**«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)**

# **Задача 1**

**Задача:** покупатель имеет n купюр достоинством: и продавец имеет m купюр достоинством: Необходимо найти максимальную стоимость товара p, которую покупатель не может купить, потому что нет возможности точно рассчитаться за этот товар с продавцом, хотя денег на покупку у него достаточно.

|  |  |
| --- | --- |
| **Входные данные** | **Выходные данные** |
| 1) Количество купюр покупателя: целое число n  2) Массив A из n целых чисел: – номиналы купюр покупателя  3) Количество купюр продавца: целое число m  4) Массив B из m целых чисел: – номиналы купюр продавца | 1) Число p – максимальная стоимость товара, которую покупатель не может оплатить точно, несмотря на то, что у него достаточно денег (целое число или None, если покупатель может рассчитаться точно за любую сумму до максимума своих средств) |

**Алгоритмы решения**

**1 реализация (неоптимизированная):**

def get\_possible\_sums(coins: list[int]) -> set[int]:

max\_sum = sum(coins)

dp = [False] \* (max\_sum + 1)

dp[0] = True

for coin in coins:

for j in range(max\_sum, coin - 1, -1):

if dp[j - coin]:

dp[j] = True

return {s for s in range(max\_sum + 1) if dp[s]}

def count\_max\_cost\_1(buyer: list[int], seller: list[int]) -> int | None:

buyer\_sums = get\_possible\_sums(buyer)

seller\_sums = get\_possible\_sums(seller)

possible\_p = set()

for b in buyer\_sums:

for s in seller\_sums:

p = b - s

if p > 0:

possible\_p.add(p)

max\_affordable = max(buyer\_sums)

for p in range(max\_affordable, 0, -1):

if p not in possible\_p:

return p

if max\_affordable == 0:

return None

if min(buyer) <= min(seller):

return None

return min(buyer) - min(seller) – 1

**Пошаговое описание алгоритма:** пользователь вводит все необходимые данные, после чего для каждого множества купюр (покупателя и продавца) вычисляется множество всех возможных сумм, которые можно составить, используя любые комбинации имеющихся купюр. Следующий шаг заключается в вычислении всех возможных точных стоимостей покупки (для каждой суммы из сумм покупателя и каждой суммы из сумм продавца считается разность, если она равна нулю, то стоимость считается точно оплачиваемой). Заключительный шаг несёт в себе логику поиска максимальной стоимости, которую нельзя оплатить (находится максимальная стоимость, которую покупатель может собрать, а потом они сравниваются со стоимостями, найденными на предыдущем шаге), если у покупателя нет денег (сумма 0), возвращается None, если у покупателя есть деньги, но все стоимости возможны, и минимальная купюра покупателя меньше либо равна минимальной купюре продавца, возвращается None.

**Принцип работы на примере:** купюры покупателя – [1, 3, 4], купюры продавца – [1, 2].

1) Подсчёт возможных сумм у покупателя

сумма всех купюр равна 1 + 3 + 4 = 8. Инициализируем массив длины 9 (от 0 до 8): [True, False, … False]. 0-ой элемент всегда True, так как сумму 0 всегда можно собрать.

2) Перебираем купюры по одной:

Купюра 1:

Проходим от j = 8 до j = 1:

if dp[j - 1]:

dp[j] = True

при j = 1: dp[1 – 1] = dp[0] = True 🡪 dp[1] = True, остальные j пока невозможны.

Текущий массив: [True, True, False, False, False, False, False, False, False].

Купюра 3:

Проходим от j = 8 до j = 3:

if dp[j - 3]:

dp[j] = True

при j = 4: dp[4 – 3] = dp[1] = True 🡪 dp[4] = True, при j = 3: dp[3 – 3] = dp[0] = True 🡪 dp[3] = True, остальные j пока невозможны.

Текущий массив: [True, True, False, True, True, False, False, False, False].

Купюра 4:

Проходим от j = 8 до j = 4:

if dp[j - 4]:

dp[j] = True

при j = 8: dp[8 – 4] = dp[4] = True 🡪 dp[8] = True, при j = 7: dp[7 – 4] = dp[3] = True 🡪 dp[7] = True, при j = 5: dp[5 – 4] = dp[1] = True 🡪 dp[5] = True, при j = 4: dp[4 – 4] = dp[0] = True 🡪 dp[4] = True, остальные j пока невозможны.

Текущий массив: [True, True, False, True, True, True, False, True, True].

**Итог: можно собрать суммы покупателя [0, 1, 3, 4, 5, 7, 8].**

3) Аналогично считаем для продавца получаем: **суммы покупателя [0, 1, 2, 3].**

4) Вычисляем все возможные стоимости покупки:

|  |  |  |
| --- | --- | --- |
| **Суммы покупателя** | **Суммы продавца («сдача, которую он может дать»)** | **Возможные покупки: сумма покупателя – сумма продавца** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 2 | -1 |
| 1 | 3 | -2 |
| 3 | 0 | 3 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |
| 3 | 3 | 0 |
| 4 | 0 | 4 |
| 4 | 1 | 3 |
| 4 | 2 | 2 |
| 4 | 3 | 1 |
| 5 | 0 | 5 |
| 5 | 1 | 4 |
| 5 | 2 | 3 |
| 5 | 3 | 2 |
| 7 | 0 | 7 |
| 7 | 1 | 6 |
| 7 | 2 | 5 |
| 7 | 3 | 4 |
| 8 | 0 | 8 |
| 8 | 1 | 7 |
| 8 | 2 | 6 |
| 8 | 3 | 5 |

Оставляем только положительные числа, получаем **итоговый список возможных покупок: [1, 2, 3, 4, 5, 6, 7, 8].**

5) Ищем наибольшую стоимость покупки, которую не сможет оплатить покупатель: берём максимум который сможет собрать покупатель и начинаем сравнивать сверху вниз по всем возможным значениям цены товара. В примере все значения цены покупатель оплатить сможет. Максиму покупателя 8.

|  |  |
| --- | --- |
| **Стоимость товара** | **Сможет оплатить** |
| 8 | Да |
| 7 | Да |
| 6 | Да |
| 5 | Да |
| 4 | Да |
| 3 | Да |
| 2 | Да |
| 1 | Да |

Исследуем сложность алгоритма: n – количество купюр у покупателя, m – количество купюр у продавца, – сумма всех купюр покупателя, – сумма всех купюр продавца. Сначала оценим функцию, которая считает возможные суммы покупателя и продавца. Она в себе содержит вложенные циклы: внешний цикл по n купюрам, внутренний по сумме от суммы всех купюр до текущей купюры. Общая сложность O(n \* ) + O(m \* ). Теперь оценим функцию, которая несёт в себе основную логику. Первый цикл в данной процедуре: вычисление всех возможных покупок. Берём худший случай, когда все суммы достижимы, тогда цикл даст сложность O(). Второй (последний цикл) в основной процедуре: поиск максимальной стоимости покупки, которую покупатель оплатить не сможет. В худшем случае (все сможет), сложность будет O(). Также есть дополнительный цикл для «особого» случая (у продавца нет купюр), он также в худшем случае даст сложность O(). В итоге получаем сложность O(n \* ) + O(m \* ) + O().

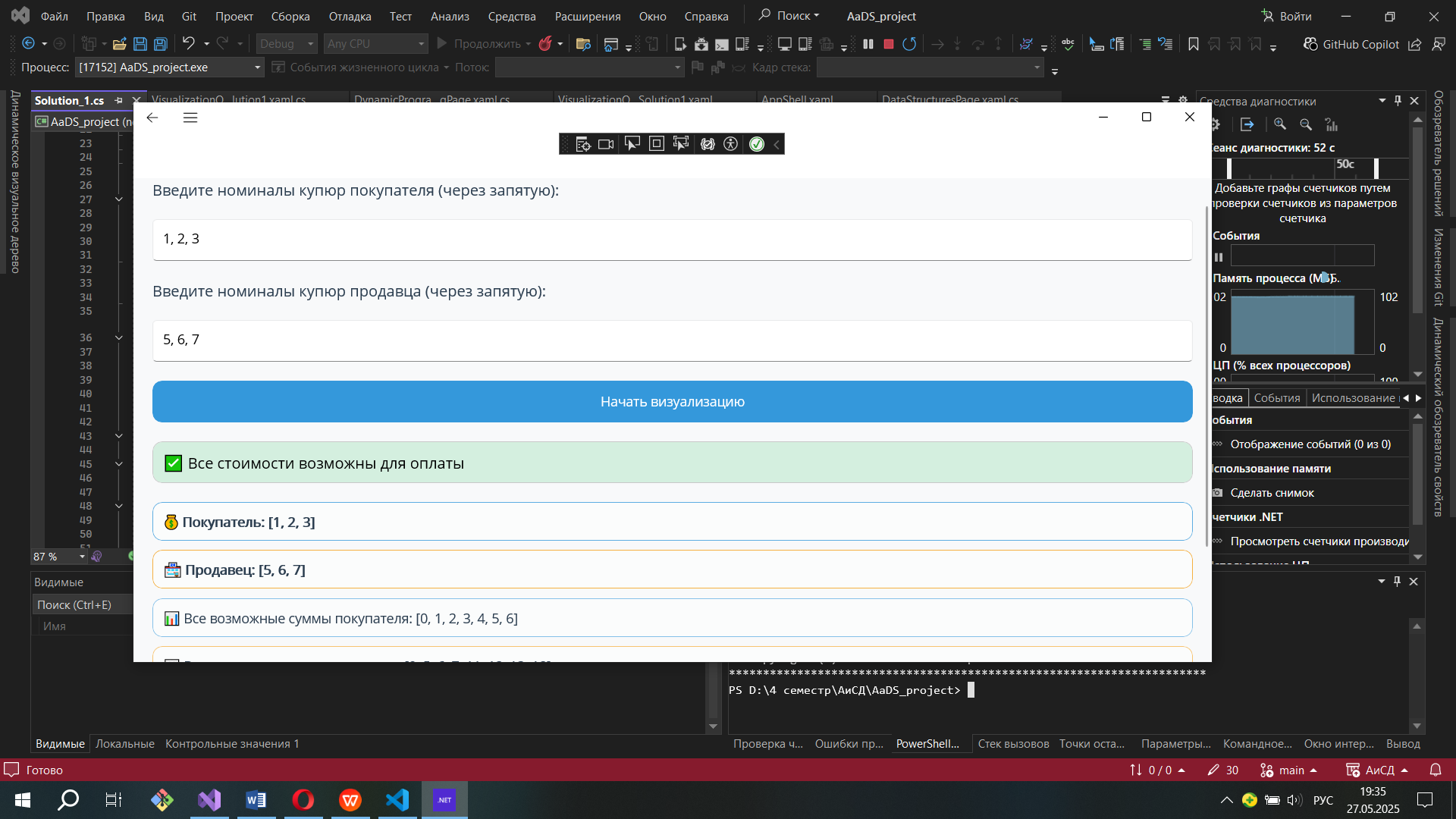
**Сложность алгоритма (1 реализация):**  Основной вклад в сложность вносит произведение ​, возникающее при переборе всех возможных разностей между достижимыми суммами покупателя и продавца.

**Дополнительная память:** – используется массив булевых значений для подсчёта возможных сумм покупателя и продавца, а также множества для хранения этих сумм.

**Преимущества алгоритма (1 реализация):** гарантирует корректный результат во всех случаях, подходит для задач, где важно учесть все возможные комбинации сумм, использует эффективную технику динамического программирования.

**Недостатки алгоритма (1 реализация):** эффективность зависит от суммы купюр, а не только их количества, может работать медленно и потреблять много памяти при больших значениях, в наихудшем случае сложность становится квадратичной по суммам, что критично для больших входных данных.

**Визуализация:**



**Улучшим реализацию (2 реализация):**

def get\_possible\_sums(coins: list[int]) -> set[int]:

possible\_sums = {0}

for coin in coins:

new\_sums = set()

for s in possible\_sums:

new\_sums.add(s + coin)

possible\_sums.update(new\_sums)

return possible\_sums

def count\_max\_cost\_2(buyer: list[int], seller: list[int]) -> int | None:

buyer\_sums = get\_possible\_sums(buyer)

if not buyer\_sums or max(buyer\_sums) == 0:

print("У покупателя нет денег!")

return None

seller\_sums = get\_possible\_sums(seller)

max\_affordable = max(buyer\_sums)

buyer\_sums\_set = buyer\_sums

if not seller\_sums:

for p in range(max\_affordable - 1, -1, -1):

if p not in buyer\_sums\_set:

return p

print("Покупатель может точно оплатить любую стоимость до своей максимальной суммы")

return None

for p in range(max\_affordable, 0, -1):

found = False

for s in seller\_sums:

if (p + s) in buyer\_sums\_set:

found = True

break

if not found:

return p

print("Покупатель может точно оплатить любую стоимость до своей максимальной суммы")

return None

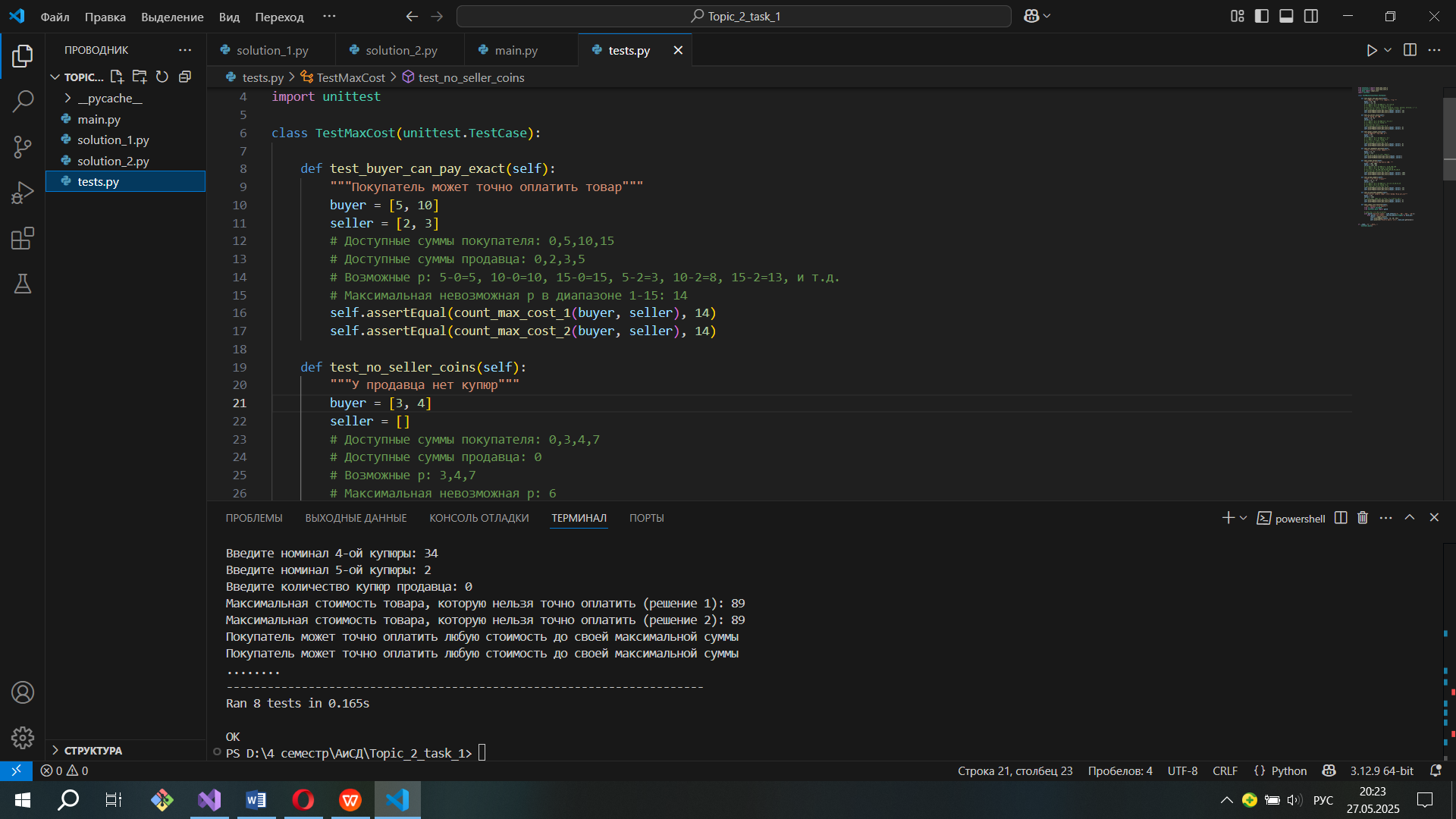
**В чём отличие данной реализации и чем она лучше?**

Первая версия создаёт дополнительное множество – все возможные суммы, которые могут быть разницей между суммой покупателя и суммой продавца. Во второй версии таких лишних множеств нет – она сразу проверяет, возможна ли покупка с учётом сдачи. Это снижает расход оперативной памяти, особенно при больших числах. Если у продавца вообще нет купюр (или у покупателя), вторая версия чётко и просто обрабатывает этот случай. В первой версии в таких ситуациях логика становится с множеством дополнительных проверок и условий.

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Первая версия** | **Вторая версия** |
| **Понятность кода** | Сложнее понять из-за множества вложенных структур | Прямолинейная логика, легче читается и поддерживается |
| **Память** | Использует дополнительное множество possible\_p | Использует только необходимые множества |
| **Обработка случая без продавца** | Обрабатывается, но с множеством условий | Обрабатывается логично и просто |
| **Сложность в худшем случае** | O(n \* S₁) + O(m \* S₂) + O(S₁ \* S₂) | O(n \* S₁) + O(m \* S₂) + O(S₁ \* S₂) |
| **Проверка невозможной оплаты** | Через сравнение p not in possible\_p | Через перебор сдач продавца и проверку p + s ∈ buyer\_sums |
| **Гибкость и расширяемость** | Менее гибкая: логика жёстко завязана на вычисление разностей | Более гибкая: можно адаптировать под другие задачи |
| **Крайние случаи (нет денег)** | Обрабатываются, но с дополнительными проверками | Обрабатываются естественно и без лишнего кода |

# 

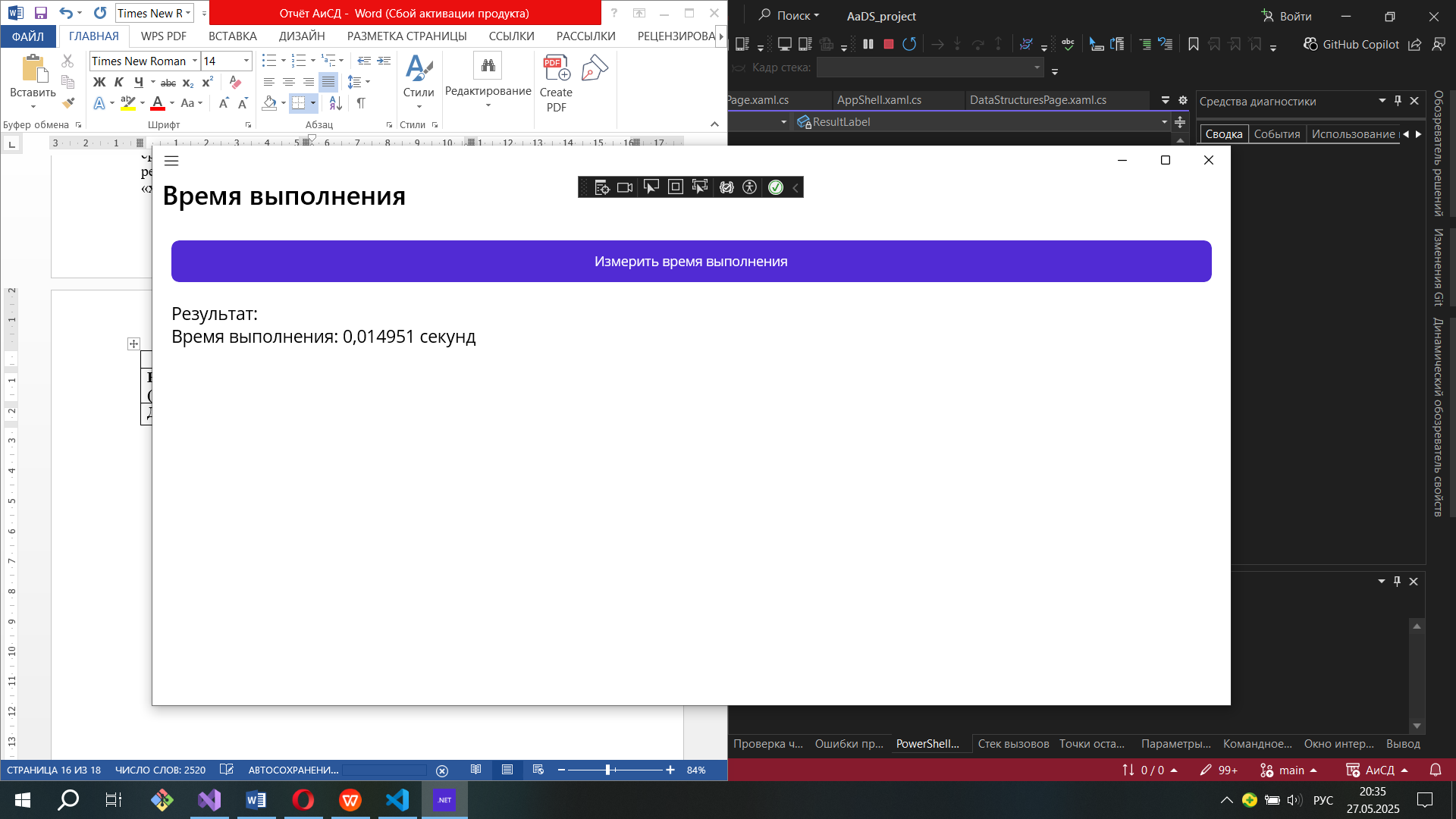
Две реализации были с помощью юнит-тестов:



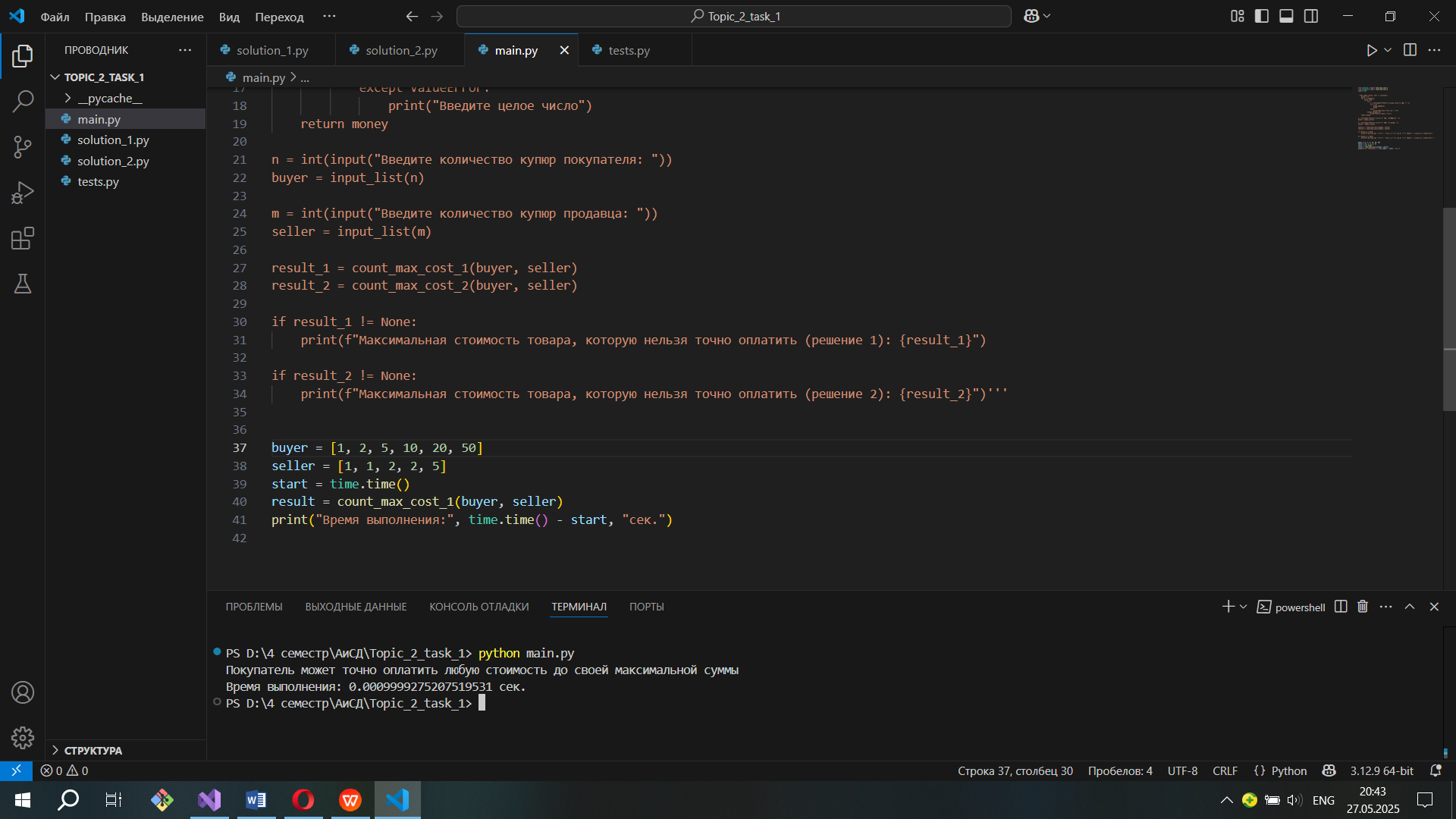
Первая реализации была написаны не только на языке программирования python, но и на C# (для визуализации). Снизу приведён сравнительный анализ времени выполнения реализации на python и C#.

|  |  |  |
| --- | --- | --- |
| **Критерий** | **C#** | **Python** |
| **Время выполнения (сек)** | 0,014951 | 0.00099993 |
| **Длина кода** | Более многословный | Короткий, лаконичный |

C#



Python



# **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:

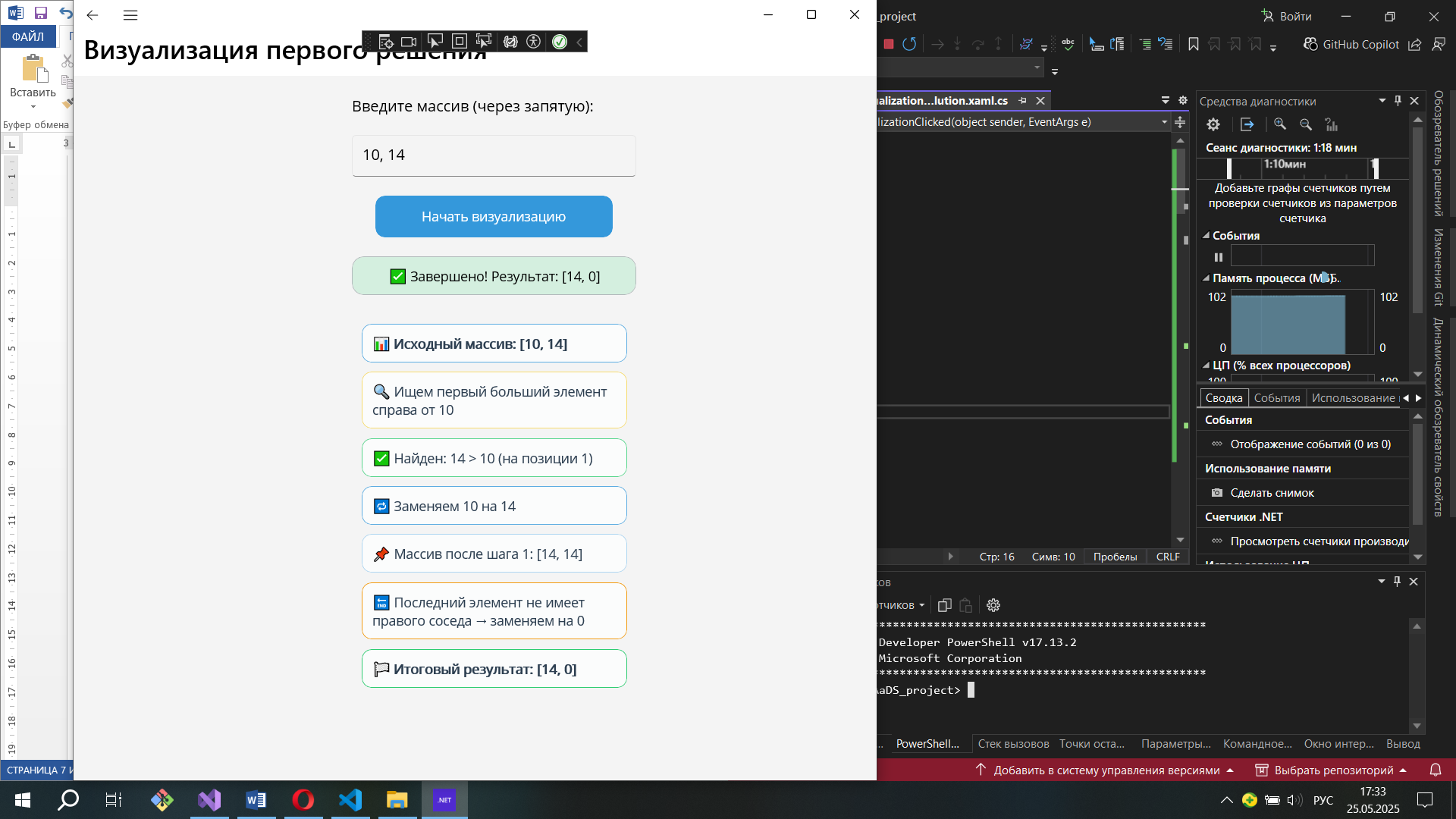
**Сложность алгоритма (1 реализация):** – для каждого элемента делается линейный проход по оставшимся.

**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

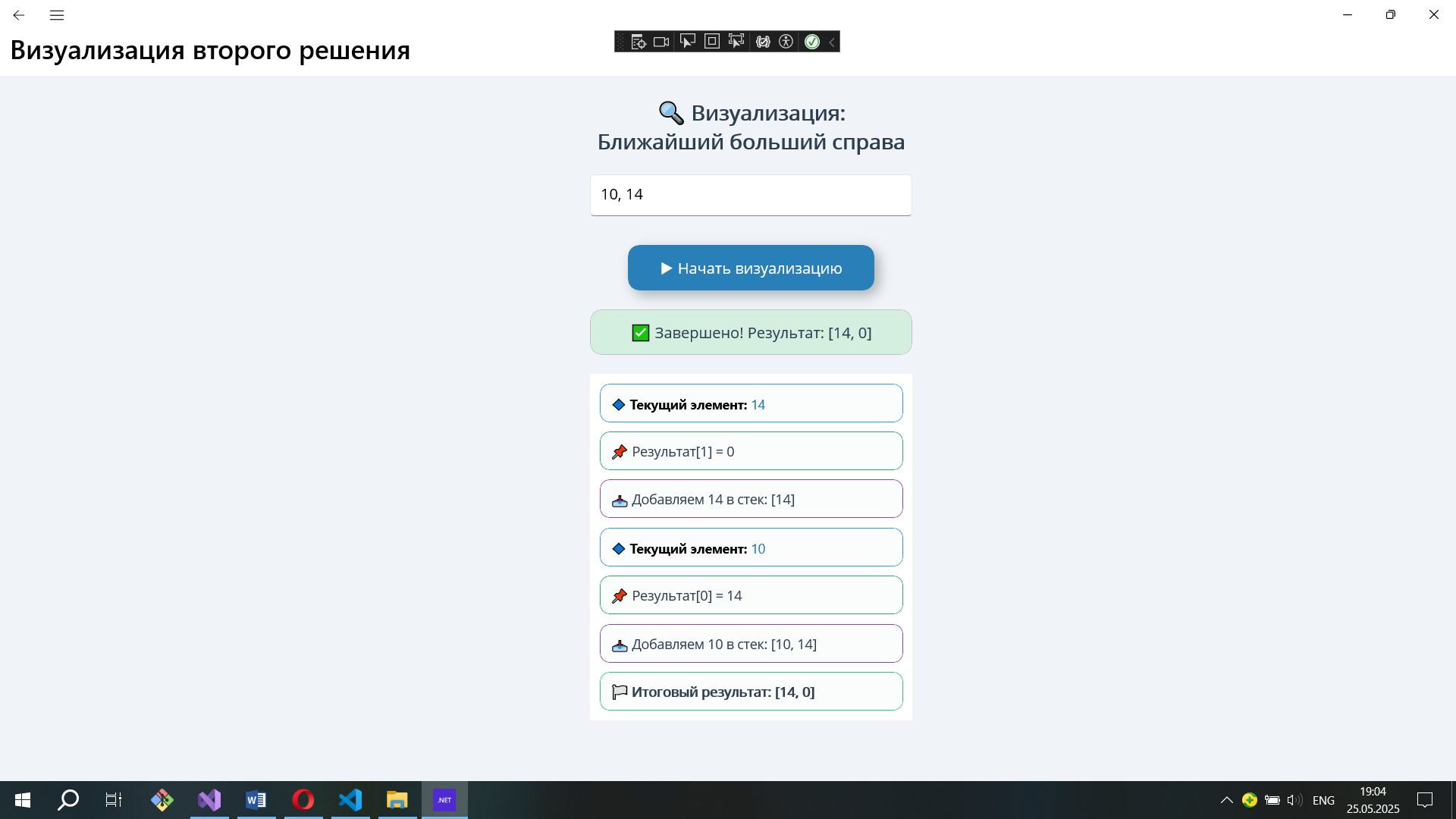
**Сложность алгоритма (2 реализация):** – линейная, каждый элемент массива добавляется в стек ровно один раз и удаляется из него не более одного раза. Благодаря этому, несмотря на вложенный цикл, суммарное число операций pop не превышает n.

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

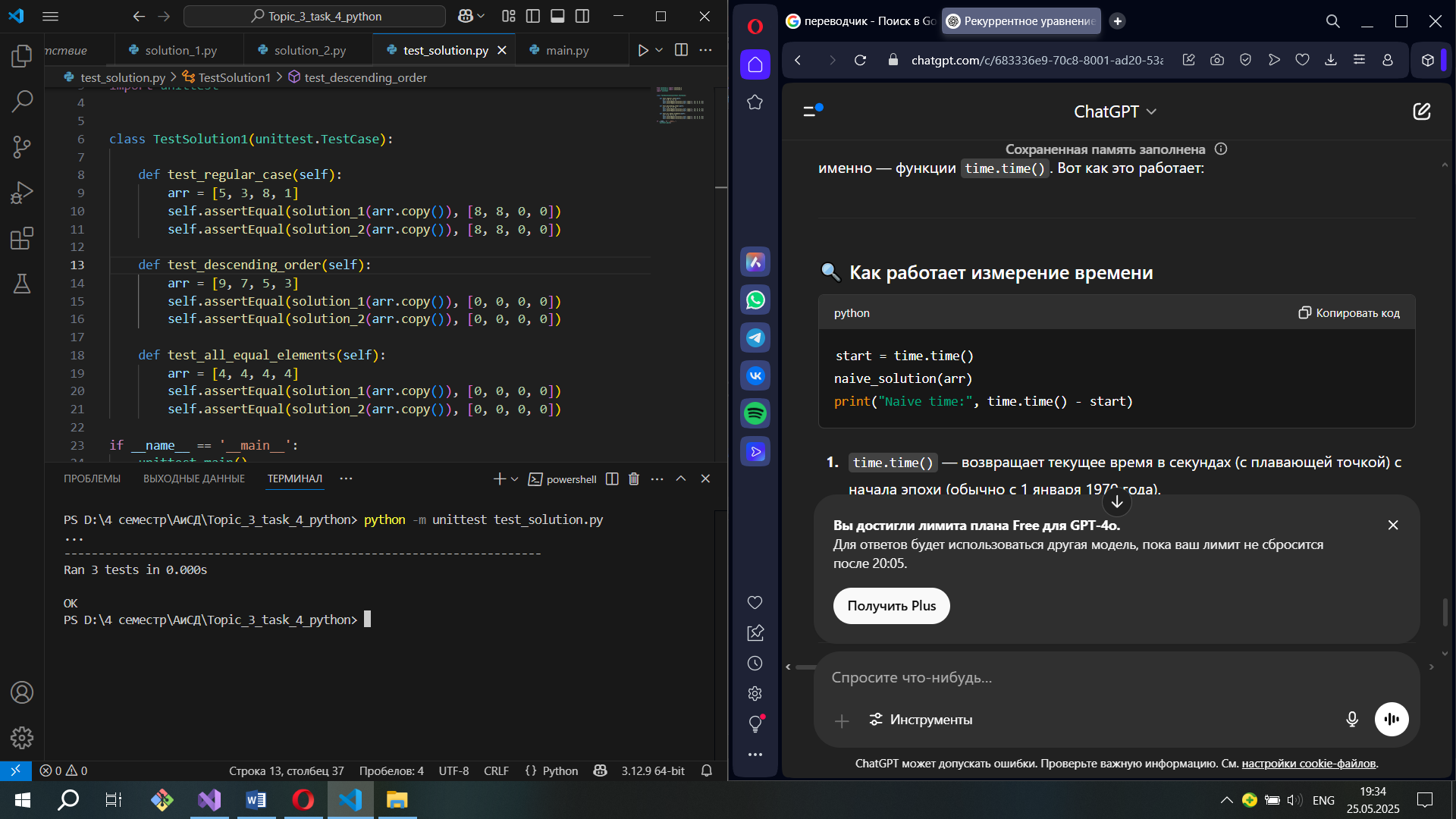
**Преимущества алгоритма (2 реализация):** эффективный, линейная сложность на входе любого размера, не требует многократных проходов по массиву.

**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:



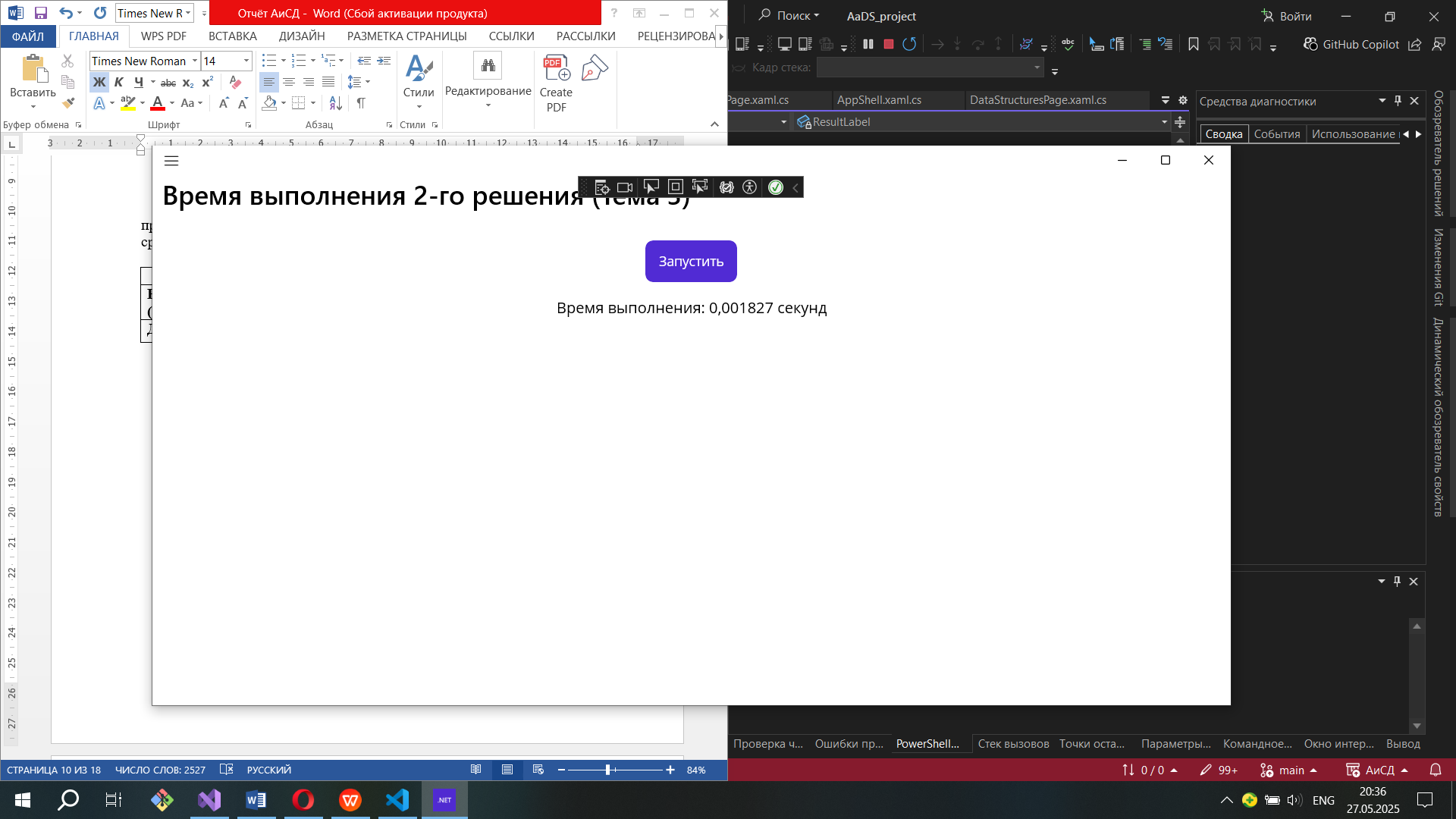
**Сравнительная таблица 2 реализаций:**

|  |  |  |
| --- | --- | --- |
| **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 |

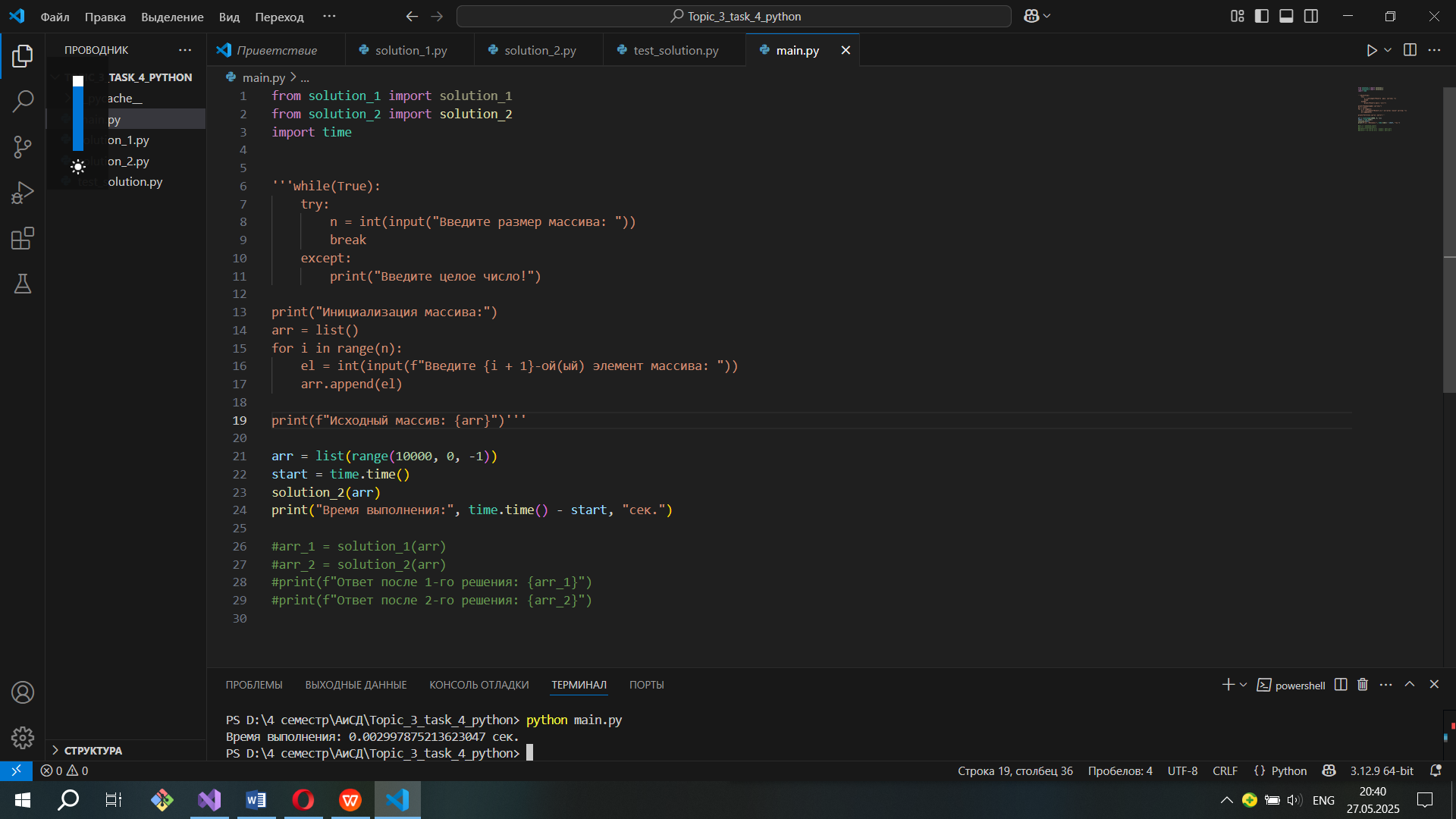
Данные реализации были написаны не только на языке программирования python, но и на C# (для визуализации). Снизу приведён сравнительный анализ 2 реализации на python и C#. Выбор пал на 2 реализацию, так как она соответствует всем условием и является более «хорошей». Массив для анализа: от 1000 до 0.

|  |  |  |
| --- | --- | --- |
| **Критерий** | **C#** | **Python** |
| **Время выполнения (сек)** | 0,001827 (быстрее, JIT-компиляция) | 0.00299788 (медленнее, интерпретатор) |
| **Длина кода** | Более многословный | Короткий, лаконичный |

C#



Python



# **Тема 4. Графы (задача 9)**