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

Отчет по лабораторной работе №1

по курсу «Алгоритмы и структуры данных»

Тема: Жадные алгоритмы. Динамическое программирование №2

Вариант 4

Выполнила:

Буй Тхук Хуен - К3139

Проверила:

Петросян А.М.

Санкт-Петербург

2024 г.

Содержание отчета

I. Задачи по варианту

Задача №2. Заправки

Вы собираетесь поехать в другой город, расположенный в d км от вашего родного города. Ваш автомобиль может проехать не более m км на полном баке, и вы начинаете с полным баком. По пути есть заправочные станции на расстояниях $stop_1, stop_2, ..., stop_n$ из вашего родного города. Какое минимальное количество заправок необходимо?

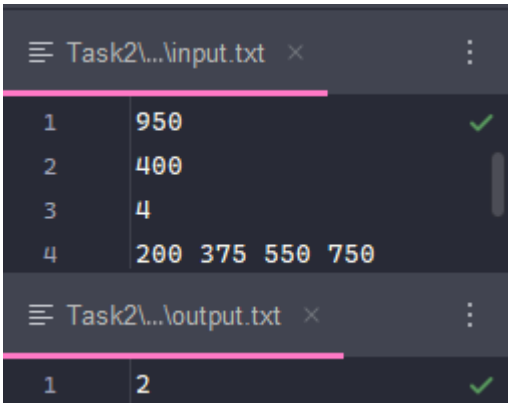
- **Формат ввода / входного файла (input.txt).** В первой строке содержится d - целое число. Во втройо строке - целое число m. В третьей находится количество заправок на пути - n. И, наконец, в последней строке - целые числа через пробел - остановки $stop_1, stop_2, ..., stop_n$.
- **Ограничения на входные данные.** $1 \leq d \leq 10^5$, $1 \leq m \leq 400$, $1 \leq n \leq 300$, $1 < stop_1 < stop_2 < ... < stop_n < d$
- **Формат вывода / выходного файла (output.txt).** Предполагая, что расстояние между городами составляет d км, автомобиль может проехать не более m км на полном баке, а заправки есть на расстояниях $stop_1, stop_2, ..., stop_n$ по пути, *выведите минимально необходимое количество заправок*. Предположим, что машина начинает ехать с полным баком. Если до места назначения добраться невозможно, выведите -1.
- Ограничение по времени. 2 сек.

- Листинг кода

```python
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def min_refuels(d, m, stops):
    # Добавляем начальную и конечную точки
    stops = [0] + stops + [d]
    refuels = 0
    last_refuel_position = 0

    for i in range(1, len(stops)):
        # Проверяем расстояние до следующей остановки
        if stops[i] - stops[i - 1] > m:
            return -1

        if stops[i] - last_refuel_position > m:
            last_refuel_position = stops[i - 1]
            refuels += 1

            # Проверяем снова возможность добраться до следующей остановки
            if stops[i] - last_refuel_position > m:
                return -1
```

```
    return refuels

def task2():
    d, m, stops = read(PATH_INPUT, 2)
    result = min_refuels(d, m, stops)
    write(PATH_OUTPUT, result, 2)

if __name__ == "__main__":
    task2()
```

- Текстовое объяснение решения
  + Добавляются начальная точка (0) и конечная точка (d) к списку остановок: stops = + stops + [d].
  + refuels: счётчик дозаправок, изначально равен 0.
  + last_refuel_position: позиция последней дозаправки, изначально 0.
  + Для каждой остановки проверяется, можно ли до неё добраться с текущим запасом топлива:
    - Если расстояние между двумя соседними остановками больше запаса хода m, возвращается -1 (невозможно доехать).
    - Если расстояние от последней дозаправки до текущей остановки превышает запас хода m, выполняется дозаправка на предыдущей остановке (last_refuel_position = stops[i - 1]), и счётчик refuels увеличивается.
  + Если даже после дозаправки расстояние до следующей остановки всё ещё больше, чем запас хода, возвращается -1.
  + Функция возвращает минимальное количество дозаправок (refuels), необходимых для достижения конечной точки.
- Результат работы код на примерах из текста задачи:



- Test:

```
Test Case: test_case_long_distance
Execution Time = 0.00057380 s, Memory Usage = 13.7578 KB

Test Case: test_example_1
Execution Time = 0.00000820 s, Memory Usage = 0.1562 KB

Test Case: test_example_2
Execution Time = 0.00000560 s, Memory Usage = 0.1562 KB

Test Case: test_example_3
Execution Time = 0.00000530 s, Memory Usage = 0.1250 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
100000 400 250 list(range(400,100001, 400))	0.00057380	13.7578
950 400 4 200 375 550 750	0.00000820	0.1562
10 3 4 1 2 5 9	0.00000560	0.1562
200 250 2 100 150	0.00000530	0.1250

- Вывод

  + Алгоритм эффективно находит минимальное количество дозаправок, необходимых для достижения конечной точки, учитывая ограничения на запас хода. Решение основано на жадном подходе: если

невозможно добраться до следующей остановки без дозаправки, выбирается последняя возможная точка для заправки.

+ Сложность алгоритма составляет O(n)

Задача №7. Проблема сапожника

- **Постановка 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.
- **Формат ввода / входного файла (input.txt).** В первой строке вводятся натуральные числа K и n. Затем во второй строке идет n натуральных чисел $t_1, ...t_n$ - количество минут, которые требуются, чтобы починить i-й сапог.
- **Ограничения на входные данные.** $1 \leq K \leq 1000, 1 \leq n \leq 500, 1 \leq t_i \leq 100$ для всех $1 \leq i \leq n$
- **Формат вывода / выходного файла (output.txt).** Выведите одно число – максимальное количество сапог, которые можно починить за один рабочий день.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

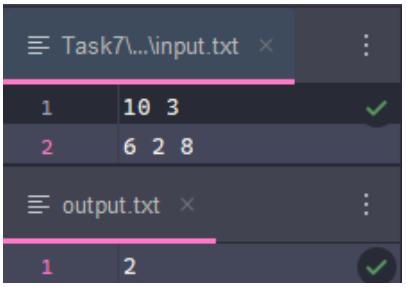
- Листинг кода

```
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def max_repaired_boots(K, repair_time):
    repair_time.sort()
    total_time = 0
    repaired = 0

    for time in repair_time:
        if total_time + time <= K:
            total_time += time
            repaired += 1

        else:
            break
```

```
    return repaired

def task7():
    K, repair_time = read(PATH_INPUT, 7)
    result = max_repaired_boots(K, repair_time)
    write(PATH_OUTPUT, result, 7)

if __name__ == "__main__":
    task7()
```

- Текстовое объяснение решения:
  + Сначала список **repair_time** сортируется в порядке возрастания. Это позволяет сначала чинить ту обувь, которая требует наименьшего времени, чтобы максимально эффективно использовать доступное время K.
  + **total_time** общее потраченное время.
  + **repaired** количество подчиненной обуви.
  + Для каждого элемента **time** из списка **repair_time** проверяется, можно ли добавить время ремонта этой пары обуви к текущему общему времени (**total_time**) без превышения лимита K.
  + Если да, то:
    - Время ремонта добавляется к **total_time**.
    - Счётчик подчиненной обуви (**repaired**) увеличивается на 1.
  + Если нет (времени уже не хватает), цикл прерывается с помощью **break**.
  + Функция возвращает общее количество пар обуви, которое удалось починить за время K.

- Результат работы код на примерах из текста задачи:



- Test

```
Test Case: test_1
Execution Time = 0.00000550 s, Memory Usage = 0.0938 KB

Test Case: test_2
Execution Time = 0.00000330 s, Memory Usage = 0.0781 KB

Test Case: test_3
Execution Time = 0.00000320 s, Memory Usage = 0.1094 KB

Test Case: test_4
Execution Time = 0.00014200 s, Memory Usage = 11.7969 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
10 3 6 2 8	0.00000550	0.0938
3 2 10 20	0.00000330	0.0781
100 5 1 2 3 4 5	0.00000320	0.1094
1000 500 range(1, 501)	0.00014200	11.7969

- Вывод

  + Сложность алгоритма составляет O(N log N) из-за этапа сортировки списка repair_time, за которым следуют O(N) циклов для перебора элементов.
  + При использовании жадного алгоритма оптимальный результат находится без перебора всех возможностей (как при динамическом программировании), алгоритм отдает приоритет ремонту обуви, на который уходит меньше всего времени, помогая максимально эффективно использовать временной бюджет K и увеличить количество отремонтированной обуви.

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.

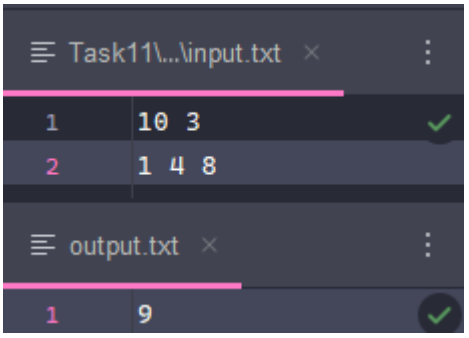
- Листинг кода

```
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def max_gold(W, n, weights):
    if W == 0 or n == 0:
        return 0

    # Создаем двумерный массив dp размером (n+1) x (W+1)
    # dp[i][w] будет хранить максимальный вес золота, который
можно набрать
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        # Перебираем все возможные вместимости сумки от 1 до W
        for w in range(1, W + 1):
            if weights[i-1] <= w:
                dp[i][w] = max( dp[i-1][w],
dp[i-1][w-weights[i-1]] + weights[i-1])

            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][w]
```

```python
def task11():
    W, n, weights = read(PATH_INPUT, 11)
    result = max_gold(W, n, weights)
    write(PATH_OUTPUT, result, 11)

if __name__ == "__main__":
    task11()
```

- Текстовое объяснение решения
  + Создается двумерная матрица dp, где dp[i][w] будет хранить максимальный вес золота, который можно набрать
  + Внешний цикл перебирает все предметы, внутренний - все возможные вместимости рюкзака.
  + Если текущий предмет помещается в рюкзак (weights[i-1] <= w), выбирается максимум из двух вариантов:
    - Не брать текущий предмет (dp[i-1][w])
    - Взять текущий предмет (dp[i-1][w-weights[i-1]] + weights[i-1])
  + Если предмет не помещается, значение остается таким же, как без этого предмета.
  + Возвращается значение в правом нижнем углу матрицы, которое представляет максимальный вес золота, который можно унести в рюкзаке вместимостью W, используя все n предметов.

- Результат работы код на примерах из текста задачи



- Test

```
Test Case: test_1
Execution Time = 0.00004080 s, Memory Usage = 0.6719 KB

Test Case: test_2
Execution Time = 0.00008070 s, Memory Usage = 0.8281 KB

Test Case: test_3
Execution Time = 0.00022230 s, Memory Usage = 3.5469 KB
```

```
Test Case: test_4
OK

Execution Time = 9.94371370 s, Memory Usage = 47682.6172 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
10 3 1 4 8	0.00004080	0.6719
15 4 2 3 4 5	0.00008070	0.8281
100 3 1 2 98	0.00022230	3.5469
10000 300 range(1, 301)	9.94371370	47682.6172

- Вывод
  + В данной задаче используется метод динамического программирования для решения задачи о рюкзаке. Создается двумерная матрица dp, где dp[i][ɯ] хранит максимальный вес золота, который можно набрать, используя первые i предметов и рюкзак вместимостью ɯ.
  + Временная сложность алгоритма составляет O(n * W), где n — количество предметов, а W — вместимость рюкзака.

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.

- Листинг кода

```
import re
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def evaluate(a, b, op):
    if op == '+':
        return a + b
    elif op == '-':
        return a - b
    elif op == '*':
        return a * b
    return 0

def max_value(exp):

    # Используем регулярное выражение для разделения чисел и
операторов
    tokens = re.findall(r'\-?\d+|\+|\-|\*|\/|\(|\)', exp)
```

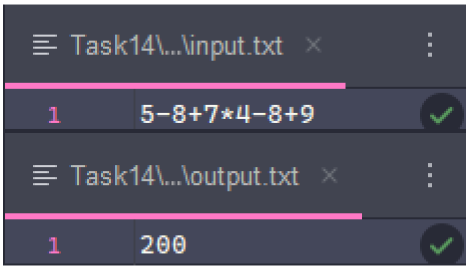
```
    #Проверяем корректность выражения
    if not tokens:
        raise ValueError("Invalid expression: empty or no valid
tokens")

    # Если выражение содержит только одно число
    if len(tokens) == 1 and tokens[0].lstrip('-').isdigit():
        return int(tokens[0])

    # Преобразовать токены в список чисел и операторов
    digits = []
    operators = []
    for token in tokens:
        if token.lstrip('-').isdigit():
            digits.append(int(token))
        elif token in "+-*/":
            operators.append(token)

    if len(operators) != len(digits) - 1:
        raise ValueError("Invalid expression: mismatch between
numbers and operators")

    n = len(digits)
    # Инициализируем таблицу для хранения максимальных и
минимальных значений
    min_val = [[0] * n for _ in range(n)]
    max_val = [[0] * n for _ in range(n)]

    for i in range(n):
        min_val[i][i] = digits[i]
        max_val[i][i] = digits[i]

    # Перебрать все подпоследовательности
    for s in range(1, n):
        for i in range(n - s):
            j = i + s
            min_val[i][j], max_val[i][j] = float('inf'),
float('-inf')

            for k in range(i, j):
                if k >= len(operators):  # Избегаем доступа к
ошибкам вне массива
                    continue

                op = operators[k]
                a = evaluate(max_val[i][k], max_val[k + 1][j],
op)
                b = evaluate(max_val[i][k], min_val[k + 1][j],
op)
```

```
                c = evaluate(min_val[i][k], max_val[k + 1][j],
op)
                d = evaluate(min_val[i][k], min_val[k + 1][j],
op)

                # Обновить минимальные и максимальные значения
                min_val[i][j] = min(min_val[i][j], a, b, c, d)
                max_val[i][j] = max(max_val[i][j], a, b, c, d)

    return max_val[0][n - 1]

def task14():
    expression = read(PATH_INPUT, 14)
    result = max_value(expression)
    write(PATH_OUTPUT, result, 14)

if __name__ == "__main__":
    task14()
```

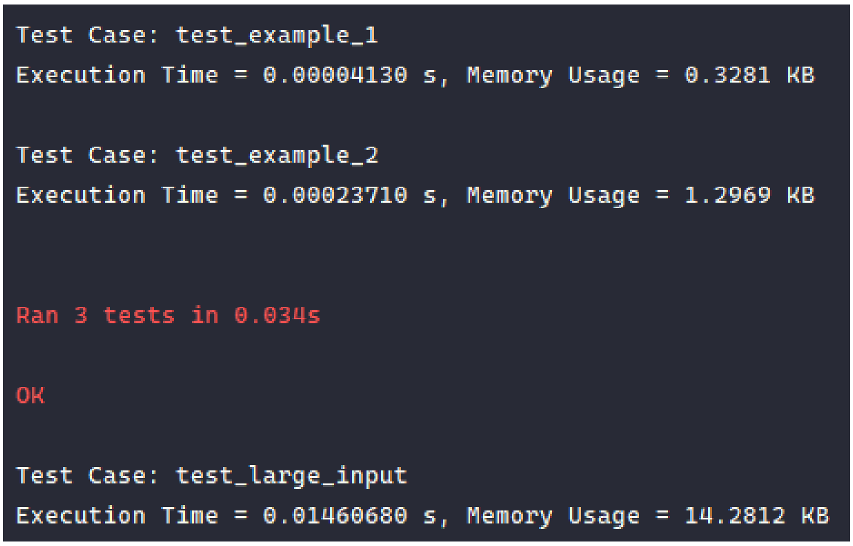
- Текстовое объяснение решения
  + Функция **evaluate**: Выполняет простые арифметические операции (+, -, *) для двух чисел.
  + Использую регулярные выражения (**regex**) для разделения чисел (включая отрицательные числа) и операторов. Если знак «-» стоит сразу после оператора или в начале выражения, то это знак отрицательного числа, а не оператор.
  + Разделяет выражение на числа (**digits**) и операторы (**operators**).
  + Создаются две двумерные таблицы **min_val** и **max_val** для хранения минимальных и максимальных значений подвыражений.
  + Динамическое программирование:
    - Перебираются все возможные подвыражения разной длины.
    - Для каждого подвыражения вычисляются минимальное и максимальное значения, перебирая все возможные места разделения выражения оператором.
  + Для каждого разделения вычисляются четыре возможных результата (a, b, c, d), комбинируя максимальные и минимальные значения левой и правой частей.
  + Обновляются минимальные и максимальные значения для текущего подвыражения.

+ Возвращается максимальное значение для всего выражения, хранящееся в max_val[n-1].

- Результат работы код на примерах из текста задачи



- Test



```
Test Case: test_example_1
Execution Time = 0.00004130 s, Memory Usage = 0.3281 KB

Test Case: test_example_2
Execution Time = 0.00023710 s, Memory Usage = 1.2969 KB


Ran 3 tests in 0.034s

OK

Test Case: test_large_input
Execution Time = 0.01460680 s, Memory Usage = 14.2812 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
1+5	0.00004130	0.3281
5-8+7*4-8+9	0.00023710	1.2969
1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1	0.01460680	14.2812

- Вывод

+ Этот алгоритм эффективно решает задачу для выражений, содержащих отрицательные цифры, многозначные числа и операторы `+`, `-`, `*`.
+ Временная сложность: $O(n^3)$, пространственная сложность: $O(n^2)$, где n — количество цифр в выражении.

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.** $1 \leq N \leq 5000, 0 \leq K \leq N$.

- **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.

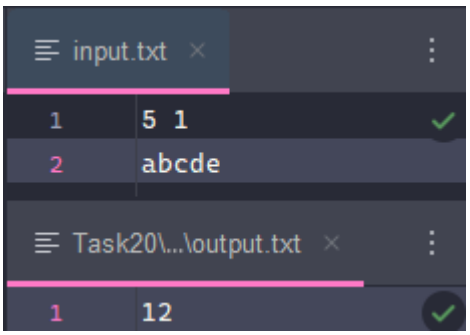
  - Листинг кода

```
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def count_palindromes(N, K, S):
    count = 0
    changes = [[0] * N for _ in range(N)]

    for length in range(1, N + 1):
        for i in range(N - length + 1):
            j = i + length - 1
```

```
            if i == j:
                changes[i][j] = 0

            elif i + 1 == j:
                changes[i][j] = 0 if S[i] == S[j] else 1

            else:
                changes[i][j] = changes[i + 1][j - 1] + (0 if
S[i] == S[j] else 1)

            if changes[i][j] <= K:
                count += 1

    return count

def task20():
    N, K, S = read(PATH_INPUT, 20)
    result = count_palindromes(N, K, S)
    write(PATH_OUTPUT, result, 20)

if __name__ == "__main__":
    task20()
```

- Текстовое объяснение решения
  + **count**: счетчик палиндромов.
  + **changes**: двумерный массив для хранения минимального количества изменений, необходимых для превращения подстроки в палиндром.
  + Динамическое программирование:
    - Перебираются все возможные длины подстрок (от 1 до N).
    - Для каждой длины перебираются все возможные начальные позиции подстроки
  + Заполнение массива **changes**:
    - Для подстрок длины 1: всегда 0 изменений.
    - Для подстрок длины 2: 0 если символы одинаковые, иначе 1.
    - Для более длинных подстрок: количество изменений для внутренней подстроки плюс 1, если крайние символы различны.
  + Если количество необходимых изменений для подстроки не превышает K, увеличиваем счетчик.
  + Функция возвращает общее количество найденных палиндромов.

- Результат работы код на примерах из текста задачи



- Test

```
Test Case: test_example_1
Execution Time = 0.00003340 s, Memory Usage = 0.3672 KB

Test Case: test_example_2
Execution Time = 0.00001220 s, Memory Usage = 0.2109 KB


Ran 3 tests in 0.045s

OK

Test Case: test_example_3
Execution Time = 0.02011170 s, Memory Usage = 321.1104 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
5 1 abcde	0.00003340	0.3672
3 3 aaa	0.00001220	0.2109
200 2 aaaaaaaaaaaa...	0.02011170	321.1104

- Вывод
  + Алгоритм эффективен для небольших значений N, но при больших входных данных может быть медленным.
  + Временная сложность $O(N^2)$, так как рассматриваются все подстроки.

+ Дополнительная память $O(N^2)$ для хранения таблицы changes.

II. Дополнительные задачи

Задача №13. Сувениры

Вы и двое ваших друзей только что вернулись домой после посещения разных стран. Теперь вы хотели бы поровну разделить все сувениры, которые все трое накупили.

- **Формат ввода / входного файла (input.txt).** В первой строке дано целое число n. Во второй строке даны целые числа $v_1, v_2, ..., v_n$, разделенные пробелами.
- **Ограничения на входные данные.** $1 \leq n \leq 20, 1 \leq v_i \leq 30$ для всех i.
- **Формат вывода / выходного файла (output.txt).** Выведите 1, если можно разбить $v_1, v_2, ..., v_n$ на три подмножества с одинаковыми суммами и 0 в противном случае.
- Ограничение по времени. 5 сек.

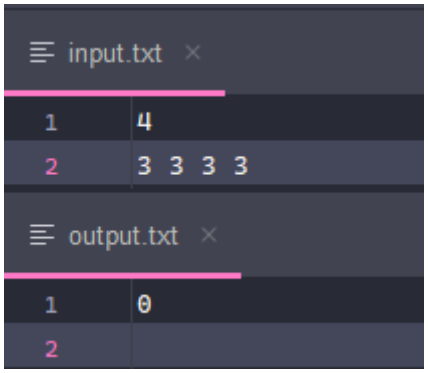
- Листинг кода

```
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def partition_subsets(souvenirs):
    total = sum(souvenirs)
    if total % 3 != 0:
        return 0

    target = total // 3
    n = len(souvenirs)

    # Создать таблицу DP
    dp = [[False] * (target + 1) for _ in range(n + 1)]
    dp[0][0] = True  # Можно получить сумму 0, ничего не выбирая

    # Просмотр каждого элемента
    for i in range(1, n + 1):
        for j in range(target, souvenirs[i - 1] - 1, -1):  # Итерация
в обратном порядке, чтобы избежать перезаписи
            dp[i][j] = dp[i - 1][j] or dp[i - 1][j - souvenirs[i - 1]]

    # Можно ли разделить тест на 3 части?
    return 1 if dp[n][target] else 0
```

```
def task13():
    souvenirs = read(PATH_INPUT, 13)
    result = partition_subsets(souvenirs)
    write(PATH_OUTPUT, result, 13)

if __name__ == "__main__":
    task13()
```

- Текстовое объяснение решения:
  + Если сумма S всех сувениров не делится на 3, сразу возвращаем 0.
  + Это позволяет быстрее найти решение, пробуя сначала большие числа.
  + Мы пытаемся разложить элементы по трем корзинам.
  + Если текущая корзина buckets[i] + текущий сувенир не превышает S/3, добавляем его.
  + Если дошли до конца списка и все три корзины имеют одинаковую сумму, возвращаем 1.
  + Если после всех попыток нельзя разделить, возвращаем 0.

- Результат работы кода:



- Test

```
Test Case: test_example_1
Execution Time = 0.00007610 s, Memory Usage = 4.3906 KB

Test Case: test_example_2
Execution Time = 0.00011500 s, Memory Usage = 11.5117 KB

Test Case: test_example_3
Execution Time = 0.00013240 s, Memory Usage = 12.1016 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
13 1 2 3 4 5 5 7 7 8 10 12 19 25	0.00007610	4.3906
11 17 59 34 57 17 23 67 1 18 2 59	0.00011500	11.5117
20 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	0.00013240	12.1016

- Вывод
  - + Сложность: O(n × target), осуществимо для n ≈ 1000.
  - + DP использует таблицу dp[i][s] размером O(n × target), где target = total // 3. Если target большой (сотни тысяч), таблица DP займет слишком много памяти.
  - + В этой задаче метод backtracking более эффективен.

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.

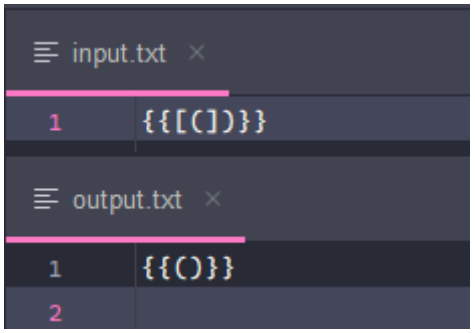
- Листинг кода

```
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def delete_bracket(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]
    pair = {')': '(', ']': '[', '}': '{'}

    # Заполняем DP
    for length in range(2, n + 1):  # Длина подстроки
        for i in range(n - length + 1):
            j = i + length - 1
             if s[j] in pair and i < j and s[i] == pair[s[j]]:  # Если
s[i] и s[j] образуют пару
                dp[i][j] = dp[i + 1][j - 1] + 2
            for k in range(i, j):  # Разбиение на две части
                dp[i][j] = max(dp[i][j], dp[i][k] + dp[k + 1][j])

    # Восстановление ответа
    def get_sequence(i, j):
        if i > j:
            return ""
```

```
        if dp[i][j] == 0:
            return ""
        if s[j] in pair and i < j and s[i] == pair[s[j]] and dp[i][j]
== dp[i + 1][j - 1] + 2:
            return s[i] + get_sequence(i + 1, j - 1) + s[j]
        for k in range(i, j):
            if dp[i][j] == dp[i][k] + dp[k + 1][j]:
                return get_sequence(i, k) + get_sequence(k + 1, j)
        return ""

    return get_sequence(0, n - 1)

def task15():
    s = read(PATH_INPUT, 15)
    result = delete_bracket(s)
    write(PATH_OUTPUT, result, 15)

if __name__ == "__main__":
    task15()
```

- Текстовое объяснение решения:
  + n = len(s): получаем длину входной строки
  + dp: создаем двумерный массив для динамического программирования
  + pair: словарь, сопоставляющий закрывающие скобки с открывающимися
  + Используется динамическое программирование для заполнения массива
  + Внешний цикл перебирает длины подстрок от 2 до n
  + Внутренний цикл перебирает начальные позиции подстрок. Проверяются два случая:
    - Если текущие символы образуют пару скобок
    - Разбиение подстроки на две части и выбор максимума
  + Функция get_sequence(i, j): Рекурсивно восстанавливает правильную последовательность скобок. Проверяет различные случаи:
    - Пустая подстрока
    - Парные скобки
    - Разбиение на две части
  + Вызывается get_sequence(0, n-1) для получения итоговой правильной последовательности скобок

- Результат работы кода:



- Test

```
Test Case: test_example_1
Execution Time = 0.00019900 s, Memory Usage = 1.0537 KB

Test Case: test_example_2
Execution Time = 0.00009480 s, Memory Usage = 1.0908 KB

Test Case: test_example_3
Execution Time = 0.00179120 s, Memory Usage = 5.1328 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
{{[(])}}	0.00019900	1.0537
(]{[}	0.00009480	1.0908
(((()))]]]][[[[{}{}]]]]	0.00179120	5.1328

- Вывод
  + Используется метод динамического программирования для оптимального решения задачи.
  + Временная сложность алгоритма составляет O(n^3), пространственная сложность алгоритма - O(n^2), где n - длина входной строки

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.

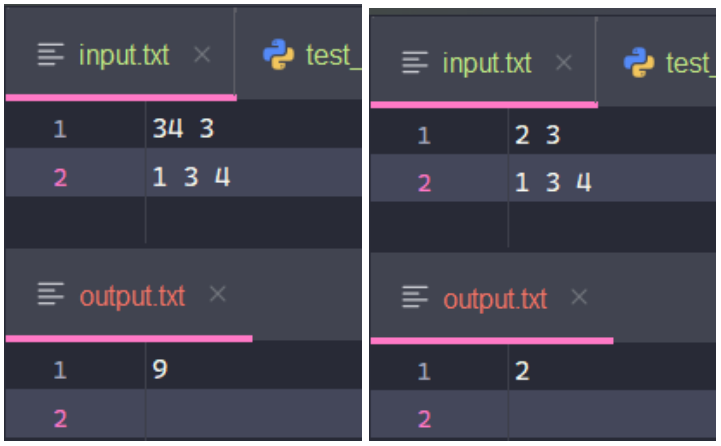
  - Листинг кода

```python
from Lab1.utils import read, write

PATH_INPUT = '../txtf/input.txt'

PATH_OUTPUT = '../txtf/output.txt'


MOD = 10**9

# Возможные ходы коня

moves = {

    0: [4, 6],

    1: [6, 8],

    2: [7, 9],

    3: [4, 8],
```

```
    4: [3, 9, 0],
    5: [],
    6: [1, 7, 0],
    7: [2, 6],
    8: [1, 3],
    9: [2, 4]
}

def phone_numbers(N):
    if N == 1:
        return 8  # Все цифры, кроме 0 и 8

    # DP таблица
    dp = [[0] * 10 for _ in range(N + 1)]

    # Начальная инициализация
    for d in range(10):
        dp[1][d] = 1 if d != 0 and d != 8 else 0

    # Заполнение DP
    for i in range(2, N + 1):
        for d in range(10):
            dp[i][d] = sum(dp[i - 1][prev] for prev in moves[d]) % MOD

    # Ответ — сумма всех допустимых окончаний
    return sum(dp[N][d] for d in range(10)) % MOD

def task17():
    N = read(PATH_INPUT, 17)
    result = phone_numbers(N)
    write(PATH_OUTPUT, result, 17)
```

```
if __name__ == "__main__":
    task17()
```

- Текстовое объяснение решения
  + Мы создаем словарь moves, который для каждой цифры содержит список цифр, на которые можно попасть ходом коня.
  + Создаем двумерный массив dp, где dp[i][j] означает количество номеров длины i+1, заканчивающихся цифрой j.
  + Инициализируем базовый случай для номеров длины 1, исключая 0 и 8.
  + Заполняем массив dp для всех длин от 2 до N. Для каждой длины и каждой цифры мы суммируем количество номеров, которые можно получить, сделав ход коня с предыдущих допустимых позиций.
  + В конце суммируем все значения в последней строке dp, что дает нам общее количество номеров длины N.
- Результат работы кода:



- Test:

```
Test Case: test_example_1
Execution Time = 0.00000620 s, Memory Usage = 0.1172 KB

Test Case: test_example_2
Execution Time = 0.00005480 s, Memory Usage = 0.8672 KB

Test Case: test_example_3
Execution Time = 0.03873780 s, Memory Usage = 205.4375 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
1	0.00000620	0.1172
2	0.00005480	0.8672
500	0.03873780	205.4375

- Вывод
  + Преимущества: Эффективное решение для больших N. Легко модифицируется для различных конфигураций клавиатуры.
  + Временная сложность: O(N), где N - длина номера.
  + Пространственная сложность: O(N), для хранения DP-таблицы.

III. Вывод

Сравнение жадных алгоритмов и динамического программирования

Критерии	Жадный алгоритм	Динамическое программирование
Принцип	Выбирает лучший шаг в каждый момент, не оглядываясь на будущее.	Рассчитает все возможности и сохранить результаты для повторного использования.
Применимые объекты	Задача имеет жадные свойства и свойства оптимальной подструктуры	Задача имеет неоптимальность, но не удовлетворяет жадному свойству.
Оптимальность	Не всегда дает оптимальные результаты	Всегда дает оптимальные результаты, если проблему можно решить с помощью DP.
Время	$O(n)$ или $O(n \log n)$	$O(n^2)$ или $O(n^3)$
Память	Низкое потребление памяти, только временное хранение.	Необходимо хранить результаты подзадач, что может потребовать $O(n)$ или $O(n^2)$ памяти