САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 1
по курсу «Алгоритмы и структуры данных»
Тема: Жадные алгоритмы. Динамическое программирование No2.
Вариант 14

Выполнил:
Нгуен Динь Нам
К3140

Проверил:

Санкт-Петербург
2024 г.

Содержание отчёта

Содержание отчёта

Задача №1. Максимальная стоимость добычи (0.5 балла)

Вор находит гораздо больше добычи, чем может поместиться в его сумке. Помогите ему найти самую ценную комбинацию предметов, предполагая, что любая часть предмета добычи может быть помещена в его сумку.

Цель - реализовать алгоритм для задачи о дробном рюкзаке.

- **Формат ввода / входного файла (input.txt).** В первой строке входных данных задано целое число n - количество предметов, и W - вместимость сумки. Следующие n строк определяют значения веса и стоимости предметов. В i-ой строке содержатся целые числа p_i и w_i – стоимость и вес i-го предмета, соответственно.

- **Ограничения на входные данные.** $1 \leq n \leq 10^3, 0 \leq W \leq 2 \cdot 10^6, 0 \leq p_i \leq 2 \cdot 10^6, 0 \leq w_i \leq 2 \cdot 10^6$ для всех $1 \leq i \leq n$. Все числа - целые.

- **Формат вывода / выходного файла (output.txt).** Выведите максимальное значение стоимости долей предметов, которые помещаются в сумку. Абсолютная погрешность между ответом вашей программы и оптимальным значением должно быть не более 10^{-3}. Для этого выведите свой ответ как минимум с четырьмя знаками после запятой (иначе ваш ответ, хотя и будет рассчитан правильно, может оказаться неверным из-за проблем с округлением).

- Ограничение по времени. 2 сек.

- Примеры:

input.txt	output.txt
3 50 60 20 100 50 120 30	180.0000

Чтобы получить значение 180, берем первый предмет и третий предмет в сумку.

input.txt	output.txt
1 10 500 30	166.6667

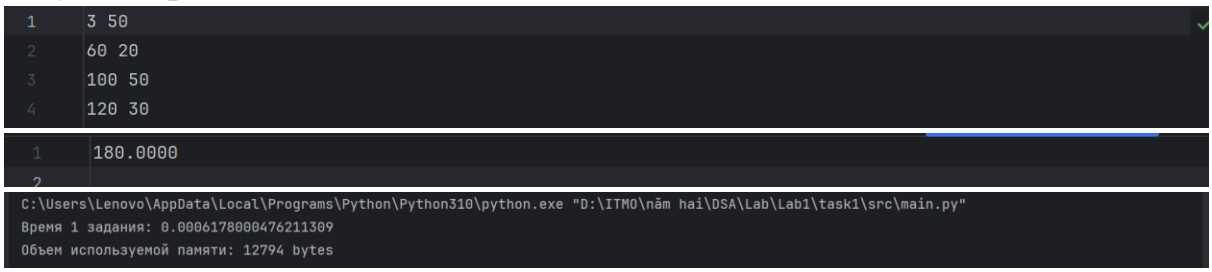
Здесь просто берем одну треть единственного доступного предмета.

```python
def Solve(n, W, items):
    arr = []
    for item in items:
        p, w = item[0], item[1]
        if w > 0:
            arr.append((p, w, p / w))

    arr.sort(key=lambda x: x[2], reverse=True)
    max_value = 0.0

    for p, w, v in arr:
        if W >= w:
            max_value += p
            W -= w
        else:
            max_value += v * W
            break

    return format(max_value, ".4f")
```

- Алгоритм:
  - Для решения этой задачи я использую жадный алгоритм (Greedy Algorithm), основанный на выборе локально оптимальных решений для достижения глобального оптимума.
  - Вычисление соотношения стоимости к весу:
    - Для определения приоритетности каждого предмета необходимо вычислить стоимость на единицу веса, то есть отношение “p/w”.
    - Каждый предмет будет представлен в виде тройки “(p, w, p/w)”, где p — это стоимость предмета, w — его вес, а p/w — отношение стоимости к весу. Этот расчет помогает легко определить, какой предмет будет наиболее выгодным для выбора в первую очередь.
    - Если у предмета w = 0, он будет исключен, чтобы избежать ошибки деления на 0 при выполнении вычислений.
  - Сортировка списка по стоимости на единицу веса:
    - После вычисления отношения стоимости к весу список предметов сортируется в порядке убывания “p/w”. Это означает, что предметы с наибольшей стоимостью на единицу веса будут рассматриваться первыми.
    - Такая сортировка позволяет гарантировать, что при добавлении предметов в рюкзак мы получим наиболее оптимальный вариант, то есть максимальную стоимость в пределах вместимости рюкзака.
  - Выбор предметов для добавления в рюкзак:

- После получения отсортированного списка предметов программа выполняет процесс их выбора для добавления в рюкзак. Изначально общая стоимость выбранных предметов (max_value) устанавливается равной 0, а вместимость рюкзака остается неизменной и равной “W”.
- Если “W >= w”, то он добавляется полностью, обновляется общая стоимость “max_value += p”, а вместимость рюкзака уменьшается “W -= w”. Если “W < w”, то в рюкзак помещается только часть предмета, которая соответствует оставшемуся свободному пространству. В этом случае стоимость добавленного предмета рассчитывается по формуле: “max_value += v * W”
- После добавления части предмета рюкзак заполняется полностью, и выполнение цикла завершается. Итоговым результатом становится максимально возможная стоимость, которую можно достичь с учетом ограниченной вместимости рюкзака.

- Результат работы кода:



- Тесты:

```
new *
def test_knapsack_large_capacity(self):
    arr = [
        [60, 10],
        [100, 20],
        [120, 15]
    ]
    n = 3
    w = 30
    expected_output = '205.0000'

    start_time = time.time()
    tracemalloc.start()
    result = Solve(n, w, arr)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    execution_time = end_time - start_time

    self.assertEqual(result, expected_output)
    self.assertLess(execution_time,  b: 2)
    self.assertLess(peak / 10 ** 6,  b: 100)
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 202
Testing started at 9:48 PM ...
Launching unittests with arguments python -m unittest D:\ITMO\năm hai\DSA\Lab\Lab1\task1\tests\test_main.p



Ran 4 tests in 0.005s

OK

Process finished with exit code 0
```

- Вывод по задаче:
  - Алгоритм жадный, выбирает предметы с наибольшим “p/w” для максимальной стоимости. Если рюкзак не может вместить весь предмет, берётся его часть. Сложность **O(NlogN)** за сортировку и **O(N)** за выбор предметов, что делает алгоритм эффективным даже для больших входных данных.

Задача №5. Максимальное количество призов

Вы организуете веселый конкурс для детей. В качестве призового фонда у вас есть n конфет. Вы хотели бы использовать эти конфеты для раздачи k лучшим местам в конкурсе с естественным ограничением, заключающимся в том, что чем выше место, тем больше конфет. Чтобы осчастливить как можно больше детей, вам нужно найти наибольшее значение k, для которого это возможно.

- **Постановка задачи.** Необходимо представить заданное натуральное число n в виде суммы как можно большего числа попарно различных натуральных чисел. То есть найти максимальное k такое, что n можно записать как $a_1 + a_2 + ... + a_k$, где $a_1, ..., a_k$ - натуральные числа и $a_i \neq a_j$ для всех $1 \leq i < j \leq k$.
- **Формат ввода / входного файла (input.txt).** Входные данные состоят из одного целого числа n.
- **Ограничения на входные данные.** $1 \leq n \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** В первой строке выведите максимальное число k такое, что n можно представить в виде суммы k попарно различных натуральных чисел. Во второй строке выведите эти k попарно различных натуральных чисел, которые в сумме дают n (если таких представлений много, выведите любое из них).
- Ограничение по времени. 2 сек.
- Примеры:

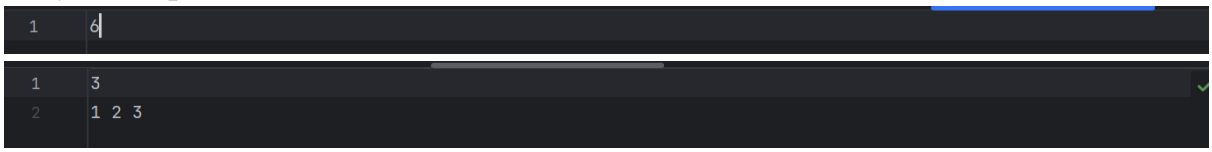
№	input.txt	output.txt
1	6	3 1 2 3

№	input.txt	output.txt
2	8	3 1 2 5

№	input.txt	output.txt
3	2	1 2

```python
def Solve(n):
    result = []
    sum_far = 0
    current = 1

    while sum_far + current <= n:
        result.append(current)
        sum_far += current
        current += 1

    result[-1] += (n - sum_far)

    return len(result), result
```

- Алгоритм:
  - Нам нужно выбрать наименьшие возможные числа, чтобы мы могли использовать как можно больше различных чисел. Если сумма выбранных чисел не равна n, последнее число будет скорректировано для получения правильного результата.
  - Создается список “result”, в котором хранятся слагаемые, а также переменные “sum_far” для отслеживания текущей суммы и current для определения следующего числа, которое можно добавить в список.
  - Процесс выбора чисел начинается с “current = 1”, и числа добавляются в список до тех пор, пока их сумма “sum_far + current” не превышает “n”. После каждого добавления число “current” увеличивается на 1, чтобы гарантировать, что все элементы списка различны.
  - Когда сумма становится близкой к “n”, но добавить следующее число без превышения невозможно, программа выполняет корректировку последнего элемента списка. Это делается путем увеличения последнего числа на (“n - sum_far”), чтобы сумма точно соответствовала n.
- Результат работы кода:

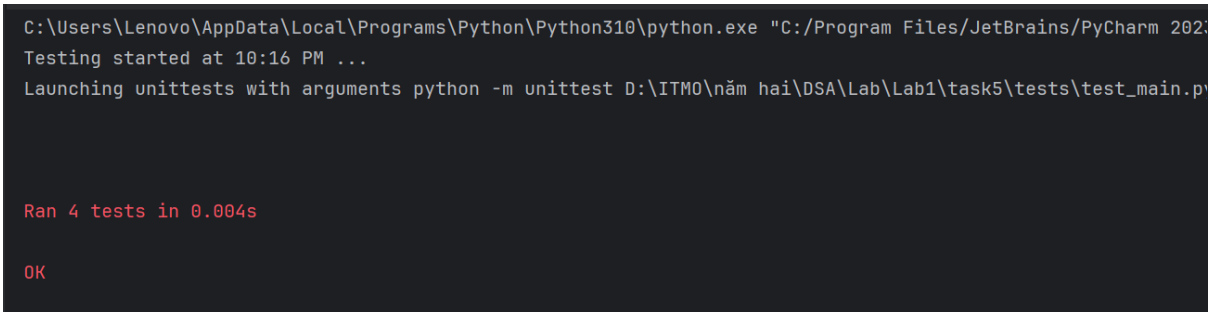


```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab1\task5\src\main.py"
Время 5 задания: 0.00039609987288713455
Объем используемой памяти: 10551 bytes
```

- Тесты:

```
new *
def test_prizes_not_exact_series(self):

    n = 8
    expected_k = 3
    expected_output = [1, 2, 5]

    start_time = time.time()
    tracemalloc.start()
    k, result = Solve(n)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    execution_time = end_time - start_time

    self.assertEqual(k, expected_k)
    self.assertEqual(result, expected_output)
    self.assertLess(execution_time, b: 2)
    self.assertLess(peak / 10 ** 6, b: 100)
```

```
new
def test_prizes_min_n(self):

    n = 1
    expected_k = 1
    expected_output = [1]

    start_time = time.time()
    tracemalloc.start()
    k, result = Solve(n)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    execution_time = end_time - start_time

    self.assertEqual(k, expected_k)
    self.assertEqual(result, expected_output)
    self.assertLess(execution_time, b: 2)
    self.assertLess(peak / 10 ** 6, b: 100)
```



- Вывод по задаче:
  - Алгоритм использует жадный метод, выбирая наименьшие возможные числа в первую очередь, чтобы максимизировать количество слагаемых. Благодаря такому подходу алгоритм работает за $O(\sqrt{N})$ и может эффективно обрабатывать даже большие значения n.

Задача №11. Максимальное количество золота

Вам дается набор золотых слитков, и ваша цель - набрать как можно больше золота в свою сумку. Существует только одна копия каждого слитка, и для каждого слитка вы можете либо взять его, либо нет (т.е. вы не можете взять часть слитка).

- **Постановка задачи.** Даны n золотых слитков, найдите максимальный вес золота, который поместится в сумку вместимостью W.
- **Формат ввода / входного файла (input.txt).** Первая строка входных данных содержит вместимость W сумки и количество n золотых слитков. В следующей строке записано n целых чисел $w_0, w_1, ..., w_{n-1}$, определяющие вес золотых слитков.
- **Ограничения на входные данные.** $1 \leq W \leq 10^4$, $1 \leq n \leq 300$, $0 \leq w_0, ..., w_{n-1} \leq 10^5$
- **Формат вывода / выходного файла (output.txt).** Выведите максимальный вес золота, который поместится в сумку вместимости W.
- Ограничение по времени. 5 сек.
- Пример:

input.txt	output.txt
10 3 1 4 8	9

Здесь сумма весов первого и последнего слитка равна 9.

- Обратите внимание, что в этой задаче все предметы имеют одинаковую стоимость на единицу веса по простой причине: все они сделаны из золота.

```python
def Solve(W, weights):
    if W == 0:
        return 0
    weights_valid = [w for w in weights if 0 < w <= W]
    dp = [0] * (W + 1)
    for w in weights_valid:
        for j in range(W, w - 1, -1):
            dp[j] = max(dp[j], dp[j - w] + w)
    return dp[W]
```

- Алгоритм:
  - Сначала я читаю число “W” из входного файла, затем загружаю список весов предметов. Чтобы обрабатывать только подходящие

предметы, я фильтрую список, оставляя только предметы с весом больше 0 и не превышающим W. Эта фильтрация позволяет мне заранее избавиться от некорректных значений и избежать ошибок при вычислениях.

- После фильтрации я создаю массив “dp” размером “W + 1”, где dp[j] хранит максимальный вес, который можно набрать при вместимости “j”. На начальном этапе я заполняю массив нулями, так как в пустом рюкзаке нет предметов.
- Затем я поочередно рассматриваю каждый предмет “w” из списка и обновляю таблицу “dp”. Чтобы не перезаписывать данные некорректно, я перебираю “j” в обратном порядке – от “W” до “w”, а не от “0” до “W”. Для каждого “j” я проверяю, можно ли добавить w в рюкзак, используя формулу: “dp[j]=max(dp[j],dp[j−w]+w)” - Я выбираю максимальное значение, чтобы рюкзак заполнялся наиболее выгодным образом.
- После обработки всех предметов я получаю “dp[W]” – это максимальный возможный вес, который можно набрать, не превышая “W”.

- Результат работы кода:

- Тесты:

```python
class TestTask11(unittest.TestCase):

    new *
    def test_knapsack_basic(self):
        W = 10
        weights = [1, 4, 8]
        expected_output = 9

        start_time = time.time()
        tracemalloc.start()
        result = Solve(W, weights)
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        end_time = time.time()

        execution_time = end_time - start_time

        self.assertEqual(result, expected_output)
        self.assertLess(execution_time, b: 5)
        self.assertLess(peak / 10 ** 6, b: 100)
```

```python
    def test_knapsack_exact_fit(self):
        W = 15
        weights = [3, 5, 7]
        expected_output = 15

        start_time = time.time()
        tracemalloc.start()
        result = Solve(W, weights)
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        end_time = time.time()

        execution_time = end_time - start_time

        self.assertEqual(result, expected_output)
        self.assertLess(execution_time, b: 5)
        self.assertLess(peak / 10 ** 6, b: 100)
```

```
✓ Tests passed: 4 of 4 tests – 5 ms

C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 202
Testing started at 10:44 PM ...
Launching unittests with arguments python -m unittest D:\ITMO\năm hai\DSA\Lab\Lab1\task11\tests\test_main.

Ran 4 tests in 0.005s

OK
```

- Вывод по задаче:

Я использовал динамическое программирование, чтобы оптимизировать выбор предметов и гарантировать, что общий вес будет максимальным, но не превысит W. Фильтрация входных данных помогла избежать

обработки неподходящих значений, что сделало алгоритм более эффективным. Сложность алгоритма O(N * W).

Задача №14. Максимальное значение арифметического выражения

В этой задаче ваша цель - добавить скобки к заданному арифметическому выражению, чтобы максимизировать его значение.

$$max(5 - 8 + 7 \times 4 - 8 + 9) = ?$$

- **Постановка задачи.** Найдите максимальное значение арифметического выражения, указав порядок применения его арифметических операций с помощью дополнительных скобок.
- **Формат ввода / входного файла (input.txt).** Единственная строка входных данных содержит строку s длины $2n + 1$ для некоторого n с символами $s_0, s_1, ..., s_{2n}$. Каждый символ в четной позиции s является цифрой (то есть целым числом от 0 до 9), а каждый символ в нечетной позиции является одной из трех операций из $+, -, *$
- **Ограничения на входные данные.** $0 \leq n \leq 14$ (следовательно, строка содержит не более 29 символов).
- **Формат вывода / выходного файла (output.txt).** Выведите максимально возможное значение заданного арифметического выражения среди различных порядков применения арифметических операций.
- Ограничение по времени. 5 сек.
- Пример:

input.txt	output.txt
1+5	6

input.txt	output.txt
5-8+7*4-8+9	200

Здесь $200 = (5 - ((8 + 7) * (4 - (8 + 9))))$.

```
4 usages  new *
def calculate(a, b, operate):
    if operate == '+':
        return a + b
    elif operate == '-':
        return a - b
    elif operate == '*':
        return a * b


new *
```

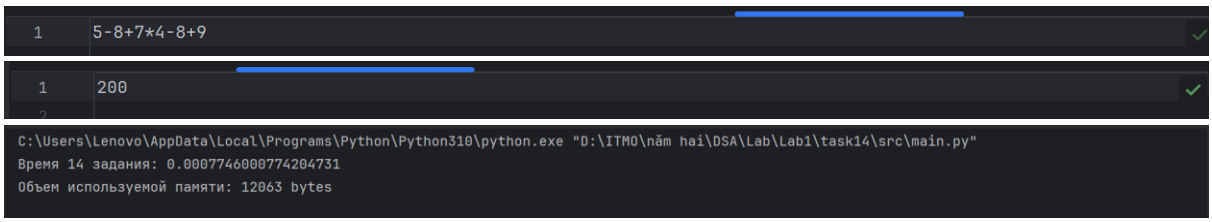
```python
def Solve(expression):
    numbers = []
    operators = []
    for char in expression:
        if char.isdigit():
            numbers.append(int(char))
        else:
            operators.append(char)

    n = len(numbers)
    dp_max = [[0] * n for _ in range(n)]
    dp_min = [[0] * n for _ in range(n)]

    for i in range(n):
        dp_max[i][i] = numbers[i]
        dp_min[i][i] = numbers[i]

    for length in range(1, n):
        for i in range(n - length):
            j = i + length
            dp_max[i][j] = float('-inf')
            dp_min[i][j] = float('inf')
```

```python
            for k in range(i, j):
                op = operators[k]

                a = calculate(dp_max[i][k], dp_max[k + 1][j], op)
                b = calculate(dp_max[i][k], dp_min[k + 1][j], op)
                c = calculate(dp_min[i][k], dp_max[k + 1][j], op)
                d = calculate(dp_min[i][k], dp_min[k + 1][j], op)

                dp_max[i][j] = max(dp_max[i][j], a, b, c, d)
                dp_min[i][j] = min(dp_min[i][j], a, b, c, d)

    return dp_max[0][n - 1]
```

- Алгоритм:
  - Сначала я разделяю числа (numbers) и операторы (operators) из входного выражения. Это позволяет мне обрабатывать каждое число и оператор отдельно.
  - Затем я использую динамическое программирование для вычисления максимального и минимального значений возможных подвыражений. Я создаю две таблицы “dp_max” и “dp_min”, где dp_max[i][j] хранит максимальное значение, которое можно получить из подвыражения numbers[i:j], а dp_min[i][j] — минимальное значение.
  - Для начала я заполняю базовые значения, где “dp_max[i][i]” и “dp_min[i][i]” равны “numbers[i]”, потому что одно число не изменяет свое значение.

- Затем я перебираю все возможные подвыражения длиной от “2” до “n”. Чтобы вычислить значение подвыражения numbers[i:j], я ищу индекс “k”, который делит выражение на две части (“numbers[i:k]” и “numbers[k+1:j]”), что соответствует расстановке скобок перед оператором “operators[k]”.
- Для каждого оператора “op” я вычисляю все возможные значения. Затем я обновляю значения “dp_max[i][j]” и “dp_min[i][j]”. После обработки всех возможных комбинаций итоговое значение “dp_max[0][n-1]” будет максимально возможным значением выражения.

- Результат работы кода:



- Тесты:

```python
class TestTask14(unittest.TestCase):

    new *
    def test_case_1(self):
        expression = "5-8+7*4-8+9"
        expected_result = 200

        start_time = time.time()
        tracemalloc.start()
        result = Solve(expression)
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        end_time = time.time()

        execution_time = end_time - start_time

        self.assertEqual(result, expected_result)
        self.assertLess(execution_time, b: 5)
        self.assertLess(peak / 10 ** 6, b: 10)
```

```
new *
def test_case_5(self):
    expression = "2*5+3-6*4"
    expected_result = 40

    start_time = time.time()
    tracemalloc.start()
    result = Solve(expression)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    execution_time = end_time - start_time

    self.assertEqual(result, expected_result)
    self.assertLess(execution_time, b: 5)
    self.assertLess(peak / 10 ** 6, b: 10)
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 202
Testing started at 11:21 PM ...
Launching unittests with arguments python -m unittest D:\ITMO\năm hai\DSA\Lab\Lab1\task14\tests\test_main.

Ran 6 tests in 0.005s

OK
```

- Вывод по задаче:
  - Я использовал динамическое программирование для перебора всех возможных расстановок скобок и поиска максимального значения выражения. Создание таблиц dp_max и dp_min позволило мне находить оптимальные результаты для подвыражений и затем комбинировать их для получения итогового решения. Алгоритм имеет сложность **$O(N^3)$.**

Задача №15. Удаление скобок.

- **Постановка задачи.** Дана строка, составленная из круглых, квадратных и фигурных скобок. Определите, какое наименьшее количество символов необходимо удалить из этой строки, чтобы оставшиеся символы образовывали правильную скобочную последовательность.
- **Формат ввода / входного файла (input.txt).** Во входном файле записана строка, состоящая из s символов: круглых, квадратных и фигурных скобок $(), [], \{\}$. Длина строки не превосходит 100 символов.
- **Ограничения на входные данные.** $1 \leq s \leq 100$.
- **Формат вывода / выходного файла (output.txt).** Выведите строку максимальной длины, являющейся правильной скобочной последовательностью, которую можно получить из исходной строки удалением некоторых символов.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

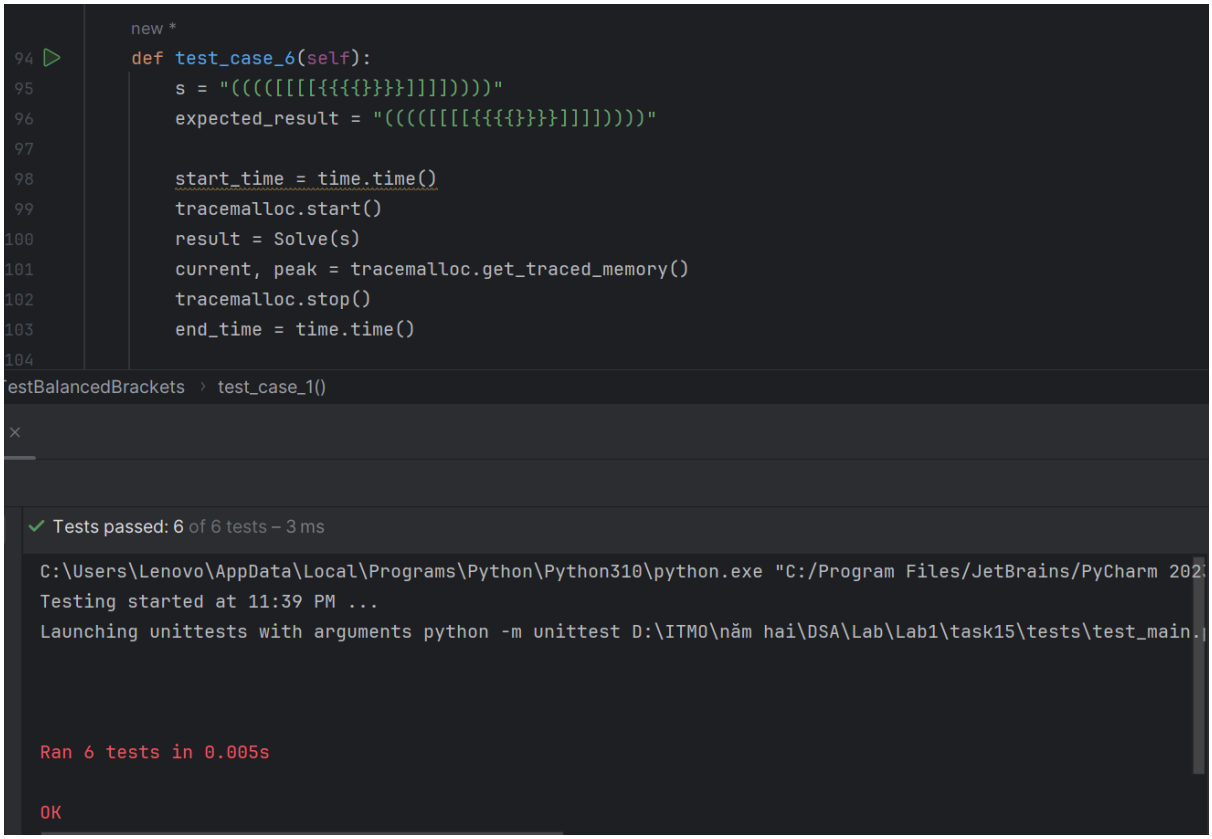
input.txt	output.txt
([)]	[]

```python
def Solve(s):
    stack = []
    marks_invalid = set()
    pairs = {')': '(', ']': '[', '}': '{'}
    open_marks = "([{"
    close_marks = ")]}"

    for i, char in enumerate(s):
        if char in open_marks:
            stack.append(i)
        elif char in close_marks:
            if stack and s[stack[-1]] == pairs[char]:
                stack.pop()
            else:
                marks_invalid.add(i)

    marks_invalid.update(stack)
    result = []
    k = "()[]{}"
```

```
for i, char in enumerate(s):
    if char not in k:
        result.append(char)
    elif i not in marks_invalid:
        result.append(char)

return ''.join(result)
```

- Алгоритм:
  - Я использую стек stack для хранения индексов открывающих скобок ((, [, {). Кроме того, я создаю множество marks_invalid для хранения индексов некорректных закрывающих скобок.
  - Если символ открывающая скобка, я сохраняю ее индекс в “stack”. Если символ закрывающая скобка, я проверяю stack: Если “stack” не пуст, и верхний элемент stack соответствует текущей закрывающей скобке, я удаляю открывающую скобку из “stack”, так как пара найдена. Если соответствующей открывающей скобки нет, я добавляю индекс текущей скобки в “marks_invalid” как некорректный.
  - После прохода по всей строке, если в stack остались элементы, это значит, что есть открывающие скобки без пар, поэтому я добавляю их в “marks_invalid”.
  - После этого в “marks_invalid” содержатся все некорректные скобки, которые нужно удалить.
  - Я создаю список “result”, в который буду добавлять корректные символы. Прохожу по строке заново и добавляю каждый символ в “result”, если выполняются условия: Если символ не является скобкой, я всегда сохраняю его. Если символ — скобка, я добавляю его только в том случае, если его индекс не находится в “marks_invalid”.
  - В конце я объединяю список result в строку и возвращаю итоговый результат.
- Результат работы кода:

- Тесты:



- Вывод по задаче:
  - Я использовал стек и множество, чтобы определить и удалить некорректные скобки. Благодаря этому алгоритм работает быстро и эффективно, обрабатывая строку за O(N). Метод гарантирует, что все корректные символы остаются неизменными, а некорректные скобки удаляются, сохраняя правильный порядок оставшихся символов

Задача №16. Продавец.

- **Постановка задачи.** Продавец техники хочет объехать n городов, посетив каждый из них ровно один раз. Помогите ему найти кратчайший путь.
- **Формат ввода / входного файла (input.txt).** Первая строка входного файла содержит натуральное число n – количество городов. Следующие n строк содержат по n чисел – длины путей между городами. В i-й строке j-е число – $a_{i,j}$ – это расстояние между городами i и j.
- **Ограничения на входные данные.** $1 \leq n \leq 13$, $0 \leq a_{i,j} \leq 10^6$, $a_{i,j} = a_{j,i}$, $a_{i,i} = 0$.
- **Формат вывода / выходного файла (output.txt).** В первой строке выходного файла выведите длину кратчайшего пути. Во второй строке выведите nчисел – порядок, в котором нужно посетить города.
- Ограничение по времени. 1 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt
5 0 183 163 173 181 183 0 165 172 171 163 165 0 189 302 173 172 189 0 167 181 171 302 167 0

output.txt
666 4 5 2 3 1

```python
def Solve(n, distance):
    INF = float('inf')
    full_mask = (1 << n) - 1
    dp = [[INF] * n for _ in range(1 << n)]
    parent = [[-1] * n for _ in range(1 << n)]

    for i in range(n):
        dp[1 << i][i] = 0

    for mask in range(1 << n):
        for u in range(n):
            if not (mask & (1 << u)):
                continue
            for v in range(n):
                if mask & (1 << v):
                    continue
                new_mask = mask | (1 << v)
                new_cost = dp[mask][u] + distance[u][v]
                if new_cost < dp[new_mask][v]:
                    dp[new_mask][v] = new_cost
                    parent[new_mask][v] = u
```

```python
    result = INF
    city_end = -1

    for i in range(n):
        if dp[full_mask][i] < result:
            result = dp[full_mask][i]
            city_end = i

    arr = []
    mask = full_mask
    current = city_end

    for _ in range(n):
        arr.append(current)
        prev = parent[mask][current]
        if prev == -1:
            break
        mask ^= (1 << current)
        current = prev

    arr.reverse()
    arr = [x + 1 for x in arr]
    return result, arr
```

- Алгоритм:
  - Я использую таблицу памяти “dp[mask][i]”, где “mask” — битовая маска (bitmask), представляющая множество посещенных городов; “i” — последний посещенный город в данном mask; “dp[mask][i]” хранит минимальную стоимость маршрута, проходящего через города в mask и заканчивающегося в “i”.

- Каждое состояние “mask” состоит из “n” битов, где 1 обозначает, что город уже посещен, а “0” — что он еще не посещен. Это позволяет эффективно проверять и обновлять состояния.
- Я создаю массив “dp”, где все элементы инициализируются значением “INF”, чтобы обозначить, что маршрут еще не рассчитан.
- Затем я устанавливаю начальные условия: “dp[1 << i][i] = 0”, что означает, что если мы начинаем в городе “i”, то стоимость пути к нему равна “0”.
- Я перебираю все возможные множества посещенных городов, “mask” от “0 до (1 << n) – 1”.
- Для каждого “mask” я проверяю все города “u”, которые уже были посещены. Затем я пытаюсь добавить новый город “v”, который еще не посещен. Если “v” не был посещен (mask & (1 << v) == 0), то обновляю минимальную стоимость маршрута: “new_cost = dp[mask][u] + distance[u][v]”.
- При этом я обновляю массив “parent”, который хранит предшествующий город на оптимальном пути.
- После завершения перебора таблица “dp” содержит оптимальные маршруты для всех состояний.
- После завершения динамического программирования я нахожу “min_cost”, проверяя все “dp[full_mask][i]”, где “full_mask = (1 << n) – 1”, то есть все города посещены. Город “city_end”, в котором заканчивается минимальный маршрут, сохраняется для дальнейшей реконструкции пути.
- Используя массив “parent”, я восстанавливаю путь из “city_end” в начальную точку. Я начинаю с “city_end”, добавляя его в список path, затем двигаюсь назад, используя “prev = parent[mask][current]”. Каждый раз я обновляю “mask ^= (1 << current)”, чтобы удалить текущий город из множества посещенных.
- Этот процесс продолжается до тех пор, пока я не достигну начального города. Затем я переворачиваю список “arr", чтобы получить маршрут в правильном порядке.

- Результат работы кода:

```
5
0 183 163 173 181
183 0 165 172 171
163 165 0 189 302
173 172 189 0 167
181 171 302 167 0
```

```
666
4 5 2 3 1
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab1\task16\src\main.py"
Время 16 задания: 0.0012823001015931368
Объем используемой памяти: 17045 bytes
```

- Тесты:

```
def test_case_3(self):
    n = 3
    distances = [
        [0, 29, 20],
        [29, 0, 15],
        [20, 15, 0]
    ]
    expected_cost = 35
    expected_path = [2, 3, 1]

    start_time = time.time()
```

TestTask16 › test_case_3()

stTask16 ×

Tests passed: 4 of 4 tests – 4 ms

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 202
Testing started at 12:32 AM ...
Launching unittests with arguments python -m unittest test_main.TestTask16 in D:\ITMO\năm hai\DSA\Lab\Lab1

Ran 4 tests in 0.006s

OK
```

- Вывод по задаче:
  - Я использовал динамическое программирование с битмаской для решения задачи. Этот метод имеет сложность O(2^n * n), что намного эффективнее, чем O(n!) полного перебора.

Задача №17. Ход конем.

- **Постановка задачи.** Шахматная ассоциация решила оснастить всех своих сотрудников такими телефонными номерами, которые бы набирались на кнопочном телефоне ходом коня. Например, ходом коня набирается телефон 340-49-27. При этом телефонный номер не может начинаться ни с цифры 0, ни с цифры 8.

1	2	3
4	5	6
7	8	9
.	0	.

  Напишите программу, определяющую количество телефонных номеров длины N, набираемых ходом коня. Поскольку таких номеров может быть очень много, выведите ответ по модулю 10^9.

- **Формат ввода / входного файла (input.txt).** Во входном файле записано одно целое число N.

- **Ограничения на входные данные.** $1 \leq N \leq 1000$.

- **Формат вывода / выходного файла (output.txt).** Выведите в выходной файл искомое количество телефонных номеров по модулю 10^9.

- Ограничение по времени. 1 сек.

- Ограничение по памяти. 256 мб.

- Примеры:

input.txt	output.txt
1	8

input.txt	output.txt
2	16

```python
def Solve(N):
    MOD = 10**9
    moves = {
        0: [4, 6],
        1: [6, 8],
        2: [7, 9],
        3: [4, 8],
        4: [0, 3, 9],
        5: [],
        6: [0, 1, 7],
        7: [2, 6],
        8: [1, 3],
        9: [2, 4]
    }
    dp = [0] * 10

    for i in range(10):
        dp[i] = 1 if i not in (0, 8) else 0

    for _ in range(1, N):
        new_dp = [0] * 10
        for j in range(10):
            if dp[j] == 0:
                continue
            for next in moves[j]:
                new_dp[next] = (new_dp[next] + dp[j]) % MOD
```

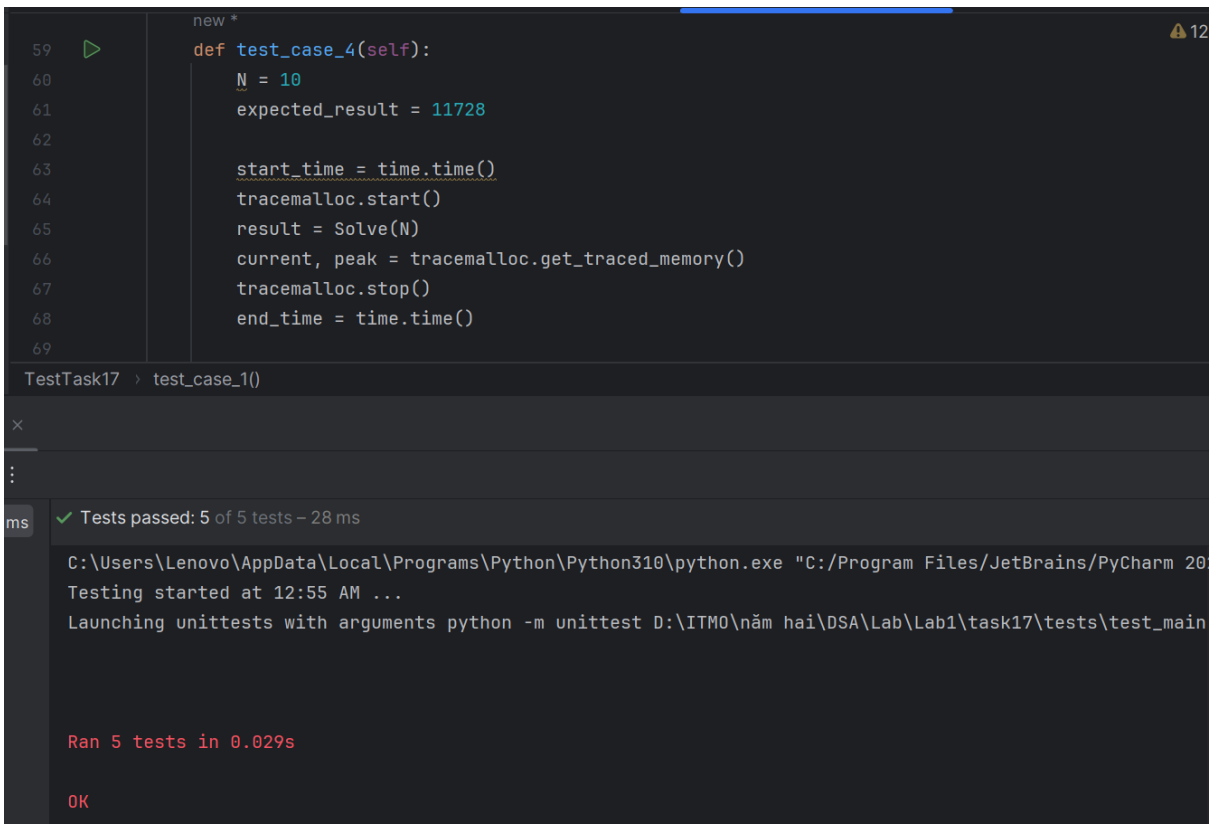
```python
    return sum(dp) % MOD
```

- Алгоритм:
  - Для решения задачи я использую массив dp, где “dp[i]” обозначает количество способов получить номер телефона, заканчивающийся на клавише “i” после определенного количества шагов.
  - На первом шаге (N = 1) каждая клавиша (0-9) может быть начальной позицией, поэтому “dp[i] = 1” для всех “i”. Однако, согласно условиям задачи, клавиши 0 и 8 не могут быть началом, поэтому “dp[0]” и “dp[8]” устанавливаются в 0.
  - На первом шаге я заполняю “dp”, устанавливая 1 для всех клавиш, кроме 0 и 8, которые равны 0, так как они не могут быть стартовыми позициями. Каждое значение 1 означает, что есть один способ начать с этой клавиши.
  - Я использую цикл “N – 1” раз, чтобы обновить “dp” на основе возможных переходов. На каждом шаге: Создаю массив new_dp, чтобы хранить новое количество комбинаций; Перебираю все клавиши j (0-9). Если “dp[j] = 0”, пропускаю, так как невозможно начать с этой клавиши. Если “dp[j] > 0”, я добавляю количество путей из “dp[j]” в “new_dp[next]”, где “next” — возможные следующие клавиши.

- После каждого шага “dp” обновляется значениями из “new_dp”.
- После завершения “N – 1” шагов, я суммирую все элементы “dp”, так как телефонный номер может заканчиваться на любой клавише (0-9). Окончательный результат вычисляется по модулю 10^9 для предотвращения переполнения чисел.

- Результат работы кода:

- Тесты:



- Вывод по задаче:
  - Я применил динамическое программирование с оптимизированным использованием памяти для решения этой

задачи. Алгоритм имеет линейную сложность O(N), что позволяет ему работать эффективно даже для больших значений N.

Задача №20. Почти палиндром.

- **Постановка задачи.** Слово называется палиндромом, если его первая буква совпадает с последней, вторая – с предпоследней и т.д. Например: «abba», «madam», «x».

  Для заданного числа K слово называется почти палиндромом, если в нем можно изменить не более K любых букв так, чтобы получился палиндром. Например, при $K = 2$ слова «$reactor$», «$kolobok$», «$madam$» являются почти палиндромами (подчеркнуты буквы, заменой которых можно получить палиндром).

  Подсловом данного слова являются все слова, получающиеся путем вычеркивания из данного нескольких (возможно, одной или нуля) первых букв и нескольких последних. Например, подсловами слова «cat» являются слова «c», «a», «t», «ca», «at» и само слово «cat» (а «ct» подсловом слова «cat» не является).

  Требуется для данного числа K определить, сколько подслов данного слова S являются почти палиндромами.

- **Формат входного файла (input.txt).** В первой строке входного файла вводятся два натуральных числа: N – длина слова и K. Во второй строке записано слово S, состоящее из N строчных английских букв.

- **Ограничения на входные данные.** $1 \le N \le 5000$, $0 \le K \le N$.

- **Формат выходного файла (output.txt).** В выходной файл требуется вывести одно число – количество подслов слова S, являющихся почти палиндромами (для данного K).

- Ограничение по времени. 2 сек.

- Ограничение по памяти. 256 мб.

- Примеры:

input.txt	output.txt
5 1 abcde	12

input.txt	output.txt
3 3 aaa	6

- Проверить можно по ссылке.

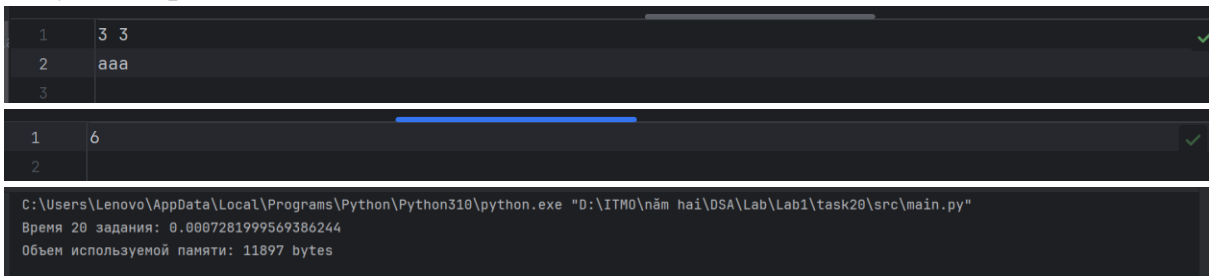
```
new *                                                    8  2
def Solve(n, k, s):
    count = 0

    for middle in range(n):
        left, right = middle, middle
        matches = 0

        while left >= 0 and right < n:
            if s[left] != s[right]:
                matches += 1
            if matches > k:
                break
            count += 1
            left -= 1
            right += 1

        left, right = middle, middle + 1
        matches = 0
```

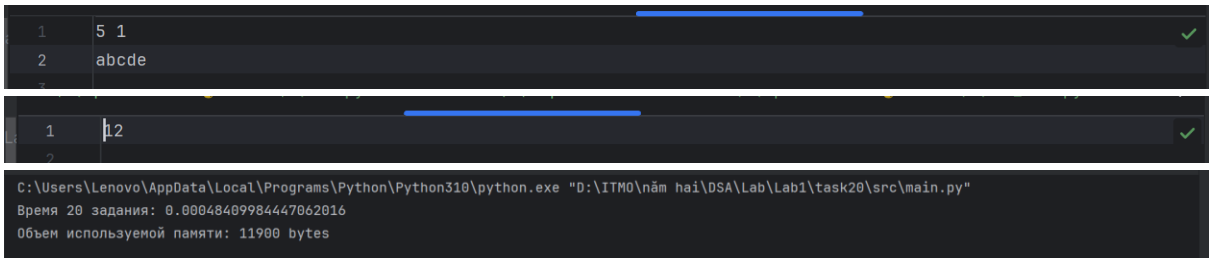
```
        while left >= 0 and right < n:
            if s[left] != s[right]:
                matches += 1
            if matches > k:
                break
            count += 1
            left -= 1
            right += 1

    return count
```

- Алгоритм:
  - Я использую метод расширения от центра, чтобы проверить все возможные подстроки и определить, являются ли они почти палиндромами.
  - Для каждой позиции в строке я проверяю две группы подстрок: Подстроки нечетной длины, где центр находится в “middle”. Подстроки четной длины, где центр находится между “middle” и “middle + 1”
  - Затем я расширяю подстроку влево и вправо, проверяя, сколько символов различаются. Если число различающихся символов (matches) превышает “k”, то данная подстрока уже не может быть почти палиндромом, и я прекращаю расширение.
  - Сначала я создаю переменную “count = 0”, чтобы отслеживать количество подходящих подстрок. Затем я перебираю каждую позицию “middle” в строке и использую ее как центр подстроки. Для каждого “middle” выполняются два типа проверок.

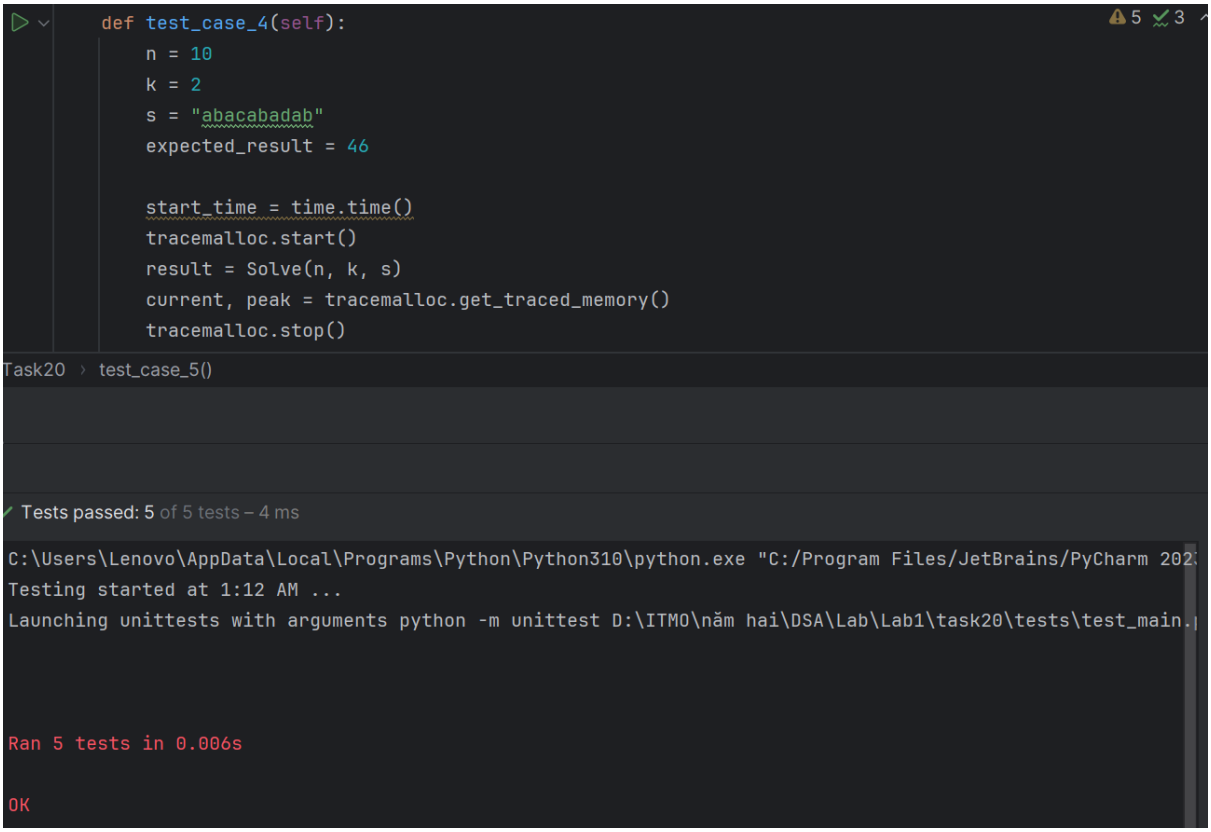
- Для подстрок нечетной длины: Устанавливаю “left = middle”, “right = middle” и начинаю расширение. Если “s[left] ≠ s[right]”, увеличиваю mismatches. Если “matches > k”, прекращаю проверку. Если “matches ≤ k”, увеличиваю count, так как подстрока удовлетворяет условиям.
- Для подстрок четной длины: Устанавливаю “left = middle”, “right = middle + 1” и выполняю аналогичную проверку.
- После завершения прохода по всей строке я возвращаю значение “count”, которое представляет общее количество подстрок, удовлетворяющих условию.

- Результат работы кода:





- Тесты:



- Вывод по задаче:
  - Я использовал метод расширения от центра, который позволяет эффективно проверять подстроки, избегая полного перебора всех возможных вариантов. Алгоритм имеет сложность $O(n^2)$. Метод удобен для задач, связанных с поиском палиндромных подстрок, и может применяться в различных вариациях.

Вывод

В этой лабораторной работе я применил алгоритмы жадного подхода и динамического программирования для решения различных задач оптимизации. Задачи, такие как задача о рюкзаке и разбиение на призы, были решены с помощью жадного метода для быстрого нахождения локально оптимального решения. В то же время задачи коммивояжера (TSP), вычисления оптимального выражения, поиска наибольшей подпоследовательности потребовали использования динамического программирования для хранения промежуточных результатов и сокращения вычислительных затрат. Я также использовал Bitmask DP для уменьшения количества состояний и метод расширения от центра для эффективного поиска симметричных подстрок. Анализ сложности, оптимизация алгоритмов и рациональное использование памяти стали ключевыми аспектами данной лабораторной работы, позволившими мне улучшить алгоритмическое мышление и навыки решения задач.