Python-5
Коллекции Python. Сортировки.

Что такое коллекция в python?

Коллекция в Python - это контейнер для хранения и работы с группой элементов одного или разных типов данных, таких как числа, строки, списки, словари и т.д. Коллекции можно изменять, добавлять, удалять элементы, сортировать и выполнять на них различные операции.

Примеры коллекций: списки, словари…

Списки.

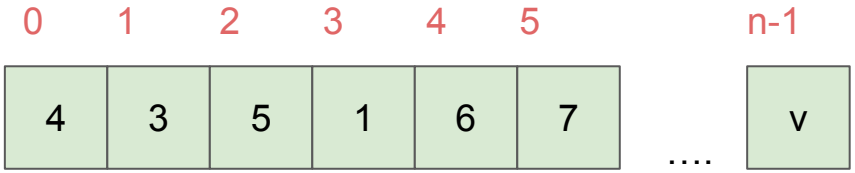
Список (List) - это упорядоченная последовательность элементов, которые могут быть изменены.

Допустим, есть коробка, и внутри этой коробки есть много маленьких ячеек, и каждая ячейка может содержать разные вещи. Эти ячейки упорядочены, что означает, что они находятся друг за другом, и у каждой есть свой номер (индекс).

YANDEX DISK (ПРИМЕРЫ): *https://disk.yandex.ru/d/lJ6oci35h-uSvQ*

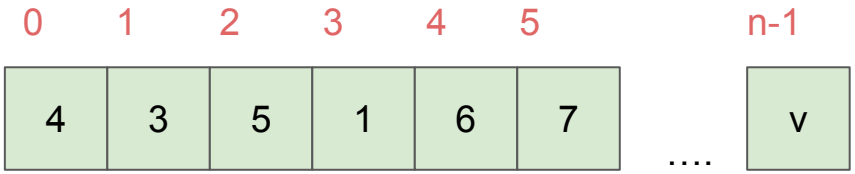
YANDEX DISK (ЗАДАЧИ): *https://disk.yandex.ru/d/GTM0aWv8m3ccoQ*

Списки. Пример.



4,3,5,1,6,7 … n - элементы массива (списка) значения какие-то
то что сверху 0,1,2.. - это индексы элементов списка

Списки. Пример изменение данных по индексу.



Например я захотел заменить 1 элемент (то есть число 3) на число -3.

list[index] = новое значение

PY: list[1] = -3


Основные методы для работы со списками!
list.append(x)
Добавляет элемент в конец списка.
list.pop([i]) Удаляет элемент на заданной позиции в списке и возвращает его.
list.clear() Удаляет все элементы из списка.
list.count(x)
Возвращает количество раз, которое x встречается в списке.
list.sort(key=None, reverse=False)
Сортирует элементы списка на месте.
list.index(x[, start[, end]]) Возвращает индекс первого элемента со значением x.

Пример работы со списками. (заполнение списка)

```
# Создаем пустой список
numbers = []

# Просим пользователя ввести три числа
for i in range(3):
    num = int(input("Введите число: "))
    # Добавляем число в список
    numbers.append(num)

# Выводим список на экран
print("Список чисел:", numbers)
```

Что такое оператор del??

Оператор del в Python *удаляет объекты (например, переменные или элементы списка) из памяти компьютера, тем самым освобождая память для других операций.*

Пример работы со списками. (удалить i-й элемент из списка)

```
# Создаем список
my_list = [1, 2, 3, 4, 5]

# Удаляем элемент с индексом 2
del my_list[2]

# Печатаем список после удаления
print(my_list)

# Результат: [1, 2, 4, 5]
```

Понятие среза в Python. Что это??

Срез (slice) в Python — *это операция извлечения подстроки из последовательности (например, строки, списка, кортежа), указывая начальный и конечный индексы.*

Синтаксис среза. sequence[start:stop:step]

где sequence — последовательность, к которой применяется операция среза,
start — индекс элемента, с которого начинается срез,
stop — индекс элемента, до которого идет срез (*не включая элемент с этим индексом*),
step (*необязательный*) — шаг, с которым нужно пройти по последовательности, чтобы выбрать элементы для среза.

Простой пример среза.

*syntax: sequence[start:stop:step]

```
s = "Hello, world!"
print(s[1:5]) # "ello"
print(s[:5]) # "Hello,"
print(s[7:]) # "world!"
print(s[::2]) # "Hlo ol!"
```

Кортежи.

Кортеж (Tuple) *- это неизменяемая упорядоченная последовательность элементов.*

Методы Кортежей.

Кортежи **не имеют методов** для изменения содержимого из-за их неизменяемости, но вы можете использовать общие функции, такие как **len**(tuple) для получения длины кортежа и tuple.**count**(x) для подсчета количества раз, которое x встречается в кортеже.

Пример работы с кортежем. (заполнение кортежа)

```
# Создаем кортеж из двух элементов
my_tuple = ('apple', 'banana')

# Выводим кортеж на экран
print(my_tuple)

# Добавляем новый элемент в кортеж
my_tuple = my_tuple + ('orange',)

# Выводим кортеж на экран с новым элементом
print(my_tuple)
```

Пример работы с кортежем. (создание кортежа из списка + объединение)

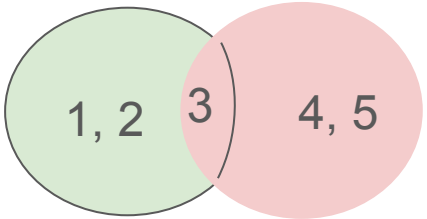
```
# Создание кортежа
my_tuple = (1, 2, 3, 4, 5)

# Создание кортежа из списка
my_list = [6, 7, 8, 9, 10]
new_tuple = tuple(my_list)

# Объединение двух кортежей
merged_tuple = my_tuple + new_tuple
```

Пример работы с кортежем. (дополнительно)

```
# Проверка наличия элемента в кортеже
if 5 in merged_tuple:
    print("5 is present")

# Нахождение индекса элемента в кортеже
print(merged_tuple.index(9))

# Подсчет количества вхождений элемента в кортеже
print(merged_tuple.count(5))

# Разбиение кортежа на две части
a, b = merged_tuple[:5], merged_tuple[5:]

# Удаление кортежа
del merged_tuple
```

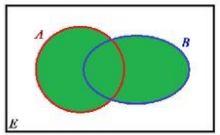
Множества.

Множество (Set) *- это неупорядоченная коллекция уникальных объектов.*



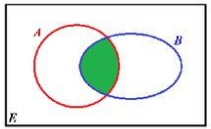
Операции над множествами

- Объединение



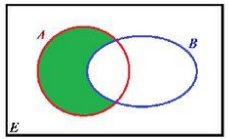
$$x \in A \cup B \Leftrightarrow (x \in A) \vee (x \in B)$$

- Пересечение



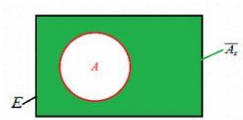
$$x \in A \cap B \Leftrightarrow (x \in A) \wedge (x \in B)$$

- Разность



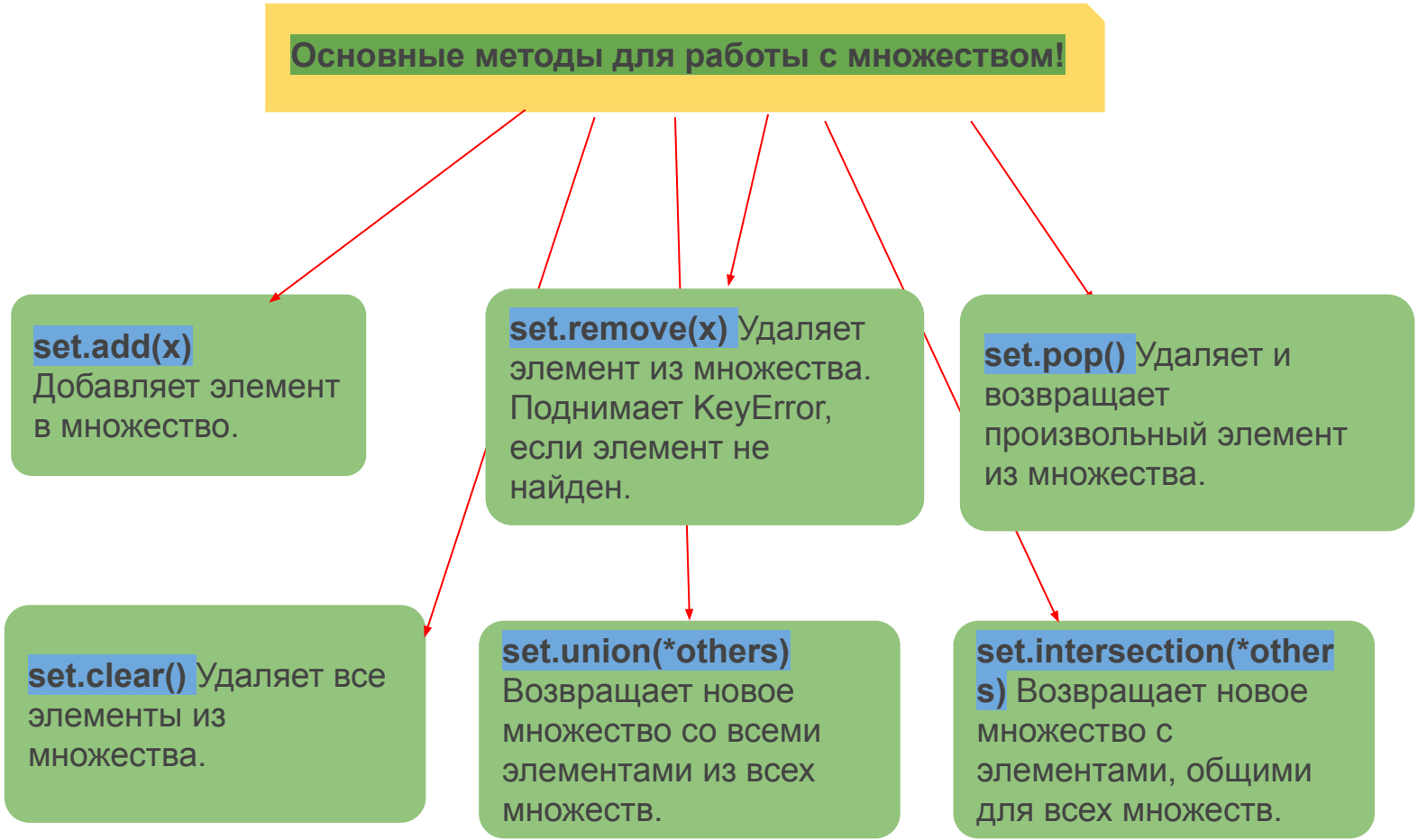
$$x \in A \setminus B \Leftrightarrow (x \in A) \wedge (x \notin B)$$

- Дополнение



$$x \in \overline{A}_E \Leftrightarrow x \notin A$$

Методы Множества.


Основные методы для работы с множеством!
set.add(x)
Добавляет элемент в множество.
set.remove(x) Удаляет элемент из множества. Поднимает KeyError, если элемент не найден.
set.pop() Удаляет и возвращает произвольный элемент из множества.
set.clear() Удаляет все элементы из множества.
set.union(*others)
Возвращает новое множество со всеми элементами из всех множеств.
set.intersection(*others) Возвращает новое множество с элементами, общими для всех множеств.

Пример работы с множеством.

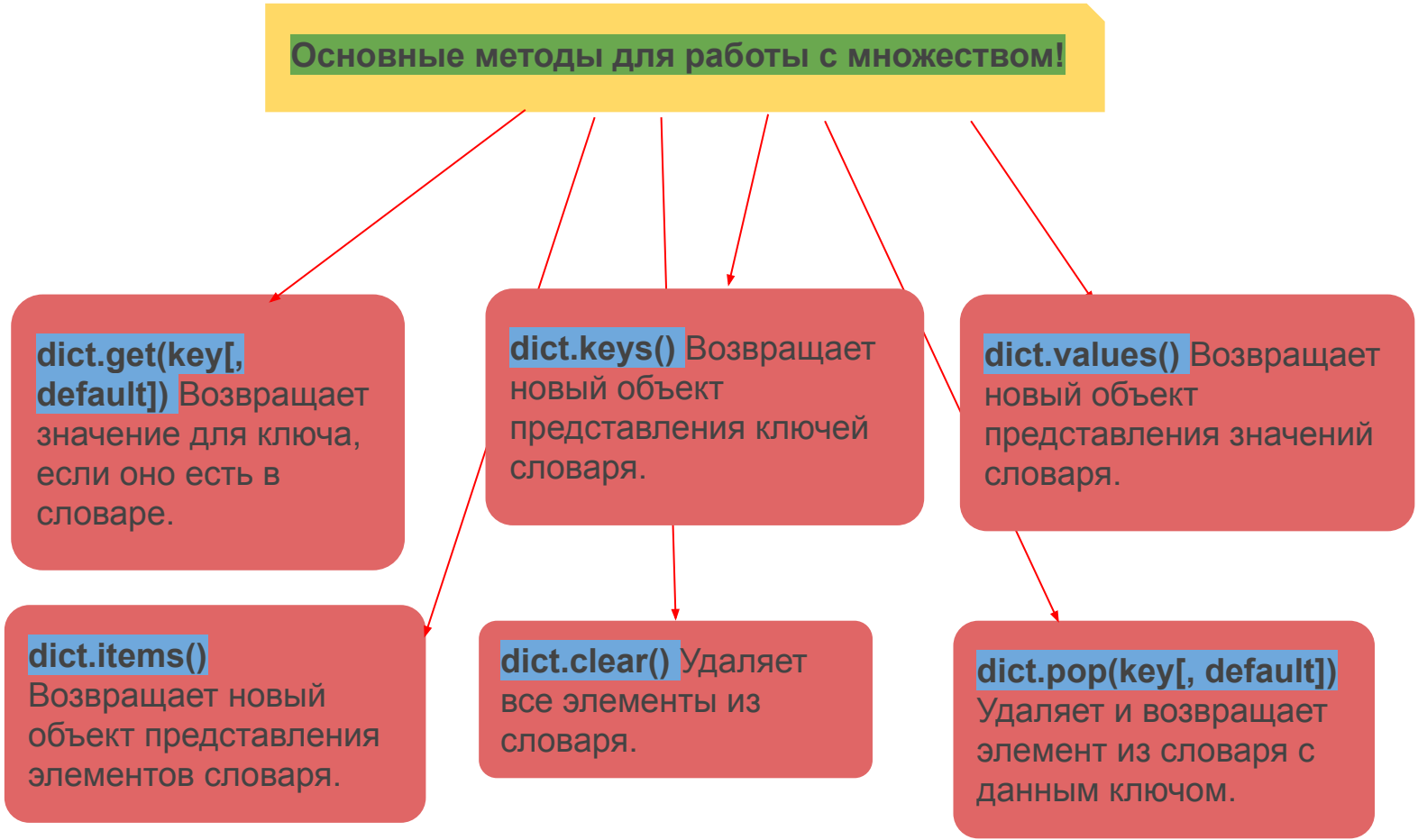
```python
# Создание множества
my_set = {1, 2, 3, 4, 5}

# Добавление элемента
my_set.add(6)

# Удаление элемента
my_set.remove(3)
```

```python
# Проверка наличия элемента в множестве
if 2 in my_set:
    print("2 is in the set")

# Пересечение двух множеств
other_set = {2, 3, 4, 7}
intersect_set = my_set.intersection(other_set)
print(intersect_set)

# Объединение двух множеств
union_set = my_set.union(other_set)
print(union_set)

# Разность двух множеств
diff_set = my_set.difference(other_set)
print(diff_set)
```

Словари.

Словарь (Dictionary) *- это неупорядоченная коллекция пар ключ-значение, где каждый элемент имеет уникальный ключ.*

Методы Словаря.


Основные методы для работы с множеством!
dict.get(key[, default]) Возвращает значение для ключа, если оно есть в словаре.
dict.keys() Возвращает новый объект представления ключей словаря.
dict.values() Возвращает новый объект представления значений словаря.
dict.items() Возвращает новый объект представления элементов словаря.
dict.clear() Удаляет все элементы из словаря.
dict.pop(key[, default]) Удаляет и возвращает элемент из словаря с данным ключом.

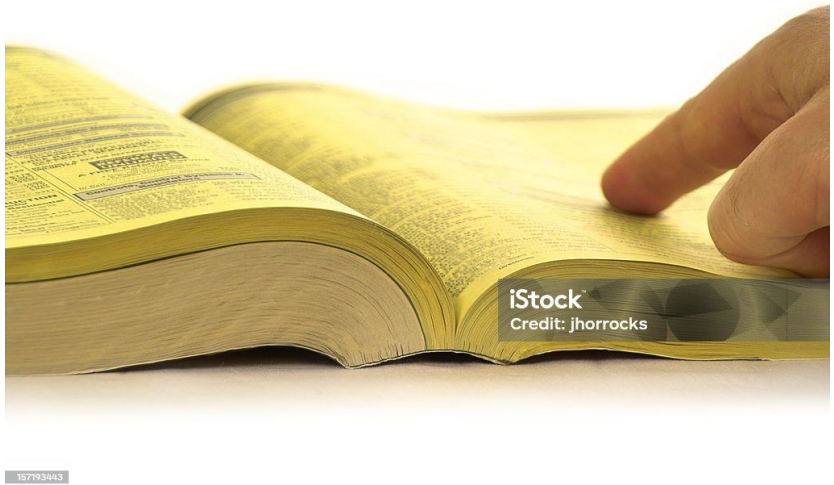
Пример работы с словарем.

```python
# Создание словаря
my_dict = {'apple': 2, 'banana': 4, 'orange': 1}

# Добавление элемента в словарь
my_dict['kiwi'] = 3

# Обращение к элементу словаря
print(my_dict['banana']) # выводит 4

# Перебор всех элементов словаря
for fruit, quantity in my_dict.items():
    print(fruit, quantity)

# Удаление элемента из словаря
del my_dict['orange']

# Проверка наличия ключа в словаре
if 'apple' in my_dict:
    print('Yes')
else:
    print('No')
```

```python
# Получение всех ключей в словаре
print(my_dict.keys())

# Получение всех значений в словаре
print(my_dict.values())

# Получение количества элементов в словаре
print(len(my_dict))
```

Поиск данных. (линейный)

Линейный поиск - это простой алгоритм поиска, при котором каждый элемент последовательности проверяется поочередно до тех пор, пока не будет найден искомый элемент или весь список не будет просмотрен. Это подходит для неотсортированных данных.



Поиск данных. (линейный). Python код.

```
lst = [2, 3, 4]
target = 3
for item in lst:
    if item == target:
        print("найден")
        break
```

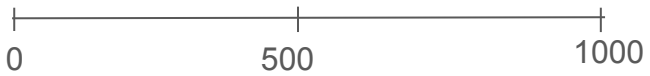
Поиск данных. (бинарный)

Бинарный поиск предназначен для поиска элемента в упорядоченном массиве. Алгоритм сравнивает искомое значение с элементом в середине массива. Если оно равно, поиск завершается. Если искомое значение меньше, поиск продолжается в левой половине массива, иначе - в правой. Процесс повторяется до тех пор, пока элемент не будет найден или область поиска не сократится до нуля. Бинарный поиск имеет временную сложность O(logn), что делает его эффективным для больших упорядоченных данных.

1000 -> страниц (450i)

1000/2 = 0..500

500/2 = 250..500



375..500

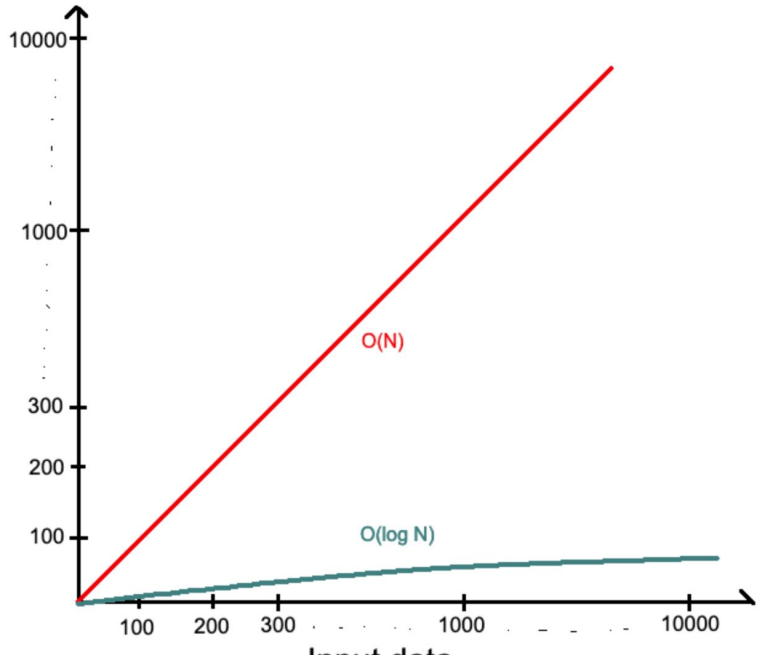
…

Поиск данных. (бинарный). Python код.

```
lst = [2, 3, 4]
target = 3
low, high = 0, len(lst) - 1
while low <= high:
    mid = (low + high) // 2
    if lst[mid] == target:
        print(mid) # 1
        break
    elif lst[mid] < target:
        low = mid + 1
    else:
        high = mid - 1
```

сравниваем значение среднего элемента и уменьшаем область поиска вдвое на каждом шаге.

Поиск данных. График.



Сортировки Данных

Алгоритмы сортировки - это специальные методы или шаги, которые позволяют упорядочить элементы в наборе данных, например, в списке или словаре, в определенном порядке (например, по возрастанию или убыванию).

Есть несколько алгоритмов сортировок например:

- **сортировка пузырьком:** этот алгоритм сравнивает пары соседних элементов и меняет их местами, если они находятся в неправильном порядке. Этот процесс продолжается до тех пор, пока весь список не будет упорядочен.

- **сортировка выбором:** этот алгоритм выбирает наименьший элемент из оставшихся и помещает его в начало списка. Затем он повторяет этот процесс для оставшихся элементов.

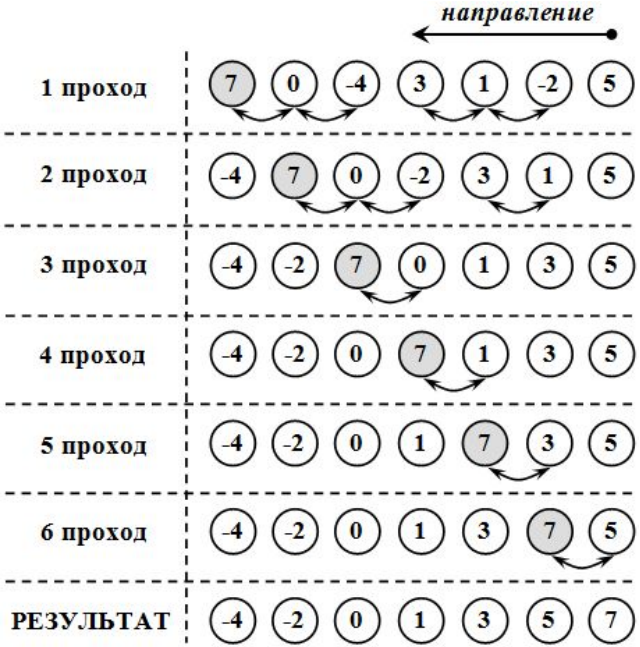
- **сортировка вставками:** этот алгоритм поочередно берет каждый элемент и вставляет его в правильное место среди уже отсортированных элементов. Он продолжает это до тех пор, пока все элементы не будут упорядочены.

и другие.

Сортировки Данных

Sl. No.	Algorithm	Runtime (ms)	Memory (MB)
1	Сортировка пузырьком	TLE	NA
2	Сортировка выбором	TLE	NA
3	Сортировка вставками	TLE	NA
4	Сортировка Шелла	408	20.2
5	Пирамидальная сортировка	532	20.1
6	Сортировка слиянием	388	21.8
7	Быстрая сортировка	236	21.5
8	Сортировка подсчетом	140	23.2

Сортировка Пузырек



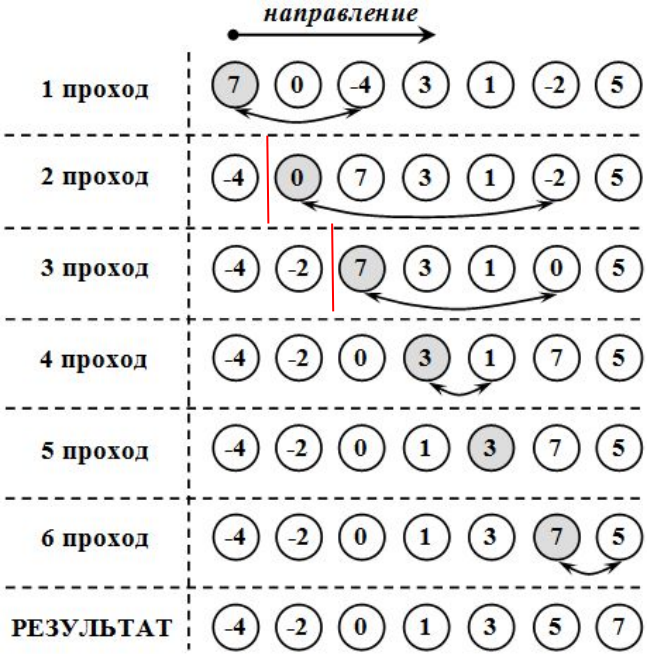
1. Сравниваем пары соседних элементов списка.
2. Если элементы находятся в неправильном порядке, меняем их местами.
3. Проходим по списку до тех пор, пока на очередном проходе не произойдет ни одной перестановки.
4. Когда не происходит перестановок, список считается отсортированным.

```python
# пример сортировки пузырьком (список) по-возрастанию
lst = [5, 6, 1, 3, -1, 0, 1, 2, -4, 4, 55, 101]
n = len(lst)
for i in range(n - 1):
    for j in range(0, n - i - 1):
        if lst[j] > lst[j + 1]:
            # swap elements
            lst[j], lst[j + 1] = lst[j + 1], lst[j]
print(lst)
```

Сортировка Шейкера

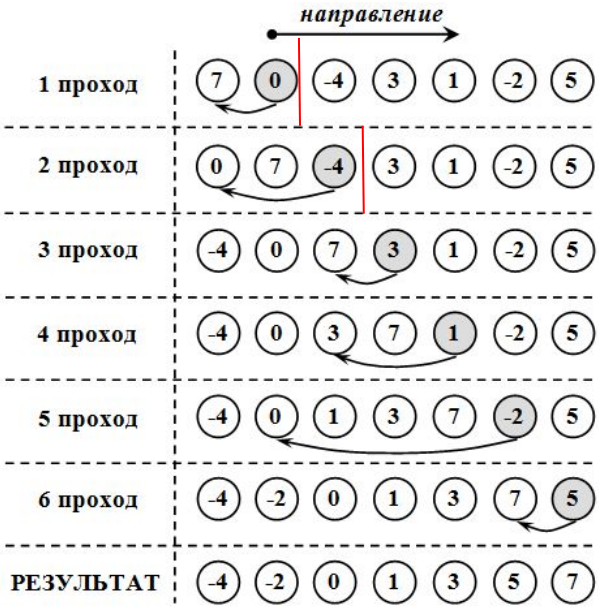
1. Начнем с начала списка.
2. Перемещаем от начала к концу, сравнивая и обменивая соседние элементы, если они находятся в неправильном порядке.
3. Перейдем к концу списка.
4. Перемещаем от конца к началу, сравнивая и обменивая соседние элементы, если они находятся в неправильном порядке.
5. Повторяем процесс, пока не будет выполнено условие, что на очередном проходе не было совершено ни одной перестановки.

Сортировка простого выбора



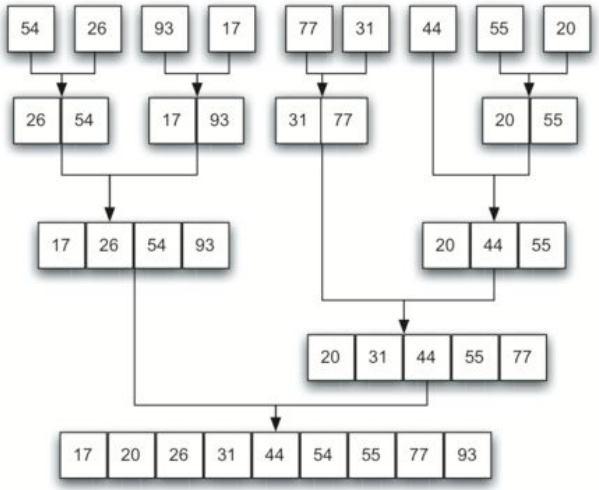
1. Начиная с текущего индекса 0.
2. Находим наименьший элемент в оставшейся части списка.
3. Обмениваем найденный минимальный элемент с элементом на текущем индексе.
4. Увеличиваем текущий индекс и повторяем шаги **2-3** для остальных элементов.
5. Повторяем процесс до тех пор, пока весь список не будет отсортирован.

Сортировка простого включения



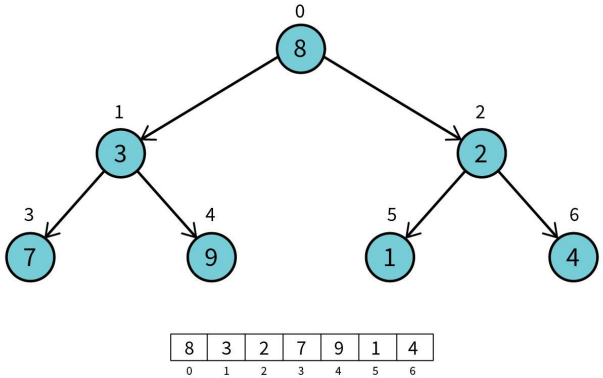
1. Начинаем с текущего индекса 1.
2. Сравниваем текущий элемент со всеми предыдущими от конца до начала.
3. Пока текущий элемент меньше предыдущего, меняем их местами.
4. Уменьшаем текущий индекс и повторяем шаги **2-3**.
5. Повторяем процесс для каждого элемента списка.

Сортировка Слияния (Merge Sort)



1. Разделим массив на две равные части.
2. Рекурсивно применить сортировку слиянием к каждой половине массива.
3. Объединим отсортированные половины в один упорядоченный массив.

Сортировка Пирамидальная (Heap Sort)



1. **Построение кучи (heapify):**
   - Преобразуем не отсортированный список в кучу. Это делается в 2 этапа:
     - Проход по списку и превращение каждого элемента в корень поддерева, удовлетворяющего свойству кучи (по возрастанию или убыванию).
     - "Просеивание" вверх: обмен элемента с его родителями до тех пор, пока свойство кучи не будет восстановлено.
2. **Извлечение элементов из кучи:**
   - Извлекаем максимальный (или минимальный) элемент из корня кучи.
   - Помещаем его в конец списка.
   - Уменьшаем размер кучи.
   - "Прогон" вниз: обмен корня с его наибольшим дочерним элементом до восстановления свойства кучи.
3. **Повторение до полной сортировки:**
   - Повторяем процесс извлечения элементов из кучи до тех пор, пока размер кучи не станет равным 1.

Оценка сложности программы.

Оценка сложности программы представляет собой анализ ресурсов, которые программа требует для выполнения в зависимости от размера входных данных. Основные виды сложности включают в себя **временную сложность** (*время выполнения*) и **пространственную сложность** (*потребление памяти*).

Обозначается обычно "О-большое" (*например, O(n), O(n2)*).

Обычно стремятся к минимизации сложности в смысле ОО-большого (Big-O), так как меньшая сложность обычно означает более эффективный алгоритм.

Оценка сложности программы. Виды.

O(1) Постоянное время - это наилучшая сложность, так как время выполнения не зависит от размера входных данных. Однако, алгоритмы с постоянной сложностью редки, и часто это связано с конкретными операциями над данными (например, доступ к элементу массива).

O(log n) Логарифмическое время также считается отличным. Бинарный поиск, например, имеет логарифмическую сложность и может быть очень эффективным для отсортированных данных.

O(n) Линейное время является хорошим показателем для многих задач. Эффективные алгоритмы линейной сложности могут обрабатывать данные пропорционально их размеру.

O(n log n) Линейное логарифмическое время также считается эффективным и широко используется в алгоритмах сортировки.

O(n^2) Квадратичное время, в целом, менее эффективно, но может быть приемлемым для небольших объемов данных или в специфических сценариях.

Оценка сложности программы.

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Как думаете какая сложность будет? #1

```
lst = [1, 2, 3]
for item in lst:
    print(item)
```

Как думаете какая сложность будет? #2

```
lst = [1, 2, 3]
mid = len(lst) // 2
first_half = lst[:mid]
second_half = lst[mid:]
print(first_half)
print(second_half)
```

Как думаете какая сложность будет? #3

```
lst = [1, 2, 3]
x, y = 1, 3
for i in range(len(lst)):
    x += 1
    y -= 2
    lst[i] *= (x + y)
print(lst)
```

Как думаете какая сложность будет? #4

```
lst = [1, 2, 3]
for i in range(len(lst)):
    for j in range(i+1, len(lst)):
        print(lst[i], lst[j])
```

Как думаете какая сложность будет? #5

```
lst = [1, 2, 3]
print(lst[1])
print(lst[2])
```

Как думаете какая сложность будет? #6

```
lst = [1, 2, 3]
for i in range(len(lst)):
    print(lst[i])
for item in lst:
    print(item)
```

Оценка сложности программы. График.