов и последовательностей
3. знать и уметь использовать простой перебор без оптимизации
4. знать и уметь использовать динамическое программирование
5. знать и уметь использовать регулярные выражения

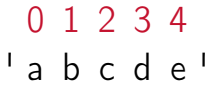
Строка

Strings

my_string = 'abcde'

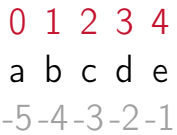
Strings

```
my_string = 'abcde'
```



Strings

my_string = 'abcde'



Индексы и срезы

Можно получить доступ к символу по его **индексу** (номеру): [index]

```
print(my_string[2])
```

Индексы и срезы

Можно получить доступ к символу по его **индексу** (номеру): $[\mathrm{index}]$
$\mathrm{print}(\mathrm{my\_string}[2]) \Rightarrow$ вывод c

Индексы и срезы

Можно получить доступ к символу по его **индексу** (номеру): [index]
print(my_string[2])⇒ вывод с
print(my_string[-2])

Индексы и срезы

Можно получить доступ к символу по его **индексу** (номеру): [index]
print(my_string[2])⇒ вывод c
print(my_string[-2])⇒ вывод d

Индексы и срезы

Срез выдает часть строки [start:stop:step]

Индексы и срезы

Срез выдает часть строки [start:stop:step]

```
s = 'Python'
s[1] # => 'y'
s[0:4] # => 'Pyth'
s[:3] # => 'Pyt'
s[3:] # => 'hon'
s[:] # => 'Python'
```

Индексы и срезы

Срез выдает часть строки [start:stop:step]

```
s = 'Python'
s[:5:2] # => 'Pto'
s[1:4:3] # => 'y'
s[::3] # => 'Ph'
s[::-1] # => 'nohtyP'
```

Операции со строками

$\mathrm{str1} + \mathrm{str2} \Rightarrow$ **Конкатенация** (сцепление) $\mathrm{str1}$ и $\mathrm{str2}$

Операции со строками

$\mathrm{str1} + \mathrm{str2} \Rightarrow$ **Конкатенация** (сцепление) $\mathrm{str1}$ и $\mathrm{str2}$

$\mathrm{str1}$ * $n \Rightarrow$ повторить $\mathrm{str1}$ n раз.

Операции со строками

$\mathrm{str1} + \mathrm{str2} \Rightarrow$ **Конкатенация** (сцепление) $\mathrm{str1}$ и $\mathrm{str2}$

$\mathrm{str1}$ * $n \Rightarrow$ повторить $\mathrm{str1}$ n раз.

Также строки можно сравнивать.

Список

Список

Представьте себе переменные, но с безграничной емкостью ...
sunnyside = ['Potato Head', 'Hamm', 'Buzz', 'Slinky Dog']

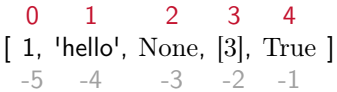
Список

```
empty_list = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]
```

```
mixed_list = [4, 13, 'hello']
```

Доступ к элементам

values = [1, 'hello', None, [3], True]



Доступ к элементам происходит по **индексу**.

```
print(values[2])
values[2] = 'new value'
```

Добавление элементов

Добавить элементы в конец списка можно с помощью **append()**

```
numbers = [1, 2, 3]
numbers.append(7) # => numbers = [1, 2, 3, 7]
numbers.append(11) # => numbers = [1, 2, 3, 7, 11]

a_list = [1, 'a', 'python', 4.2]
a_list.append(3) # => a_list = [1, 'a', 'python', 4.2, 3]
a_list.append('hello')
# => a_list = [1, 'a', 'python', 4.2, 3, 'hello']
```

Добавление элементов

Добавить элементы в конец списка можно с помощью **append()**

```
x = [1, 2, 3]
y = [4, 5]
x.append(y) # => x = [1, 2, 3, [4, 5]]
```

Срезы

```
x = [1, 2, 3, 4, 5]

x[2:4] # => [3,4]
x[3:4] # => [4]
x[1:-1] # => [2,3,4]
```

Срезы

```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[:3] # => ['a', 'b', 'c']
y[2:] # => ['c', 'd', 'e', 'f']
y[:-1] # => ['a', 'b', 'c', 'd', 'e']
y[:] # => ['a', 'b', 'c', 'd', 'e', 'f']
```

Срезы

```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[1:5:2] # => ['b', 'd']
y[::3] # => ['a', 'd']
```

Срезы

```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[::-1] # => ['f', 'e', 'd', 'c', 'b', 'a']
```

Удаление элемента remove()

```
numbers = [1, 2, 3, 4]
letters = ['a', 'b', 'c']
numbers_repeated = [1, 2, 5, 4, 2, 6]

numbers.remove(2) # => [1, 3, 4]
letters.remove('b') # => ['a', 'c']
numbers_repeated.remove(2) # => [1, 5, 4, 2, 6]

my_list = [1, 'a']
my_list.remove('b') # => ValueError
```

Оператор in

Найти элемент в списке можно с помощью оператора **in**

Оператор in

Найти элемент в списке можно с помощью оператора **in**

```
0 in [] # => False
'y' in 'Python' # => True
23 in ['hello', 40, 'a', 5] # => False
23 in ['hello', 40, 'a', 23] # => True
23 in ['hello', 40, 'a', '23'] # => False
```

Оператор in

Найти элемент в списке можно с помощью оператора **in**

```
0 in [] # => False
'y' in 'Python' # => True
23 in ['hello', 40, 'a', 5] # => False
23 in ['hello', 40, 'a', 23] # => True
23 in ['hello', 40, 'a', '23'] # => False
```

Работает со списками, строками и range

Функция len()

— служит для определения размера списков, строк и т. д.

Функция len()

— служит для определения размера списков, строк и т. д.

$\mathrm{len}()$

Функция len()

— служит для определения размера списков, строк и т. д.

len()

```
s = 'Python'
len(s) # => 6

my_list = [0, 1, 2, 3]
len(my_list) # => 4
```

Операции с list

list.append(x): Добавить x в конец списка

Операции с list

list.append(x): Добавить x в конец списка
list.insert(i, x): Добавить x в позицию i

Операции с list

list.append(x): Добавить x в конец списка
list.insert(i, x): Добавить x в позицию i
list.pop(i=-1): Вернуть и удалить элемент с индексом i (0 по умолчанию)

Операции с list

list.append(x): Добавить x в конец списка
list.insert(i, x): Добавить x в позицию i
list.pop(i=-1): Вернуть и удалить элемент с индексом i (0 по умолчанию)
list.remove(x): Удалить первое совпадение x

Операции с list

list.append(x): Добавить x в конец списка
list.insert(i, x): Добавить x в позицию i
list.pop(i=-1): Вернуть и удалить элемент с индексом i (0 по умолчанию)
list.remove(x): Удалить первое совпадение x
list.extend(iterable): Добавить коллекцию iterable в конец списка

Операции с list

list.append(x): Добавить x в конец списка
list.insert(i, x): Добавить x в позицию i
list.pop(i=-1): Вернуть и удалить элемент с индексом i (0 по умолчанию)
list.remove(x): Удалить первое совпадение x
list.extend(iterable): Добавить коллекцию iterable в конец списка
list[i] = new_value: Изменить значение элемента с номером i

Операции с list

list.index(value, start=0, stop=len(list)):

Операции с list

list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.

Операции с list

list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.
list.count(value): Подсчет сколько раз встречается value.

Операции с list

list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.
list.count(value): Подсчет сколько раз встречается value.
list.reverse(): реверс списка

Операции с list

list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.
list.count(value): Подсчет сколько раз встречается value.
list.reverse(): реверс списка
list.sort(): сортировка списка

Операции с list

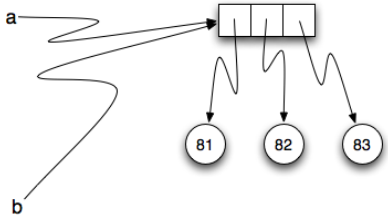
list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.
list.count(value): Подсчет сколько раз встречается value.
list.reverse(): реверс списка
list.sort(): сортировка списка
list[basic_slice] = iterable: Заменить стандартный срез на коллекцию iterable (возможно другого размера): numbers[:] = []

Операции с list

list.index(value, start=0, stop=len(list)):
Возвращает позицию первого вхождения value.
list.count(value): Подсчет сколько раз встречается value.
list.reverse(): реверс списка
list.sort(): сортировка списка
list[basic_slice] = iterable: Заменить стандартный срез на коллекцию iterable (возможно другого размера): numbers[:] = []
list[extended_slice] = iterable: Заменить расширенный срез (с шагом) на коллекцию iterable такого же размера.

Псевдонимы и клонирование

- К **одному и тому же объекту** может относиться более одной переменной.!
- Это известно как **aliasing** — псевдоним.
- Для списков $\mathrm{list.copy}() \Rightarrow$ возвращает неполную копию списка.
- Копируем только ссылки, а не внутренние значения.



Клонирование

$$\text{my\_secret\_box} = [0, 1, 2]$$

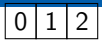
Клонирование

0	1	2

$\mathrm{my\_secret\_box} = [0, 1, 2]$

Клонирование

my_secret_box



$\text{my\_secret\_box} = [0,\ 1,\ 2]$

Клонирование

$\mathrm{my\_secret\_box} = [0,\ 1,\ 2]$

Клонирование

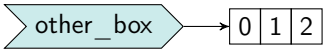

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
```

Клонирование

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
```

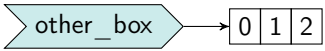
Клонирование

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
```



Клонирование

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
other_box.remove(2)
```



Клонирование

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
other_box.remove(2)
```

Клонирование

```
my_secret_box = [0, 1, 2]
other_box = my_secret_box.clone()
other_box.remove(2)
print(my_secret_box)
```

Словарь

Словарь

- Коллекция пар **key**−**value**.

Словарь

- Коллекция пар **key**$-$**value**.
- **Невозможно** использовать **срезы**, **можно** выполнять просмотр с помощью циклов for.

Словарь

- Коллекция пар **key**−**value**.
- **Невозможно** использовать **срезы**, **можно** выполнять просмотр с помощью циклов for.
- В общем случае **не упорядочены**.

Словарь

- Коллекция пар **key**−**value**.
- **Невозможно** использовать **срезы**, **можно** выполнять просмотр с помощью циклов for.
- В общем случае **не упорядочены**.
- В новых версиях Python 3.7 сохраняется порядок по добавлению (но это не точно).

Как создавать словари?

- $\{\}/\mathrm{dict}()$: пустой словарь

Как создавать словари?

- { }/dict(): пустой словарь
- d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}

Как создавать словари?

- { }/dict(): пустой словарь
- d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
  print(d['one']) # ⇒ 1

Пример

```
my_students = {'Анна': ['economics', 'freshman'],
               'Иван': ['psychology', 'master'],
               'Антон': ['law', 'junior'],
               'Нестор': ['international relations', 'freshman']}

for student, info in my_students.items():
    print(f'{student} studies {info[0]}')
# Иван закончил обучение :(
my_students.pop('Иван')
# Новый студент
my_students['Арсений'] = ['industrial engineering', 'junior']
# Заменим предмет
my_students['Антон'][1] = 'sophomore'
```

Кортеж

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.
- Элементы нельзя добавлять/удалять/изменять после создания кортежа.

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.
- Элементы нельзя добавлять/удалять/изменять после создания кортежа. my_tuple = (1, [1, 2], 'a')

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.
- Элементы нельзя добавлять/удалять/изменять после создания кортежа. my_tuple = (1, [1, 2], 'a')
- len(my_tuple) $\Rightarrow$

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.
- Элементы нельзя добавлять/удалять/изменять после создания кортежа. my_tuple = (1, [1, 2], 'a')
- len(my_tuple) $\Rightarrow$ 3
- my_tuple.append(3) $\Rightarrow$

Кортеж

- **Неизменяемая** упорядоченная последовательность элементов.
- Также как list, можно использовать **индексы**, **срезы** и цикл for.
- Элементы нельзя добавлять/удалять/изменять после создания кортежа. my_tuple = (1, [1, 2], 'a')
- len(my_tuple) ⇒ 3
- my_tuple.append(3) ⇒ AttributeError: 'tuple' object has no attribute 'append'

Кортеж

() / tuple(): пустой кортеж,

Кортеж

() / tuple(): пустой кортеж,

```
my_list = [1, 2, 3]
my_tuple = ('a', my_list) # ('a', [1, 2, 3, 4])
my_list.append(4)
print(my_tuple)
my_list += [5, 6, 7]  # my_list.extend(...)
print(my_tuple)
my_tuple += (1, 2)# my_tuple = my_tuple + (1, 2)
print(my_tuple)
```

Множество

Множество

- **Неупорядоченная** совокупность **различных** элементов.

Множество

- **Неупорядоченная** совокупность **различных** элементов.
- **Невозможно** использовать **индексы/срезы**, **возможно** перебирать элементы с помощью цикла for.

Множество

- **Неупорядоченная** совокупность **различных** элементов.
- **Невозможно** использовать **индексы/срезы**, **возможно** перебирать элементы с помощью цикла for.
- **Изменяемая структура данных**: методы add(element), remove(element) .

Множество

- **Неупорядоченная** совокупность **различных** элементов.
- **Невозможно** использовать **индексы/срезы**, **возможно** перебирать элементы с помощью цикла for.
- **Изменяемая структура данных**: методы $\mathrm{add(element)}$, $\mathrm{remove(element)}$. $\mathrm{my\_set} = \{1, 2, 3, 4, 2\}$

Множество

- **Неупорядоченная** совокупность **различных** элементов.
- **Невозможно** использовать **индексы/срезы**, **возможно** перебирать элементы с помощью цикла for.
- **Изменяемая структура данных**: методы add(element), remove(element) . my_set = {1, 2, 3, 4, 2} set() ({ } занято для dict)

Множество

- **Неупорядоченная** совокупность **различных** элементов.
- **Невозможно** использовать **индексы/срезы**, **возможно** перебирать элементы с помощью цикла for.
- **Изменяемая структура данных**: методы add(element), remove(element) . my_set = {1, 2, 3, 4, 2} set() ({ } занято для dict)
- Использует операции над множествами: **union**, **intersection**, **difference**, **symmetric difference**.

Множества

```
anna = {'Иван', 'Сергей', 'Михаил'}
mike = {'Наталья', 'Зина', 'Даша'}
steve = {'Наталья', 'Николай', 'Петр'}
mary = {'Сергей', 'Дима', 'Фома'}
```

Множества

```
# пересечение &
print(mike.intersection(steve))  # => {'Наталья'}
print(anna & mike)  # => set()
# объединение |
print(anna.union(mary))  # => {'Фома', 'Сергей', 'Дима',
                         #  'Иван', 'Михаил'}
print(steve | anna | mike | mary)  # => all names
# разность -
print((mike - steve))  # => {'Даша', 'Зина'}
# симметрическая разность ^
print(anna.symmetric_difference(mary))
# => {'Иван', 'Фома', 'Михаил', 'Дима'}}
```

Числовые алгоритмы

Числовые алгоритмы

1. Битовые операции
2. Большее из двух, из трех
3. Алгоритм Евклида
4. Сумма цифр числа
5. Перевод в другую систему счисления
6. Нахождение делителей числа
7. Определение простоты числа
8. Решение линейных и квадратных уравнений
9. Табулирование функции
10. Поиск максимума функции

Битовые операции

Над битами двух целых операндов можно выполнять логические операции: not, and, or, xor.

Отличие между побитовыми и логическими операциями состоит в том, что побитовые (поразрядные) операции выполняются над отдельными битами операндов, а не над их значением в десятичном (обычно) представлении.

Битовые операции

В Паскаль определены еще две операции shl и shr, которые сдвигают последовательность битов на заданное число позиций влево или вправо соответственно.

При этом биты, которые выходят за разрядную сетку, теряются.

При выполнении операции shl освободившиеся справа биты заполняются нулями.

При выполнении операции shr освободившиеся слева биты заполняются единицами при сдвиге вправо отрицательных значений и нулями в случае положительных значений.

Пример битовых операций

```
1  a = 10
2  b = 4
3  # AND
4  print("a & b =", a & b)
5  # OR
6  print("a | b =", a | b)
7  # NOT
8  print("~a =", ~a)
9
10 # XOR
11 print("a ^ b =", a ^ b)
12 # right
13 print("a >> 1 =", a >> 1)
14 print("b >> 1 =", b >> 1)
15 a = 5
16 b = -10
17 # left
18 print("a << 1 =", a << 1)
19 print("b << 1 =", b << 1)
```

Задачи на битовые операции

- Определить является ли число степенью двойки
- Проверить число на четность
- вычислить $(-1)^n$
- Обнулить крайний правый бит числа
- Получить следующее большее число с тем же количеством единичных бита
- и другие (Г. Уоррен Алгоритмические трюки для программистов)

Большее из двух

```
1  if a > b:
2      M = a
3  else:
4      M = b
5
6  print("Maximum", M)
```

При всей простоте это довольно популярный алгоритм. Сравнение двух переменных может быть реализовано достаточно сложным способом.

Например, попробуйте придумать сравнение при котором $7 > 65$.

Большее из трех

```
#1
if (a>=b) and (a>=c):
    zmax=a
if (b>=a) and (b>=c):
    zmax=b
if (c>=a) and (c>=b):
    zmax=c
#2
if a>=b:
    zmax=a
else:
    zmax=b
if zmax<c:
    zmax=c
#3
if a>=b:
    if a>=c:
        zmax=a
    else:
        zmax=c
else:
    if b>=c:
        zmax=b
    else:
        zmax=c
```

Первый вариант хуже чем второй и третий, так как в некоторых случаях выполнится несколько условных операторов.

Алгоритм Евклида

Алгоритм Евклида — это способ нахождения наибольшего общего делителя (НОД) двух целых чисел. Оригинальная версия алгоритма, когда НОД находится вычитанием, была открыта Евклидом (III в. до н. э). В настоящее время чаще при вычислении НОД алгоритмом Евклида используют деление, так как данный метод эффективнее.

Наибольший общий делитель пары чисел – это самое большое число, которое нацело делит оба числа пары.

Алгоритм Евклида

Вычитание

```
1  while (a != 0) and (b != 0):
2      if a > b:
3          a = a - b
4      else:
5          b = b - a
6
7  gcd=a + b
```

Алгоритм Евклида

Деление

```
1  while (a != 0) and (b != 0):
2      if a > b:
3          a = a mod b
4      else:
5          b = b mod a
6
7  gcd=a + b
```

Алгоритм Евклида

Рекурсия

```
1  def gcd(a, b):
2      if a == 0 :
3          return b
4
5      return gcd(b%a, a)
```

Сумма цифр числа

Алгоритм решения задачи:

1. сумме присвоить ноль.
2. Пока n больше нуля
3. найти остаток от деления n на 10 (т.е. последнюю цифру числа), добавить его к сумме и увеличить произведение;
4. избавиться от последнего разряда числа n путем деления нацело на число 10.

Сумма цифр числа

```
1  def getSum(n):
2      sum = 0
3      while (n != 0):
4          sum = sum + (n % 10)
5          n = n//10
6
7      return sum
8
9  n = 12345
10 print(getSum(n))
```

Как можно модифицировать алгоритм, чтобы найти количество цифр? наибольшую цифру?

Что произойдет если 10 заменить на 8?

Перевод в другую систему счисления

Используем стандартный алгоритм: деление с остатком на основание системы счисления. Последовательность остатков образует цифры нового числа.

Основная проблема: цифры нового числа получаются в обратном порядке.

Способы решения:

1. рекурсивная функция
2. запись в новое число
3. массив

Перевод из десятичной в двоичную

Рекурсия

```
1  def DecimalToBinary(num):
2
3      if num >= 1:
4          DecimalToBinary(num // 2)
5      print(num % 2, end = '')
```

Перевод из десятичной в двоичную

Запись в новое число

```
1  n=13
2  res=0
3  t=1
4  while n>0:
5      b=n%2
6      res=b*t+res
7      n=n//2
8      t=t*10
9
10 print(res)
```

Делители числа

Вариант 1

```
1  def printDivisors(n):
2      i = 1
3      while i <= n:
4          if (n % i==0):
5              print(i)
6          i = i + 1
7
8  printDivisors(100)
```

Делители числа

Вариант 2

```
1  def printDivisors(n) :
2      i = 1
3      while i <= math.sqrt(n):
4          if (n % i == 0) :
5              if (n / i == i) :
6                  print(i)
7              else :
8                  print(i , n/i)
9          i = i + 1
10
11 printDivisors(100)
```

Простое число

Используем тот факт, что если у числа n есть делитель, то он находится в интервале $[2; \sqrt{n}]$

```
1  def is_prime (n):
2      flag = True
3
4      k=2
5      while k*k <=n and flag==True:
6          if (n % k == 0):
7              flag = False
8      else:
9              k=k+1
10     return flag
```

Простое число

1. использовать меньший интервал (гипотеза Римана)
2. использовать список простых числел в качестве делителей
3. решето Эратосфена

Решение линейного уравнения

Решим уравнение $ax + b = 0$.

```
a,b=map(float,input().split())
if a==0:
    if b==0:
        print('INF')
    else:
        print('NO')
else:
    x=-b/a
    print(x)
```

Решение квадратного уравнения

Решим уравнение $ax^2 + bx + c = 0$.

```
1  a,b,c=map(int,input())
2  if a==0:
3      #bx+c=0
4  else:
5  d=b*b-4*a*c
6  if d<0:
7      print('NO')
8  if d==0:
9      print('x= ',-b/(2*a))
10 if d>0:
11     print('x1= ',(-b+sqrt(d))/(2*a))
12     print('x2= ',(-b-sqrt(d))/(2*a))
```

Табулирование функции

Задача табулирования (вывод значений с шагом) функции $f(x) = x^2 - 3x + 2$ на отрезке $[a,b]$ с шагом h.

```
1  def f(x):
2    return x*x - 3*x + 2
3
4  a,b,h=map(float,input().split())
5  x=a
6  while x<=b:
7     print('x= ',x,' f(x)= ',f(x))
8     x=x+h
```

Наибольшее заначение функции на отезке с шагом

Задача поиск максимума функции $f(x) = x^2 - 3x + 2$ на отрезке $[a,b]$ с шагом h.

```
1  def f(x):
2    return x*x - 3*x + 2
3
4  a,b,h=map(float, input().split())
5
6  x=a
7  R=f(a)
8  t=a
9  while x<=b:
10    if (f(x)>R):
11      t=x
12      R=f(x)
13    x=x+h
14 print('maximum',t,' ',R)
```

Базовые алгоритмы обработки последовательностей

Базовые алгоритмы обработки последовательностей

1. Сумма элементов
2. Сумма и количество элементов по условию
3. Среднее арифметическое
4. Среднее арифметическое по условию
5. Наибольший и наименьший элементы и их номера
6. Наибольший/наименьший по условию
7. Количество элементов равных наибольшему
8. Поиск двух наибольших элементов (второго максимума)
9. Поиск элемента равного заданному
10. Поиск подпоследовательности по условию

Базовые алгоритмы обработки последовательностей

Для определенности будем считать, что последовательность задана количеством элементов и это количество уже прочитано в переменную n.

Мы рассмотрим только сам алгоритм и его реализацию без детализации ввод вывода и источника данных (консоль, файл и пр.)

Аккумулятор — переменная, которая изменяет свое значение на значение добавляемой величины. Пример: сумма, произведение.

Счетчик — переменная, которая изменяет свое значение на 1 при определенном условии. Пример: подсчет количества, счетчик цикла.

Сумма элементов

Описание алгоритма

1. Предусловия: аккумулятор = 0
2. В цикле читаем элементы последовательности и добавляем их в аккумулятор.
3. Постусловия: вывод аккумулятора на экран

Сумма элементов

Псевдокод

Данные: Последовательность чисел размера n
Результат: Сумма элементов
аккумулятор=0;
цикл *от 1 до n* **выполнять**
    прочитать элемент;
    аккумулятор=аккумулятор+элемент;
конец
вывод на экран;

Алгоритм 1: Сумма элементов

Сумма и количество элементов по условию

Описание алгоритма

1. Предусловия: аккумулятор $= 0$, счетчик $= 0$.
2. В цикле читаем элементы последовательности.
3. Для каждого элемента проверяем условие и если оно выполнено добавляем элемент в аккумулятор и увеличиваем счетчик.
4. Постусловия: вывод аккумулятора и счетчика на экран.

Сумма и количество элементов по условию

Псевдокод

Данные: Последовательность чисел размера n
Результат: Сумма и количество элементов по условию
аккумулятор=0;
счетчик=0;
цикл *от 1 до n* **выполнять**
    прочитать элемент;
    если *условие* **тогда**
        аккумулятор=аккумулятор+элемент;
        счетчик=счетчик+1;
    конец
конец
вывод на экран;

Алгоритм 2: Сумма и количество элементов по условию

Среднее арифметическое

Описание алгоритма

1. Предусловия: аккумулятор = 0
2. В цикле читаем элементы последовательности и добавляем их в аккумулятор.
3. Постусловия: расчет среднего арифметического и вывод на экран

Данные: Последовательность чисел размера n
Результат: Сумма элементов
аккумулятор=0;
цикл *от 1 до n* **выполнять**
    прочитать элемент;
    аккумулятор=аккумулятор+элемент;
конец
среднее арифметическое=$\dfrac{\text{аккумулятор}}{\text{количество элементов}}$
вывод на экран;

Алгоритм 3: Среднее арифметическое

Среднее арифметическое по условию

Описание алгоритма

1. Предусловия: аккумулятор = 0
2. Для каждого элемента проверяем условие и если оно выполнено добавляем элемент в аккумулятор и увеличиваем счетчик.
3. Постусловия:
   - 3.1 если элементов удовлетворяющих условию нет, то ОШИБКА
   - 3.2 иначе расчет среднего арифметического и вывод на экран

Данные: Последовательность чисел размера n
Результат: Сумма элементов
аккумулятор=0;
счетчик=0;
цикл *от 1 до n* **выполнять**
  прочитать элемент;
  если *условие* **тогда**
    аккумулятор=аккумулятор+элемент;
    счетчик=счетчик+1;
  конец
конец
если *счетчик>0* **тогда**
  среднее арифметическое=$\frac{\text{аккумулятор}}{\text{количество элементов}}$
иначе
  ОШИБКА
конец

Алгоритм 4: Среднее арифметическое по условию

Наибольший и наименьший элементы и их номера

Описание алгоритма

1. Предусловия: инициализация наибольшего элемента начальным значением.
2. Для каждого элемента если он больше наибольшего, то запоминаем этот элемент.
3. Постусловия: вывод на экран

Для поиска наименьшего все аналогично.

Наибольший и наименьший элементы и их номера

Псевдокод

Данные: Последовательность чисел размера n
Результат: Наибольший и его номер
наибольший=первый элемент;
номер=1;
цикл *от 2 до n* **выполнять**
    прочитать элемент;
    если *элемент>наибольший* **тогда**
        наибольший=элемент;
        номер=текущее значение счетчика цикла;
    конец
конец
вывод на экран;

Алгоритм 5: Наибольший и наименьший элементы и их номера

Наибольший/наименьший по условию

Анализ задачи

Поиск наибольшего элемента, как мы видели раньше, состоит из двух этапов:

1. Инициализация наибольшего элемента начальным значением.
2. Поиск наибольшего

Тут возникает проблема: какое начальное значение взять? Начальным значением для наибольшего, в этом случае, должен быть первый элемент удовлетворяющий условию.

В предыдущем примере мы брали первый элемент последовательности. Но первый не обязан удовлетворять условию. Хуже того в последовательности может не быть нужных нам элементов вообще.

В решении такой задачи возможны два варианта:

1. Известен диапазон, в котором находятся элементы последовательности.
2. Общий случай (произвольные элементы).

Рассмотрим их по отдельности.

Наибольший по условию: Известен диапазон

Описание алгоритма

Пусть элементы последовательности лежат на отрезке [a;b].

1. Предусловия: наибольший это число слева от левой границы отрезка.
2. Если элемент удовлетворяет условию и он больше наибольшего, то запоминаем этот элемент.
3. Постусловия: вывод на экран

Для поиска наименьшего все аналогично.

Наибольший по условию: Известен диапазон

Псевдокод

Пусть элементы последовательности лежат на отрезке [a;b].

Данные: Последовательность чисел размера *n*
Результат: Наибольший и по условию
наибольший=a-1;
цикл *от 2 до n* **выполнять**
    прочитать элемент;
    если *условие И элемент>наибольший* **тогда**
        наибольший=элемент;
    конец
конец
вывод на экран;

Алгоритм 6: Наибольший по условию: Известен диапазон

Наибольший по условию: Общий случай

Описание алгоритма

1. Найти первый элемент удовлетворяющий условию для инициализации наибольшего
2. Если элемент удовлетворяет условию и он больше наибольшего, то запоминаем этот элемент.
3. Постусловия: вывод на экран

Для поиска наименьшего все аналогично.

Наибольший по условию: Общий случай

Псевдокод

Пусть элементы последовательности лежат на отрезке [a;b].

Данные: Последовательность чисел размера *n*
Результат: Наибольший по условию
цикл *от 1 до n* **выполнять**
    прочитать элемент;
    если *условие выполнилось первый раз* **тогда**
        наибольший=элемент;
        условие уже было;
    иначе
        если *условие И элемент>наибольший* **тогда**
            наибольший=элемент;
        конец
    конец
конец
вывод на экран;

Алгоритм 7: Наибольший по условию: Общий случай

Количество элементов равных наибольшему

Описание алгоритма

1. Предусловия: наибольший это первый, количество равно 1.
2. Если элемент больше наибольшего, то запоминаем этот элемент и сбрасываем количество на 1
3. Если элемент равен наибольшему, то увеличиваем количество на 1
4. Постусловия: вывод на экран

Количество элементов равных наибольшему

Псевдокод

Данные: Последовательность чисел размера n
Результат: Количество равных наибольшему
наибольший=первый элемент;
количество=1;
цикл *от 2 до n* **выполнять**
    прочитать элемент;
    если *элемент>наибольшего* **тогда**
        наибольший=элемент;
        количество=1;
    иначе
        если *элемент = наибольший* **тогда**
            количество=количество+1;
        конец
    конец
конец
вывод на экран;

Алгоритм 8: Количество элементов равных наибольшему

Поиск двух наибольших элементов (второго максимума)

Описание алгоритма

1. Предусловия: первый и второй наибольшие это максимум и минимум из первых двух элементов последовательности.
2. Если элемент больше наибольшего, то второй это первый, а первый это элемент
3. Если элемент не больше наибольшего, но больше второго, то второй это элемент
4. Постусловия: вывод на экран

Поиск двух наибольших элементов

Псевдокод

Данные: Последовательность чисел размера n
Результат: Два максимума
наибольший1=максимум(первый, второй);
наибольший2=минимум(первый, второй);
цикл *от 3 до n* **выполнять**
    прочитать элемент;
    если *элемент>наибольшего1* **тогда**
        наибольший2=наибольший1;
        наибольший1=элемент;
    иначе
        если *элемент > наибольший2* **тогда**
            наибольший2=элемент;
        конец
    конец
конец
вывод на экран;

Алгоритм 9: Поиск двух наибольших элементов

Поиск элемента равного заданному

Описание алгоритма

1. Предусловия: признак того, что элемент найден
2. Если элемент равен заданному, то поиск завершен иначе перейти к следующему элементу
3. Постусловия: вывод на экран

Поиск элемента равного заданному

Псевдокод

Данные: Последовательность чисел размера n, число
Результат: Есть или нет равный заданному
нашли=ЛОЖЬ;
счетчик=1; **до тех пор, пока** *не просмотрены все элементы и заданный не найден* **выполнять**
    прочитать элемент;
    если *элемент=заданному* **тогда**
        нашли=ИСТИНА;
    иначе
        перейти к следующему;
    конец
конец
вывод на экран;

Алгоритм 10: Поиск элемента равного заданному

Поиск подпоследовательности по условию

Описание алгоритма

Подпоследовательность это группа идущих подряд элементов последовательности. Например для [1,1,2,3,4,1,2,3] одна из подпоследовательностей будет [2,3,4,1].

1. Предусловия: инициализация максимальной и текущей длины, чтение первого элемента
2. Если текущий элемент больше предыдущего, то увеличиваем текущую длину
3. Иначе сравниваем текущую длину с максимальной и сбрасываем счетчик текущей длины
4. Постусловия: вывод на экран

Поиск подпоследовательности по условию

Псевдокод

Данные: Последовательность чисел размера n
Результат: Наибольший п
максимальная длина=0; текущая длина=1; прочитать предыдущий;
цикл *i:=2 to n* **выполнять**
    прочитать текущий элемент;
    если *условие* **тогда**
        текущая длина=текущая длина+1;
    иначе
        если *Текущая длина>максимальной* **тогда**
            максимальная длина=текущая длина;
        конец
        текущая длина=1;
    конец
    предыдущий=текущий;
конец
вывод на экран;

Алгоритм 11: Поиск подпоследовательности по условию

Базовые алгоритмы обработки массивов

Базовые алгоритмы обработки массивов

1. Сумма элементов
2. Сумма и количество элементов по условию
3. Среднее арифметическое
4. Среднее арифметическое по условию
5. Циклический сдвиг массива влево/вправо
6. Реверс массива
7. Количество различных элементов в монотонном массиве
8. Поиск различных элементов массива
9. Вставка элемента
10. Удаление элемента

Базовые алгоритмы поиска и сортировки

1. Наибольший и наименьший элементы и их номера
2. Наибольший/наименьший по условию
3. Количество элементов равных наибольшему
4. Поиск двух наибольших элементов (второго максимума)
5. Поиск элемента равного заданному
6. Поиск подпоследовательности по условию
7. Поиск элементов с заданным расстоянием между индексами
8. Пересечение и разность массивов
9. Бинарный поиск
10. Пузырьковая сортировка
11. Сортировка выбором
12. Сортировка вставками
13. Сортировка за линейное время

Сумма элементов

Описание алгоритма

1. Предусловия: аккумулятор = 0
2. В цикле проходим по элементам массива и добавляем их в аккумулятор.
3. Постусловия: вывод аккумулятора на экран

Данные: Массив arr размера n
Результат: Сумма элементов
аккумулятор=0;
цикл *i от 1 до n* **выполнять**
  аккумулятор=аккумулятор+arr[i];
конец
вывод на экран;

Алгоритм 12: Сумма элементов

Сумма и количество элементов по условию

Описание алгоритма

1. Предусловия: аккумулятор = 0, счетчик = 0.
2. В цикле проходим по элементам массива.
3. Для каждого элемента проверяем условие и если оно выполнено добавляем элемент в аккумулятор и увеличиваем счетчик.
4. Постусловия: вывод аккумулятора и счетчика на экран.

Сумма и количество элементов по условию

Псевдокод

Данные: Массив arr размера n
Результат: Сумма и количество элементов по условию
аккумулятор=0;
счетчик=0;
цикл *i от 1 до n* **выполнять**
    если *условие* **тогда**
        аккумулятор=аккумулятор+arr[i];
        счетчик=счетчик+1;
    конец
конец
вывод на экран;

Алгоритм 13: Сумма и количество элементов по условию

Среднее арифметическое

Описание алгоритма

1. Предусловия: аккумулятор = 0
2. В цикле проходим по элементам и добавляем их в аккумулятор.
3. Постусловия: расчет среднего арифметического и вывод на экран

Данные: Массив arr размера n
Результат: Сумма элементов
аккумулятор=0;
цикл *i от 1 до n* **выполнять**
  аккумулятор=аккумулятор+элемент;
конец
среднее арифметическое=$\frac{\text{аккумулятор}}{\text{количество элементов}}$
вывод на экран;

Алгоритм 14: Среднее арифметическое

Циклический сдвиг массива влево/вправо

Описание алгоритма

Первоначальное состояние массива A:

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
2	3	-4	1	0	9	12

Состояние массива A после циклического сдвига на один элемент влево:

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
3	-4	1	0	9	12	2

Наибольший и наименьший элементы и их номера

Описание алгоритма

1. Предусловия: инициализация наибольшего элемента начальным значением.
2. Для каждого элемента если он больше наибольшего, то запоминаем этот элемент.
3. Постусловия: вывод на экран

Для поиска наименьшего все аналогично.

Наибольший и наименьший элементы и их номера

Псевдокод

Данные: Массив arr размера *n*
Результат: Наибольший и его номер
наибольший=первый элемент;
номер=1;
цикл *i от 2 до n* **выполнять**
    если *arr[i]>наибольший* **тогда**
        наибольший=arr[i];
        номер=текущее значение счетчика цикла;
    конец
конец
вывод на экран;

Алгоритм 15: Наибольший и наименьший элементы и их номера

Наибольший/наименьший по условию

Анализ задачи

Поиск наибольшего элемента, как мы видели раньше, состоит из двух этапов:

1. Инициализация наибольшего элемента начальным значением.
2. Поиск наибольшего

Тут возникает проблема: какое начальное значение взять? Начальным значением для наибольшего, в этом случае, должен быть первый элемент удовлетворяющий условию.

В предыдущем примере мы брали первый элемент массива. Но первый не обязан удовлетворять условию. Хуже того в массиве может не быть нужных нам элементов вообще.

Наибольший по условию: Общий случай

Описание алгоритма

1. Найти первый элемент удовлетворяющий условию для инициализации наибольшего
2. Если элемент удовлетворяет условию и он больше наибольшего, то запоминаем этот элемент.
3. Постусловия: вывод на экран

Для поиска наименьшего все аналогично.

Наибольший по условию: Общий случай

Псевдокод

Данные: Массив arr размера *n*
Результат: Наибольший по условию
цикл *i от 1 до n* **выполнять**
    если *условие выполнилось первый раз* **тогда**
        наибольший=arr[i];
        условие уже было;
    иначе
        если *условие И элемент>наибольший* **тогда**
            наибольший=arr[i];
        конец
    конец
конец
вывод на экран;

Алгоритм 16: Наибольший по условию: Общий случай

Количество элементов равных наибольшему

Описание алгоритма

1. Предусловия: наибольший это первый, количество равно 1.
2. Если элемент больше наибольшего, то запоминаем этот элемент и сбрасываем количество на 1
3. Если элемент равен наибольшему, то увеличиваем количество на 1
4. Постусловия: вывод на экран

Количество элементов равных наибольшему

Псевдокод

Данные: Массив arr размера *n*
Результат: Количество равных наибольшему
наибольший=первый элемент;
количество=1;
цикл *i от 2 до n* **выполнять**
    если *arr[i]>наибольшего* **тогда**
        наибольший=arr[i];
        количество=1;
    иначе
        если *arr[i] = наибольший* **тогда**
            количество=количество+1;
        конец
    конец
конец
вывод на экран;

Алгоритм 17: Количество элементов равных наибольшему

Поиск двух наибольших элементов (второго максимума)

Описание алгоритма

1. Предусловия: первый и второй наибольшие это максимум и минимум из первых двух элементов массива.
2. Если элемент больше наибольшего, то второй это первый, а первый это элемент
3. Если элемент не больше наибольшего, но больше второго, то второй это элемент
4. Постусловия: вывод на экран

Поиск двух наибольших элементов

Псевдокод

Данные: Массив arr размера *n*
Результат: Два максимума
наибольший1=максимум(первый, второй);
наибольший2=минимум(первый, второй);
цикл *i от 3 до n* **выполнять**
    если *arr[i]>наибольшего1* **тогда**
        наибольший2=наибольший1;
        наибольший1=arr[i];
    иначе
        если *arr[i] > наибольший2* **тогда**
            наибольший2=arr[i];
        конец
    конец
конец
вывод на экран;

Алгоритм 18: Поиск двух наибольших элементов

Поиск элемента равного заданному

Описание алгоритма

1. Предусловия: признак того, что элемент найден
2. Если элемент равен заданному, то поиск завершен иначе перейти к следующему элементу
3. Постусловия: вывод на экран

Поиск элемента равного заданному

Псевдокод

Данные: Массив arr размера n, число
Результат: Есть или нет равный заданному
нашли=ЛОЖЬ;
счетчик=1; **до тех пор, пока** *не просмотрены все элементы и заданный не найден* **выполнять**
    если *элемент=заданному* **тогда**
        нашли=ИСТИНА;
    иначе
        перейти к следующему;
    конец
конец
вывод на экран;

Алгоритм 19: Поиск элемента равного заданному

Поиск подпоследовательности по условию

Описание алгоритма

Подпоследовательность это группа идущих подряд элементов массива. Например для [1,1,2,3,4,1,2,3] одна из подпоследовательностей будет [2,3,4,1].

1. Предусловия: инициализация максимальной и текущей длины
2. Если текущий элемент больше предыдущего, то увеличиваем текущую длину
3. Иначе сравниваем текущую длину с максимальной и сбрасываем счетчик текущей длины
4. Постусловия: вывод на экран

Поиск подпоследовательности по условию

Псевдокод

Данные: Массив arr размера n
Результат: Наибольший n
максимальная длина=0; текущая длина=1; **цикл** $i:=2$ *to* n **выполнять**
    если *условие* **тогда**
        текущая длина=текущая длина+1;
    иначе
        если *Текущая длина>максимальной* **тогда**
            максимальная длина=текущая длина;
        конец
        текущая длина=1;
    конец
    предыдущий=текущий;
конец
если *Текущая длина>максимальной* **тогда**
    максимальная длина=текущая длина;
конец
вывод на экран;

Алгоритм 20: Поиск подпоследовательности по условию

Поиск подпоследовательности по условию

Описание алгоритма

Подпоследовательность это группа идущих подряд элементов массива. Например для [1,1,2,3,4,1,2,3] одна из подпоследовательностей будет [2,3,4,1].

1. Предусловия: инициализация максимальной и текущей длины
2. Если текущий элемент больше предыдущего, то увеличиваем текущую длину
3. Иначе сравниваем текущую длину с максимальной и сбрасываем счетчик текущей длины
4. Постусловия: вывод на экран

При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях массива.

Алгоритм может быть определен в рекурсивной и нерекурсивной формах.

Бинарный поиск также называют поиском методом деления отрезка пополам или дихотомии.

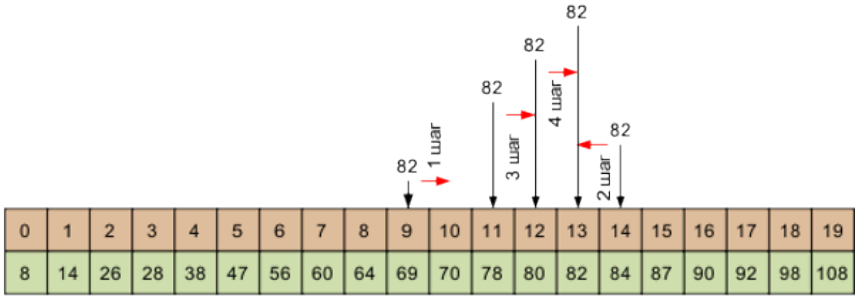
На каждом шаге осуществляется поиск середины отрезка по формуле

$$\text{mid} = (\text{left} + \text{right})/2$$

Если искомый элемент равен элементу с индексом mid, поиск завершается. В случае если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.

Бинарный поиск

Описание алгоритма



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
8	14	26	28	38	47	56	60	64	69	70	78	80	82	84	87	90	92	98	108

Сортировка

Задача сортировки является такой же базовой, как задача поиска. В практических условиях эти задачи взаимосвязаны. Решению проблем, связанных с сортировкой, посвящено множество фундаментальных научных исследований, разработано множество алгоритмов.

В общем случае сортировку следует понимать как процесс перегруппировки, заданного множества объектов в определенном порядке.

Сортировка применяется для облегчения поиска элементов в упорядоченном множестве. Задача сортировки одна из фундаментных в программировании. В большинстве случаев необходимо отсортировать массив, элементами которого являются целые числа.

Сортировка

Различают два вида сортировки: внутреннюю и внешнюю.

Под внутренней сортировкой понимают сортировку массивов, так как массив можно поместить на хранение в оперативную (внутреннюю) память.

Под внешней сортировкой понимают сортировку файлов, т.к. файлы хранящиеся во внешней памяти, не всегда могут поместиться в оперативной памяти.

Мы будем рассматривать только внутреннюю сортировку.

Сортировка

Общая задача сортировки: пусть имеется множество элементов $\mathrm{A} = \{a_1, a_2, \ldots, a_n\}$. Под сортировкой будем понимать перестановку этих элементов в множество $A' = \{a_{k1}, a_{k2}, \ldots, a_{kp}\}$, где при некоторой упорядочивающей функции $f : A \to A'$ выполняется соотношение $f(a_{ki}) \Leftarrow f(a_{kj})$, при, $j > i$.

Описание алгоритма: будем рассматривать линейный массив как вектор столбец.

11
3
5
4
1

Сначала рассмотрим элементы от первого до последнего. Начиная с последнего элемента, будем рассматривать пары соседних элементов, если «верхний» элемент больше(тяжелее) «нижнего», то обменяем их местами. Указанный процесс обмена повторим для всех множеств от i до n.

Пусть имеется множество $\mathrm{A} = \{a_1, a_2, \ldots, a_n\}$ выберем среди элементов $a_1, a_2, \ldots, a_n$ наибольший и поменяем его местами с первым. Выберем среди элементов $a_2, \ldots, a_n$ наименьший и обменяем его местами со вторым и т.д. На i -ом шаге выберем среди элементов $a_i, \ldots, a_n$ наименьший и обменяем его местами с i -ым, продолжать будем до тех пор, пока не достигнем последнего элемента.

Описание алгоритма: пусть имеется множество $\mathrm{A} = \{a_1, a_2, \ldots, a_n\}$

- Разделим множество A на две части. $A = \{a_1, a_2, \ldots, a_i\} U \{a_{i+1}, \ldots, a_n\}$, обозначим их AL и AR. Предполагается, что часть AL уже отсортирована.
- Возьмем первый элемент части AR и поместим его в часть AL так, чтобы его порядок не нарушился.
- Продолжаем, указанный процесс, до тех пор, пока не будет исчерпана часть AR .

На начальном этапе предполагается, что AL содержит только первый элемент множества A.

Для помещения элемента в отсортированную часть, используется процедура просеивания, которая состоит в следующем: пусть имеется элемент x, сравним его с последним элементом, отсортированной части.

Если порядок не нарушается, т.е. $x \geq a_i$, то x станет последним элементом множества.

Если выполняется $\mathrm{x} < a_i$, то сдвинем на позицию вправо $a_i \to a_{i+1}$ и будем сравнивать x с a_i Продолжаем процесс до тех пор, пока не найдем такой элемент, что $\mathrm{x} \geq a_i$.

Базовые алгоритмы обработки двумерных массивов

Базовые алгоритмы обработки двумерных массивов

1. Сумма строки или столбца по условию
2. Формирование одномерного массива на основе двумерного
3. Обмен местами двух строк или столбцов
4. Линейное преобразование строк
5. Максимумы по строкам
6. Транспонирование матрицы
7. Обработка областей матрицы
8. Умножение матриц

Обработка областей матрицы

Закономерности номеров элементов

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

1. Индексы элементов, лежащих на главной диагонали равны т.е. $i = j$
2. Первый индекс всех элементов выше главной диагонали меньше второго, т.е. $i < j$
3. Первый индекс всех элементов ниже главной диагонали больше второго, т.е. $i > j$

Обработка областей матрицы

Закономерности номеров элементов

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

1. Для элементов побочной диагонали сумма первого и второго индексов равна увеличенному на единицу порядку матрицы, т.е. $i + j = N+1$
2. Для элементов, находящихся выше побочной диагонали должно выполняться равенство: $i + j < N+1$
3. Для элементов, находящихся ниже побочной диагонали должно выполняться равенство: $i + j > N+1$

Методы построения алгоритмов

Методы построения алгоритмов

В настоящее время в информатике накоплен огромный арсенал приемов, методов и способов решения прикладных задач. Существует множество узкоспециальных методов, имеющих ограниченное применение, но кроме того существуют методы применимые к большому кругу задач. К таким методам относятся:

- полный перебор
- перебор с возвратом
- динамическое программирование
- жадные алгоритмы
- рекурсия

Задача о рюкзаке (также задача о ранце) — NP-полная задача комбинаторной оптимизации. Своё название получила от конечной цели: уложить как можно большее число ценных вещей в рюкзак при условии, что вместимость рюкзака ограничена. С различными вариациями задачи о рюкзаке можно столкнуться в экономике, прикладной математике, криптографии и логистике.

В общем виде задачу можно сформулировать так: из заданного множества предметов со свойствами «стоимость» и «вес» требуется отобрать подмножество с максимальной полной стоимостью, соблюдая при этом ограничение на суммарный вес.

Пусть имеется набор предметов, каждый из которых имеет два параметра — масса и ценность. Также имеется рюкзак определённой грузоподъёмности. Задача заключается в том, чтобы собрать рюкзак с максимальной ценностью предметов внутри, соблюдая при этом ограничение рюкзака на суммарную массу.

Вор пробрался в магазин, у него есть рюкзак который выдерживает 8 килограмм. В магазине, он увидел вещи. Каждый предмет имеет: название, вес и стоимость (ценность). Количество каждого предмета 1 (но может быть и неодинаковое):

Название	Вес	Стоимость
Бинокль	3	2
Ноутбук	4	3
Котелок	6	1
Золото	5	4

Вещи которые есть в магазине. Его задача оптимально уместить эти вещи в рюкзаке так что бы он унес вещей на максимальную стоимость.

Полный перебор

Полный перебор

Полный перебор (или метод «грубой силы» от англ. brute force) — метод решения задачи путем перебора всех возможных вариантов. Сложность полного перебора зависит от количества всех возможных решений задачи. Если пространство решений очень велико, то полный перебор может не дать результатов в течение нескольких лет или даже столетий.

«Метод исчерпывания» включает в себя целый класс различных методов. Обычно постановка задачи подразумевает рассмотрение конечного числа состояний данной логической системы с целью выявления истинности логического утверждения посредством независимого анализа каждого состояния. Методика доказательства утверждения состоит из двух частей:

1. Доказательство возможности исчерпания всех состояний системы. Требуется показать, что любое конкретное состояние системы (например, значение доказываемого логического выражения) соответствует хотя бы одному из рассматриваемых кандидатов в решения.
2. Проверка каждого варианта и доказательство того, что рассматриваемый вариант является или не является решением поставленной задачи.

Полный перебор

Для задачи о рюкзаке мы можем применить метод полного перебора. Сформируем все возможные наборы и подсчитаем стоимость C и вес W:

$x_1 = \{0, 0, 0, 1\}, W = 5, C = 4,$
$x_2 = \{0, 0, 1, 0\}, W = 6, C = 1,$
$x_3 = \{0, 0, 1, 1\}, W = 11, C = 5,$
$x_4 = \{0, 1, 0, 0\}, W = 4, C = 3,$
$x_5 = \{0, 1, 0, 1\}, W = 9, C = 7,$
$\dots$
$x_9 = \{1, 0, 0, 1\}, W = 8, C = 6$ (решение),
$\dots$
$x_{15} = \{1, 1, 1, 0\}, W = 13, C = 6,$
$x_{16} = \{1, 1, 1, 1\}, W = 18, C = 10$

Перебор с возвратом

Перебор с возвратом

Метод разработки алгоритма, известный как **перебор с возвратом** (программирование с отходом назад, метод ветвей и границ), можно описать как организованный исчерпывающий поиск, который часто позволяет избежать исследования всех возможностей. Этот метод особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений.

Классическим примером использования алгоритма поиска с возвратом является задача о восьми ферзях. Её формулировка такова: «Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Сперва на доску ставят одного ферзя, а потом пытаются поставить каждого следующего ферзя так, чтобы его не били уже установленные ферзи. Если на очередном шаге такую установку сделать нельзя — возвращаются на шаг назад и пытаются поставить ранее установленного ферзя на другое место.

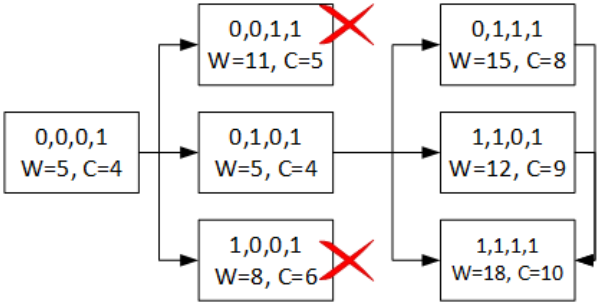
Помимо этого, метод поиска с возвратом позволяет решать множество других переборных задач. Например, с помощью него можно получить все подмножества, размещения, перестановки, сочетания данного множества M.

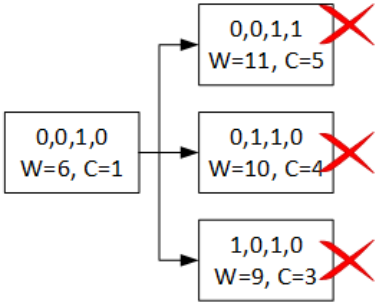
Перебор с возвратом

Для задачи о рюкзаке мы можем применить метод перебора с возвратом. Для тех наборов у которых вес уже больше заданного добавлять предметы бессмысленно:

$x_1 = \{0, 0, 0, 1\}, W = 5, C = 4,$
$x_2 = \{0, 0, 1, 0\}, W = 6, C = 1,$
$x_3 = \{0, 0, 1, 1\}, W = 11, C = 5$ (тупик),
$x_4 = \{0, 1, 0, 0\}, W = 4, C = 3,$
$x_5 = \{0, 1, 0, 1\}, W = 9, C = 7$ (тупик),
$\dots$
$x_{12} = \{1, 1, 0, 0\}, W = 7, C = 5$ (тупик),

Перебор с возвратом





Динамическое программирование

Нередко не удается разбить задачу на небольшое число подзадач, объединение решений которых позволяет получить решение исходной задачи. В таких случаях мы можем попытаться разделить задачу на столько подзадач, сколько необходимо, затем каждую подзадачу разделить на еще более мелкие подзадачи и т.д. Если бы весь алгоритм сводился именно к такой последовательности действий, мы бы получили в результате алгоритм с экспоненциальным временем выполнения. Во многих случаях удается получить лишь полиномиальное число подзадач и поэтому ту или иную подзадачу приходится решать многократно. Если бы вместо этого мы отслеживали решения каждой решенной подзадачи и просто отыскивали в случае необходимости соответствующее решение, мы бы получили алгоритм с полиномиальным временем выполнения.

С точки зрения реализации иногда бывает проще создать таблицу решений всех подзадач, которые нам когда-либо придется решать. Мы заполняем эту таблицу независимо от того, нужна ли нам на самом деле конкретная подзадача для получения общего решения. Заполнение таблицы подзадач для получения решения определенной задачи получило название динамического программирования (это название про исходит из теории управления).

Динамическое программирование

по Р. Беллману

Динамическим программированием (в наиболее общей форме) называют процесс пошагового решения задач, когда на каждом шаге выбирается одно решение из множества допустимых (на этом шаге) решений, причем такое, которое оптимизирует заданную целевую функцию или функцию критерия. В основе теории динамического программирования лежит принцип оптимальности Беллмана: «каково бы ни было начальное состояние на любом шаге и решение, выбранное на этом шаге, последующие решения должны выбираться оптимальными относительно состояния, к которому придет система в конце данного шага».

Использование этого принципа гарантирует, что решение, выбранное на любом шаге, является не локально лучшим, а лучшим с точки зрения задачи в целом.

Динамическое программирование

по Р. Беллману

Динамическое программирование решает задачу, разбивая её на подзадачи и объединяя их решения. Рекурсивные алгоритмы также делят задачу на независимые подзадачи, эти подзадачи — на более мелкие подзадачи и так далее, а затеи собирают решение основной задачи «снизу вверх». Динамическое программирование применимо тогда, когда подзадачи не являются независимыми, иными словами, когда у подзадач есть общие «подподзадачи». В этом случае при использовании рекурсии будут решаться одни и те же подзадачи несколько раз (классический пример –вычисление чисел Фибоначчи на основе рекуррентного соотношения). Формы алгоритма динамического программирования могут быть разными — общим для них является заполняемая таблица и порядок формирования ее элементов.

Алгоритм, основанный на динамическом программировании, решает каждую из подзадач только один раз и запоминает ответы в специальной таблице. Это позволяет не вычислять заново ответ к уже встречавшейся подзадаче. Типичными для динамического программирования являются задачи оптимизации.

У подобных задач может быть много возможных решений а, их «качество» определяется значением какого-то параметра, и требуется выбрать оптимальное решение, при котором значение параметра будет минимальным или максимальным (в зависимости от постановки задачи). В общем случае, оптимум может достигаться для нескольких разных решений.

Можно предложить следующую систему шагов для построения алгоритма, основанного на динамическом программировании:

1. описать строение оптимальных решений,
2. выписать рекуррентное соотношение, связывающее оптимальные значения параметра для подзадач.
3. двигаясь снизу вверх, вычислить оптимальное значение параметра для подзадач,
4. пользуясь полученной информацией, построить оптимальное решение.

В верхней строке указаны варианты веса. От 0 до 8.
В нижнем правом углу указана оптимальная стоимость.
$K[i][w] = \max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])$

вес-предмет	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2
2	0	0	2	3	3	3	5	5
3	0	0	2	3	3	3	5	5
4	0	0	2	3	4	4	5	6

Расшифровка: Чтобы заполнить вес w=4, предмет i=2, мы ищем максимум
$K[2][4] = \max(val[1] + K[1][4 - 3], K[1][4])$
$K[2][4] = \max(3 + 0, 2) = 3$

Жадные алгоритмы

Жадные алгоритмы

Рассмотрим небольшую задачу. Допустим, что у нас есть монеты достоинством 25, 10, 5 копеек и 1 копейка и нужно вернуть сдачу 63 копейки. Почти не раздумывая, мы преобразуем эту величину в две монеты по 25 копеек, одну монету в 10 копеек и три монеты по одной копейке.

Нам не только удалось быстро определить перечень монет нужного достоинства, но и, по сути, мы составили самый короткий список монет требуемого достоинства. Данный алгоритм заключался в выборе монеты самого большого достоинства (25 копеек), но не больше 63 копеек, добавлению ее в список сдачи и вычитанию ее стоимости из 63 (получается 38 копеек). Затем снова выбираем монету самого большого достоинства, но не больше остатка (38 копеек): этой монетой опять оказывается монета в 25 копеек. Эту монету мы опять добавляем в список сдачи, вычитаем ее стоимость из остатка и т.д.

Жадные алгоритмы

Этот метод внесения изменений называется **«жадным»** алгоритмом. На каждой отдельной стадии «жадный» алгоритм выбирает тот вариант, который является локально оптимальным в том или ином смысле. Обратите внимание, что алгоритм для определения сдачи обеспечивает в целом оптимальное решение лишь вследствие особых свойств монет. Следует подчеркнуть, что не каждый «жадный» алгоритм позволяет получить оптимальный результат в целом. Как нередко бывает «жадная стратегия» подчас обеспечивает лишь сиюминутную выгоду, в то время как в целом результат может оказаться неблагоприятным.

Как узнать, даст ли жадный алгоритм оптимум применительно к данной задаче? Общих рецептов тут нет, но существуют две особенности, характерные для задач, решаемых жадными алгоритмами. Это принцип жадного выбора и свойство оптимальности для подзадач.

Говорят, что к оптимизационной задаче применим принцип жадного выбора, если последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение.

Жадные алгоритмы

Непрерывная задача о рюкзаке отличается от дискретной тем, что вор может дробить краденые товары на части и укладывать в рюкзак эти части, а не обязательно вещи целиком (если в дискретной задаче вор имеет дело с золотыми слитками, то в непрерывной — с золотым песком).

Решение непрерывной задачи о рюкзаке с помощью жадного алгоритма выглядит так. Вычислим цены (в расчёте на килограмм) всех товаров (цена товара номер i равна v_i/w_i). Сначала вор берёт по максимуму самого дорогого товара: если весь этот товар кончился, а рюкзак не заполнен, вор берёт следующий по цене товар, затем следующий, и так далее, пока не наберёт вес W.

Жадный алгоритм даёт оптимум в непрерывной задаче о рюкзаке и не даёт в дискретной.

Жадные алгоритмы

Основной областью применения «жадных» алгоритмов являются задачи поиска и оптимизации.

Классическим является алгоритм Дейкстры поиска кратчайшего пути между двумя вершинами неориентированного графа с положительными весами.

В отдельных случаях «жадная» стратегия используется для получения «хорошего» начального приближения к оптимальному решению.

Понятие рекурсии

Понятие рекурсии

Алгоритм принято называть рекурсивным, если в его определении содержится прямой или косвенный вызов этого же алгоритма.

Вычисления, проводимые с помощью рекурсивных алгоритмов (процедур, функций) называют рекурсивными. Различают два базовых вида рекурсии: **прямая** и **косвенная**.

Понятие рекурсии

Под **прямой** рекурсией принято понимать непосредственный вызов алгоритма (функции, процедуры) F из текста самого алгоритма F.

При **косвенной** рекурсии мы имеем циклическую последовательность вызовов нескольких алгоритмов $F_1, F_2, \ldots, F_k$ (функций, процедур) друг друга: F_1 вызывает F_2, F_2 вызывает F_3, ..., F_k вызывает F_1 ($k > 1$).

Рекурсивная триада

А.Р Есаяном предложена схема решения задач с помощью рекурсии, основным компонентом которой является рекурсивная триада, состоящая из трех базовых этапов:

1. параметризация,
2. выделение базы (или выделение начальной базы и правил её изменения),
3. декомпозиция.

Параметризация

Параметризация задачи заключается в выявлении совокупности исходных величин, определяющих постановку и решение задачи.

Значения этих параметров или некоторых из них влияют на трудоемкость решения задачи.

Иногда бывает полезно ввести в рассмотрение дополнительные параметры, напрямую с постановкой задачи не связанные, но помогающие организовать рекурсию.

Выделение базы

Выделение базы — поиск одной или нескольких подзадач, которые бывают решены непосредственно без рекурсивного вызова. В случае если база будет меняться в процессе вычислений, то должен быть указан алгоритм её изменения.

Как правило, подобная динамическая база расширяется за счет получения решений промежуточных задач и облегчает выполнение процесса отложенных вычислений. Возможно и сужение рекурсивной базы.

Декомпозиция

Декомпозиция общего случая — это процесс последовательного разложения задачи на серию подзадач двух типов: тех, которые мы решать умеем и тех, которые в чем-то аналогичны исходной задаче.

В последнем случае каждая из полученных подзадач должна быть упрощенным вариантом предыдущей задачи. При этом декомпозицию следует осуществлять так, чтобы можно было доказать, что при любом допустимом наборе значений параметров рано или поздно она приведет нас к одному из выделенных тривиальных случаев, то есть к базе.

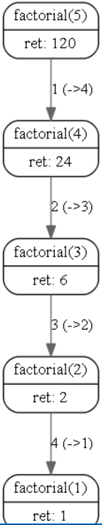
Пример: Факториал

1. Параметризация: n — число, факториал которого ищем
2. Выделение базы: $1! = 1$, $0! = 1$.
3. Декомпозиция: $n! = n \cdot (n-1)!$

Пример: Факториал

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

n=int(input("n= "))
print(factorial(n))
```



Запись 3 (->2) означает, что номер рекурсивного вызова 3, номер

Пример: Числа Фибоначчи

Числа Фибоначчи — элементы числовой последовательности

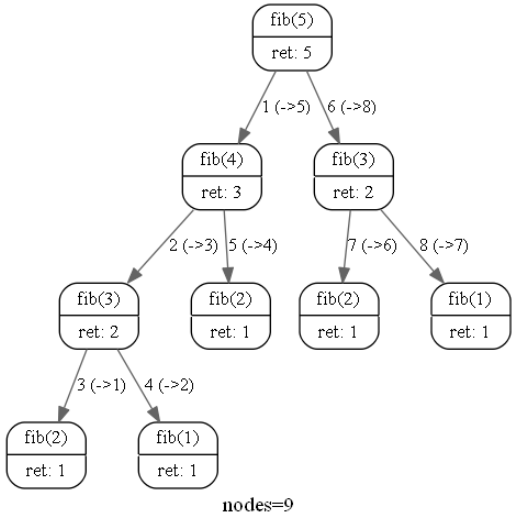
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,...
(последовательность A000045 в OEIS)

в которой первые два числа равны либо 1 и 1, либо 0 и 1, а каждое последующее число равно сумме двух предыдущих чисел. Названы в честь средневекового математика Леонардо Пизанского (известного как Фибоначчи).

1. Параметризация: n — номер числа, Фибоначчи
2. Выделение базы: $f_1 = 1$, $f_2 = 1$.
3. Декомпозиция: $f_n = f_{n-1} + f_{n-2}$

Пример: Числа Фибоначчи

```
1  def fib(n):
2      if n in {0, 1}:  # base
3          return n
4      return fib(n - 1) + fib(n - 2)  # decompose
5
6  [print(fib(n)) for n in range(15)]
```



Сопоставление с образцом

Очень часто при обработке текстовых данных возникает задача о сопоставлении строки с образцом. Самый простой пример — проверка корректности вводимых пользователем данных. Допустим, у нас имеются следующие требования к паролю, используемому в системе:

- пароль должен иметь длину не меньше 6 символов, но не больше 15;
- пароль может содержать как строчные, так и прописные буквы, а также цифры;
- пароль должен начинаться с заглавной буквы, а заканчиваться строчной буквой.

В принципе, можно написать функцию, которая будет проверять, соответствует ли строка предъявляемым требованиям:

```
1  def check_password(s):
2      if not 6 <= len(s) <= 15:
3          return False
4      if not 'A' <= s[0] <= 'Z':
5          return False
6      if not 'a' <= s[-1] <= 'z':
7          return False
8      for c in s [1:-1]:
9          if not ('a' <= c <= 'z' or 'A'
       <= c <= 'Z' or '0' <= c <= '9'):
10             return False
11     return True
```

Сопоставление собразцом

Очень часто при обработке текстовых данных возникает задача о сопоставлении строки с образцом. Самый простой пример — проверка корректности вводимых пользователем данных. Допустим, у нас имеются следующие требования к паролю, используемому в системе:

- пароль должен иметь длину не меньше 6 символов, но не больше 15;
- пароль может содержать как строчные, так и прописные буквы, а также цифры;
- пароль должен начинаться с заглавной буквы, а заканчиваться строчной буквой.

А можно пойти другим путём и воспользоваться специально предназначенным для этого инструментом — регулярными выражениями:

```
1 import re
2
3 def check_password_with_regexp(s):
4     return re.match(r'^[A-Z]{1}[a-zA-Z0-9]{4,13}[a-z]{1}$', s)
```

Сопоставление собразцом

Как видно из примера, вид функции $\_\text{password}$ сильно изменился.

Мы не будем на данном этапе пытаться разобраться в том, как именно работает новая версия функции, а лишь отметим одно важное отличие: первая версия функции **выполняла ряд проверок**, пытаясь найти символ, **не соответствующий** предъявляемым требованиям, вторая же версия функции вместо этого выясняет, **соответствует** ли переданная строка заранее заданному **образцу**.

Именно в этом и заключается основное предназначение регулярных выражений — они позволяют описать, как **должна** выглядеть строка, если она удовлетворяет определённым критериям.

Где используются регулярные выражения?

Регулярные выражения получили весьма широкое распространение благодаря тому, что являются достаточно мощным инструментом работы с текстовыми данными.

Так, их поддержка есть в большинстве языков программирования (Python, Ruby, Perl, Java и т.д.), они используются во многих утилитах командой строки UNIX-подобных операционных систем (grep, sed, awk и т.д.).

Помимо этого поддержку регулярных выражений можно встретить во многих текстовых редакторах (Vim, Emacs, Sublime Text, Notepad++ и т.д.), поскольку этот механизм позволяет достаточно просто выполнять замены символов в тексте.

Где используются регулярные выражения?

Модуль re (**R**egular **E**xpressions) стандартной библиотеки языка Python предоставляет набор функций и классов для работы с регулярными выражениями. Вот список наиболее часто используемых из них:

Функция	Описание
re.match	Ищет соответствие заданному шаблону в начале строки
re.search	Ищет соответствие заданному шаблону в произвольном месте строки и возвращает первое найденное совпадение
re.findall	Находит и возвращает список всех непересекающихся подстрок исходной строки, соответствующих шаблону
re.split	Разбивает строку на набор подстрок с использованием шаблона для поиска разделителей
re.sub	Заменяет подстроку, соответствующую шаблону, указанным значением
re.compile	«Компилирует» регулярное выражение для дальнейшего использования

Где используются регулярные выражения?

Теперь рассмотрим несколько примеров работы с регулярными выражениями. Пока что мы не знаем синтаксиса, используемого для описания регулярных выражений, поэтому рассмотрим самый простой случай: когда в качестве регулярного выражения выступает непосредственно искомая строка.

```
1  import re
2  s = 'с самого начала у меня была какаято- тактика, и я её придерживался'
3  re.match('с самого', s)
4  <_sre.SRE_Match object; span=(0, 8), match='с самого'>
5
6  re.match('была', s)
7  re.search('была', s)
8  <_sre.SRE_Match object; span=(23, 27), match='была'>
9
10 re.findall('и', s)
11 ['и', 'и', 'и', 'и']
12
13 re.split(' ', s)
```

Где используются регулярные выражения?

Как видно из примера, функции re.match и re.search возвращают в случает совпадения экземпляр класса SRE_Match или None, если совпадение не найдено, в то время как функции re.findall, re.split возвращают список, состоящий из строк, функция re.sub возвращает строку, получающуюся после выполнения замены.

Специальные символы

В предыдущих примерах мы использовали искомую строку в качестве регулярного выражения. Теперь настало время познакомиться более подробно с синтаксисом описания регулярных выражений.

Помимо непосредственно искомых символов регулярное выражение может содержать специальные символы, которые позволяют задать **шаблон**. Вот краткий список основных из них:

Специальные символы	Описание
.	Любой символ, кроме символа новой строки
^	Начало строки
$	Конец строки
?	0 или 1 соответствие шаблона слева
*	0 или больше соответствий шаблона слева
+	1 или более соответствий шаблона слева
{m}	Ровно m соответствий шаблона слева

Примеры использования:

```
>>> import re
>>> re.match('.', 'a')
<_sre.SRE_Match object; span=(0, 1), match='a'>
>>> re.match('.', 'b')
<_sre.SRE_Match object; span=(0, 1), match='b'>
>>> re.match('.', '.')
<_sre.SRE_Match object; span=(0, 1), match='.'>
>>> re.search('a', 'bab')
<_sre.SRE_Match object; span=(1, 2), match='a'>
>>> re.search('^a', 'bab')
>>> re.search('^a$', 'bab')
>>> re.search('a$', 'bab')
>>> re.search('^.a.$', 'bab')
<_sre.SRE_Match object; span=(0, 3), match='bab'>
>>> re.search('a?', 'bbb')
<_sre.SRE_Match object; span=(0, 0), match=''>
>>> re.search('a?', 'bab')
```

Примеры использования:

```
>>> re.search('a*', 'baaab')
<_sre.SRE_Match object; span=(0, 0), match=''>
>>> re.search('ba*b', 'baaab')
<_sre.SRE_Match object; span=(0, 5), match='baaab'>
>>> re.search('ba+b', 'baaab')
<_sre.SRE_Match object; span=(0, 5), match='baaab'>
>>> re.search('ba?b', 'baaab')
>>> re.search('a{2}', 'baaab')
<_sre.SRE_Match object; span=(1, 3), match='aa'>
>>> re.search('a{3}', 'baaab')
<_sre.SRE_Match object; span=(1, 4), match='aaa'>
>>> re.search('a{4}', 'baaab')
>>> re.search('a{1}', 'baaab')
<_sre.SRE_Match object; span=(1, 2), match='a'>
>>> re.search('a{1,2}', 'baaab')
<_sre.SRE_Match object; span=(1, 3), match='aa'>
```

Примеры использования:

```
>>> re.search(r'\*', r'*')
<_sre.SRE_Match object; span=(0, 1), match='*'>
>>> re.search(r'[abc]', r'0123ccaabb275')
<_sre.SRE_Match object; span=(4, 5), match='c'>
>>> re.search(r'[abc]?', r'0123ccaabb275')
<_sre.SRE_Match object; span=(0, 0), match=''>
>>> re.search(r'[abc]+', r'0123ccaabb275')
<_sre.SRE_Match object; span=(4, 10), match='ccaabb'>
>>> re.search(r'[0-9]{4}[abc]+[0-9]{3}', r'0123ccaabb275')
<_sre.SRE_Match object; span=(0, 13), match='0123ccaabb275'>
>>> re.search(r'[0-9]{4}[^0-9]+[0-9]{3}', r'0123ccaabb275')
<_sre.SRE_Match object; span=(0, 13), match='0123ccaabb275'>
>>> re.search(r'a|b', r'ccadd')
<_sre.SRE_Match object; span=(2, 3), match='a'>
>>> re.search(r'a|b', r'ccbdd')
<_sre.SRE_Match object; span=(2, 3), match='b'>
```

Больше специальных символов

Теперь перейдём к рассмотрению более сложных специальных символов, поддержка которых присутствует в библиотеке re. Эти символы нужны как для написания сложных, так и для сокращения длинных регулярных выражений. Неполный список специальных символов приведён в таблице ниже:

Специальный символ	Описание
\A	Начало строки; эквивалент ^
\b	Начало слова
\B	**Не** начало слова
\d	Цифра; расширенный вариант [0-9]
\D	**Не** цифра; «отрицание» \d
\s	Пробельный символ; расширенный вариант [\t\n\r\f\v]
\S	**Не** пробельный символ; «отрицание» \s
\w	«Буква» в слове расширенный вариант [a-zA-Z0-9_]
\W	**Не** «буква»; «отрицание» \w
\Z	Конец строки; эквивалент $

Примеры использования:

```
1
2  >>> import re
3  >>> re.match(r'\Aab', 'abcd')
4  <_sre.SRE_Match object; span=(0, 2), match='ab'>
5  >>> re.match(r'\Aab', 'dabcd')
6  >>> re.search(r'\bbbb', 'abbba bbb ccc')
7  <_sre.SRE_Match object; span=(6, 9), match='bbb'>
8  >>> re.search(r'\Bbbb', 'abbba bbb ccc')
9  <_sre.SRE_Match object; span=(1, 4), match='bbb'>
10 >>> re.search(r'\d+', 'ab123cd')
11 <_sre.SRE_Match object; span=(2, 5), match='123'>
12 >>> re.search(r'\D+', 'ab123cd')
13 <_sre.SRE_Match object; span=(0, 2), match='ab'>
14 >>> re.sub(r'\s+', '_', 'aa bb cc dd')
15 'aa_bb_cc_dd'
```

Примеры использования:

```
1
2  >>> re.findall(r'\S+', 'aa bb cc dd')
3  ['aa', 'bb', 'cc', 'dd']
4  >>> re.search(r'\w+', 'ab123cd')
5  <_sre.SRE_Match object; span=(0, 7), match='ab123cd'>
6  >>> re.search(r'\W+', 'ab123cd')
7  >>> re.search(r'\w+', 'ab123cd  aaa')
8  <_sre.SRE_Match object; span=(0, 7), match='ab123cd'>
9  >>> re.search(r'\W+', 'ab123cd  aaa')
10 <_sre.SRE_Match object; span=(7, 9), match='  '>
11 >>> re.search(r'aa\Z', 'bbaa')
12 <_sre.SRE_Match object; span=(2, 4), match='aa'>
13 >>> re.search(r'aa\Z', 'bbaab')
```

Группы

Зачастую бывает необходимо, чтобы специальный символ (+, * и т.д.) применялся не к одному символу слева, а определённой **группе** символов. Для этого нужно заключить интересующую часть шаблона в круглые скобки ‘(. . .)‘:

```
>>> import re
>>> re.search('(ab){3}', 'ab ab ababab ab ab')
<_sre.SRE_Match object; span=(6, 12), match='ababab'>
>>> re.search('(ab){3}', 'ab ab abab ab ab') is None
True
```

В некоторых случаях при обработке строковых данных при помощи регулярных выражений возникает необходимость выделить определённую часть подстроки, соответствующей шаблону. Например, рассмотрим следующую задачу: извлечь из строки ’какой-то текст, текст жирным шрифтом, и снова какой-то текст’ подстроку, заключённую внутри *тега* ... Для этого также удобно использовать группы:

```
>>> import re
```

Группы

Обратите внимание, что всему шаблону соответствует подстрока текст жирным шрифтом', в то время как в группу попадает только нужная нам строка 'текст жирным шрифтом'.

Группы нумеруются в том порядке, в котором они перечислены в шаблоне:

```
>>> import re
>>> re.match('(a|b)(c|d)', 'ac').groups()
('a', 'c')
>>> re.match('(a|b)(c|d)', 'ac').group(1)
'a'
>>> re.match('(a|b)(c|d)', 'ac').group(2)
'c'
>>> re.match('(a|b)(c|d)', 'bd').groups()
('b', 'd')
>>> re.match('(a|b)(c|d)', 'bd').group(1)
'b'
```

Группы

Группы можно использовать внутри выражения, например, для поиска повторяющихся букв:

```
>>> import re
>>> re.search(r'(a|b|c)\1', 'aabc')
<_sre.SRE_Match object; span=(0, 2), match='aa'>
>>> re.search(r'(a|b|c)\1', 'abbc')
<_sre.SRE_Match object; span=(1, 3), match='bb'>
>>> re.search(r'(a|b|c)\1', 'abcc')
<_sre.SRE_Match object; span=(2, 4), match='cc'>
```

Группы можно использовать внутри строки замены в функции re.sub:

```
>>> import re
>>> re.sub('(a|b|c)', r'*\1*', 'a')
'*a*'
>>> re.sub('(a|b|c)', r'*\1*', 'abc')
'*a**b**c*'
```

Группы

Если групп в выражении достаточно много, их можно именовать при помощи конструкции (?P<имя>), а затем обращаться к ним при помощи (?P=имя) внутри выражения, при помощи g<имя> внутри строки замены или непосредственно по имени при работе с объектом SRE_match:

```
>>> import re
>>> re.match('(?P<group1>a|b)(?P<group2>c|d)', 'ac').groups()
('a', 'c')
>>> re.match('(?P<group1>a|b)(?P<group2>c|d)', 'ac').group('group1')
'a'
>>> re.match('(?P<group1>a|b)(?P<group2>c|d)', 'ac').group('group2')
'c'
>>> re.match('(?P<group1>a|b)(?P<group2>c|d)', 'ac').group(1)
'a'
>>> re.match('(?P<group1>a|b)(?P<group2>c|d)', 'ac').group(2)
'c'
>>> re.sub('(?P<group1>a|b)(?P<group2>c|d)', r'!\1!*\g<group2>*', 'xxacxx')
```

Выводы

Регулярные выражения, как и любой другой инструмент, стоит использовать там, где они действительно уместны. Во многих случаях использование альтернативного подхода может существенно упростить процесс решения задачи, а также сделать текст программы более понятным. В качестве примера плохого регулярного выражения можно привести вот этот модуль для языка Perl, который проверяет корректность введённого адреса электронной почты. Мало того, что это выражение огромно, так ещё и потребуется не один час, чтоб разобраться, как именно оно устроено.

В остальных случаях, особенно когда регулярное выражение получается понятным и лаконичным, его использование оказывается более предпочтительным. Однако не стоит забывать о том, что зачастую на написание корректно работающего регулярного выражения может потребоваться больше времени, чем на решение задачи альтернативным способом.

Хороший сервис проверки регулярных выражений и генерации кода

Выводы

1. Нужна практика программирования (решение большого количества однотипных задач)
2. Нужно знать и уметь применять стандартные алгоритмы для решения задач
3. Если вы можете описать решение задачи в терминах действий, то вы задачу успешно решите на любом языке программирования