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

Отчет по лабораторной работе №1 (семестр 2)
по курсу «Алгоритмы и структуры данных»
Тема: Жадные алгоритмы. Динамическое программирование №2
Вариант 10

Выполнила:
Коновалова Кира Романовна
К3139

Проверил:

Санкт-Петербург
2025 г.

Содержание отчета

Задачи по варианту. Задачи по выбору

Задача №3. Максимальная стоимость добычи (0.5 балла)

Текст задачи:

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.

Листинг кода:

```
def max_revenue_calculate(a, b):
    '''вычисляет маскимальный доход от рекламы'''
    a.sort(reverse=True) #список прибыли за клик
    b.sort(reverse=True) #список среднего количества кликов
    return sum(a[i] * b[i] for i in range(len(a)))

def max_ad_revenue(input_file: str, output_file: str):
    '''читает, вычисляет и записывает данные'''
    with open(input_file, 'r') as f:
        n = int(f.readline().strip())
        a = list(map(int, f.readline().split()))
        b = list(map(int, f.readline().split()))

    max_revenue = max_revenue_calculate(a, b)

    with open(output_file, 'w') as f:
        f.write(str(max_revenue) + "\n")

if __name__ == "__main__":
    max_ad_revenue('../txtf/input.txt', '../txtf/output.txt')
```

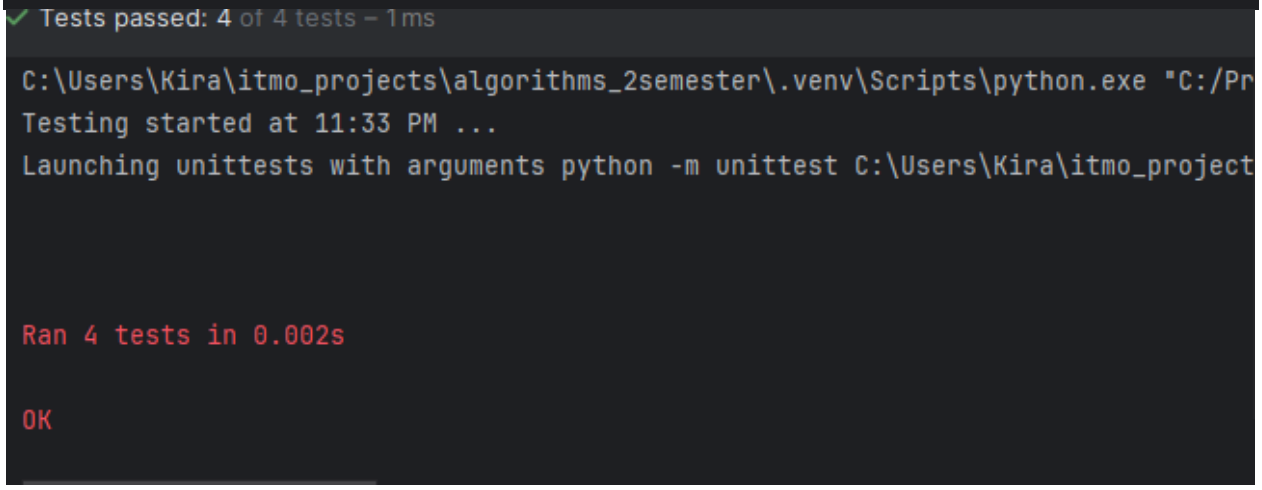
Текстовое объяснение задачи:

Алгоритм решает задачу о дробном рюкзаке, где нужно выбрать предметы (рекламные объекты) с максимальной стоимостью (доходом) при условии, что рюкзак (сумка) может вмещать любую часть предмета. В данном случае наибольший доход можно получить, если упорядочить предметы по убыванию отношения "доход за клик" и "среднее количество кликов" для каждого предмета

Сначала сортируются два списка — список доходов за клик и список среднего количества кликов. После сортировки пары доходов и кликов с наибольшими значениями оказываются в начале этих списков. Далее идет вычисление максимального дохода. Максимальный доход вычисляется как сумма произведений элементов из обоих списков с одинаковыми индексами

Тесты:

```python
import unittest
from lab1.task3.src.main import max_revenue_calculate

class TestMaxAdRevenue(unittest.TestCase):
    def test_case_1(self):
        self.assertEqual(max_revenue_calculate([23], [39]), 897)

    def test_case_2(self):
        self.assertEqual(max_revenue_calculate([1, 3, -5], [-2, 4, 1]), 23)

    def test_case_3(self):
        self.assertEqual(max_revenue_calculate([-1, -2, -3], [-3, -2, -1]),
14)

    def test_case_4(self):
        self.assertEqual(max_revenue_calculate([0, 0, 0], [0, 0, 0]), 0)

if __name__ == "__main__":
    unittest.main()
```



Вывод по задаче:

Алгоритм решает задачу о дробном рюкзаке, эффективно вычисляя максимальный возможный доход от выбранных предметов. Основной сложностью является сортировка списков, что приводит к временной сложности **O(n * log(n))**. Алгоритм может применяться для задач, где нужно максимизировать прибыль при ограничениях на количество предметов, их стоимости или доступности

Задача №7. Проблема сапожника (0.5 балла)

Текст задачи:

В некоей воинской части есть сапожник. Рабочий день сапожника длится K минут. Заведующий складом оценивает работу сапожника по количеству починенной обуви, независимо от того, насколько сложный ремонт требовался в каждом случае. Дано n сапог, нуждающихся в починке. Определите, какое

максимальное количество из них сапожник сможет починить за один рабочий день.

Листинг кода:

```
def max_boots(k, n, times):
    times.sort()

    total_time = 0
    cnt = 0

    for time in times:
        if total_time + time <= k:
            total_time += time
            cnt += 1
        else:
            break #выход если время работы превышает K
    return cnt

if __name__ == "__main__":
    with open('../txtf/input.txt', 'r') as f:
        k, n = map(int, f.readline().split())
        times = list(map(int, f.readline().split()))

    result = max_boots(k, n, times)

    with open('../txtf/output.txt', 'w') as f:
        f.write(str(result))
```

Текстовое объяснение задачи:

Алгоритм решает задачу, определяя максимальное количество сапог, которые сапожник может починить за один рабочий день, при ограничении по времени, отведенному на работу

Алгоритм использует жадный подход, решая задачу путем поочередного выбора сапог, которые могут быть отремонтированы в оставшееся время.

Шаги алгоритма:

1. Сортировка времени, где алгоритм сортирует список времени, необходимого для починки каждого сапога, в порядке возрастания. Это позволяет сначала починить те сапоги, которые требуют меньше времени
2. Перебор времени. Алгоритм проходит по отсортированному списку и, начиная с сапога, требующего минимального времени, пытается добавить его время починки к общему времени. Если время работы не превышает ограничения K, сапожник чинит этот сапог, и счетчик починенных сапог увеличивается
3. Если время работы превышает K, алгоритм останавливает выполнение и возвращает количество починенных сапог.

Тесты:

```python
import unittest
from lab1.task7.src.main import max_boots

class TestMaxBoots(unittest.TestCase):

    def test_case_1(self):
        k, n = 10, 3
        times = [2, 6, 8]
        self.assertEqual(max_boots(k, n, times), 2)

    def test_case_2(self):
        k, n = 3, 2
        times = [5, 7]
        self.assertEqual(max_boots(k, n, times), 0)

    def test_case_3(self):
        k, n = 5, 1
        times = [3]
        self.assertEqual(max_boots(k, n, times), 1)

    def test_case_4(self):
        k, n = 10, 4
        times = [3, 3, 3, 3]
        self.assertEqual(max_boots(k, n, times), 3)

    def test_case_5(self):
        k, n = 1000, 500
        times = [1] * 500
        self.assertEqual(max_boots(k, n, times), 500)

    def test_case_6(self):
        k, n = 1, 3
        times = [1, 2, 3]
        self.assertEqual(max_boots(k, n, times), 1)

if __name__ == '__main__':
    unittest.main()
```

```
✓ Tests passed: 6 of 6 tests – 1 ms

C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pr
Testing started at 11:34 PM ...
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_projects



Ran 6 tests in 0.002s

OK
```

Вывод по задаче:

Алгоритм эффективно решает задачу с использованием жадного подхода, сортируя время починки и поочередно добавляя сапоги в список починенных, пока время работы не превышает установленный лимит. Тесты подтверждают корректность работы алгоритма в различных случаях

Время работы алгоритма: сортировка занимает O(n*log(n)), где n это количество сапог. Прохождение по списку занимает O(n). Таким образом, общая временная сложность алгоритма O(n*log(n))

Задача №9. Распечатка (1 балл)

Текст задачи:

Диссертация дело сложное, особенно когда нужно ее печатать. При этом вам нужно распечатать не только текст самой диссертации, так и другие материалы (задание, рецензии, отзывы, авторефераты для защиты и т.п.). Вы оценили объём печати в N листов. Фирма, готовая размножить печатные материалы, предлагает следующие финансовые условия. Один лист она печатает за A1 рублей, 10 листов - за A2 рублей, 100 листов - за A3 рублей, 1000 листов - за A4 рублей, 10000 листов - за A5 рублей, 100000 листов - за A6 рублей, 1000000 листов - за A7 рублей. При этом не гарантируется, что один лист в более крупном заказе обойдется дешевле, чем в более мелком. И даже может оказаться, что для любой партии будет выгодно воспользоваться тарифом для одного листа. Печать конкретного заказа производится или путем комбинации нескольких тарифов, или путем заказа более крупной партии. Например, 980 листов можно распечатать, заказав печать 9 партий по 100 листов плюс 8 партий по 10 листов, сделав 98 заказов по 10 листов, 980 заказов по 1 листу или заказав печать 1000 (или даже 10000 и более) листов, если это окажется выгоднее. Требуется по заданному объему заказа в листах N определить минимальную сумму денег в рублях, которой будет достаточно для выполнения заказа.

Листинг кода:

```
def min_print(N, A):
    batch_sizes = [1, 10, 100, 1000, 10000, 100000, 1000000]  # Размеры
партий
    min_cost = float('inf')

    #Перебираем все доступные тарифы
    for i in range(7):
        batch_size = batch_sizes[i]
        batch_cost = A[i]

        #Определяем необходимое количество партий
        num_batches = (N + batch_size - 1) // batch_size
        total_cost = num_batches * batch_cost

        min_cost = min(min_cost, total_cost)

    return min_cost



if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as file:
```

```
        N = int(file.readline().strip())
        A = [int(file.readline().strip()) for _ in range(7)]

    result = min_print(N, A)

    with open("../txtf/output.txt", "w") as file:
        file.write(str(result) + "\n")
```

Текстовое объяснение задачи:

Алгоритм решает задачу минимизации стоимости печати материала с учётом разных тарифов, предлагаемых для различных партий листов.

Алгоритм работает следующим образом:

1. Для каждого возможного тарифа (1 лист, 10 листов, 100 листов и так далее) вычисляется стоимость печати нужного количества листов, если использовать только этот тариф
2. Для каждого тарифа считается, сколько партий нужно заказать, и умножается на цену этой партии. Это даёт стоимость печати для этого тарифа
3. Среди всех полученных стоимостей выбирается минимальное значение

Тесты:

```
import unittest
from lab1.task9.src.main import min_print

class TestMinPrintCost(unittest.TestCase):

    def test_case_1(self):
        N = 980
        A = [1, 9, 90, 900, 1000, 10000, 10000]
        result = min_print(N, A)
        self.assertEqual(result, 882)

    def test_case_2(self):
        N = 980
        A = [1, 10, 100, 1000, 900, 10000, 10000]
        result = min_print(N, A)
        self.assertEqual(result, 900)

    '''минимальный случай'''
    def test_case_3(self):
        N = 1
        A = [5, 40, 300, 2500, 10000, 15000, 20000]
        result = min_print(N, A)
        self.assertEqual(result, 5)

    '''граничный случай между тарифами (n=10)'''
    def test_case_4(self):
        N = 10
        A = [2, 15, 120, 1100, 10500, 90000, 800000]
        result = min_print(N, A)
        self.assertEqual(result, 15)

    '''комбинация выгоднее'''
    def test_case_5(self):
```

```
        N = 15
        A = [2, 18, 100, 1000, 9000, 80000, 700000]
        result = min_print(N, A)
        self.assertEqual(result, 30)

    '''более крупная партия'''
    def test_case_6(self):
        N = 9999
        A = [1, 9, 80, 700, 6500, 50000, 400000]
        result = min_print(N, A)
        self.assertEqual(result, 6500)

    '''A1 всегда выгоднее'''
    def test_case_7(self):
        N = 500
        A = [1, 20, 250, 3000, 35000, 50000, 400000]
        result = min_print(N, A)
        self.assertEqual(result, 500)

    '''граничный случай n=1000000'''
    def test_case_8(self):
        N = 1000000
        A = [5, 40, 350, 2500, 20000, 150000, 900000]
        result = min_print(N, A)
        self.assertEqual(result, 900000)

    '''дешевле брать больше листов'''
    def test_case_9(self):
        N = 9500
        A = [2, 15, 140, 1200, 11000, 100000, 900000]
        result = min_print(N, A)
        self.assertEqual(result, 11000)

    def test_case_10(self):
        N = 5
        A = [1, 10, 100, 1000, 900, 10000, 10000]
        result = min_print(N, A)
        self.assertEqual(result, 5)


if __name__ == "__main__":
    unittest.main()
```

```
✔ Tests passed: 10 of 10 tests – 2 ms

C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pro
Testing started at 11:35 PM ...
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_projects



Ran 10 tests in 0.004s

OK
```

Вывод по задаче:

Алгоритм эффективно решает задачу выбора оптимального тарифа для печати, минимизируя затраты. Он работает за время O(1), так как количество тарифов фиксировано (7 тарифов), что делает его очень быстрым и подходящим для данной задачи

Задача №14. Максимальное значение арифметического выражения (2 балла)

Текст задачи:

В этой задаче ваша цель - добавить скобки к заданному арифметическому выражению, чтобы максимизировать его значение. max(5 − 8 + 7 × 4 − 8 + 9) =?

Найдите максимальное значение арифметического выражения, указав порядок применения его арифметических операций с помощью дополнительных скобок.

Листинг кода:

```
def max_expression_value(expression):
    nums = [int(expression[i]) for i in range(0, len(expression), 2)]
    ops = [expression[i] for i in range(1, len(expression), 2)]

    n = len(nums)

    #Инициализация DP таблиц
    dp_max = [[0] * n for _ in range(n)]
    dp_min = [[0] * n for _ in range(n)]

    #Инициализация для выражений длины 1
    for i in range(n):
        dp_max[i][i] = nums[i]
        dp_min[i][i] = nums[i]

    #Заполняем таблицы для всех подвыражений
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            dp_max[i][j] = float('-inf')
            dp_min[i][j] = float('inf')
            for k in range(i, j):
                op = ops[k]
                if op == '+':
                    dp_max[i][j] = max(dp_max[i][j], dp_max[i][k] + dp_max[k
+ 1][j])
                    dp_min[i][j] = min(dp_min[i][j], dp_min[i][k] + dp_min[k
+ 1][j])
                elif op == '-':
                    dp_max[i][j] = max(dp_max[i][j], dp_max[i][k] - dp_min[k
+ 1][j])
                    dp_min[i][j] = min(dp_min[i][j], dp_min[i][k] - dp_max[k
+ 1][j])
                elif op == '*':
                    dp_max[i][j] = max(dp_max[i][j], dp_max[i][k] * dp_max[k
+ 1][j])
                    dp_min[i][j] = min(dp_min[i][j], dp_min[i][k] * dp_min[k
+ 1][j])
```

```
    #Ответом будет максимальное значение на всем интервале от 0 до n-1
    return dp_max[0][n - 1]


if __name__ == '__main__':
    with open("../txtf/input.txt", "r") as file:
        expression = file.readline().strip()

    result = max_expression_value(expression)

    with open("../txtf/output.txt", "w") as file:
        file.write(str(result) + "\n")
```

Текстовое объяснение задачи:

Используется динамическое программирование (DP) для вычисления максимальных и минимальных значений выражений на каждом интервале. Алгоритм разбивает выражение на подвыражения, вычисляя для каждого интервала минимальное и максимальное возможное значение.

- Сначала создаются два массива DP: dp_max для хранения максимальных значений и dp_min для хранения минимальных значений подвыражений.
- Далее инициализируются таблицы для выражений длины 1, то есть для каждого числа в выражении.
- Для выражений длины 2 и больше заполняются таблицы, перебирая все возможные разбиения выражения и применяя операцию между числами (сложение, вычитание, умножение).
- В зависимости от операции выбираются соответствующие минимальные и максимальные значения для подвыражений.
- В конце алгоритм возвращает максимальное значение для всего выражения, находя его на интервале от 0 до n-1

Для каждого подвыражения рассматриваются все возможные разбиения и вычисляются максимальные и минимальные значения с учетом операций между подвыражениями. Операции обновляют значения для текущего подвыражения в зависимости от того, является ли операция сложением, вычитанием или умножением.

Тесты:

```
import unittest
from lab1.task14.src.main import max_expression_value

class MaxExpressionValueTest(unittest.TestCase):

    '''Тест из примера 1'''
    def test_case_1(self):
        expression = '1+5'
        result = max_expression_value(expression)
```

```
        self.assertEqual(result, 6)

    '''Тест из примера 2'''
    def test_case_2(self):
        expression = '5-8+7*4-8+9'
        result = max_expression_value(expression)
        self.assertEqual(result, 200)

    '''Одни скобки'''
    def test_case_3(self):
        expression = '3+2*5'
        result = max_expression_value(expression)
        self.assertEqual(result, 25)

    '''Две скобки'''
    def test_case_4(self):
        expression = '1+2*3+4'
        result = max_expression_value(expression)
        self.assertEqual(result, 21)

    def test_case_5(self):
        expression = '2+3*4-6+2*5'
        result = max_expression_value(expression)
        self.assertEqual(result, 80)


if __name__ == "__main__":
    unittest.main()
```

```
✓ Tests passed: 5 of 5 tests – 1 ms

C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pro
Testing started at 11:37 PM ...
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_projects



Ran 5 tests in 0.002s

OK
```

Вывод по задаче:

Алгоритм использует динамическое программирование для нахождения максимального значения арифметического выражения с расставленными скобками. Он разделяет выражение на подвыражения, вычисляя для каждого максимальные и минимальные значения в зависимости от операций. Для этого создаются две таблицы: одна для максимальных значений, другая для минимальных. Алгоритм перебирает все возможные разбиения выражения, и на основе текущих значений операций обновляет таблицы.

Временная сложность составляет O(n^3), где n количество чисел в выражении, так как для каждого подвыражения алгоритм проверяет все возможные разбиения, что даёт кубическую сложность.

Задача №15. Удаление скобок (2 балла)

Текст 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

Листинг кода:

```
def remove_invalid_brackets(s: str) -> str:
    stack = []
    to_remove = set()

    for i, char in enumerate(s):
        if char in '([{':
            stack.append((char, i))  # добавляем в стек с индексом
        elif char in ')]}':
            if stack and ((char == ')' and stack[-1][0] == '(') or
                          (char == ']' and stack[-1][0] == '[') or
                          (char == '}' and stack[-1][0] == '{')):
                stack.pop()  # находим пару, убираем из стека
            else:
                to_remove.add(i)  # добавляем в список на удаление

    # Добавляем оставшиеся символы из стека
    to_remove.update(i for _, i in stack)

    return ''.join(char for i, char in enumerate(s) if i not in to_remove)


if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as file:
        s = file.readline().strip()

    result = remove_invalid_brackets(s)

    with open("../txtf/output.txt", "w") as file:
        file.write(result + "\n")
```

Текстовое объяснение задачи:

Алгоритм решает задачу с помощью стека. Идея заключается в том, чтобы пройти по строке и использовать стек для поиска пар скобок. При этом символы, которые не образуют правильную пару, будут помечены для удаления.

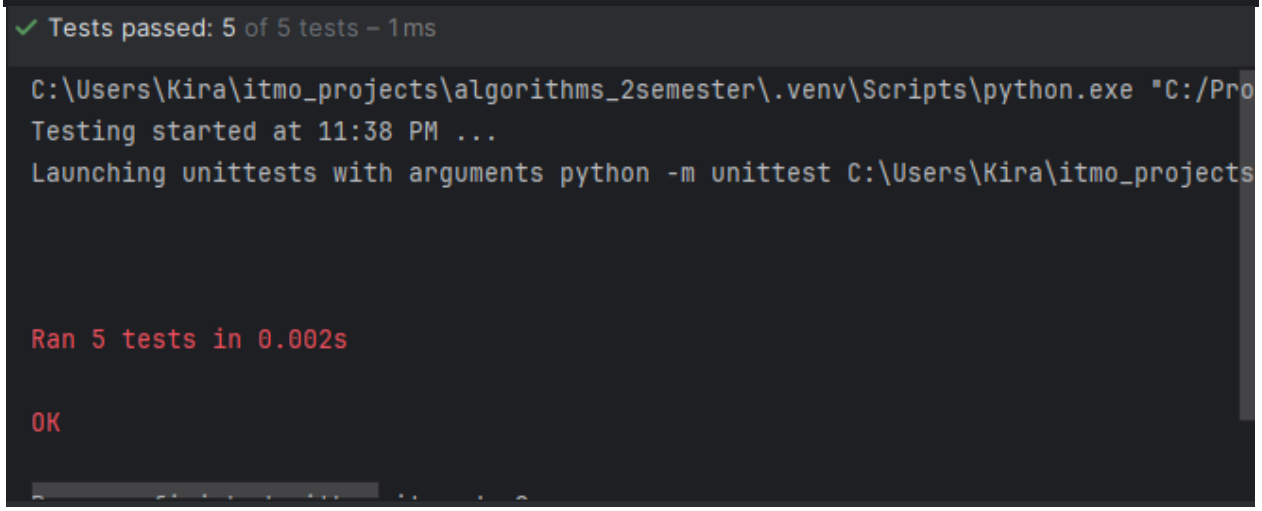
1. Инициализируем пустой стек и множество to_remove для хранения индексов символов, которые нужно удалить
2. Проходим по строке и для каждого символа: Если это открывающая скобка, добавляем её в стек с индексом. Если это закрывающая скобка, проверяем: если в стеке есть соответствующая открывающая скобка (т.е.

пара), удаляем её из стека. А если пары нет, добавляем текущий индекс в множество to_remove

3. После завершения обхода строки, все символы, которые остались в стеке (не закрытые), также добавляются в to_remove
4. Возвращаем строку, в которой исключены все символы, находящиеся в множестве to_remove

Тесты:

```python
import unittest
from lab1.task15.src.main import remove_invalid_brackets

class TestRemoveInvalidBrackets(unittest.TestCase):

    def test_example1(self):
        self.assertEqual(remove_invalid_brackets("()"), "()")

    def test_example2(self):
        self.assertEqual(remove_invalid_brackets("([)]"), "[]")

    def test_example3(self):
        self.assertEqual(remove_invalid_brackets("{[()]}"), "{[()]}")

    def test_example4(self):
        self.assertEqual(remove_invalid_brackets("{[("), "")

    def test_example5(self):
        self.assertEqual(remove_invalid_brackets("((()))"), "((()))")

if __name__ == '__main__':
    unittest.main()
```



Вывод по задаче:

Алгоритм корректно обрабатывает различные входные строки, удаляя все некорректные скобки. Этот алгоритм эффективно решает задачу с помощью стека и имеет линейную сложность O(n), где n длина строки

Задача №17. Ход конем (2.5 балла)

Текст 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.

Напишите программу, определяющую количество телефонных номеров длины N, набираемых ходом коня. Поскольку таких номеров может быть очень много, выведите ответ по модулю 109.

Листинг кода:

```
MOD = 1000000000

def horse(n):
    dp = [[0] * (n + 1) for _ in range(10)]  #инициализация массива dp
    for i in range(10):
        dp[i][1] = 1  #для длины 1

    #заполнение массива dp для всех длин от 2 до n
    for j in range(2, n + 1):
        for i in range(10):
            if i == 0:
                dp[0][j] = (dp[4][j - 1] + dp[6][j - 1]) % MOD
            elif i == 1:
                dp[1][j] = (dp[6][j - 1] + dp[8][j - 1]) % MOD
            elif i == 2:
                dp[2][j] = (dp[9][j - 1] + dp[7][j - 1]) % MOD
            elif i == 3:
                dp[3][j] = (dp[8][j - 1] + dp[4][j - 1]) % MOD
            elif i == 4:
                dp[4][j] = (dp[0][j - 1] + dp[3][j - 1] + dp[9][j - 1]) % MOD
            elif i == 6:
                dp[6][j] = (dp[0][j - 1] + dp[1][j - 1] + dp[7][j - 1]) % MOD
            elif i == 7:
                dp[7][j] = (dp[6][j - 1] + dp[2][j - 1]) % MOD
            elif i == 8:
                dp[8][j] = (dp[1][j - 1] + dp[3][j - 1]) % MOD
            elif i == 9:
                dp[9][j] = (dp[2][j - 1] + dp[4][j - 1]) % MOD

    #подсчитываем сумму всех возможных номеров длины n
    result = 0
    for i in range(1, 10):
        if i != 8:  # Исключаем 8
            result = (result + dp[i][n]) % MOD

    return result

if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as file:
        n = int(file.readline().strip())

    result = horse(n)

    with open("../txtf/output.txt", "w") as file:
        file.write(str(result))
```

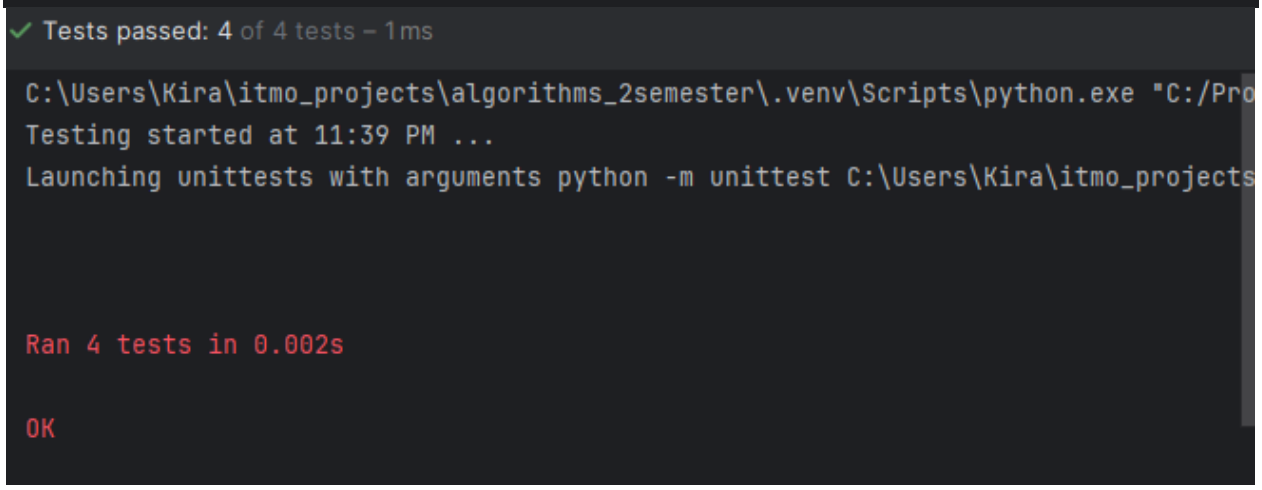
Текстовое объяснение задачи:

Задача требует подсчёта количества телефонных номеров длины N, которые можно набрать "ходом коня" на кнопочном телефоне.

Алгоритм решения задачи использует динамическое программирование. Я создаю массив dp, где dp[i][j] хранит количество номеров длины j, заканчивающихся на цифре i. Для каждого шага (для каждого числа длины от 2 до N) я вычисляю, на какие цифры можно перейти из каждой цифры предыдущей длины.

Тесты:

```python
import unittest
from lab1.task17.src.main import horse

class TestHorse(unittest.TestCase):
    def test_case_1(self):
        self.assertEqual(horse(1), 8)

    def test_case_2(self):
        self.assertEqual(horse(2), 16)

    def test_case_3(self):
        self.assertEqual(horse(3), 36)

    def test_case_4(self):
        self.assertEqual(horse(4), 82)

if __name__ == "__main__":
    unittest.main()
```



Вывод по задаче:

Алгоритм решает задачу с использованием динамического программирования, эффективно вычисляя возможные комбинации телефонных номеров длиной N, которые могут быть набраны "ходом коня". Алгоритм обрабатывает переходы между цифрами с учётом правил ходов и исключает числа, начинающиеся с цифры 8

Решение работает за время O(n 10), что является достаточно быстрым для решения задачи при больших значениях N.

Задача №19. Произведение матриц (3 балла)

Текст задачи:

В произведении последовательности матриц полностью расставлены скобки, если выполняется один из следующих пунктов:

– Произведение состоит из одной матрицы.

– Оно является заключенным в скобки произведением двух произведений с полностью расставленными скобками.

Полная расстановка скобок называется оптимальной, если количество операций, требуемых для вычисления произведения, минимально. Требуется найти оптимальную расстановку скобок в произведении последовательности матриц.

Листинг кода:

```
def matrix_chain_order(dimensions):
    n = len(dimensions) - 1  #количество матриц
    dp = [[0] * n for _ in range(n)]  #массив для хранения минимальных
операций
    s = [[0] * n for _ in range(n)]  #массив для хранения мест разделения
(где ставить скобки)


    for length in range(2, n + 1):  #длина цепочки матриц от 2 до n
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                q = dp[i][k] + dp[k + 1][j] + dimensions[i] * dimensions[k +
1] * dimensions[j + 1]
                if q < dp[i][j]:
                    dp[i][j] = q
                    s[i][j] = k

    #рекурсивная функция для построения скобочной структуры
    def build_order(s, i, j):
        if i == j:
            return 'A'  #вывод только букв A
            # return chr(ord('A') + i)  #обозначение каждой матрици своей
буквой
        k = s[i][j]
        left = build_order(s, i, k)
        right = build_order(s, k + 1, j)
        return f"({left}{right})"  #скобки вокруг левого и правого
подвыражений

    return build_order(s, 0, n - 1)


if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as file:
        n = int(file.readline().strip())
        dimensions = []
```

```
        for i in range(n):
            ai, bi = map(int, file.readline().split())
            dimensions.append(ai)
        dimensions.append(bi)

    result = matrix_chain_order(dimensions)

    with open("../txtf/output.txt", "w") as file:
        file.write(result)
```

Текстовое объяснение задачи:

Задача решается с помощью динамического программирования. Основная цель это найти оптимальную расстановку скобок для произведения матриц, минимизируя количество операций умножения

Функция *matrix_chain_order(dimensions):*

1. Принимает список dimensions, который содержит размеры матриц.
2. Инициализирует два двумерных массива dp и s. Где dp[i][j] минимальное количество операций умножения для умножения матриц с индексами от i до j. A [i][j] — индекс разбиения для оптимального умножения матриц в диапазоне от i до j
3. Для каждой возможной длины цепочки матриц от 2 до `n`, для каждой подцепочки определяется минимальное количество операций и место для разбиения

Функция *build_order(s, i, j).* Эта функция строит строку с оптимальной расстановкой скобок. Если это одна матрица, возвращается символ "A". Если это несколько матриц, функция рекурсивно строит левую и правую части выражения и добавляет скобки вокруг них.

Тесты:

```
import unittest
from lab1.task19.src.main import matrix_chain_order


class TestMatrixChainOrder(unittest.TestCase):
    """пример из условия задачи"""
    def test_case_1(self):
        dimensions = [10, 50, 90, 20]
        expected = "((AA)A)"
        result = matrix_chain_order(dimensions)
        self.assertEqual(result, expected)

    def test_case_2(self):
        dimensions = [10, 20, 30] #(10, 20, 20, 30)
        expected = "(AA)"
        result = matrix_chain_order(dimensions)
        self.assertEqual(result, expected)

    '''одна матрица'''
    def test_case_3(self):
```

```
        dimensions = [10, 20]
        expected = "A"
        result = matrix_chain_order(dimensions)
        self.assertEqual(result, expected)

if __name__ == '__main__':
    unittest.main()
```

```
✓ Tests passed: 3 of 3 tests – 1 ms

C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pr
Testing started at 11:40 PM ...
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_project


Ran 3 tests in 0.002s

OK
```

Вывод по задаче:

Задача решена с помощью динамического программирования для поиска оптимальной расстановки скобок в произведении последовательности матриц. Основной идеей решения является минимизация количества операций умножения матриц, что достигается через учет всех возможных вариантов разбиения цепочки матриц и выбор оптимального

Алгоритм эффективно находит минимальное количество операций, используя двумерные массивы для хранения промежуточных результатов и рекурсивно строит оптимальную расстановку скобок. Время работы алгоритма составляет O(n^3)

Задача №21. Игра в дурака (3 балла)

Текст задачи:

Петя очень любит программировать. Недавно он решил реализовать популярную карточную игру «Дурак». Но у Пети пока маловато опыта, ему срочно нужна Ваша помощь. Как известно, в «Дурака» играют колодой из 36 карт. В Петиной программе каждая карта представляется в виде строки из двух символов, где первый символ означает ранг (‘6’, ‘7’, ‘8’, ‘9’, ‘T’, ‘J’, ‘Q’, ‘K’, ‘A’) карты, а второй символ означает масть (‘S’, ‘C’, ‘D’, ‘H’). Ранги перечислены в порядке возрастания старшинства.

Пете необходимо решить следующую задачу: сможет ли игрок, обладая набором из N карт, отбить M карт, которыми под него сделан ход? Для того чтобы отбиться, игроку нужно покрыть каждую из карт, которыми под него сделан ход, картой из своей колоды. Карту можно покрыть либо старшей

картой той же масти, либо картой козырной масти. Если кроющаяся карта сама является козырной, то её можно покрыть только старшим козырем. Одной картой можно покрыть только одну карту.

Листинг кода:

```
import time

def can_defend(N, M, trump_suit, hand_cards, attack_cards):
    #Функция для преобразования карты в пару (ранг, масть)
    def parse_card(card):
        return card[0], card[1]

    hand_cards = [parse_card(card) for card in hand_cards]
    attack_cards = [parse_card(card) for card in attack_cards]

    #Разделяем карты на козырные и некозырные
    trump_cards = [card for card in hand_cards if card[1] == trump_suit]
    non_trump_cards = [card for card in hand_cards if card[1] != trump_suit]

    #Сортируем карты по рангу
    rank_order = ['6', '7', '8', '9', 'T', 'J', 'Q', 'K', 'A']
    rank_to_value = {rank: idx for idx, rank in enumerate(rank_order)}

    def card_value(card):
        rank, suit = card
        return rank_to_value[rank]

    trump_cards.sort(key=card_value)
    non_trump_cards.sort(key=card_value)

    #Для каждой атакующей карты проверяем, можем ли мы ее отбить
    for attack_card in attack_cards:
        attack_rank, attack_suit = attack_card

        covered = False

        if attack_suit == trump_suit:
            #Если атакующая карта козырная она может быть покрыта только
старшим козырем
            for i, card in enumerate(trump_cards):
                hand_rank, hand_suit = card
                if card_value(card) > card_value(attack_card):
                    trump_cards.pop(i)
                    covered = True
                    break
        else:
            #Если атакующая карта не козырная
            #Попробуем покрыть картой той же масти, если старше
            for i, card in enumerate(non_trump_cards):
                hand_rank, hand_suit = card
                if attack_suit == hand_suit and card_value(card) >
card_value(attack_card):
                    non_trump_cards.pop(i)
                    covered = True
                    break

            if not covered:
                #Попробуем покрыть козырной картой
                for i, card in enumerate(trump_cards):
                    hand_rank, hand_suit = card
                    if card_value(card) > card_value(attack_card):
```

```
                        trump_cards.pop(i)
                        covered = True
                        break

        if not covered:
            return "NO"  #Если не нашли, чем покрыть, возвращаем "NO"

    return "YES"  #Если все карты отбиты


if __name__ == "__main__":
    start_time = time.time()

    with open("../txtf/input.txt", "r") as file:
        N, M, trump_suit = file.readline().split()
        N, M = int(N), int(M)
        hand_cards = file.readline().split()
        attack_cards = file.readline().split()

    result = can_defend(N, M, trump_suit, hand_cards, attack_cards)

    with open("../txtf/output.txt", "w") as file:
        file.write(result + "\n")

    end_time = time.time()
    print(f'Время выполнения: {end_time - start_time} секунд')
```

Текстовое объяснение задачи:

Мой код решает эту задачу следующим образом:

1. Мы начинаем с того, что преобразуем каждую карту в пару (ранг, масть), используя вспомогательную функцию parse_card.
2. Разделяем карты на козырные и некозырные.
3. Сортируем карты по возрастанию их силы с помощью словаря rank_to_value, который отображает каждый ранг на его индекс в порядке возрастания.
4. Для каждой атакующей карты пытаемся найти подходящую карту из своей колоды для защиты:

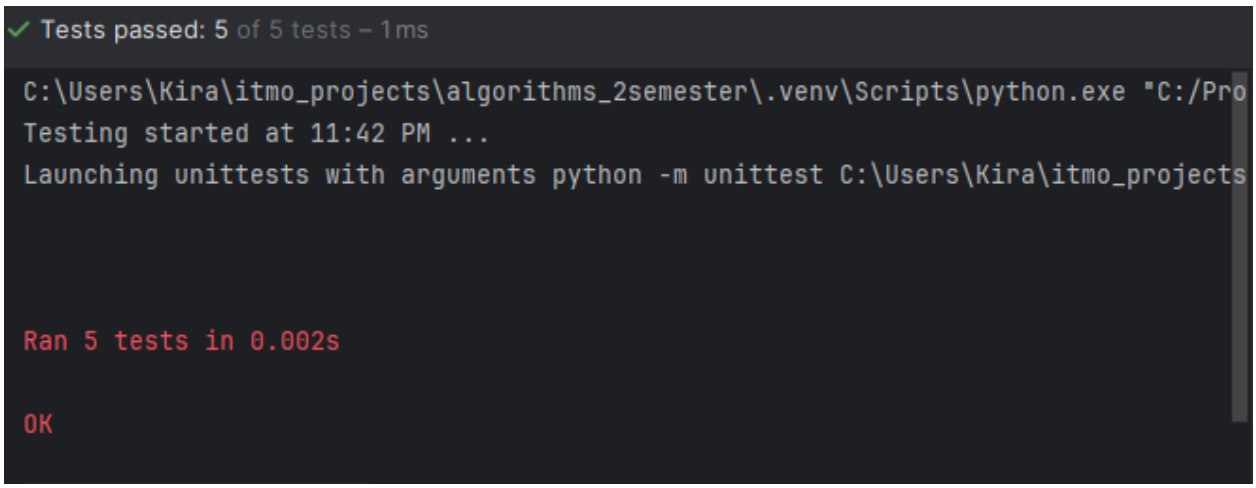
    - Если атакующая карта козырная, она может быть покрыта только старшим козырем.
    - Если атакующая карта не козырная, пытаемся покрыть её картой той же масти, если она старше, или козырной картой, если таковой нет.

5. Если для каждой атакующей карты найдено подходящее покрытие, возвращается "YES", иначе "NO"

Тесты:

```
import unittest
from lab1.task21.src.main import can_defend
```

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

    '''Тест из примера 1'''
    def test_case_1(self):
        N = 6
        M = 2
        trump_suit = 'C'
        hand_cards = ['KD', 'KC', 'AD', '7C', 'AH', '9C']
        attack_cards = ['6D', '6C']
        result = can_defend(N, M, trump_suit, hand_cards, attack_cards)
        self.assertEqual(result, "YES")

    '''Тест из примера 2'''
    def test_case_2(self):
        N = 4
        M = 1
        trump_suit = 'D'
        hand_cards = ['9S', 'KC', 'AH', '7D']
        attack_cards = ['8D']
        result = can_defend(N, M, trump_suit, hand_cards, attack_cards)
        self.assertEqual(result, "NO")

    '''Игрок может покрыть все атакующие карты'''
    def test_case_3(self):
        N = 5
        M = 3
        trump_suit = 'S'
        hand_cards = ['6H', '7C', '9S', 'KH', 'QS']
        attack_cards = ['6S', '7S', '8S']
        result = can_defend(N, M, trump_suit, hand_cards, attack_cards)
        self.assertEqual(result, "NO")

    def test_case_4(self):
        N = 4
        M = 4
        trump_suit = 'D'
        hand_cards = ['6S', '7C', '9H', 'KH']
        attack_cards = ['6S', '7D', '8D', 'KD']
        result = can_defend(N, M, trump_suit, hand_cards, attack_cards)
        self.assertEqual(result, "NO")

    def test_case_5(self):
        N = 6
        M = 2
        trump_suit = 'S'
        hand_cards = ['6S', '7S', '9H', 'KH', 'QS', 'AH']
        attack_cards = ['6H', '7H']
        result = can_defend(N, M, trump_suit, hand_cards, attack_cards)
        self.assertEqual(result, "YES")


if __name__ == "__main__":
    unittest.main()
```



Для алгоритма были написаны юнит-тесты, которые проверяют различные случаи игры:

Тест 1: когда игрок может отбить все атакующие карты с помощью карт из своей колоды.

Тест 2: когда игрок не может отбить хотя бы одну атакующую карту.

Тест 3: когда все карты в руке могут быть использованы для защиты.

Тест 4: когда игрок не может защититься от всех атакующих карт. Тесты демонстрируют, что алгоритм корректно обрабатывает различные сценарии, включая минимальные и максимальные размеры входных данных

Вывод по задаче:
Алгоритм решает задачу с использованием сортировки карт и поиска соответствующих карт для защиты. Он эффективно и правильно обрабатывает различные варианты игры, включая козырные и некозырные карты. Время выполнения зависит от количества карт и атакующих карт, и алгоритм работает за время O(M * N), что подходит для решения задачи в рамках поставленных ограничений

Задача №22. Симпатичные узоры (4 балла)
Текст задачи:
Компания BrokenTiles планирует заняться выкладыванием во дворах у состоятельных клиентов узор из черных и белых плиток, каждая из которых имеет размер 1×1 метр. Известно, что дворы всех состоятельных людей имеют наиболее модную на сегодня форму прямоугольника M × N метров. Однако

при составлении финансового плана у директора этой организации появилось целых две серьезных проблемы: во первых, каждый новый клиент очевидно захочет, чтобы узор, выложенный у него во дворе, отличался от узоров всех остальных клиентов этой фирмы, а во вторых, этот узор должен быть симпатичным. Как показало исследование, узор является симпатичным, если в нем нигде не встречается квадрата 2×2 метра, полностью покрытого плитками одного цвета. Для составления финансового плана директору необходимо узнать, сколько клиентов он сможет обслужить, прежде чем симпатичные узоры данного размера закончатся. Помогите ему!

Листинг кода:

```
def cute_patterns(x, y, n):
    '''Функция проверяет можно ли разместить плитки
    на текущие строки (x, y): нигде не должно быть 2x2 одного цвета
    '''

    b = [0] * 5  #Вспомогательный массив для хранения значений плиток

    for i in range(n - 1):
        #Определяем цвета плиток в текущем и следующем столбцах для двух
строк
        b[1] = 0 if (x & (1 << i)) == 0 else 1
        b[2] = 0 if (x & (1 << (i + 1))) == 0 else 1
        b[3] = 0 if (y & (1 << i)) == 0 else 1
        b[4] = 0 if (y & (1 << (i + 1))) == 0 else 1

        #Если четыре плитки образуют квадрат одного цвета то False
        if b[1] == b[2] == b[3] == b[4]:
            return False

    return True

def cute_patterns_main(n, m):
    '''Основная функция для подсчета количества возможных узоров'''
    res = 0
    length = 1 << n  #Количество возможных строк (2^n)
    #Возможные переходы
    a = [[0] * length for _ in range(m)]  #Количество допустимых узоров на
каждом шаге
    dp = [[0] * length for _ in range(length)]  #Матрица допустимых переходов
между строками

    #Заполняем dp: можно ли переходить между строками i и j
    for i in range(length):
        for j in range(length):
            dp[i][j] = 1 if cute_patterns(i, j, n) else 0

    #Инициализируем базовый случай: первая строка может быть любой
    for i in range(length):
        a[0][i] = 1

    #Заполняем массив a
    for x in range(1, m):
        for i in range(length):
            for j in range(length):
                a[x][i] += a[x - 1][j] * dp[j][i]
```

```
    for i in range(length):
        res += a[m - 1][i]

    return res

if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as f:
        n, m = map(int, f.readline().split())

    if n > m:
        n, m = m, n

    result = cute_patterns_main(n, m)

    with open("../txtf/output.txt", "w") as f:
        f.write(str(result))
```

Текстовое объяснение задачи:

Алгоритм решает задачу методом динамического программирования, перебирая все возможные состояния строк плиток и проверяя, могут ли две строки быть соседними, не образуя квадрата 2×2 одинакового цвета. Для этого используется вспомогательная функция *cute_patterns*, которая проверяет, можно ли размещать плитки в строках (x, y), чтобы не нарушить симпатичность узора

Функция *cute_patterns*. Эта функция проверяет, можно ли разместить плитки на текущих строках (x и y), чтобы в их пересечении не образовывался квадрат 2×2, полностью покрытый плитками одного цвета. Строки представляют собой двоичные числа, где 0 соответствует белой плитке, а 1 черной. Для каждой пары строк (x, y) функция проверяет все соседние плитки на двух строках, сравнивая их значения. Если в какой-либо момент четыре плитки (две из первой строки и две из второй) образуют квадрат одного цвета (например, все плитки черные или все белые), функция возвращает False, иначе True

Функция *cute_patterns_main*. В этой функции происходит основная работа по подсчету количества симпатичных узоров. Создается массив всех возможных строк (каждая строка — это двоичное число длиной n). Строится матрица переходов (dp). Для каждой строки в столбце массива a подсчитывается количество допустимых узоров, используя динамическое программирование

Тесты:

```
import unittest
from lab1.task22.src.main import cute_patterns_main, cute_patterns

class TestNicePatterns(unittest.TestCase):
    def test_case_1(self):
        """Минимальный случай 1x1
        поле можно покрыть только двумя способами"""
```

```
        self.assertEqual(cute_patterns_main(1, 1), 2)

    def test_case_2(self):
        """Поле 2x2
        2^4=16 Минус два узора (все черные и все белые)"""
        self.assertEqual(cute_patterns_main(2, 2), 14)

    def test_case_3(self):
        """Длинный узкий двор 1x4
        каждая клетка может быть либо белой либо черной, запрещенных нет.
2^4=16"""
        self.assertEqual(cute_patterns_main(1, 4), 16)

    def test_case_4(self):
        """двор 4x1"""
        self.assertEqual(cute_patterns_main(4, 1), 16)

    def test_case_5(self):
        """Прямоугольное поле 3x3"""
        self.assertEqual(cute_patterns_main(3, 3), 322)


if __name__ == "__main__":
    unittest.main()
```

```
✓ Tests passed: 5 of 5 tests – 1 ms

C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pro
Testing started at 11:43 PM ...
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_projects



Ran 5 tests in 0.003s

OK
```

Для алгоритма мною были написаны юнит-тесты, которые проверяют различные случаи применения функции подсчета симпатичных узоров на дворе. Тесты охватывают как минимальные случаи (например, поле 1x1), так и более сложные прямоугольные формы (например, 3x3). Все тесты обеспечивают корректность работы алгоритма, проверяя правильность подсчета возможных симпатичных узоров для разных размеров дворов

Вывод по задаче:

Алгоритм динамического программирования позволяет эффективно подсчитать количество возможных симпатичных узоров на дворе размером M×N. Несмотря на значительное количество возможных конфигураций (2^n для каждой строки), использование матрицы переходов и динамическое

программирование значительно ускоряет процесс подсчета, что позволяет решать задачу для дворов размером до 10×10

Временная сложность:

Алгоритм решает задачу методом динамического программирования, где:

- для каждой строки можно сформировать 2^n возможных состояний, где n это количество плиток в строке
- для каждого перехода между строками необходимо проверять все возможные пары строк, что в худшем случае требует O((2^n)^2) операций
- Динамическое программирование для подсчета всех возможных узоров выполняется за O(m * (2^n)^2), где m — количество строк (сторон в прямоугольном дворе)

Таким образом, временная сложность алгоритма составляет **O(m * (2^n)^2)**, что может быть достаточно эффективно для размеров n до 10

Вывод

В ходе выполнения лабораторной работы я изучил и применил жадные алгоритмы и динамическое программирование для решения различных задач. Жадные алгоритмы оказались эффективными в задачах, где можно принимать локально оптимальные решения, как, например, в задаче о дробном рюкзаке. Я использовал сортировку по убыванию ценности на единицу веса для нахождения максимальной прибыли, что позволило получить решение с высокой скоростью

Для задач, где необходимо учитывать зависимости между шагами, я применил динамическое программирование. Например, в задаче о максимальной стоимости добычи оно бы позволило найти решение с учетом всех возможных вариантов

Также я разобрался, как обрабатывать крайние случаи, например, когда данные содержат нулевые значения, и как адаптировать алгоритмы для специфических условий. В результате работы я понял, как эффективно решать задачи с различными структурами данных и требованиями к скорости