**«Алгоритмы и структуры данных»**

**Калугина Марина Алексеевна**

**Выполнили студенты гр. 353502:**

**Кашко Анастасия,**

**Згирская Дарья**

**СОДЕРЖАНИЕ**

[Тема 1. Бинарные поисковые деревья (задача 3) 3](#_Toc199069996)

[Тема 2. Разработка эффективных алгоритмов (задача 1, задача 9) 4](#_Toc199069997)

[Тема 3. Структуры данных (задача 4) 5](#_Toc199069998)

[Тема 4. Графы (задача 9) 6](#_Toc199069999)

# **Тема 1. Бинарные поисковые деревья (задача 3)**

# **Тема 2. Разработка эффективных алгоритмов (задача 1, задача 9)**

# **Тема 3. Структуры данных (задача 4)**

**Задача:** в массиве A размера n за одно обращение к каждому элементу массива необходимо каждый элемент заменить ближайшим большим, следующим за ним. Если такого элемента нет, то необходимо заменить его нулём. Можно использовать дополнительную память.

|  |  |
| --- | --- |
| **Входные данные** | **Выходные данные** |
| 1) Размер массива n  2) Массив A из n целых чисел: | 1) Массив B из n целых чисел, в котором: равен ближайшему большему элементы справа от если такого элемента нет – |

**Алгоритмы решения**

**1 реализация («плохая»):**

def solution(arr: list):

def find\_near\_max(first: int, arr: list):

for i in range(first + 1, len(arr)):

if arr[i] > arr[first]:

return arr[i]

return 0

for i in range(len(arr) - 1):

arr[i] = find\_near\_max(i, arr)

arr[-1] = 0

return arr

**Пошаговое описание алгоритма:** проходимся по массиву от начала до предпоследнего элемента, для каждого из которых во вложенной функции ищем в исходном массиве первый элемент справа, который больше текущего, если такой найден, возвращаем его, иначе – 0, присваиваем найденное значение текущему элементу, последний элемент всегда равен 0.

**Принцип работы на примере:** [3, 1, 4, 2]

|  |  |  |
| --- | --- | --- |
| **Шаг** | **Массив (до)** | **Массив (после)** |
| 0 | [3, 1, 4, 2] | [4, 1, 4, 2] |
| 1 | [4, 1, 4, 2] | [4, 4, 4, 2] |
| 2 | [4, 4, 4, 2] | [4, 4, 0, 2] |
| 3 | [4, 4, 0, 2] | [4, 4, 0, 0] |

Исследуем сложность алгоритма, используя рекуррентные уравнения: T(n) – количество операций для массива длины n. Для каждого i-го элемента вызывается функция поиска первого большего, метод проходит от i + 1 до n – 1, то есть делает n – i – 1 сравнений. Функция вызывается n – 1 раз для каждого i = 0, 1, … n – 2. T(1) = 0, поскольку в массиве из одного элемента нет соседних для сравнения.

Преобразуем сумму в рекуррентное уравнение. Выразим T(n) через T(n – 1). T(n – 1) – это результат той же суммы, но для массива длины n - 1:

Решим, полученное рекуррентное уравнение методом итераций:

Пусть k = n – 1:

Пусть j = n – i, i = 1 🡪 j = n – 1, i = n – 1 🡪 j = 1:

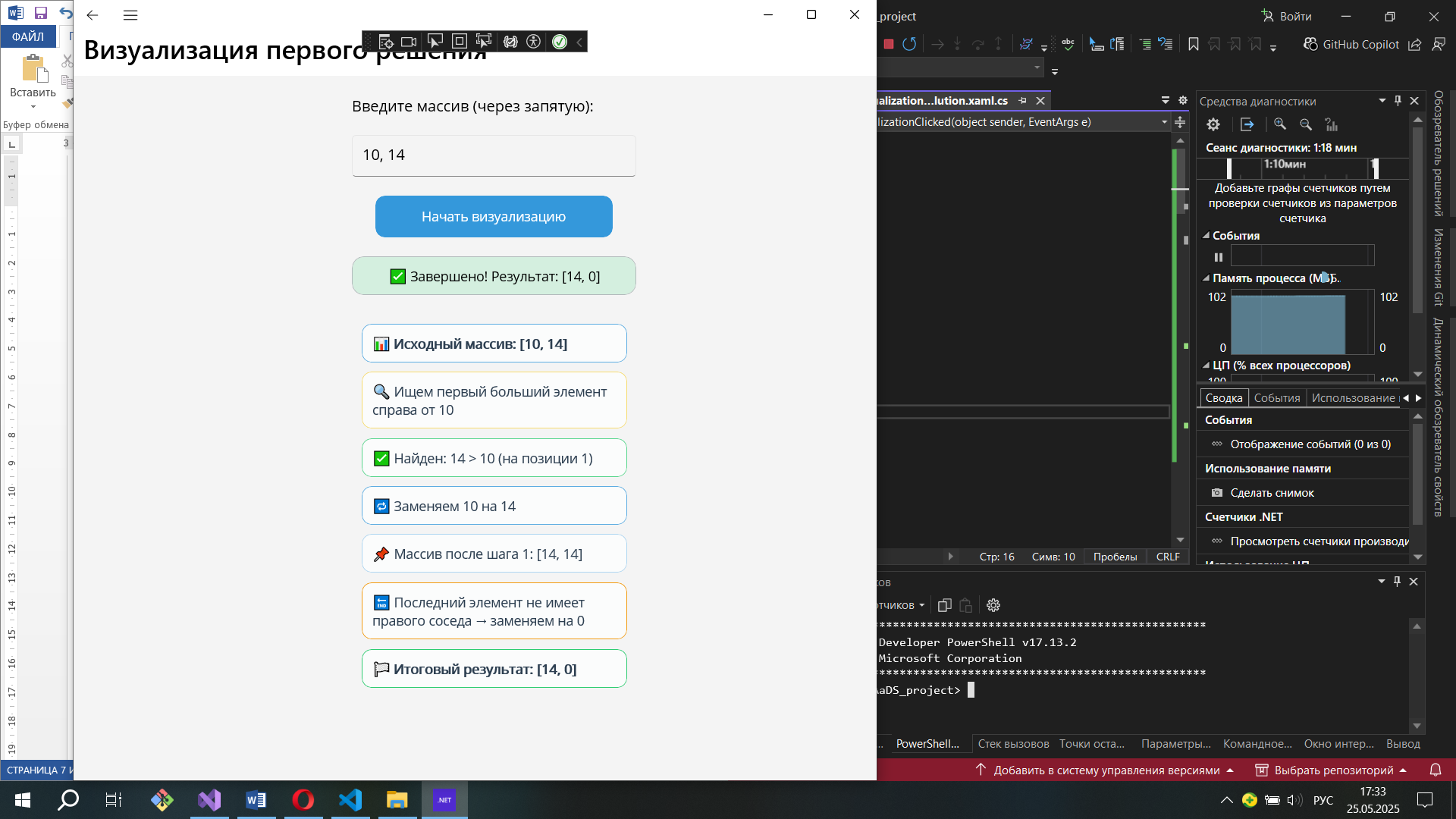
**Сложность алгоритма (1 реализация):** – для каждого элемента делается линейный проход по оставшимся.

**Дополнительная память:** – при выполнении не используются новые структуры данных, а также ресурсоёмкие механизмы.

**Преимущества алгоритма (1 реализация):** простой в реализации, не требует дополнительных структур данных.

**Недостатки алгоритма (1 реализация):** медленный на больших входных данных из-за квадратичной сложности. Данная реализация является «плохой», так как не учитывает условие «за одно обращение к каждому элементу.

**Визуализация:**



**2 реализация:**

def solution\_2(arr):

n = len(arr)

result = [0] \* n

stack = []

for i in range(n - 1, -1, -1):

while stack and stack[-1] <= arr[i]:

stack.pop()

result[i] = stack[-1] if stack else 0

stack.append(arr[i])

return result

**Пошаговое описание алгоритма:** инициализируем пустой стек и результирующий массив такого же размера, как исходный, заполняем нулями. Проходимся по массиву справа налево (от конца к началу). **Сначала обрабатываем последний элемент** — у него справа ничего нет, поэтому для него ответ — 0, и он (элемент) кладётся в стек. **Переходим к предпоследнему элементу**, смотрим на стек — в нём лежит последний элемент массива. Если последний элемент больше текущего, значит он и есть "следующий больший элемент справа" для текущего. Если последний элемент меньше или равен текущему — значит он нам не подходит, мы его убираем из стека (он "перекрывается" текущим элементом, потому что текущий больше). **Таким образом, в стеке всегда лежат элементы, которые «могут быть следующими большими» для элементов слева**. После обработки всех элементов в результирующем массиве окажутся нужные значения.

**Принцип работы на примере:** [3, 1, 4, 2]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Шаг** | **Массив** | **Состояние стека (до)** | **Состояние стека (после)** | **Результат** |
| 0 | [3, 1, 4, ***2***] | [] | [2] | [0] |
| 1 | [3, 1, ***4***, 2] | [2] | [4] | [0, 0] |
| 2 | [3, ***1***, 4, 2] | [4] | [1, 4] | [4, 0, 0] |
| 3 | [***3***, 1, 4, 2] | [1, 4] | [3, 4] | [4, 4, 0, 0] |

Исследуем сложность алгоритма, используя рекуррентные уравнения: T(n) – время работы алгоритма при массиве длины n (общее число операций). В основном цикле n итераций (проход от n – 1 до 0), на каждой итерации делается постоянная работа (проверка условия, добавления в стек, присваивание результата), может быть вызван цикл while, который делает несколько pop (достаёт элементы из стека). Каждый элемент добавляется в стек ровно один раз, а удалён может быть не более одного раза. Таким образом, за весь алгоритм pop() не больше n раз.

где P(n) – стоимость pop() на шаге n, О(1) – постоянная работа, T(n – 1) – время обработки массива длины n – 1. Решим рекуррентное уравнение методом итераций:

(авнение методом итерацийвкитодом подстановкимассиве окажутся нужные значения. такого же размера как

Пусть k = n:

Так как

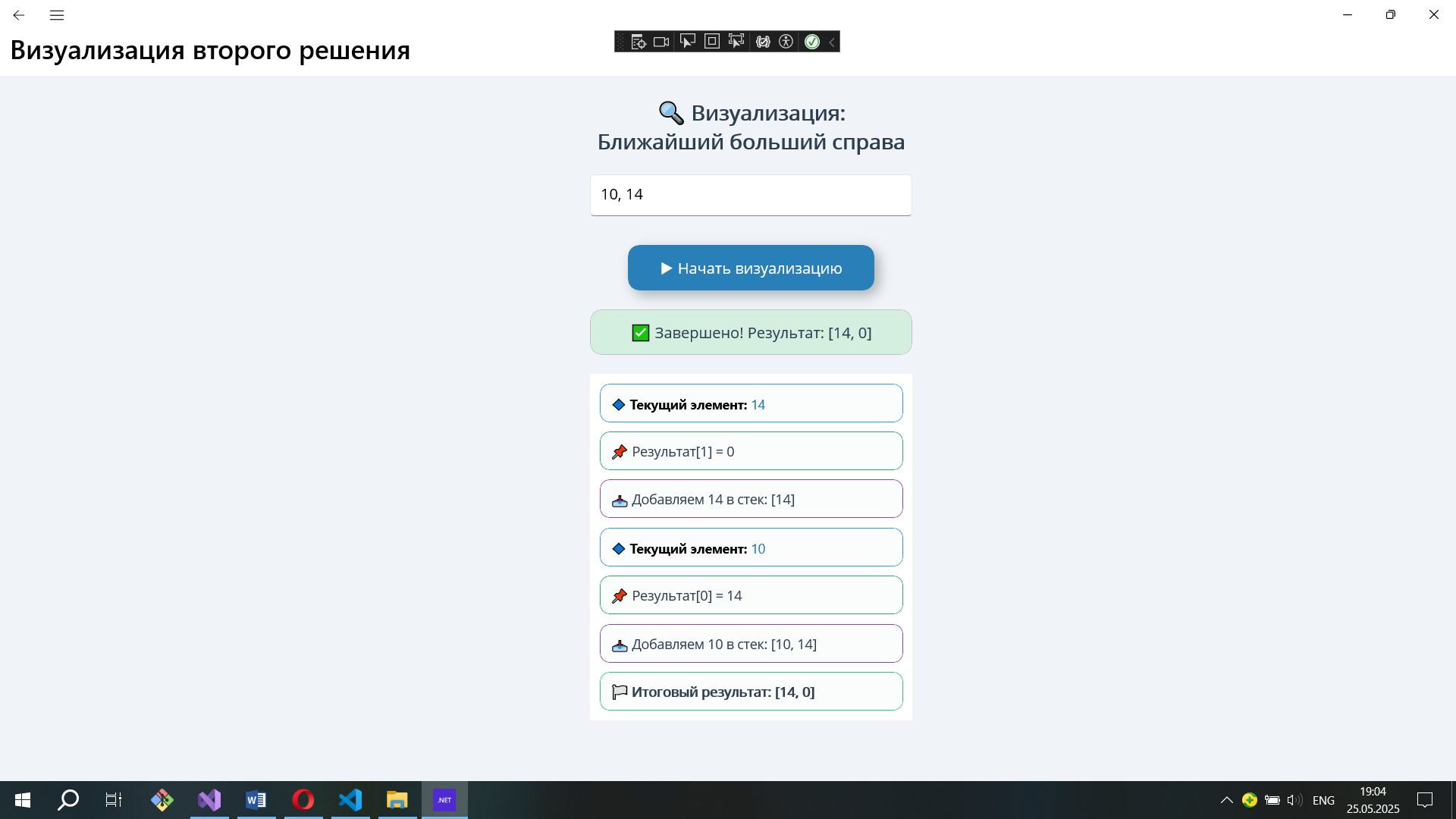
**Сложность алгоритма (2 реализация):** – линейная, каждый элемент массива добавляется в стек ровно один раз и удаляется из него не более одного раза. Благодаря этому, несмотря на вложенный цикл, суммарное число операций pop не превышает n.

**Дополнительная память:** O(n) – стек может содержать до n элементов в худшем случае.

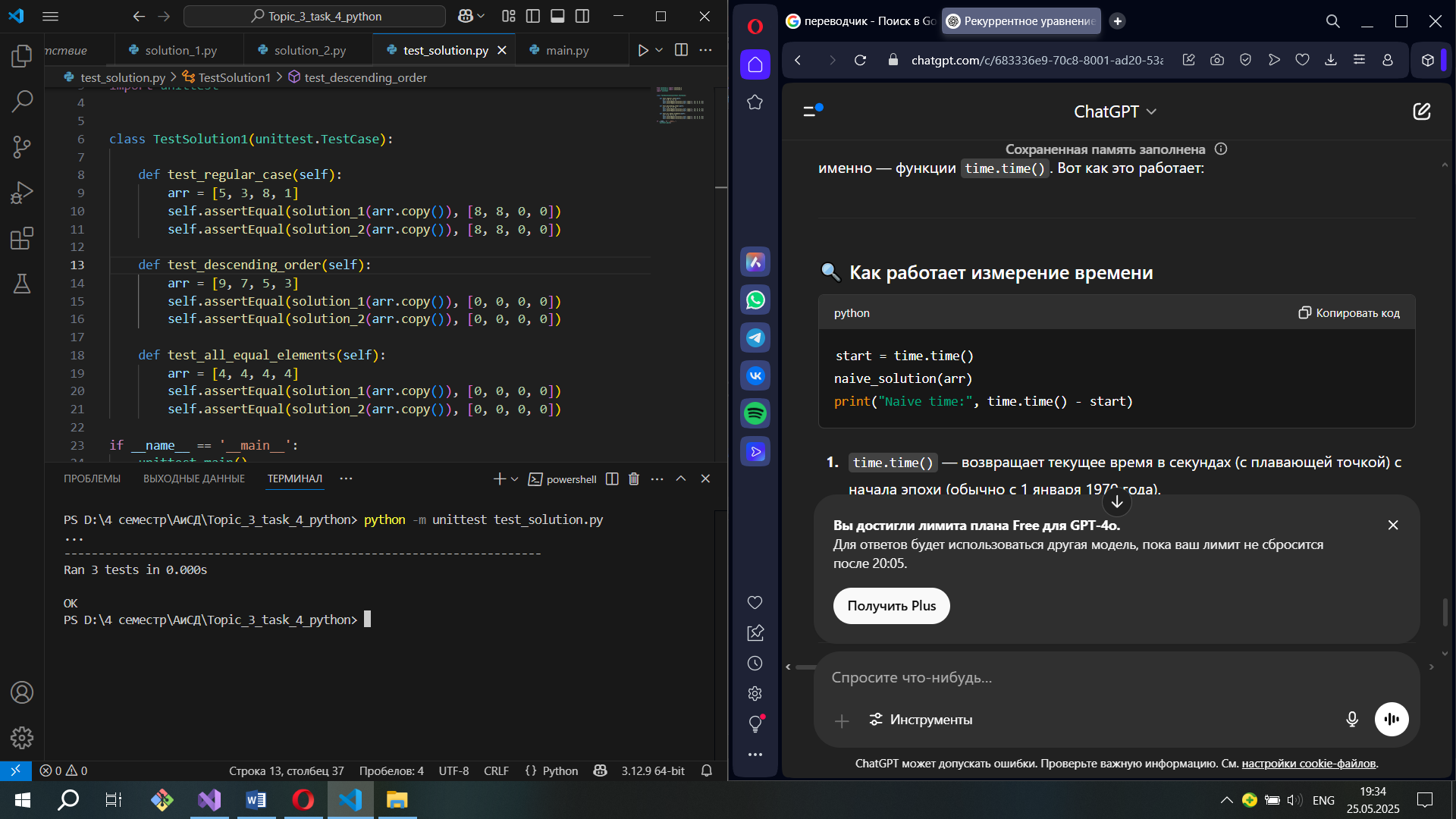
**Преимущества алгоритма (2 реализация):** эффективный, линейная сложность на входе любого размера, не требует многократных проходов по массиву.

**Недостатки алгоритма (2 реализация):** требуется дополнительная память под стек, что может быть критично при ограниченных ресурсах, алгоритм более сложен для понимания и реализации по сравнению с простым двойным циклом, не всегда подходит для задач, где важно минимальное использование памяти.

**Визуализация:**



Две реализации были протестированы вручную и с помощью юнит-тестов:



**Сравнительная таблица 2 реализаций:**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Наивная реализация (двойной цикл)** | **Реализация со стеком** |
| **Временная сложность** |  |  |
| **Дополнительная память** | O(1) | O(n) (для стека) |
| **Сложность реализации** | Простая, легко понять и реализовать | Более сложная, требует понимания работы стека |
| **Преимущества** | Минимум дополнительной памяти, простота | Быстрая работа даже на больших данных |
| **Недостатки** | Медленная на больших данных, квадратичная сложность | Требует дополнительной памяти под стек, сложнее для понимания |

Данные реализации были написаны не только на языке программирования python, но и на C# (для визуализации). Снизу приведён сравнительный анализ 2 реализации на python и C#. Выбор пал на 2 реализацию, так как она соответствует всем условием и является более «хорошей». Массив для анализа: от 1000 до 0.

|  |  |  |
| --- | --- | --- |
| **Критерий** | **C#** | **Python** |
| **Время выполнения (сек)** | 0,00167 (быстрее, JIT-компиляция) | 0.00245547 (медленнее, интерпретатор) |
| **Длина кода** | Более многословный | Короткий, лаконичный |

# **Тема 4. Графы (задача 9)**