Cartoons for Open Skies - the Emirates InFlight magazine

Разминка

https://pythonist.ru/test-stroki-i-mnozhestva-v-python/

Что можно посмотреть про расстояние Левенштейна

https://www.geeksforgeeks.org/python/introduction-to-python-levenshtein-module/

Для закрепления ООП

Стек (Stack)

Стек - структура данных LIFO (Last-In-First-Out), где элементы добавляются и удаляются с одного конца (вершины).

Основные операции:

- push(x) - добавить элемент на вершину
- pop() - удалить и вернуть верхний элемент
- peek() - посмотреть верхний элемент без удаления
- is_empty() - проверка на пустоту

In [4]:
```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        """Добавление элемента на вершину стека"""
        self.items.append(item)

    def pop(self):
        """Удаление и возврат верхнего элемента"""
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("pop from empty stack")

    def peek(self):
        """Просмотр верхнего элемента без удаления"""
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("peek from empty stack")

    def is_empty(self):
        """Проверка на пустоту"""
        return len(self.items) == 0
```

```
    def size(self):
        """Размер стека"""
        return len(self.items)

    def __str__(self):
        return f"Stack({self.items})"

# Пример использования
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(4)
stack.push(3)
print(stack)  # Stack([1, 2, 3])
print(stack.pop())  # 3
print(stack.peek())  # 2
print(stack.is_empty())  # False
```

```
Stack([1, 2, 4, 3])
3
4
False
```

Очередь (Queue)

Очередь - структура данных FIFO (First-In-First-Out), где элементы добавляются в конец и удаляются из начала.

Основные операции:

- enqueue(x) - добавить элемент в конец
- dequeue() - удалить и вернуть первый элемент
- front() - посмотреть первый элемент
- is_empty() - проверка на пустоту

In [5]:
```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Добавление элемента в конец очереди"""
        self.items.append(item)

    def dequeue(self):
        """Удаление и возврат первого элемента"""
        if not self.is_empty():
            return self.items.pop(0)
        raise IndexError("dequeue from empty queue")
```

```
    def front(self):
        """Просмотр первого элемента без удаления"""
        if not self.is_empty():
            return self.items[0]
        raise IndexError("front from empty queue")

    def is_empty(self):
        """Проверка на пустоту"""
        return len(self.items) == 0

    def size(self):
        """Размер очереди"""
        return len(self.items)

    def __str__(self):
        return f"Queue({self.items})"

# Пример использования
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue)  # Queue([1, 2, 3])
print(queue.dequeue())  # 1
print(queue.front())    # 2
print(queue.size())     # 2
```

```
Queue([1, 2, 3])
1
2
2
```

Дек (Deque - Double Ended Queue)

Дек - двусторонняя очередь, позволяющая добавлять и удалять элементы с обоих концов.

Основные операции:

- add_front(x) - добавить в начало
- add_rear(x) - добавить в конец
- remove_front() - удалить из начала
- remove_rear() - удалить из конца
- peek_front() - посмотреть первый элемент
- peek_rear() - посмотреть последний элемент

In [ ]:

```python
class Deque:
    def __init__(self):
        self.items = []

    def add_front(self, item):
        """Добавление в начало"""
        self.items.insert(0, item)

    def add_rear(self, item):
        """Добавление в конец"""
        self.items.append(item)

    def remove_front(self):
        """Удаление из начала"""
        if not self.is_empty():
            return self.items.pop(0)
        raise IndexError("remove_front from empty deque")

    def remove_rear(self):
        """Удаление из конца"""
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("remove_rear from empty deque")

    def peek_front(self):
        """Просмотр первого элемента"""
        if not self.is_empty():
            return self.items[0]
        raise IndexError("peek_front from empty deque")

    def peek_rear(self):
        """Просмотр последнего элемента"""
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("peek_rear from empty deque")

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

    def __str__(self):
        return f"Deque({self.items})"

# Пример использования
deque = Deque()
deque.add_rear(1)
deque.add_front(2)
deque.add_rear(3)
print(deque)  # Deque([2, 1, 3])
print(deque.remove_front())  # 2
print(deque.remove_rear())   # 3
```

```
Deque([2, 1, 3])
2
3
```

In [ ]:
```python
from collections import deque

# Создание дека
d = deque([1, 2, 3])

# Основные операции
d.append(4)           # добавить в конец
d.appendleft(0)       # добавить в начало
d.pop()               # удалить с конца
d.popleft()           # удалить с начала

# Дополнительные возможности
d.rotate(1)           # циклический сдвиг вправо
d.rotate(-1)          # циклический сдвиг влево
d.extend([5, 6])      # добавить несколько в конец
d.extendleft([-1, -2]) # добавить несколько в начало

print(d)  # deque([-2, -1, 0, 1, 2, 3, 5])
```

```
deque([-2, -1, 1, 2, 3, 5, 6])
```

In [ ]:
```python
def is_palindrome(word):
    """Проверка строки на палиндром с использованием дека"""
    char_deque = Deque()

    # Добавляем все символы в дек
    for char in word.lower():
        if char.isalpha():  # игнорируем пробелы и знаки препинания
            char_deque.add_rear(char)

    # Сравниваем символы с обоих концов
    while char_deque.size() > 1:
        front = char_deque.remove_front()
        rear = char_deque.remove_rear()
        if front != rear:
            return False

    return True

# Тесты
print(is_palindrome("radar"))        # True
print(is_palindrome("А роза упала на лапу Азора"))  # True
print(is_palindrome("hello"))        # False
```

```
True
True
False
```

Структура	Принцип	Основные операции	Сложность операций
Стек	LIFO	push, pop	O(1)
Очередь	FIFO	enqueue, dequeue	O(1)
Дек	Двусторонний	add_front, add_rear, remove_front, remove_rear	O(1)

Упражнение 8.1: Двусвязный список

Вам предлагается реализовать **двусвязный** список, поддерживающий операции вставки элемента (в произвольную позицию, по индексу, куда происходит вставка), получения значения в элементе по индексу, удаления элемента (по индексу), а также для которого правильно определена базовая функция len().

В качестве основы и примера можете использовать реализацию **односвязного** списка и модифицировать ее под нашу задачу.

Функции от произвольного количества аргументов

Наверняка в ходе решения задач вы задавались вопросом: как можно сделать функцию, которая способна принимать разное, заранее неизвестное количество аргументов?

Что же, пора вскрыть покровы, познакомьтесь с *args и **kwargs!

- `*args` используется для передачи неопределенного числа неименованных аргументов. Если поставить звездочку перед именем, это имя будет представлять собой кортеж из всех переданных аргументов функции.

```
def function(*args):
    for i in args:
        print(i)

function(1, 2, 3, 4)
```

В приведенном выше примере, args — это кортеж (1, 2, 3, 4).

- `**kwargs` работает так же, как и `*args`, но вместо кортежа используется словарь. Это позволяет функции принимать любое количество именованных аргументов.

```
def function(**kwargs):
    for name, value in kwargs.items():
        print(f'{name} = {value}')

function(a=1, b=2, c=3)
```
В этом примере, kwargs — это словарь {'a': 1, 'b': 2, 'c': 3}.

- Функция может быть определена как принимающая и `*args`, и `**kwargs`. В таком случае она будет принимать сначала произвольное количество неименованных элементов, а потом произвольное количество именованных. Важно: конструкцию `**kwargs` нельзя располагать до `*args`. Если это сделать — будет выдано сообщение об ошибке.

```
def function(*args, **kwargs):
    for i in args:
        print(i)
    for name, value in kwargs.items():
        print(f'{name} = {value}')

function(1,2,3,a=1, b=2, c=3)
```

Функции как объекты

В Python все является объектом, включая функции. Это означает, что функции можно передавать в качестве аргументов другим функциям как и любой другой объект. Функции, которые можно передавать в качестве аргументов или возвращать из других функций, известны как объекты первого класса. Это означает, что их можно передавать в качестве аргументов другим функциям, возвращать как значения из других функций и хранить в переменных или структурах данных как любой другой объект. В принципе, функции можно воспринимать просто как класс, для которого определена **операция вызова (call)** -- `(arg1, arg2,...)` . Пример использования функции как объекта первого класса:

```
def hello(name):
    print(f"Hello, {name}!")

greeting_function = hello
greeting_function("User")
```
Рассмотрим другой пример с передачей функции в качестве аргумента другой функции:

```
def apply_function(numbers, function):
    results = []
```

```
    for number in numbers:
        result = function(number)
        results.append(result)
    return results

def square(number):
    return number ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = apply_function(numbers, square)
print(squared_numbers)
```

В этом примере функция `apply_function` принимает в качестве аргументов список чисел и функцию. Функция `apply_function` применяет переданную функцию к каждому числу в списке и возвращает новый измененный список. Функция `square` возводит число в квадрат, и используется в качестве аргумента функции `apply_function`.

Объекты первого класса также позволяют возвращать функции из другой функции:

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

times_2 = make_multiplier(2)
times_3 = make_multiplier(3)

print(times_2(5)) # 10
print(times_3(5)) # 15
```

В этом примере функция `make_multiplier` принимает число `n` и возвращает новую функцию, которая умножает свой аргумент на `n`. Затем эта функция применяется, чтобы создать две новые функции `times_2` и `times_3`, которые умножают свой аргумент на `2` и `3` соответственно. В итоге вызывается функция с аргументом `5`, чтобы увидеть их результаты.

Функции, которые принимают другие функции в качестве аргументов и/или возвращают функции в качестве результатов, называются функциями высшего порядка. Их можно использовать для инкапсуляции многократно используемого поведения и создания более абстрактного кода, о котором легче рассуждать.

Например, встроенные функции `map` и `filter` в Python являются функциями высшего порядка, которые работают с итерируемыми объектами и применяют принимаемую функцию к каждому элементу итерируемого объекта.

Декораторы

Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код. Пример:

```
def my_decorator(function_to_decorate):
     # Внутри себя декоратор определяет функцию-"обёртку". Она будет обёрнута вокруг декорируемой,
     # получая возможность исполнять произвольный код до и после неё.
    def wrapper():
        print("Я - код, который отработает до вызова функции")
        function_to_decorate() # Сама функция
        print("А я - код, срабатывающий после")
     # Вернём эту функцию
     return wrapper

def alone_function():
     print("Я простая одинокая функция, ты ведь не посмеешь меня изменять?")

# Однако, чтобы изменить её поведение, мы можем декорировать её, то есть просто передать декоратору,
# который обернет исходную функцию в любой код, который нам потребуется, и вернёт новую,
# готовую к использованию функцию:
function_decorated = my_decorator(alone_function)
function_decorated()
```

Можем даже сделать вот так, чтобы каждый раз во время вызова alone_function, вместо неё вызывалась function_decorated:

```
alone_function = my_decorator(alone_function)
alone_function()
```

Для упрощения таких действий существует специальный синтаксис декораторов. Вот так можно было записать предыдущий пример, используя его:

```
@my_decorator
def another_alone_function():
     print("Оставь меня в покое")

another_alone_function()
```

Один из важных фактов, которые следует понимать, заключается в том, что функции и методы в Python — это практически одно и то же, за исключением того, что методы всегда ожидают первым параметром ссылку на сам объект (self). Это значит, что мы можем создавать декораторы для методов точно так же, как и для функций, просто не забывая про self.

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie -= 3
        return method_to_decorate(self, lie)
    return wrapper

class Lucy:
    def __init__(self):
        self.age = 32
    @method_friendly_decorator
    def sayYourAge(self, lie):
        print(f"Мне {self.age + lie} лет")

l = Lucy()
l.sayYourAge(-3)
Мне 26 лет, а ты бы сколько дал?
```

Конечно, если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то можно воспользоваться распаковкой аргументов:

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    # Данная "обёртка" принимает любые аргументы
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print("Передали ли мне что-нибудь?:")
        print(args)
        print(kwargs)
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print("Python is cool, no argument here.")

function_with_no_argument()
#Передали ли мне что-нибудь?:
#()
#{}
#Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print(a, b, c)

function_with_arguments(1, 2, 3)
#Передали ли мне что-нибудь?:
#(1, 2, 3)
#{}
#1 2 3
```

In [ ]:
```python
def smart_divide(func):
    def wrapper(*args, **kwargs):
        print(f"Вызывается функция {func.__name__} с аргументами {args} {kwarg
        result = func(*args, **kwargs)
        print(f"Результат: {result}")
        return result
    return wrapper

@smart_divide
def divide(a, b):
    return a / b

divide(10, 2)
```

```
Вызывается функция divide с аргументами (10, 2) {}
Результат: 5.0
```

Out[ ]: 5.0

Асимптотика алгоритмов (потребление ресурсов)

В асимптотическом анализе мы оцениваем производительность алгоритма с точки зрения размера входных данных (мы не измеряем фактическое время выполнения). Мы вычисляем порядок роста времени (или пространства), затраченного алгоритмом с точки зрения размера входных данных. Например, линейный поиск растет линейно, а двоичный поиск растет логарифмически с точки зрения размера входных данных.

Примеры с анализом их сложности:

1. Алгоритм линейного поиска:

```python
# Linearly search target in arr.
# If target is present, return the index;
# otherwise, return -1
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

arr = [1, 10, 30, 15]
x = 30
print(search(arr, x))
```

- Лучший случай: Постоянное время, независимо от размера входных данных. Это будет иметь место, если элемент для поиска

находится в первом индексе указанного списка. Таким образом, количество сравнений в этом случае равно 1.
- Средний случай: Линейное время. Это будет иметь место, если элемент для поиска находится в среднем индексе (при среднем поиске) указанного списка.
- Худший случай: Элемент для поиска отсутствует в списке.

2. **Специальная сумма массива**:
   в этом примере мы возьмем массив длины (n) и рассмотрим следующие случаи:

- Если (n) четное, то наш вывод будет равен 0.
- Если (n) нечетно, то наш вывод будет представлять собой сумму элементов массива.

Ниже представлена реализация данной задачи:

```
def getSum(arr1):
    n = len(arr1)
    if n % 2 == 0:  # (n) is even
        return 0

    sum = 0
    for i in range(n):
        sum += arr1[i]
    return sum  # (n) is odd

# Declaring two lists, one with an odd length
# and the other with an even length
arr1 = [1, 2, 3, 4]
arr2 = [1, 2, 3, 4, 5]
print(getSum(arr1))
print(getSum(arr2))
```

Как анализировать циклы для анализа сложности алгоритмов

Вот общие шаги по анализу циклов для анализа сложности:

- Определить количество итераций цикла. Обычно это делается путем анализа переменных управления циклом и условия завершения цикла.

- Определите количество операций, выполняемых в каждой итерации цикла. Это может включать как арифметические

операции, так и операции доступа к данным, такие как доступ к массиву или доступ к памяти.

- Выразите общее количество операций, выполненных циклом, как функцию размера входных данных. Это может включать использование математических выражений или нахождение выражения в замкнутой форме для количества операций, выполненных циклом.
- Определите порядок роста выражения для числа операций, выполняемых циклом.

Постоянная временная сложность $O(1)$:

Временная сложность функции (или набора операторов) считается равной O(1), если она не содержит циклов, рекурсий и вызовов других функций с непостоянным временем выполнения. То есть набор нерекурсивных и нециклических операторов.

В информатике O(1) относится к постоянной временной сложности, что означает, что время выполнения алгоритма остается постоянным и не зависит от размера входных данных. Это означает, что время выполнения алгоритма O(1) всегда будет занимать одинаковое количество времени независимо от размера входных данных. Примером алгоритма O(1) является доступ к элементу массива с использованием индекса.

Пример:

- Функция swap() имеет временную сложность O(1).
- Цикл или рекурсия, которая выполняется постоянное число раз, также считается O(1). Например, следующий цикл — O(1).

```
# Here c is a constant
for i in range(1, c+1):
        # some O(1) expressions

        # This code is contributed by Pushpesh Raj.
```

Анализ временной сложности:

- Лучший случай: порядок роста будет постоянным , поскольку в лучшем случае мы предполагаем, что (n) четное. Средний случай: В этом случае мы предположим, что четные и нечетные события

равновероятны, поэтому порядок роста будет линейным.

- Худший случай: порядок роста будет линейным , поскольку в этом случае мы предполагаем, что (n) всегда нечетное.

Линейная временная сложность $O(n)$:

Временная сложность цикла считается O(n), если переменные цикла увеличиваются/уменьшаются на постоянную величину.

Линейная временная сложность, обозначаемая как O(n), является мерой роста времени выполнения алгоритма пропорционально размеру входных данных. В алгоритме O(n) время выполнения увеличивается линейно с размером входных данных.

Например, поиск элемента в несортированном массиве или итерация по массиву и выполнение постоянного объема работы для каждого элемента будет O(n) операций. Проще говоря, для входных данных размера n алгоритму требуется n шагов для завершения операции.

```
# Here c is a positive integer constant
for i in range(1, n+1, c):
        # some O(1) expressions

for i in range(n, 0, -c):
        # some O(1) expressions

        # This code is contributed by Pushpesh Raj
```

Квадратичная временная сложность $O(n^2)$:

Квадратичная временная сложность, обозначаемая как $O(n^2)$, относится к алгоритму, время выполнения которого увеличивается пропорционально квадрату размера ввода. Другими словами, для ввода размера n алгоритму требуется n * n шагов для завершения операции.

Примером алгоритма $O(n^2)$ является вложенный цикл, который итерирует по всему вводу для каждого элемента, выполняя постоянный объем работы для каждой итерации. Это приводит к общему количеству итераций n * n, что делает время выполнения квадратичным по размеру ввода.

```
for i in range(1, n+1, c):
        for j in range(1, n+1, c):
```

```
                # some O(1) expressions

for i in range(n, 0, -c):
        for j in range(i+1, n+1, c):
                # some O(1) expressions

                # This code is contributed by Pushpesh Raj
```

Пример: Сортировка выбором и сортировка вставкой имеют временную сложность $O(n^2)$.

Логарифмическая временная сложность $O(Log N)$:

Временная сложность цикла считается как $O(Logn)$, если переменные цикла делятся/умножаются на постоянную величину. А также для рекурсивных вызовов в рекурсивной функции временная сложность считается как $O(Logn)$.

```
i = 1
while(i <= n):
        # some O(1) expressions
        i = i*c

i = n
while(i > 0):
        # some O(1) expressions
        i = i//c

# This code is contributed by Pushpesh Raj
# Recursive function
def recurse(n):
        if(n <= 0):
                return
        else:
                # some O(1) expressions
        recurse(n/c)
# Here c is positive integer constant greater than 1
# This code is contributed by Pushpesh Raj
```

Пример: двоичный поиск имеет временную сложность $O(Logn)$

Логарифмическая временная сложность $O(Log(LogN))$:

Временная сложность цикла считается равной $O(Log(Logn))$, если переменные цикла уменьшаются/увеличиваются экспоненциально на постоянную величину.

```
# Here c is a constant greater than 1
```

```
i = 2
while(i <= n):
        # some O(1) expressions
        i = i**c

# Here fun is sqrt or cuberoot or any other constant root
i = n
while(i > 1):
        # some O(1) expressions
        i = fun(i)

# This code is contributed by Pushpesh Raj
```

Как объединить временные сложности последовательных циклов?

При наличии последовательных циклов мы рассчитываем временную сложность как сумму временных сложностей отдельных циклов.

Чтобы объединить временные сложности последовательных циклов, необходимо учесть количество итераций, выполняемых каждым циклом, и объем работы, выполняемой в каждой итерации. Общую временную сложность алгоритма можно рассчитать, умножив количество итераций каждого цикла на временную сложность каждой итерации и взяв максимум из всех возможных комбинаций.

Например, рассмотрим следующий код:

```
for i in range(n):
  for j in range(m):
    # some constant time operation
```

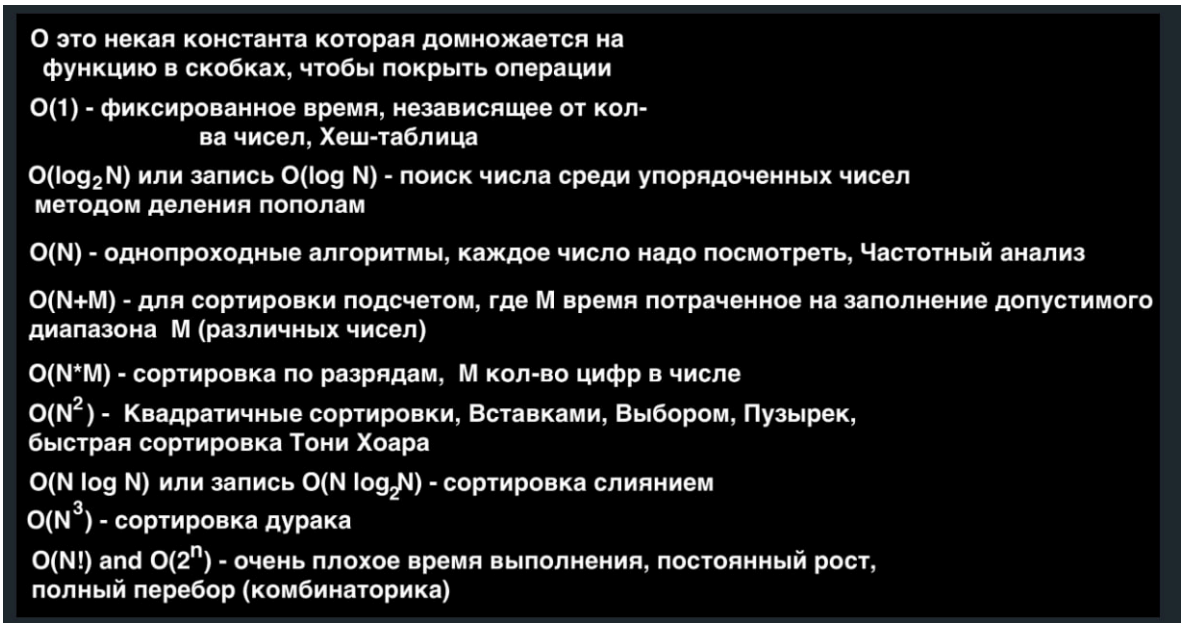
Здесь внешний цикл выполняет n итераций, а внутренний цикл выполняет m итераций для каждой итерации внешнего цикла. Таким образом, общее количество итераций, выполняемых внутренним циклом, равно n * m, а общая временная сложность равна $O(n * m)$.

В другом примере рассмотрим следующий код:

```
for i in range(n):
  for j in range(i):
    # some constant time operation
```

Здесь внешний цикл выполняет n итераций, а внутренний цикл выполняет i итераций для каждой итерации внешнего цикла, где i — текущий счетчик итераций внешнего цикла. Общее количество итераций, выполненных внутренним циклом, можно вычислить, суммируя количество итераций, выполненных в каждой итерации внешнего цикла, которое определяется формулой sum(i) от i=1 до n, что равно n * (n + 1) / 2.

Кусочек квиза!: https://www.geeksforgeeks.org/quizzes/quiz-on-complexity-analysis-for-dsa/?page=1



Сортировки - часть 1

Случайная сортировка

In [1]:
```
import random

def shuffle(a):
  n = len(a)
  for i in range(n):
    j = random.randint(0, n-1)
    a[i], a[j] = a[j], a[i]

def is_sorted(a):
  n = len(a)
  for i in range(n-1):
    if a[i] > a[i+1]:
      return False
  return True

def random_sort(a):
  while not is_sorted(a):
    shuffle(a)

a = [4, 2, 5, 0]
random_sort(a)
print(a)
```

```
[0, 2, 4, 5]
```

Как работает алгоритм:

1. Проверяем, отсортирован ли массив
2. Если нет - полностью перемешиваем его случайным образом
3. Повторяем до тех пор, пока не получим отсортированный массив

- Сложность: $O(n × n!)$ в среднем случае
- Худший случай: Бесконечное время (теоретически)

Сортировка пузырьком

In [2]:
```
def bubble_sort(a):
  n = len(a)
  for i in range(n):
    for j in range(n-i-1):
      if a[j] > a[j+1]:
        a[j], a[j+1] = a[j+1], a[j]

a = [15, -6, 3, 14, -23, 3, 12]
bubble_sort(a)
print(a)
```

```
[-23, -6, 3, 3, 12, 14, 15]
```

In [3]:
```
def bubble_sort(a):
  n = len(a)
  i = 0
  swapped = True
  while swapped:
    swapped = False
    for j in range(n-i-1):
      if a[j] > a[j+1]:
        a[j], a[j+1] = a[j+1], a[j]
        swapped = True
    i += 1

a = [15, -6, 3, 14, -23, 3, 12]
bubble_sort(a)
print(a)
```

```
[-23, -6, 3, 3, 12, 14, 15]
```

1. Оптимизация: Алгоритм останавливается, когда за полный проход не было ни одного обмена (массив уже отсортирован)
2. Уменьшение области прохода: После каждого прохода самый

большой элемент "всплывает" в конец, поэтому в следующем проходе можно не проверять последние i элементов

- Лучший случай: $O(n)$ - когда массив уже отсортирован
- Средний и худший случай: $O(n^2)$
- Память: $O(1)$ (сортировка на месте)
- Устойчивость: Сохраняет порядок равных элементов

Сортировка выбором

In [4]:
```
def selection_sort(a):
  n = len(a)
  for i in range(n-1):
    k = i
    for j in range(i+1, n):
      if a[j] < a[k]:
        k = j
    a[k], a[i] = a[i], a[k]

a = [15, -6, 3, 14, -23, 3, 12]
selection_sort(a)
print(a)
```

```
[-23, -6, 3, 3, 12, 14, 15]
```

Как работает алгоритм:

1. Находим минимальный элемент в неотсортированной части массива
2. Меняем его с первым элементом неотсортированной части
3. Увеличиваем границу отсортированной части на 1 элемент
4. Повторяем до полной сортировки

- Сложность: $O(n^2)$ в лучшем, среднем и худшем случаях
- Память: $O(1)$ (сортировка на месте)
- Неустойчивость: Может менять порядок равных элементов

Сортировка вставками

In [5]:
```python
def insertion_sort(a):
  n = len(a)
  for i in range(1, n):
    for j in range(i, 0, -1):
      if a[j] < a[j-1]:
        a[j], a[j-1] = a[j-1], a[j]

a = [15, -6, 3, 14, -23, 3, 12]
insertion_sort(a)
print(a)
```

```
[-23, -6, 3, 3, 12, 14, 15]
```

Как работает алгоритм:

1. Начинаем со второго элемента (i = 1)
2. Для каждого элемента "проталкиваем" его влево, пока он не займет правильную позицию в отсортированной части
3. Повторяем для всех элементов массива

- Лучший случай: $O(n)$ - когда массив уже отсортирован
- Средний и худший случай: $O(n^2)$
- Память: $O(1)$ (сортировка на месте)
- Устойчивость: Сохраняет порядок равных элементов

https://habr.com/ru/companies/kts/articles/727528/