

***Дисциплина: Разработка программных модулей***

**Отчет по выполнению задания на тему: «Техническая документация к проекту курсовой работы».**

**Группа: 3-ИСП11-18**

**Выполнили студенты:**

Зверев Н.Д.

**Проверил преподаватель:**

Джегюеде М.Э.

Москва 2024 г.

1. **Ссылка на репозиторий GitHub:**

<https://github.com/ZverevNikita/sort_visualizer>

1. Техническая документация проекта курсовой работы

* Описание каждого класса, метода и переменной

1. **Класс «Quick Sort» -** класс, представляющий алгоритм быстрой сортировки

**Переменные:**

* array: массив для сортировки
* fig, ax: объекты графического окна и осей для визуализации данных

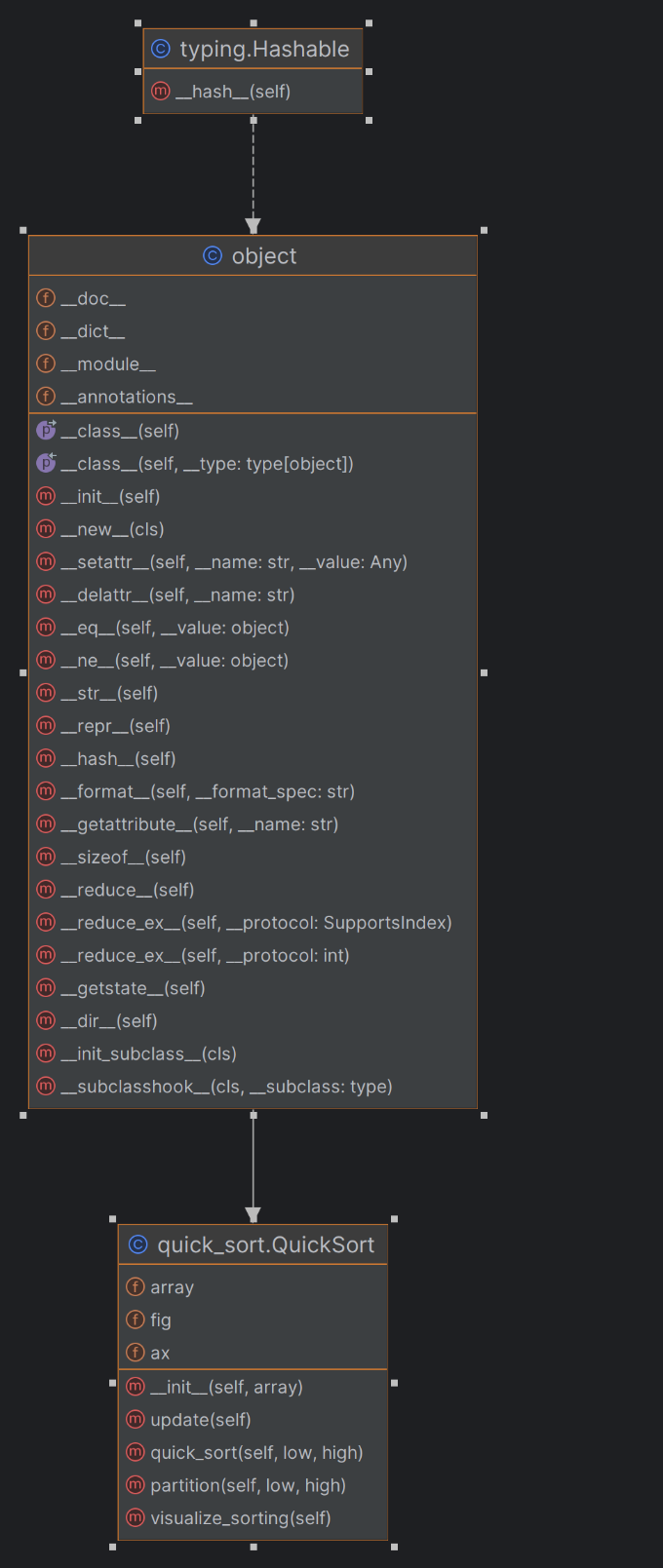
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, который инициализирует объект QuickSort с массивом array и создает графическое окно для визуализации сортировки
* update(self): метод для обновления графического отображения текущего состояния сортировки путем очистки графика и добавления новых столбцов на график
* quick\_sort(self, low, high): рекурсивный метод для выполнения быстрой сортировки. Разбивает массив на подмассивы и сортирует их
* partition(self, low, high): метод для разделения подмассива на две части относительно центрального элемента (опорного элемента) pivot. Возвращает индекс опорного элемента
* visualize\_sorting(self): метод для отображения окончательного результата сортировки

**Комментирование кода:**

# Импорт модуля random для генерации случайных чисел  
import random  
# Импорт модуля time для работы с временем  
import time  
# Импорт модуля networkx для работы с графами  
import networkx as nx  
# Импорт модуля matplotlib.pyplot для визуализации данных  
import matplotlib.pyplot as plt  
# Импорт класса deque из модуля collections  
from collections import deque   
  
# Определение класса QuickSort  
class QuickSort:  
 # Определение конструктора класса, который инициализирует массив  
 def \_\_init\_\_(self, array):  
 # Инициализация массива в объекте класса  
 self.array = array  
 # Создание объекта графика для визуализации  
 self.fig, self.ax = plt.subplots()  
  
 # Метод для обновления графика визуализации  
 def update(self):  
 # Очистка текущего графика  
 self.ax.clear()  
 # Отображение столбцов диаграммы на графике  
 self.ax.bar(range(len(self.array)), self.array, color='b')  
 # Установка заголовка графика  
 self.ax.set\_title('Быстрая сортировка (Quick Sort)')  
 # Установка подписи оси X  
 self.ax.set\_xlabel('Количество элементов в массиве')  
 # Установка подписи оси Y  
 self.ax.set\_ylabel('Значения элементов в массиве')  
 # Пауза для обновления графика  
 plt.pause(0.1)  
  
 # Метод быстрой сортировки с параметрами low и high  
 def quick\_sort(self, low, high):  
 # Проверка условия выхода из рекурсии  
 if low < high:  
 # Находим опорный элемент и разделяем массив  
 pivot = self.partition(low, high)  
 # Рекурсивный вызов быстрой сортировки для левой части  
 self.quick\_sort(low, pivot)  
 # Рекурсивный вызов быстрой сортировки для правой части  
 self.quick\_sort(pivot + 1, high)  
  
 # Метод разделения массива на две части  
 def partition(self, low, high):  
 # Опорный элемент - первый элемент массива  
 pivot = self.array[low]  
 # Индекс i для прохода по элементам слева  
 i = low - 1  
 # Индекс j для прохода по элементам справа  
 j = high + 1  
 # Бесконечный цикл разделения массива  
 while True:  
 i += 1  
 # Поиск элемента больше или равного опорному слева  
 while self.array[i] < pivot:  
 i += 1  
 j -= 1  
 # Поиск элемента меньше или равного опорному справа  
 while self.array[j] > pivot:  
 j -= 1  
 # Проверка условия завершения разделения  
 if i >= j:  
 # Возвращаем индекс j (новый опорный элемент)  
 return j  
 # Обмен элементов для разделения  
 self.array[i], self.array[j] = self.array[j], self.array[i]  
 # Обновление графика после обмена  
 self.update()  
  
 # Метод для запуска визуализации сортировки  
 def visualize\_sorting(self):  
 # Отображение окна с графиком  
 plt.show()

**UML-диаграмма класса «Quick Sort»:**



**Анализ быстрой сортировки:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0 секунд

128 элементов массива - 0.0009970664978027344 секунд

256 элементов массива - 0.0009989738464355469 секунд

512 элементов массива - 0.0009980201721191406 секунд

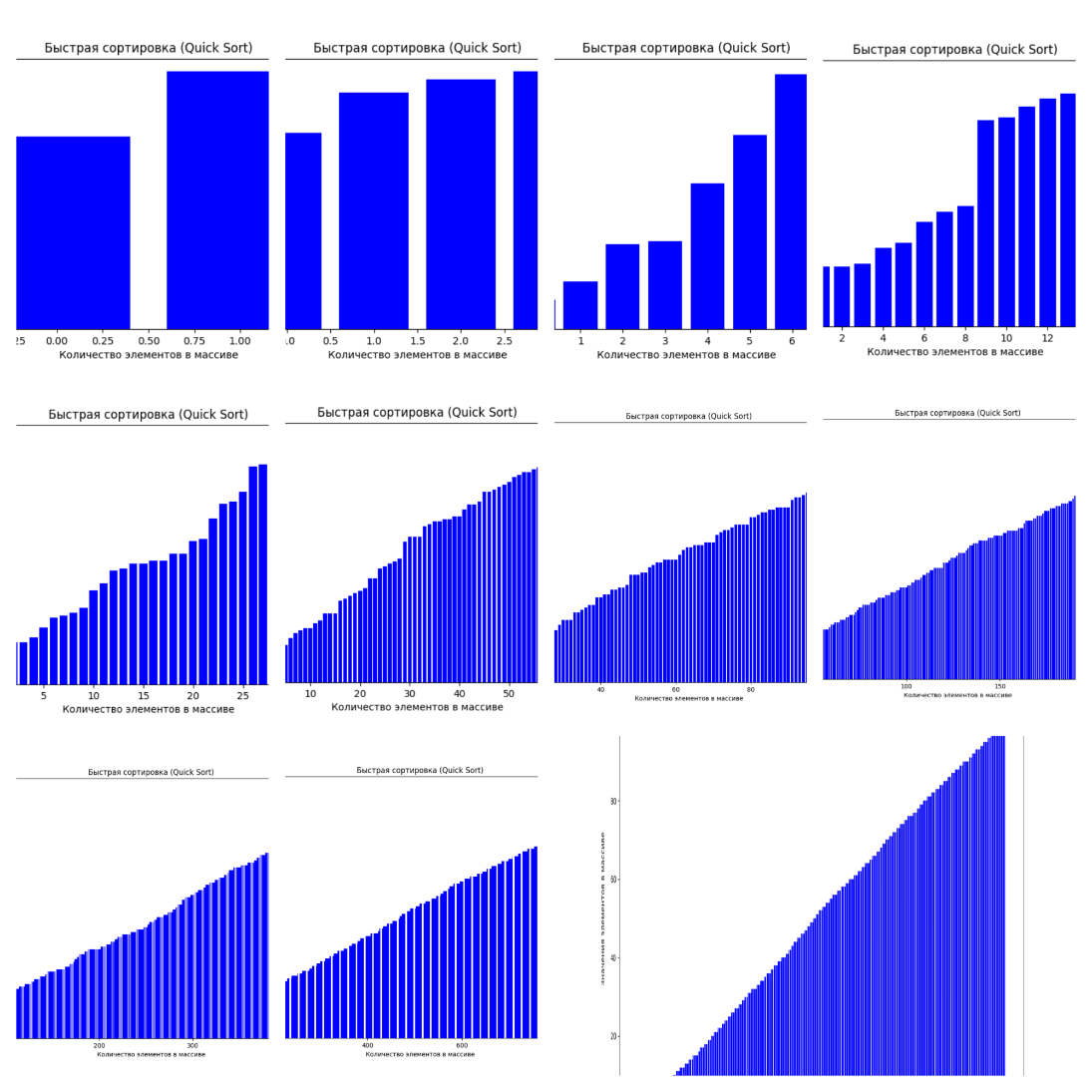
1024 элементов массива - 0.003026247024536133 секунд

2048 элементов массива - 0.0060176849365234375 секунд

**График анализа быстрой сортировки:**



**Визуальный анализ алгоритма быстрой сортировки:**

****

1. **Класс «MergeSort» -** класс, представляющий алгоритм сортировки слиянием

**Переменные:**

* array: массив для сортировки
* fig, ax: объекты графического окна и осей для визуализации данных

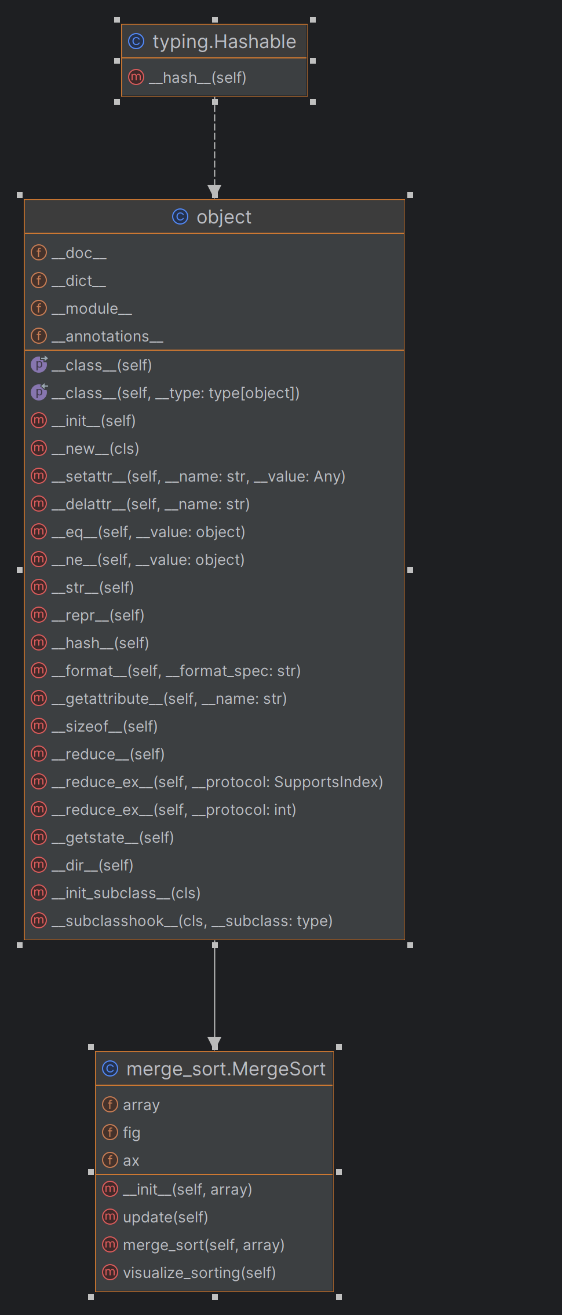
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, который принимает массив array для сортировки и создает графическое окно для визуализации
* update(): метод для обновления графического отображения текущего состояния сортировки путем очистки графика, добавления новых столбцов и обновления заголовка и подписей осей
* merge\_sort(self, array): рекурсивный метод реализующий алгоритм сортировки слиянием, делит массив на две половины, рекурсивно сортирует и объединяет их
* visualize\_sorting(self): метод для отображения окончательного результата сортировки

**Комментирование кода:**

# Определение класса MergeSort  
class MergeSort:  
 # Определение конструктора класса, который инициализирует массив  
 def \_\_init\_\_(self, array):  
 # Инициализация массива в объекте класса  
 self.array = array  
 # Создание объекта графика для визуализации  
 self.fig, self.ax = plt.subplots()   
  
 # Метод для обновления графика визуализации  
 def update(self):  
 # Очистка текущего графика  
 self.ax.clear()   
 # Отображение столбцов диаграммы на графике  
 self.ax.bar(range(len(self.array)), self.array, color='b')  
 # Установка заголовка графика  
 self.ax.set\_title('Сортировка слиянием (Merge Sort)')  
 # Установка подписи оси X  
 self.ax.set\_xlabel('Количество элементов в массиве')  
 # Установка подписи оси Y  
 self.ax.set\_ylabel('Значения элементов в массиве')  
 # Пауза для обновления графика  
 plt.pause(0.01)   
  
 # Рекурсивный метод сортировки слиянием  
 def merge\_sort(self, array):  
 # Проверка, что массив содержит более одного элемента  
 if len(array) > 1:  
 # Находим середину массива  
 mid = len(array) // 2  
 # Делим массив на левую половину  
 left\_half = array[:mid]  
 # Делим массив на правую половину  
 right\_half = array[mid:]  
  
 # Рекурсивный вызов сортировки для левой половины  
 self.merge\_sort(left\_half)  
 # Рекурсивный вызов сортировки для правой половины  
 self.merge\_sort(right\_half)  
  
 # Инициализация индексов для слияния  
 i = j = k = 0   
  
 # Слияние двух отсортированных половин массива  
 while i < len(left\_half) and j < len(right\_half):  
 if left\_half[i] < right\_half[j]:  
 array[k] = left\_half[i]  
 i += 1  
 else:  
 array[k] = right\_half[j]  
 j += 1  
 k += 1  
  
 # Добавление оставшихся элементов из левой половины  
 while i < len(left\_half):  
 array[k] = left\_half[i]  
 i += 1  
 k += 1  
  
 # Добавление оставшихся элементов из правой половины  
 while j < len(right\_half):  
 array[k] = right\_half[j]  
 j += 1  
 k += 1  
  
 # Обновление графика после слияния  
 self.update()   
  
 # Метод для запуска визуализации сортировки  
 def visualize\_sorting(self):  
 # Отображение окна с графиком  
 plt.show()

**UML-диаграмма класса «MergeSort»:**



**Анализ сортировки слиянием:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0 секунд

128 элементов массива - 0.0 секунд

256 элементов массива - 0.0 секунд

512 элементов массива - 0.001079559326171875 секунд

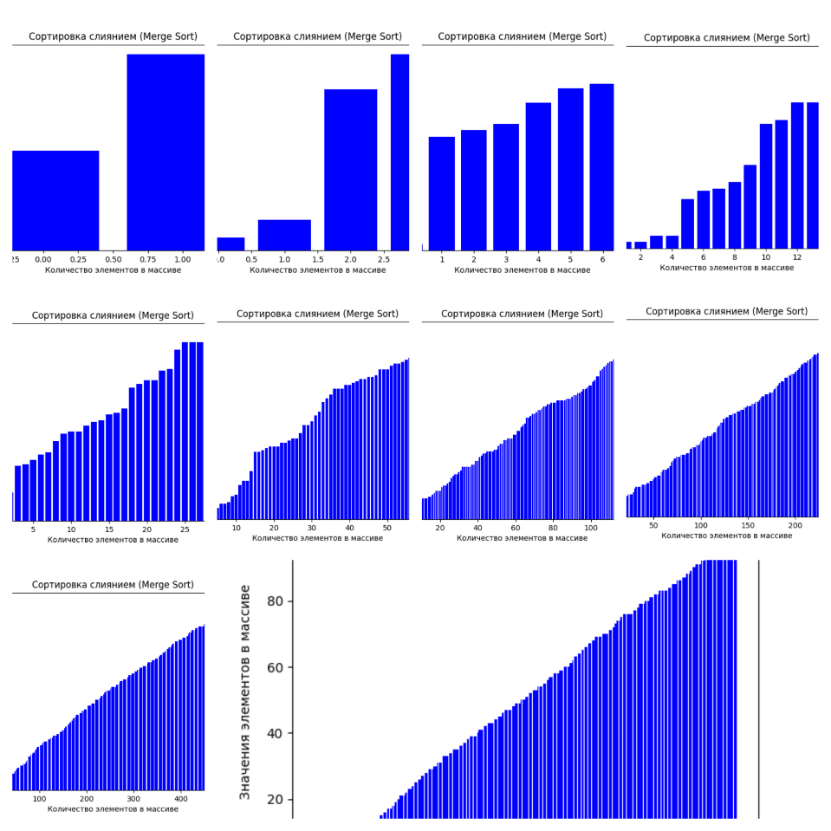
1024 элементов массива - 0.0029926300048828125 секунд

2048 элементов массива - 0.0049860477447509766 секунд

**График анализа сортировки слиянием:**



**Визуальный анализ алгоритма сортировки слиянием:**

****

1. **Класс «BubbleSort»** - класс, представляющий алгоритм сортировки пузырьком

**Переменные**:

* array: массив для сортировки
* fig, ax: объекты графического окна и осей для визуализации данных
* bar: столбики в графике для каждого элемента массива

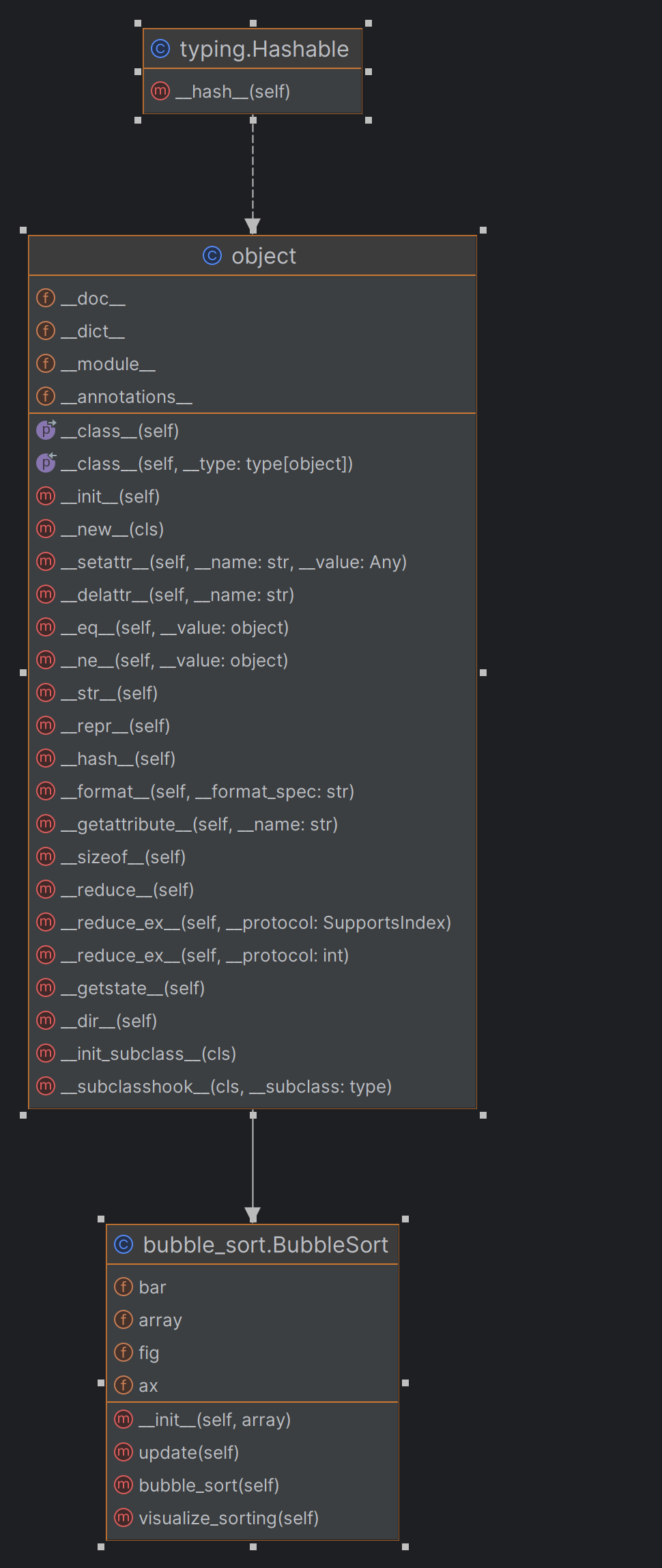
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, который принимает массив array для сортировки и инициализирует графическое окно для визуализации сортировки пузырьком. Устанавливает заголовок и подписи осей, создает столбики для графика
* update(self): метод для обновления графического отображения текущего состояния сортировки, изменяя высоту столбцов на графике
* bubble\_sort(self): метод, реализующий алгоритм сортировки пузырьком. Проходит по массиву и меняет местами элементы, пока массив не будет отсортирован
* visualize\_sorting(self): метод для отображения окончательного результата сортировки

**Комментирование кода:**

# Определяем класс BubbleSort для реализации алгоритма сортировки пузырьком  
class BubbleSort:  
 # Метод инициализации класса с массивом в качестве параметра  
 def \_\_init\_\_(self, array):  
 # Инициализируем массив для сортировки  
 self.array = array  
 # Создаем фигуру и оси для визуализации  
 self.fig, self.ax = plt.subplots()  
 # Устанавливаем заголовок графика  
 self.ax.set\_title('Сортировка пузырьком (Bubble Sort)')  
 # Устанавливаем подпись по оси X  
 self.ax.set\_xlabel('Количество элементов в массиве')  
 # Устанавливаем подпись по оси Y  
 self.ax.set\_ylabel('Значения элементов в массиве')  
 # Создаем столбчатую диаграмму для массива  
 self.bar = self.ax.bar(range(len(array)), array, color='b')  
  
 # Метод для обновления графика визуализации  
 def update(self):  
 for rect, height in zip(self.bar, self.array):  
 # Устанавливаем высоту столбцов диаграммы  
 rect.set\_height(height)  
 # Приостанавливаем выполнение на короткое время для обновления графика  
 plt.pause(0.01)  
  
 # Метод для сортировки массива методом пузырька  
 def bubble\_sort(self):  
 # Получаем длину массива  
 n = len(self.array)  
 for i in range(n):  
 for j in range(0, n - i - 1):  
 if self.array[j] > self.array[j + 1]:  
 # Обмениваем элементы, если нужно  
 self.array[j], self.array[j + 1] = self.array[j + 1], self.array[j]  
 # Обновляем визуализацию после каждого шага сортировки  
 self.update()  
  
 # Метод для визуализации сортировки  
 def visualize\_sorting(self):  
 # Отображаем окончательный результат сортировки  
 plt.show()

**UML-диаграмма класса «BubbleSort»:**



**Анализ сортировки пузырьком:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0009999275207519531 секунд

128 элементов массива - 0.0009951591491699219 секунд

256 элементов массива - 0.003024578094482422 секунд

512 элементов массива - 0.010006427764892578 секунд

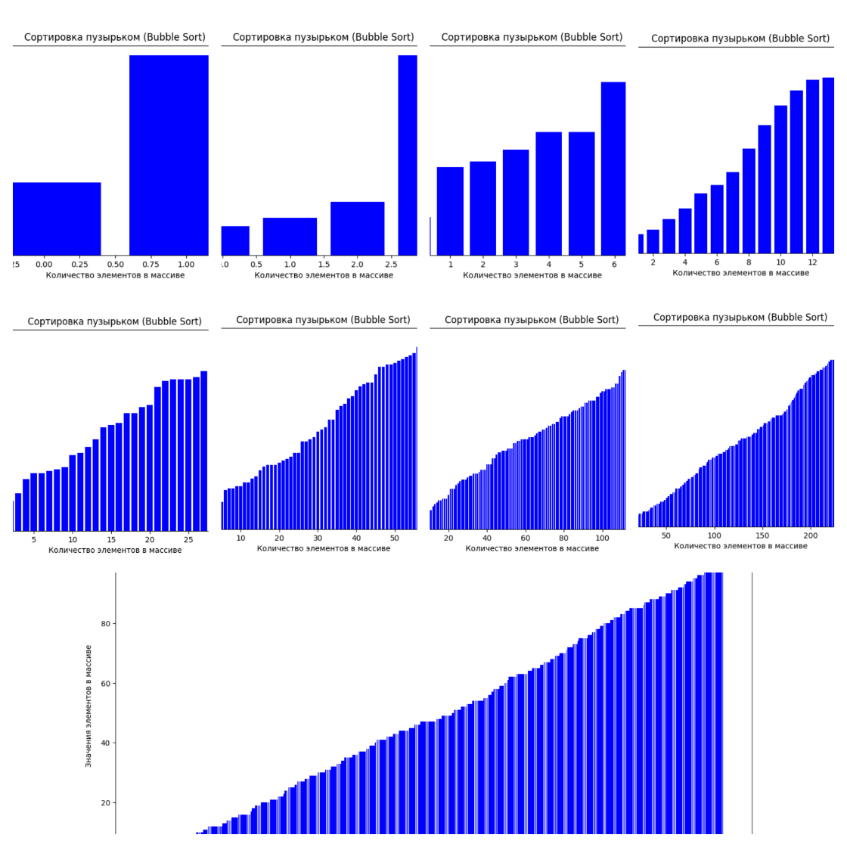
1024 элементов массива - 0.03688764572143555 секунд

2048 элементов массива - 0.15661287307739258 секунд

**График анализа сортировки пузырьком:**

****

**Визуальный анализ алгоритма сортировки пузырьком:**

****

1. **Класс «InsertionSort»** - класс, представляющий алгоритм сортировки вставками

**Переменные**:

* array: массив для сортировки
* fig, ax: объекты графического окна и осей для визуализации данных

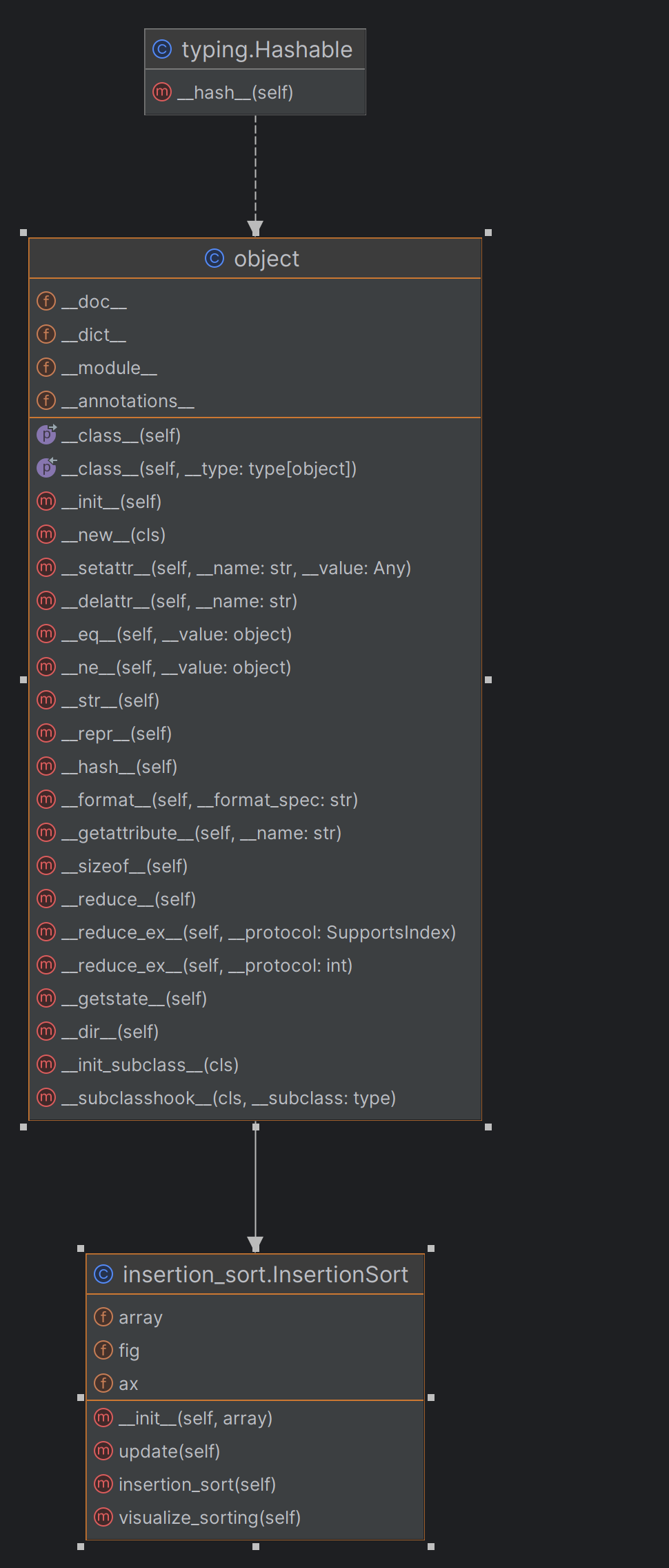
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, принимающий массив array для сортировки и инициализирующий графическое окно для визуализации
* update(self): метод для обновления графического отображения текущего состояния сортировки вставкой, очищает ось и строит столбчатую диаграмму с текущими значениями массива
* insertion\_sort(self): метод, который реализует алгоритм сортировки вставкой. Проходит по массиву, вставляя элементы в отсортированное подмножество
* visualize\_sorting(self): метод для отображения окончательного результата сортировки

**Комментирование кода:**

# Определяем класс InsertionSort для реализации алгоритма сортировки вставкой  
class InsertionSort:  
 # Метод инициализации класса с массивом в качестве параметра  
 def \_\_init\_\_(self, array):  
 # Инициализируем массив для сортировки  
 self.array = array  
 # Создаем фигуру и оси для визуализации  
 self.fig, self.ax = plt.subplots()  
  
 # Метод для обновления графика визуализации  
 def update(self):  
 # Очищаем текущий график  
 self.ax.clear()  
 # Отображаем столбчатую диаграмму для массива  
 self.ax.bar(range(len(self.array)), self.array, color='b')  
 self.ax.set\_title('Сортировка вставкой (Insertion Sort)')  
 self.ax.set\_xlabel('Количество элементов в массиве')  
 self.ax.set\_ylabel('Значения элементов в массиве')  
 # Приостанавливаем выполнение на короткое время для обновления графика  
 plt.pause(0.01)  
  
 # Метод для сортировки массива методом вставок  
 def insertion\_sort(self):  
 for i in range(1, len(self.array)):  
 key = self.array[i]  
 j = i - 1  
 while j >= 0 and self.array[j] > key:  
 self.array[j + 1] = self.array[j]  
 j -= 1  
 self.update()  
 self.array[j + 1] = key  
 # Обновляем визуализацию после каждого шага сортировки  
 self.update()  
  
 # Метод для визуализации сортировки  
 def visualize\_sorting(self):  
 # Отображаем окончательный результат сортировки  
 plt.show()

**UML-диаграмма класса «InsertionSort»:**



**Анализ сортировки вставкой:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0 секунд

128 элементов массива - 0.0 секунд

256 элементов массива - 0.0009815692901611328 секунд

512 элементов массива - 0.0051534175872802734 секунд

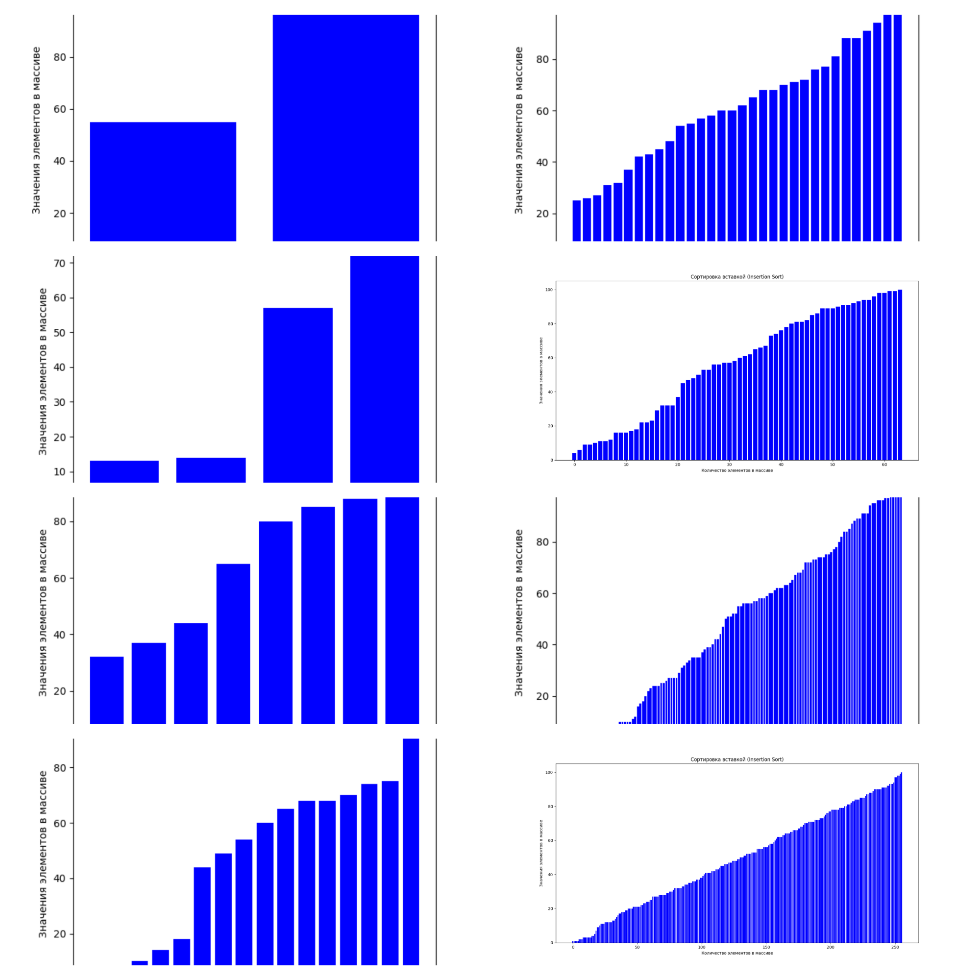
1024 элементов массива - 0.018196582794189453 секунд

2048 элементов массива - 0.07284212112426758 секунд

**График анализа сортировки вставкой:**

****

**Визуальный анализ алгоритма сортировки вставками:**

****

1. **Класс «BinarySearch» -** класс, представляющий алгоритм двоичного поиска

**Переменные:**

* array: массив для поиска
* fig, ax: объекты графического окна и осей для визуализации данных

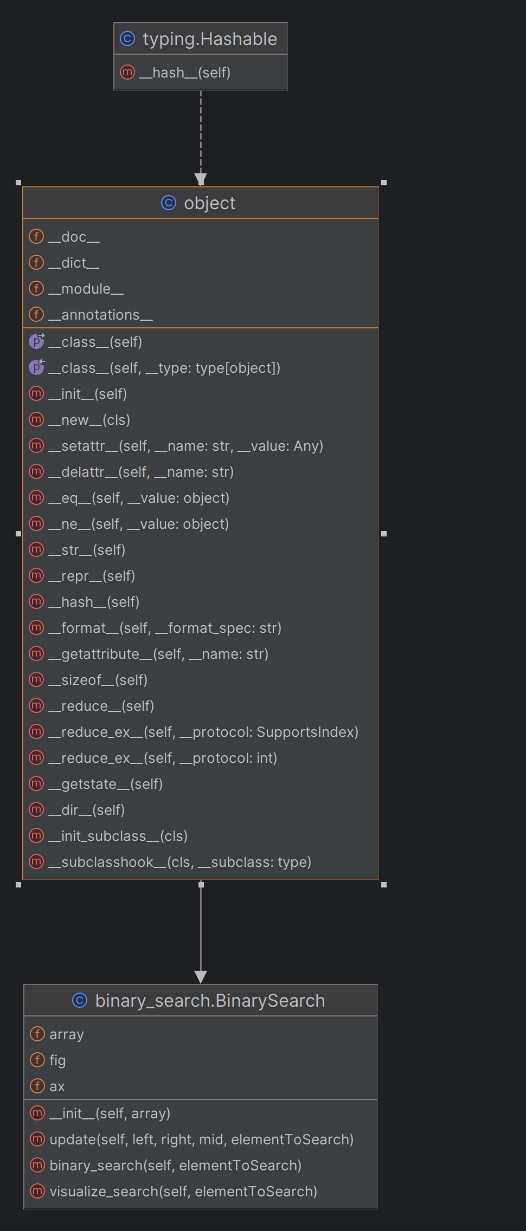
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, принимающий массив array для поиска и инициализирующий графическое окно для визуализации. Устанавливает заголовок, подписи осей
* update(self, left, right, mid, elementToSearch): метод для обновления графического отображения состояния бинарного поиска. Очищает график, строит столбчатую диаграмму массива, добавляет горизонтальную линию для целевого элемента и столбец для сравниваемого элемента
* binary\_search(self, elementToSearch): метод реализации алгоритма бинарного поиска. Осуществляет поиск целевого элемента elementToSearch в массиве и возвращает его индекс
* visualize\_search(self, elementToSearch): метод для визуализации поиска переданного элемента elementToSearch, вызывает binary\_search и отображает окончательный результат

**Комментирование кода:**

# Определяем класс BinarySearch для реализации алгоритма бинарного поиска  
class BinarySearch:  
 # Метод инициализации класса с массивом в качестве параметра  
 def \_\_init\_\_(self, array):  
 # Инициализируем массив для поиска  
 self.array = array  
 # Создаем фигуру и оси для визуализации  
 self.fig, self.ax = plt.subplots()  
 # Устанавливаем заголовок и подписи осей на графике  
 self.ax.set\_title('Бинарный поиск (Binary Search)')  
 self.ax.set\_xlabel('Индексы элементов в массиве')  
 self.ax.set\_ylabel('Значения элементов в массиве')  
  
 # Метод для обновления графика визуализации  
 def update(self, left, right, mid, elementToSearch):  
 # Очищаем текущий график  
 self.ax.clear()  
 # Отображаем столбчатую диаграмму для массива  
 self.ax.bar(range(len(self.array)), self.array, color='b', alpha=0.7)  
 # Добавляем горизонтальную линию для целевого элемента  
 self.ax.axhline(y=elementToSearch, color='r', linestyle='--', label='Целевой элемент массива')  
 if left <= right:  
 # Отображаем сравниваемый элемент, если левая граница меньше или равна правой  
 self.ax.bar(mid, self.array[mid], color='g', label='Сравниваемый элемент', alpha=0.7)  
 # Добавляем легенду  
 self.ax.legend()  
 # Приостанавливаем выполнение на короткое время для обновления графика  
 plt.pause(0.5)  
  
 # Метод реализующий алгоритм бинарного поиска  
 def binary\_search(self, elementToSearch):  
 left = 0  
 # Устанавливаем начальные границы поиска  
 right = len(self.array) - 1  
 while left <= right:  
 # Находим середину массива  
 middle = (left + right) // 2  
 # Обновляем визуализацию  
 self.update(left, right, middle, elementToSearch)  
 if self.array[middle] == elementToSearch:  
 # Если элемент найден, возвращаем его индекс  
 return middle  
 elif self.array[middle] < elementToSearch:  
 # Сдвигаем левую границу поиска  
 left = middle + 1  
 else:  
 # Сдвигаем правую границу поиска  
 right = middle - 1  
 # Если элемент не найден, возвращаем -1  
 return -1  
  
 def visualize\_search(self, elementToSearch):  
 result = self.binary\_search(elementToSearch)  
 # Визуализируем результат поиска  
 plt.show()

**UML-диаграмма класса «BinarySearch»:**



**Анализ двоичного поиска итерационным методом:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0 секунд

128 элементов массива - 0.0 секунд

256 элементов массива - 0.0009922981262207031 секунд

512 элементов массива - 0.0009644031524658203 секунд

1024 элементов массива - 0.0010004043579101562 секунд

2048 элементов массива - 0.002959012985229492 секунд

**График анализа двоичного поиска итерационным методом:**



**Визуальный анализ алгоритма двоичного поиска:**



1. **Класс «JumpingSearch»** - класс, представляющий алгоритм поиска прыжками

**Переменные**:

* array: массив для поиска
* fig, ax: объекты графического окна и осей для визуализации данных

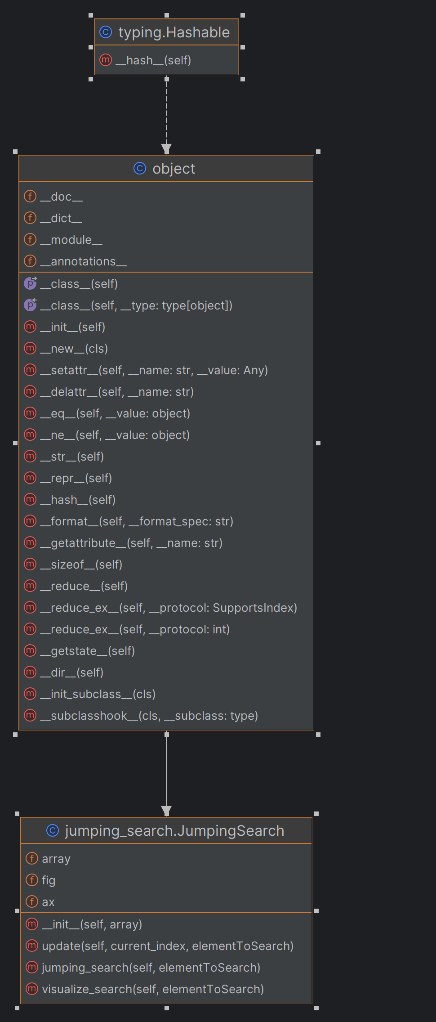
**Методы:**

* \_\_init\_\_(self, array): конструктор класса, который принимает массив array для поиска и инициализирует графическое окно для визуализации. Устанавливает заголовок, подписи осей
* update(self, current\_index, elementToSearch): метод для обновления графического отображения текущего состояния поиска прыжками. Очищает график, строит столбчатую диаграмму массива, добавляет горизонтальную линию для целевого элемента и столбец для найденного элемента
* jumping\_search(self, elementToSearch): метод для реализации алгоритма поиска прыжками. Осуществляет поиск целевого элемента elementToSearch в массиве и выдаёт его индекс
* visualize\_search(self, elementToSearch): метод для визуализации поиска переданного элемента elementToSearch, вызывает jumping\_search и отображает окончательный результат

**Комментирование кода:**

# Определяем класс JumpingSearch для реализации алгоритма поиска прыжками  
class JumpingSearch:  
 # Метод инициализации класса с массивом в качестве параметра  
 def \_\_init\_\_(self, array):  
 # Инициализируем массив для поиска  
 self.array = array  
 # Создаем фигуру и оси для визуализации  
 self.fig, self.ax = plt.subplots()  
 # Устанавливаем заголовок и подписи осей на графике  
 self.ax.set\_title('Поиск прыжками (Jumping Search)')  
 self.ax.set\_xlabel('Количество элементов в массиве')  
 self.ax.set\_ylabel('Значения элементов в массиве')  
  
 # Метод для обновления графика визуализации  
 def update(self, current\_index, elementToSearch):  
 # Очищаем текущий график  
 self.ax.clear()  
 # Отображаем столбчатую диаграмму для массива  
 self.ax.bar(range(len(self.array)), self.array, color='b', alpha=0.7)  
 # Добавляем горизонтальную линию для целевого элемента  
 self.ax.axhline(y=elementToSearch, color='r', linestyle='--', label='Целевой элемент массива')  
 if current\_index != -1:  
 # Отображаем найденный элемент, если он существует  
 self.ax.bar(current\_index, self.array[current\_index], color='g', label='Найденный элемент массива', alpha=0.7)  
 # Добавляем легенду  
 self.ax.legend()  
 # Приостанавливаем выполнение на короткое время для обновления графика  
 plt.pause(0.5)  
  
 # Метод реализующий алгоритм поиска прыжками  
 def jumping\_search(self, elementToSearch):  
 # Получаем длину массива  
 arrayLength = len(self.array)  
 jumpStep = int(arrayLength \*\* 0.5)  
 # Инициализируем переменные для шагов алгоритма  
 previousStep = 0  
  
 # Выполняем прыжки пока значение на текущем шаге меньше целевого элемента  
 while self.array[min(jumpStep, arrayLength) - 1] < elementToSearch:  
 previousStep = jumpStep  
 # Обновляем шаги  
 jumpStep += int(arrayLength \*\* 0.5)  
 if previousStep >= arrayLength:  
 return -1  
 # Обновляем визуализацию  
 self.update(previousStep, elementToSearch)  
  
 while self.array[previousStep] < elementToSearch:  
 previousStep += 1  
 if previousStep == min(jumpStep, arrayLength):  
 return -1  
 # Продолжаем искать целевой элемент  
 self.update(previousStep, elementToSearch)  
  
 if self.array[previousStep] == elementToSearch:  
 self.update(previousStep, elementToSearch)  
 # Если найден целевой элемент, обновляем и возвращаем индекс  
 return previousStep  
  
 def visualize\_search(self, elementToSearch):  
 result = self.jumping\_search(elementToSearch)  
 # Визуализируем результат поиска  
 plt.show()

**UML-диаграмма класса «JumpingSearch»:**



**Анализ поиска прыжками:**

2 элемента массива - 0.0 секунд

4 элемента массива - 0.0 секунд

8 элементов массива - 0.0 секунд

16 элементов массива - 0.0 секунд

32 элемента массива - 0.0 секунд

64 элемента массива - 0.0 секунд

128 элементов массива - 0.0 секунд

256 элементов массива - 0.0009989738464355469 секунд

512 элементов массива - 0.005254983901977539 секунд

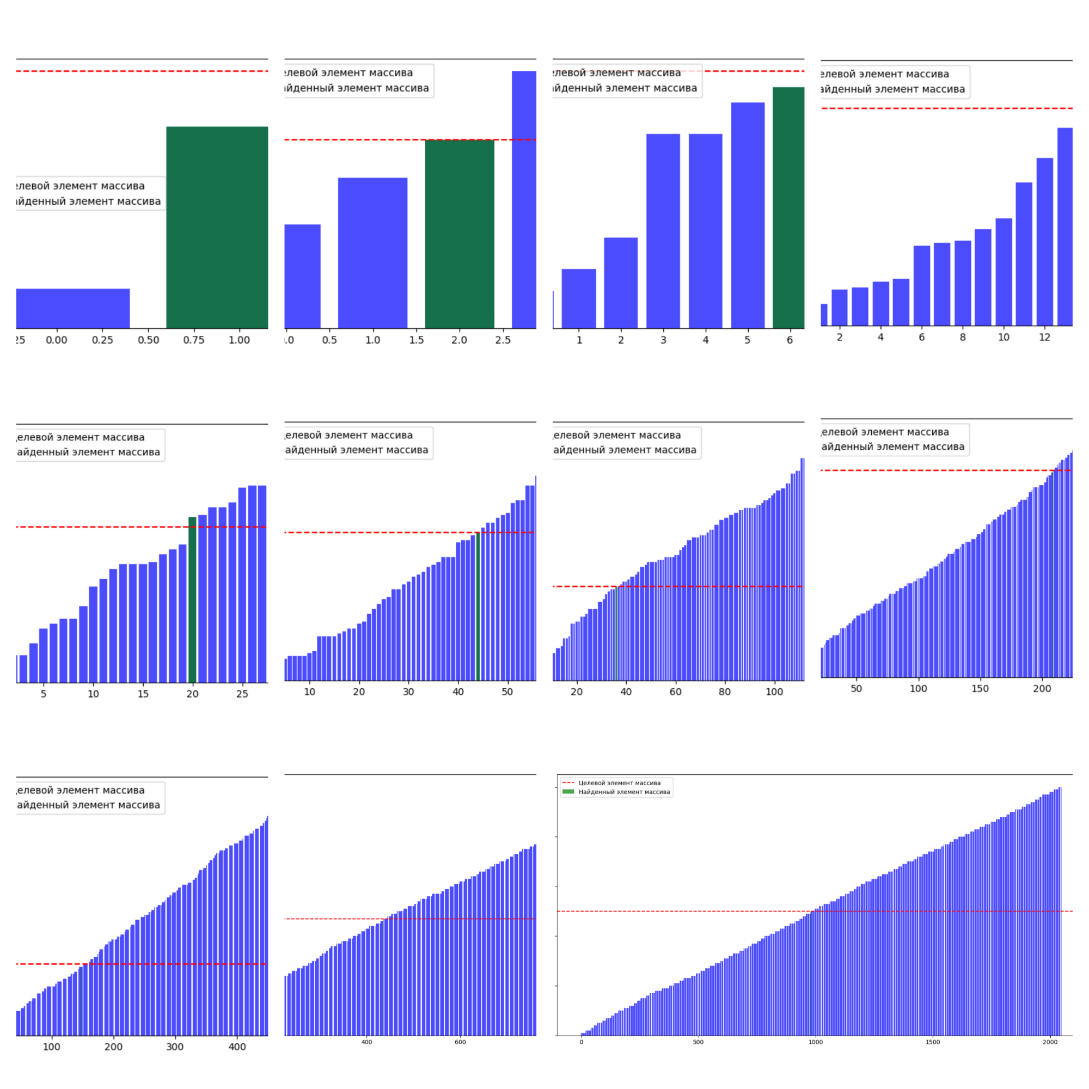
1024 элементов массива - 0.005371809005737305 секунд

2048 элементов массива - 0.005230903625488281 секунд

**График анализа поиска прыжками:**



**Визуальный анализ алгоритма поиска прыжками:**

****

1. **Класс «BFS»** - класс, представляющий алгоритм сортировки в графе в ширину

**Переменные:**

* graph: словарь, представляющий граф, где ключи - узлы, а значения - список смежных узлов
* nodes: список строковых значений, представляющих узлы графа
* visited: множество, хранящее посещенные узлы
* queue: очередь для обхода графа в ширину
* result: список для хранения узлов в порядке обхода
* intermediate\_results: список списков для записи промежуточных результатов обхода

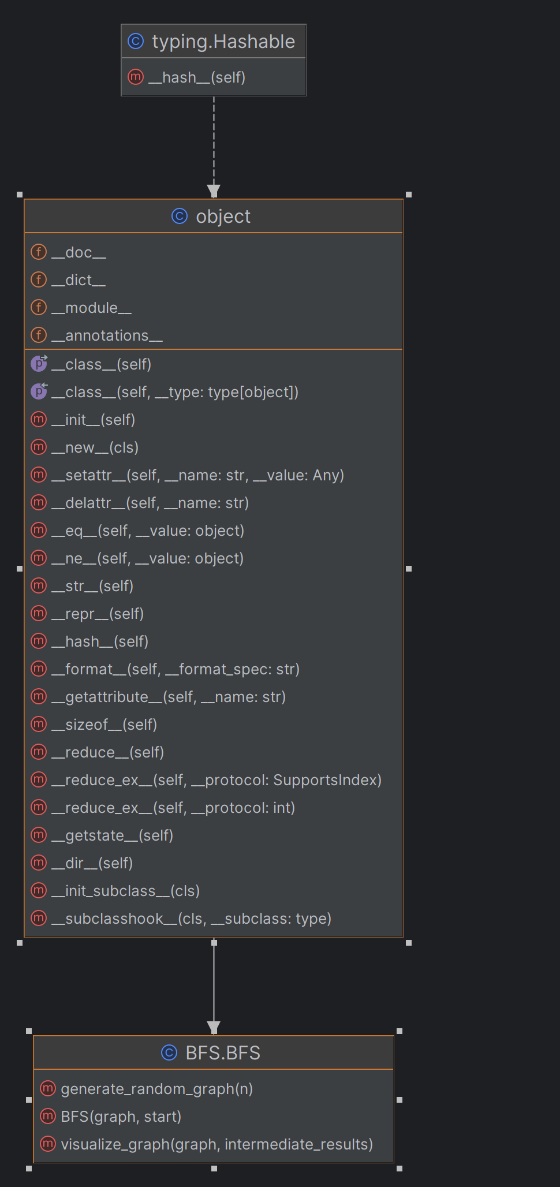
**Методы:**

* generate\_random\_graph(n): создает случайный граф с n узлами
* BFS(graph, start): выполняет поиск в ширину в графе graph с начальным узлом start
* visualize\_graph(graph, intermediate\_results): визуализирует промежуточные результаты BFS на графе

**Комментирование кода:**

# Определяем класс BFS для реализации алгоритма поиска в ширину (BFS)  
class BFS:  
 # Метод для генерации случайного графа  
 def generate\_random\_graph(n):  
 # Инициализируем пустой словарь для представления графа  
 graph = {}  
 # Создаем список узлов графа  
 nodes = [str(i) for i in range(n)]  
  
 # Проходим по каждому узлу  
 for node in nodes:  
 # Добавляем узел в граф с пустым списком смежных узлов  
 graph[node] = []  
  
 # Проходим по узлам с индексами  
 for i, node in enumerate(nodes):  
 # Выбираем случайные соседние узлы для текущего узла  
 for other\_node in random.sample(nodes[:i] + nodes[i + 1:], k=random.randint(0, n - 1)):  
 # Добавляем соседние узлы в список смежности текущего узла  
 graph[node].append(other\_node)  
  
 # Возвращаем сгенерированный граф  
 return graph  
  
 # Метод для выполнения поиска в ширину  
 def BFS(graph, start):  
 # Инициализируем множество для отслеживания посещенных узлов  
 visited = set()  
 # Инициализируем очередь с начальным узлом и путем  
 queue = deque([(start, [start])])  
 # Добавляем начальный узел в посещенные  
 visited.add(start)  
 # Список для хранения результата поиска  
 result = []  
 # Список для промежуточных результатов  
 intermediate\_results = []  
  
 # Пока очередь не пуста  
 while queue:  
 # Извлекаем узел и путь из очереди  
 node, path = queue.popleft()  
 # Добавляем узел в результат  
 result.append(node)  
 # Добавляем текущий результат в список промежуточных результатов  
 intermediate\_results.append(list(result))  
  
 # Перебираем соседей текущего узла  
 for neighbor in graph[node]:  
 # Если сосед не был посещен  
 if neighbor not in visited:  
 # Добавляем соседа в очередь с обновленным путем  
 queue.append((neighbor, path + [neighbor]))  
 # Добавляем соседа в посещенные  
 visited.add(neighbor)  
 # Добавляем соседа в результат  
 result.append(neighbor)  
 # Добавляем текущий результат в список промежуточных результатов  
 intermediate\_results.append(list(result))  
  
 # Возвращаем список промежуточных результатов  
 return intermediate\_results  
  
 # Метод для визуализации графа и промежуточных результатов  
 def visualize\_graph(graph, intermediate\_results):  
 G = nx.Graph()  
 plt.figure(figsize=(8, 6))  
  
 for node in intermediate\_results[0]:  
 # Добавляем узлы из первого промежуточного результата в граф  
 G.add\_node(node)  
  
 for i in range(1, len(intermediate\_results)):  
 # Очищаем текущий граф  
 plt.clf()  
  
 for edge in graph.items():  
 for n in edge[1]:  
 if edge[0] in intermediate\_results[i] and n in intermediate\_results[i]:  
 # Добавляем ребра между узлами из текущего промежуточного результата  
 G.add\_edge(edge[0], n)  
  
 pos = nx.spring\_layout(G)  
 nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='b')  
 # Визуализируем граф на текущем шаге с задержкой и отображаем его  
 plt.title(f'Шаг {i}')  
 plt.pause(0.5)  
 plt.show()

**UML-диаграмма класса «BFS»:**



**Анализ сортировки в графе методом поиска в ширину:**

2 элемента в графе - 0.0 секунд

4 элемента в графе - 0.0 секунд

8 элементов в графе - 0.0 секунд

16 элементов в графе - 0.0 секунд

32 элемента в графе - 0.0 секунд

64 элемента в графе - 0.0 секунд

128 элементов в графе - 0.0009946823120117188 секунд

256 элементов в графе - 0.0019948482513427734 секунд

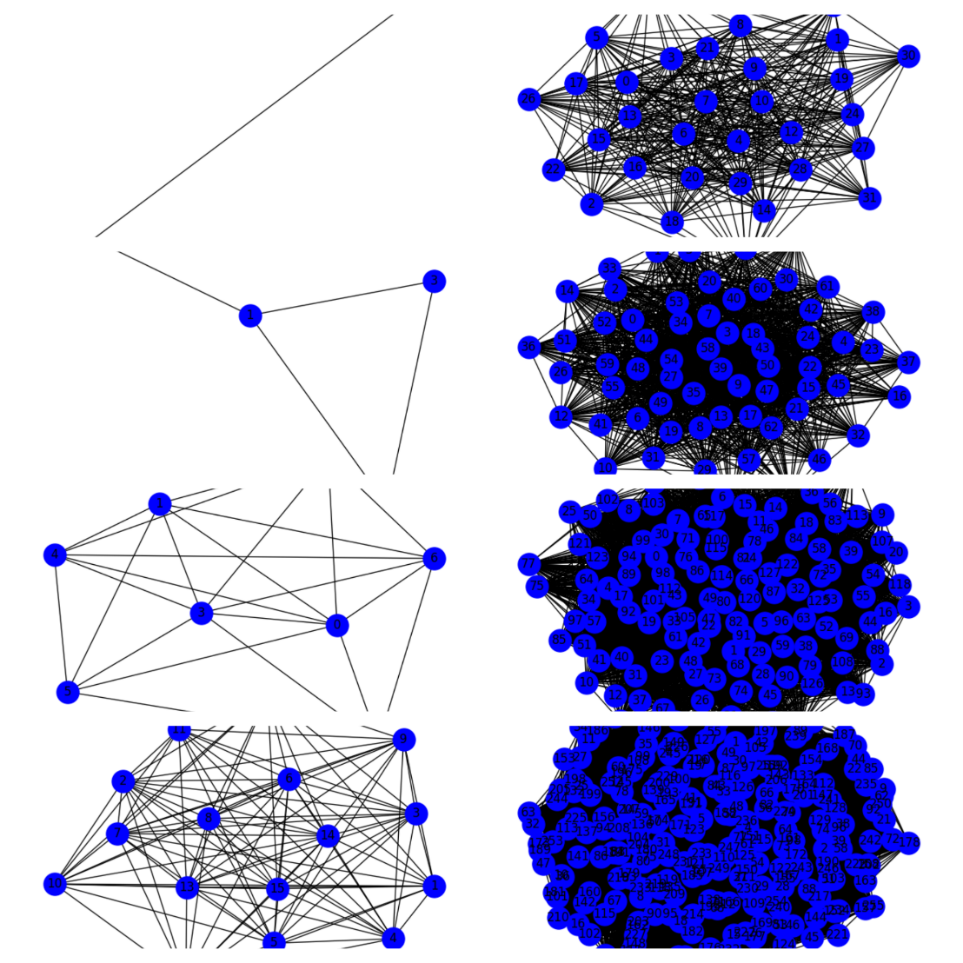
512 элементов в графе - 0.012963056564331055 секунд

1024 элементов в графе - 0.06192517280578613 секунд

2048 элементов в графе - 0.26728367805480957 секунд

**График анализа сортировки в графе методом поиска в ширину:**

**Визуальный анализ сортировки в графе методом поиска в ширину:**

****

1. **Класс «DFS»** - класс, представляющий алгоритм сортировки в глубину

**Переменные**:

* graph: словарь, представляющий граф (узлы и их соседи)
* nodes: список узлов
* visited: множество для отслеживания посещенных узлов
* stack: стек для обхода узлов в глубину
* result: список для хранения порядка посещения узлов в DFS
* intermediate\_results: список списков для промежуточных результатов DFS
* G: объект графа NetworkX
* pos: переменная для хранения позиций узлов на графе

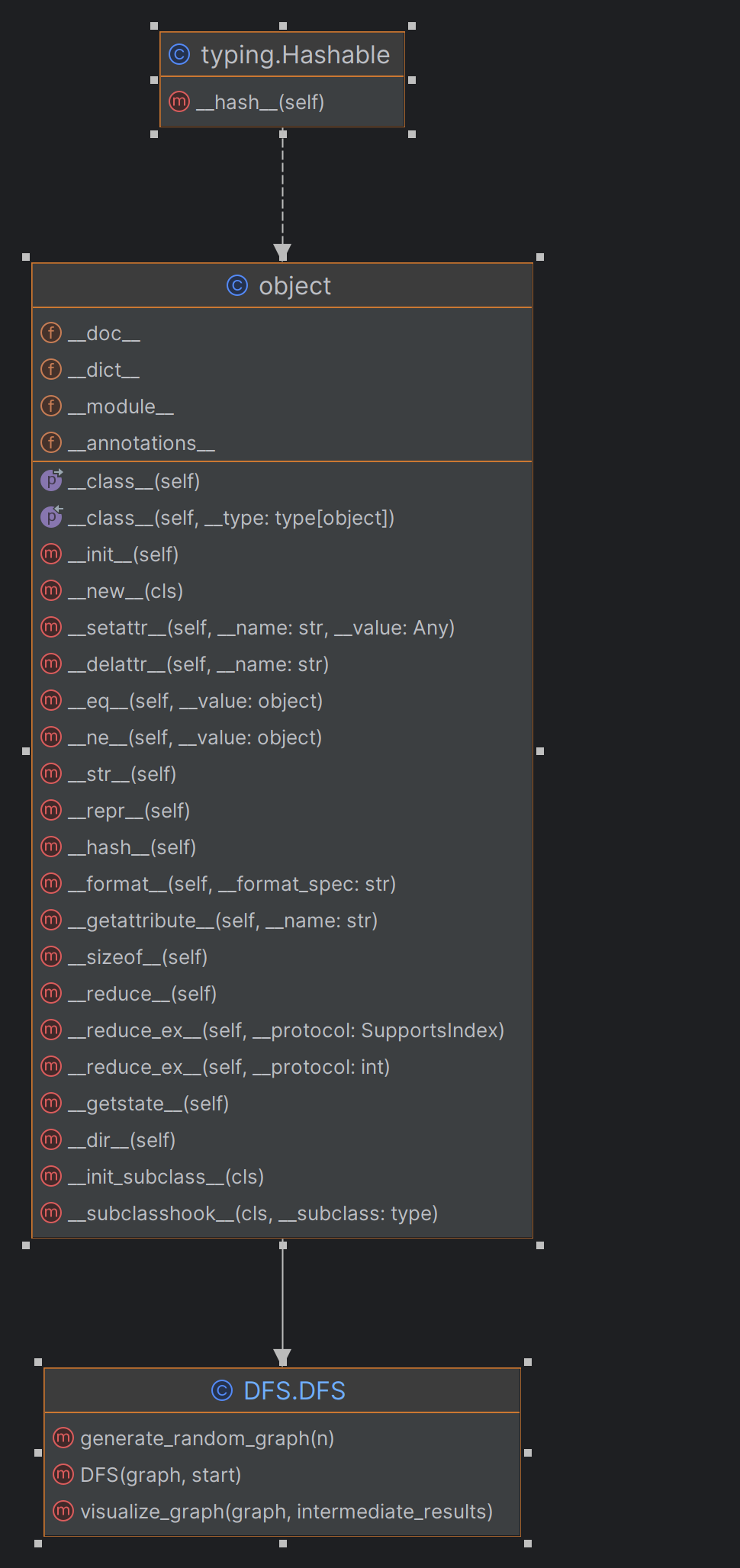
**Методы:**

* generate\_random\_graph(n): генерирует случайный граф с n узлами и возвращает его
* DFS(graph, start): выполняет обход в глубину начиная с узла start в графе graph и возвращает промежуточные результаты
* visualize\_graph(graph, intermediate\_results): визуализирует граф и промежуточные результаты обхода в глубину

**Комментирование кода:**

# Определяем класс DFS для реализации алгоритма поиска в глубину (DFS)  
class DFS:  
 # Метод для генерации случайного графа  
 def generate\_random\_graph(n):  
 # Инициализируем пустой словарь для представления графа  
 graph = {}  
 # Создаем список узлов графа  
 nodes = [str(i) for i in range(n)]  
  
 # Проходим по каждому узлу  
 for node in nodes:  
 # Добавляем узел в граф с пустым списком смежных узлов  
 graph[node] = []  
  
 # Проходим по узлам с индексами  
 for i, node in enumerate(nodes):  
 # Выбираем случайные соседние узлы для текущего узла  
 for other\_node in random.sample(nodes[:i] + nodes[i + 1:], k=random.randint(0, n - 1)):  
 # Добавляем соседние узлы в список смежности текущего узла  
 graph[node].append(other\_node)  
  
 # Возвращаем сгенерированный граф  
 return graph  
  
 # Метод для выполнения поиска в глубину  
 def DFS(graph, start):  
 # Инициализируем множество для отслеживания посещенных узлов  
 visited = set()  
 # Инициализируем стек с начальным узлом и путем  
 stack = [(start, [start])]  
 # Добавляем начальный узел в посещенные  
 visited.add(start)  
 # Список для хранения результата поиска  
 result = []  
 # Список для промежуточных результатов  
 intermediate\_results = []  
  
 # Пока стек не пуст  
 while stack:  
 # Извлекаем узел и путь из стека  
 node, path = stack.pop()  
 # Добавляем узел в результат  
 result.append(node)  
 # Добавляем текущий результат в список промежуточных результатов  
 intermediate\_results.append(list(result))  
  
 # Перебираем соседей текущего узла  
 for neighbor in graph[node]:  
 # Если сосед не был посещен  
 if neighbor not in visited:  
 # Добавляем соседа в стек с обновленным путем  
 stack.append((neighbor, path + [neighbor]))  
 # Добавляем соседа в посещенные  
 visited.add(neighbor)  
 # Добавляем соседа в результат  
 result.append(neighbor)  
 # Добавляем текущий результат в список промежуточных результатов  
 intermediate\_results.append(list(result))  
  
 # Возвращаем список промежуточных результатов  
 return intermediate\_results  
  
 # Метод для визуализации графа и промежуточных результатов  
 def visualize\_graph(graph, intermediate\_results):  
 G = nx.Graph()  
 plt.figure(figsize=(8, 6))  
  
 for node in intermediate\_results[0]:  
 # Добавляем узлы из первого промежуточного результата в граф  
 G.add\_node(node)  
  
 for i in range(1, len(intermediate\_results)):  
 # Очищаем текущий граф  
 plt.clf()  
  
 for edge in graph.items():  
 for n in edge[1]:  
 if edge[0] in intermediate\_results[i] and n in intermediate\_results[i]:  
 # Добавляем ребра между узлами из текущего промежуточного результата  
 G.add\_edge(edge[0], n)  
  
 pos = nx.spring\_layout(G)  
 nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='b')  
 # Визуализируем граф на текущем шаге с задержкой и отображаем его  
 plt.title(f'Шаг {i}')  
 plt.pause(0.5)  
 plt.show()

**UML-диаграмма класса «DFS»:**



**Анализ сортировки в графе методом поиска в глубину:**

2 элемента в графе - 0.0 секунд

4 элемента в графе - 0.0 секунд

8 элементов в графе - 0.0 секунд

16 элементов в графе - 0.0 секунд

32 элемента в графе - 0.0 секунд

64 элемента в графе - 0.0 секунд

128 элементов в графе - 0.0 секунд

256 элементов в графе - 0.0 секунд

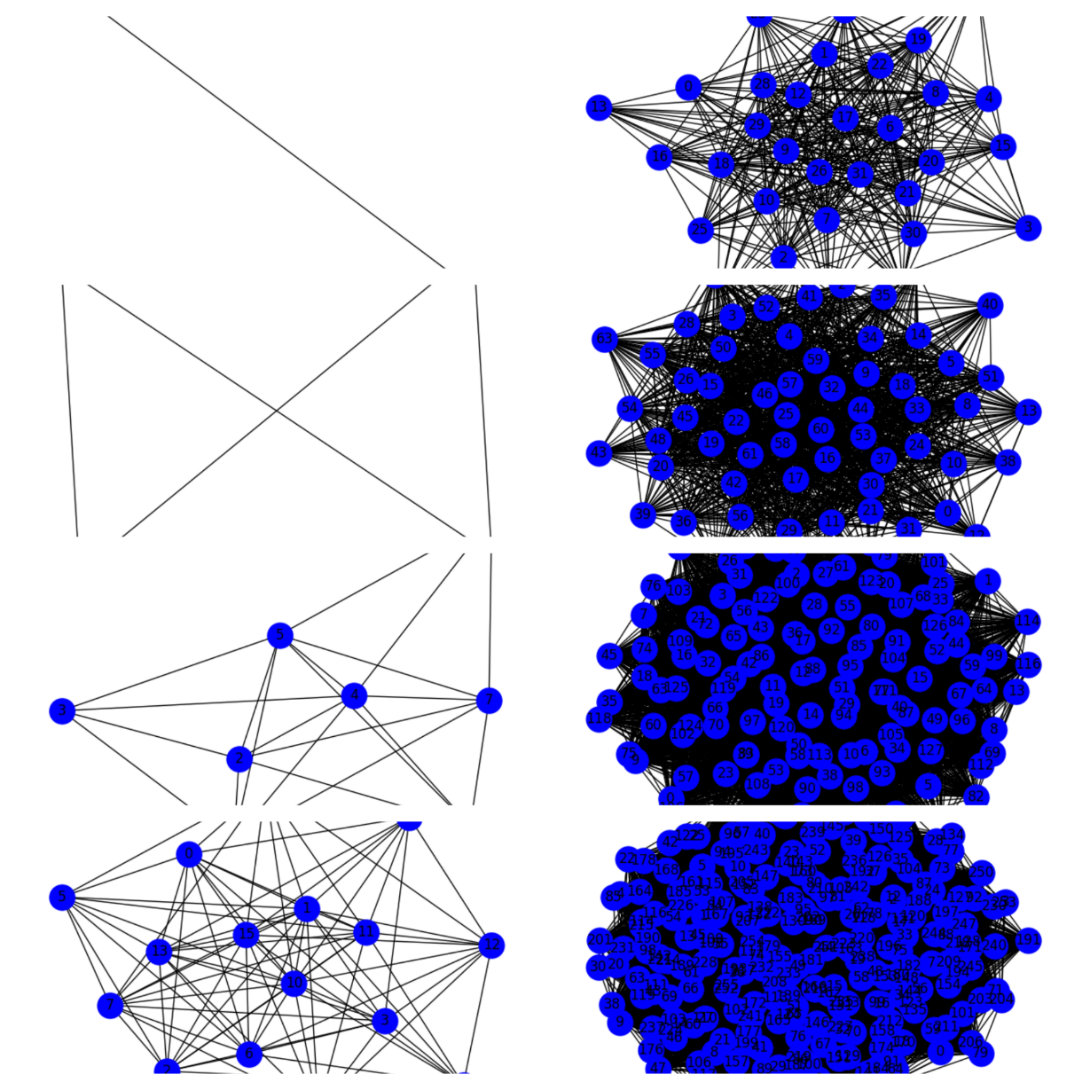
512 элементов в графе - 0.003989219665527344 секунд

1024 элементов в графе - 0.016954660415649414 секунд

2048 элементов в графе - 0.0718076229095459 секунд

**График анализа сортировки в графе методом поиска в глубину:**

**Визуальный анализ сортировки в графе методом поиска в глубину:**

****

1. **Класс «KMP»** - класс, представляющий алгоритм Кнута-Морриса-Пратта

**Переменные**:

* patternLength: длина шаблона
* compliedPatternArray: массив, содержащий информацию о совпадениях префиксов и суффиксов шаблона
* k: индекс для обновления массива compliedPatternArray
* n: длина текста
* m: длина шаблона
* foundIndexes: список индексов, где найдена подстрока
* patternIndex: индекс для сравнения шаблона и текста

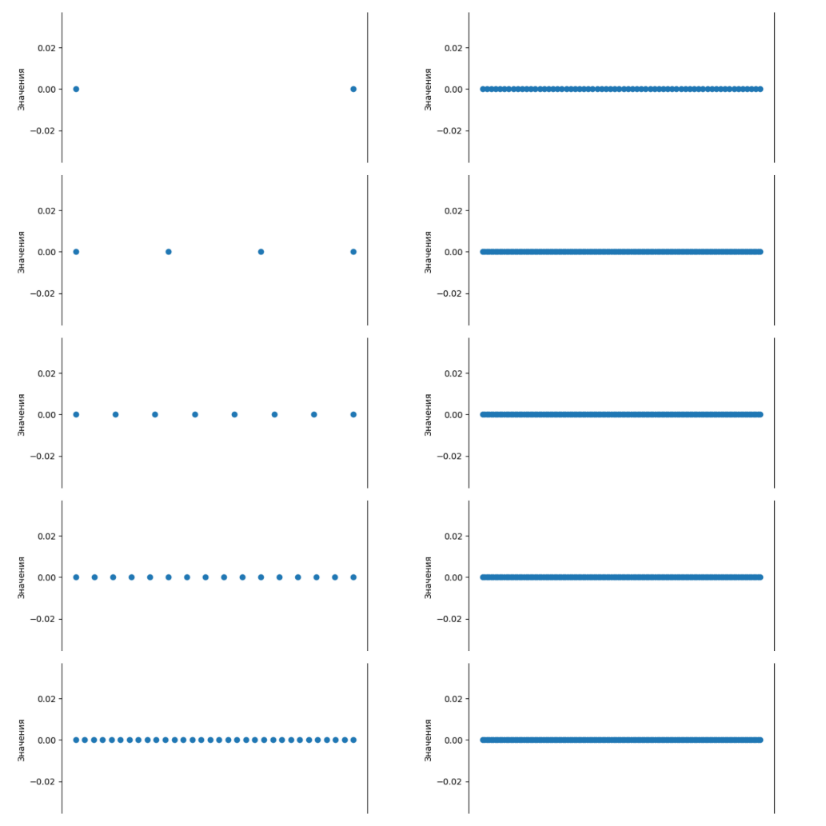
**Методы:**

* compilePatternArray(self, pattern): компилирует массив compliedPatternArray, который содержит информацию о совпадениях префиксов и суффиксов шаблона
* performKMPSearch(self, text, pattern): выполняет поиск подстроки в тексте с использованием алгоритма KMP
* visualize(indexes): визуализирует результаты поиска подстроки в виде точек на графике. Если подстрока найдена, отображает позиции, в которых она найдена, иначе выводит сообщение о том, что подстрока не найдена

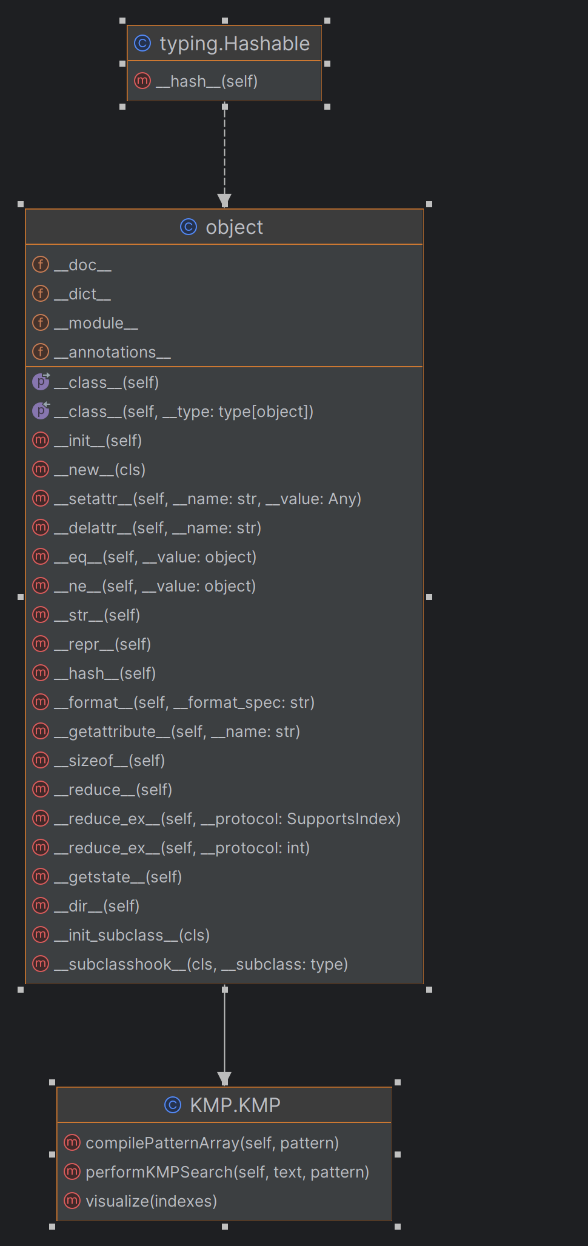
**Комментирование кода:**

# Определяем класс KMP для реализации алгоритма Кнута-Морриса-Пратта (KMP)  
class KMP:  
 # Метод для создания массива compliedPatternArray  
 def compilePatternArray(self, pattern):  
 # Вычисляем длину шаблона  
 patternLength = len(pattern)  
 # Создаем массив compliedPatternArray длиной равной длине шаблона и заполняем его нулями  
 compliedPatternArray = [0] \* patternLength  
 # Инициализируем переменную k  
 k = 0  
 # Проходим по шаблону, начиная с индекса 1  
 for i in range(1, patternLength):  
 # Пока k больше 0 и символы шаблона не совпадают  
 while k > 0 and pattern[k] != pattern[i]:  
 # Обновляем k по значению из compliedPatternArray  
 k = compliedPatternArray[k-1]  
 # Если символы шаблона совпадают  
 if pattern[k] == pattern[i]:  
 # Увеличиваем k  
 k += 1  
 # Заполняем массив compliedPatternArray  
 compliedPatternArray[i] = k  
 # Возвращаем массив compliedPatternArray  
 return compliedPatternArray  
  
 # Метод для выполнения поиска с использованием алгоритма KMP  
 def performKMPSearch(self, text, pattern):  
 # Длина текста  
 n = len(text)  
 # Длина шаблона  
 m = len(pattern)  
 # Получаем массив compliedPatternArray из метода compilePatternArray  
 compliedPatternArray = self.compilePatternArray(pattern)  
 # Список для хранения найденных индексов  
 foundIndexes = []  
 # Инициализируем переменную для индекса шаблона  
 patternIndex = 0  
 # Проходим по тексту  
 for i in range(n):  
 # Пока patternIndex больше 0 и символы не совпадают  
 while patternIndex > 0 and pattern[patternIndex] != text[i]:  
 # Обновляем patternIndex по значению из compliedPatternArray  
 patternIndex = compliedPatternArray[patternIndex - 1]  
 # Если символы совпадают  
 if pattern[patternIndex] == text[i]:  
 # Увеличиваем patternIndex  
 patternIndex += 1  
 # Если весь шаблон найден  
 if patternIndex == m:  
 # Добавляем индекс начала найденного шаблона в список  
 foundIndexes.append(i - m + 1)  
 # Обновляем patternIndex по значению из compliedPatternArray  
 patternIndex = compliedPatternArray[patternIndex - 1]  
 # Возвращаем список найденных индексов  
 return foundIndexes  
  
 # Метод для визуализации результатов поиска  
 def visualize(indexes):  
 # Если найдены индексы  
 if indexes:  
 # Выводим сообщение о найденных позициях  
 print('Подстрока найдена в позициях:', ', '.join(map(str, indexes)))  
 x = indexes  
 # Создаем данные для графика  
 y = [0] \* len(indexes)  
 # Строим точечный график для визуализации индексов  
 plt.scatter(x, y)  
 # Добавляем заголовок и метки осей, затем отображаем график  
 plt.title('Алгоритм Кнута-Морриса-Пратта (KMP)')  
 plt.xlabel('Позиции')  
 plt.ylabel('Значения')  
 plt.show()  
 # Если индексы не найдены  
 else:  
 # Выводим сообщение о том, что подстрока не найдена  
 print('Подстрока не найдена!')

**Визуальный анализ алгоритма Кнута-Морриса-Пратта:**



**UML-диаграмма класса «KMP»:**



1. **Класс «main»** - точка входа в программу

**Переменные:**

* choice: хранит выбор пользователя относительно алгоритма сортировки
* n: размер массива, вводимый пользователем
* array: массив случайных чисел, который будет сортироваться
* start\_time: время начала выполнения алгоритма сортировки
* end\_time: время окончания выполнения алгоритма сортировки
* elementToSearch: элемент, который будет искаться в массиве при использовании двоичного поиска

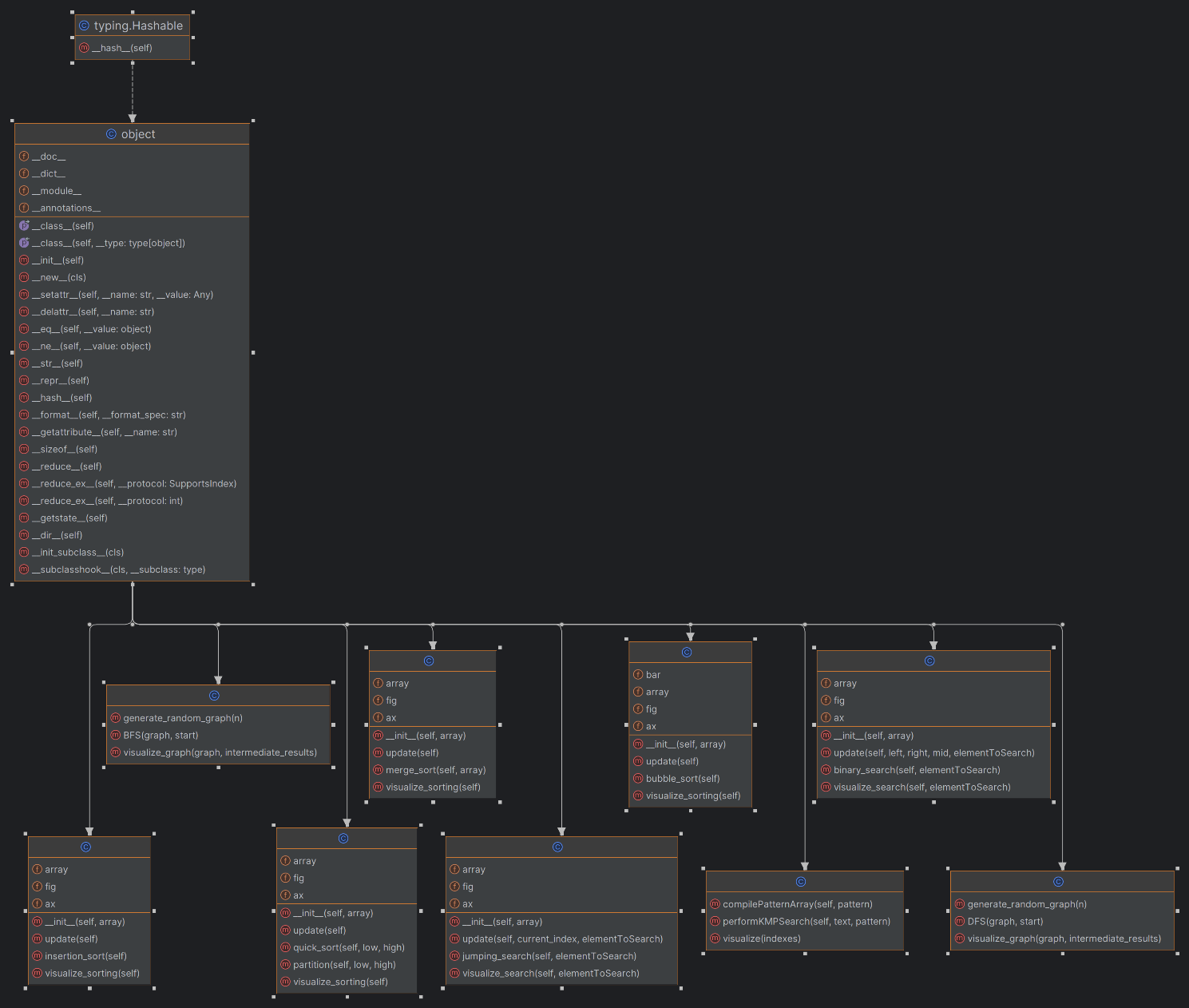
**Методы:**

* quick\_sort.quick\_sort(0, len(array) - 1): вызывает метод быстрой сортировки для объекта quick\_sort
* merge\_sort.merge\_sort(merge\_sort.array): вызывает метод сортировки слиянием для объекта merge\_sort
* bubble\_sort.bubble\_sort(): вызывает метод сортировки пузырьком для объекта bubble\_sort
* insertion\_sort.insertion\_sort(): вызывает метод сортировки вставкой для объекта insertion\_sort
* array.sort(): сортирует массив для использования двоичного поиска
* binary\_search.binary\_search(elementToSearch): вызывает метод двоичного поиска для объекта binary\_search

**Комментирование кода:**

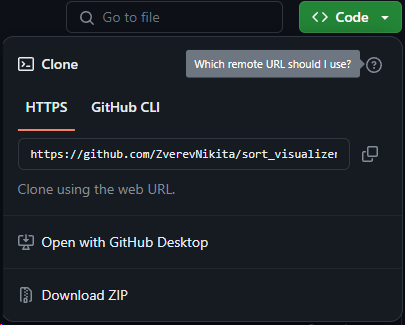
# Проверка, что имя программы - 'main'  
if name == 'main':  
 # Цикл для вывода меню выбора алгоритма сортировки  
 while True:  
 # Вывод списка алгоритмов  
 print('Выберите алгоритм сортировки:')  
 print('1. Быстрая сортировка (Quick Sort)')  
 print('2. Сортировка слиянием (Merge Sort)')  
 print('3. Сортировка пузырьком (Bubble Sort)')  
 print('4. Сортировка вставкой (Insertion Sort)')  
 print('5. Двоичный поиск (Binary Search)')  
 print('6. Поиск прыжками (Jumping Search)')  
 print('7. Сортировка в графе (BFS)')  
 print('8. Сортировка в графе (DFS)')  
 print('9. Алгоритм Кнута-Морриса-Пратта (KMP)')  
 print('0. Выход')  
 print('')  
 # Ввод выбора пользователя  
 choice = int(input('Введите номер алгоритма сортировки: '))  
 if choice == 0:  
 break  
 # Обработка выбранного алгоритма  
 elif choice == 1:  
 # Выбрана быстрая сортировка  
 print('Быстрая сортировка (Quick Sort)')  
 # Генерация массива случайных чисел  
 n = int(input('Введите размер массива: '))  
 array = [random.randint(1, 100) for \_ in range(n)]  
 print('Исходный массив:', array)  
 # Создание объекта QuickSort и запуск сортировки  
 quick\_sort = QuickSort(array)  
 start\_time = time.time()  
 quick\_sort.quick\_sort(0, len(array) - 1)  
 end\_time = time.time() - start\_time  
 print('Отсортированный массив:', quick\_sort.array)  
 print('Время выполнения программы:', end\_time, 'секунд')  
 quick\_sort.visualize\_sorting()  
 print('')  
 if input("Желаете продолжить работу? (да/нет): ").lower() != 'да':  
 break  
 # Обработка других алгоритмов сортировки аналогичным образом

**Общая UML-диаграмма классов:**



* Инструкция по установке и запуску проекта

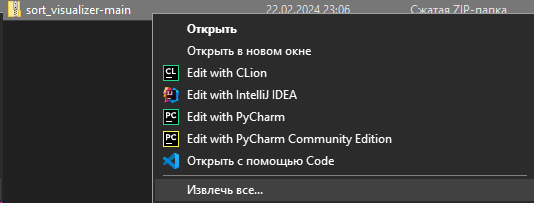
1. Необходимо перецти в репозиторий GitHub, используя данную ссылку: <https://github.com/ZverevNikita/sort_visualizer>
2. После перехода на репозиторий проекта GitHub необходимо выполнить следующие действия: Code -> Download ZIP (скачать ZIP-архив репозитория проекта GitHub)



1. После установки ZIP-архива репозитория проекта GitHub, необходимо его распаковать для дальнейшего его использования и запуска, для этого необходимо сделать следующие действия:

* Найти данный ZIP-архив репозитория проекта GitHub в «Проводник» -> «Загрузки»
* После нахождения ZIP-архив репозитория проект

GitHub необходимо нажать на ПКМ и нажать на «Извлечь все…»



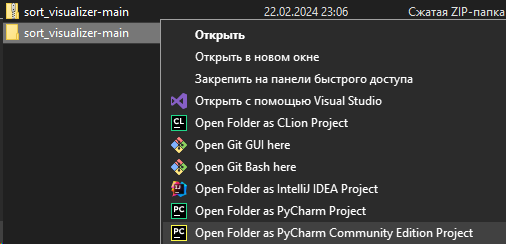
* После чего в «Проводник» -> «Загрузки» должен появиться разархивированный ZIP-архив репозитория

проекта GitHub, который представляет из себя обычную папку



* Теперь для запуска проекта из репозитория GitHub необходимо сделать несколько простых действий:

1. Нажать ПКМ на разархивированный ZIP-архив репозитория проекта GitHub и нажать на «Открыть с помощью <PyCharm> (В зависимости от того, какая среда разработки предварительно установлена)



1. После перехода в среду разработки необходимо запустить проект из репозитория GitHub, нажав на кнопку «Run», после чего проект из репозитория GitHub должен запуститься (в зависимости от среды разработки иногда требуется установить некоторые инструменты для запуска проекта из репозитория GitHub)