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

Выполнил:
Нгуен Хыу Жанг
Мобильные и сетевые технологии
К3140

Проверила:
Петросян Анна Мнацакановна

Санкт-Петербург
2025 г

Содержание

Задачи по варианту

Задание 1 : Максимальная стоимость добычи

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 дан- ных 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.** $1 \le n \le 10^3$, $0 \le W \le 2 \cdot 10^6$, $0 \le p_i \le 2 \cdot 10^6$, $0 \le w_i \le 2 \cdot 10^6$ для всех $1 \le i \le n$. Все числа - целые.
- **Формат вывода / выходного файла (output.txt).** Выведите максимальное значение стоимости долей предметов, которые помещаются в сумку. Аб- солютная 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 округле- нием).
- 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.

```
import os

def fractional_knapsack(n, W, items):
    for i in range(n):
        items[i] = (items[i][0], items[i][1], items[i][0] / items[i][1])

    items.sort(key=lambda x: x[2], reverse=True)

    total_value = 0.0
    remaining_capacity = W

    for value, weight, unit_value in items:
        if remaining_capacity == 0:
            break

        if weight <= remaining_capacity:
            total_value += value
```

```
            remaining_capacity -= weight
        else:
            total_value += unit_value * remaining_capacity
            remaining_capacity = 0

    return total_value

current_dir = os.path.dirname(os.path.abspath(__file__))
input_file_path = os.path.join(current_dir, '..', 'txtf', 'input.txt')
output_file_path = os.path.join(current_dir, '..', 'txtf', 'output.txt')

with open(input_file_path, 'r') as file:
    n, W = map(int, file.readline().split())
    items = [tuple(map(int, file.readline().split())) for _ in range(n)]

max_value = fractional_knapsack(n, W, items)

with open(output_file_path, 'w') as file:
    file.write(f"{max_value:.4f}\n")
```

input.txt:

```
3 50
60 20
100 50
120 30
```

output.txt:

```
180.0000
```

Объяснение кода задачи о дробном рюкзаке:

- В данной задаче мы помогаем вору максимизировать стоимость добычи, используя алгоритм для решения задачи дробного рюкзака. Основная суть заключается в том, что вор может брать части предметов, что позволяет нам использовать жадный подход.

Структура кода:

1. **Импортирование необходимых библиотек**:
   - Импортируем библиотеку `os` для работы с файловой системой.
2. **Определение функции `fractional_knapsack`**:
   - Эта функция принимает количество предметов `n`, вместимость рюкзака `W` и список `items`, содержащий пары (стоимость, вес) каждого предмета.
3. **Расчет удельной стоимости**:
   - В цикле мы добавляем третью составляющую к каждому предмету: удельную стоимость, которая вычисляется как `стоимость / вес`. Это позволит нам сортировать предметы по их ценности на единицу веса.
4. **Сортировка предметов**:
   - После вычисления удельной стоимости, мы сортируем список предметов в порядке убывания удельной стоимости. Это позволяет первыми рассматривать самые ценные предметы.
5. **Заполнение рюкзака**:
   - Инициализируем переменную `total_value` для хранения общей стоимости и `remaining_capacity` для отслеживания оставшейся вместимости рюкзака.
   - Проходим по отсортированным предметам:
     - Если оставшаяся вместимость равна нулю, выходим из цикла.

- Если вес предмета меньше или равен оставшейся вместимости, добавляем его полную стоимость к `total_value` и уменьшаем `remaining_capacity`.
- Если вес предмета превышает оставшуюся вместимость, добавляем пропорциональную стоимость (удельная стоимость умноженная на оставшую вместимость) и выходим из цикла.

6. **Чтение входных данных**:
   - Открываем файл `input.txt` и считываем количество предметов и вместимость рюкзака. Затем считываем стоимость и вес каждого предмета, сохраняя их в списке.
7. **Запись результата в файл**:
   - После вычислений записываем максимальную стоимость в файл `output.txt`, форматируя результат с четырьмя знаками после запятой для обеспечения необходимой точности.

Примеры работы:

- Для входных данных:

```
3 50
60 20
100 50
120 30
```

- Мы имеем три предмета. После сортировки по удельной стоимости, наилучший вариант — взять третий и первый предмет, что дает общую стоимость 180.0000.
- Для входных данных:

```
1 10
500 30
```

- Мы имеем три предмета. После сортировки по удельной стоимости, наилучший вариант — взять третий и первый предмет, что дает общую стоимость 166.6667.

Заключение:

- Данный код эффективно решает задачу дробного рюкзака, используя жадный алгоритм, что позволяет быстро находить решение даже при больших входных данных.

Задание 6 : Максимальная зарплата

В качестве последнего вопроса успешного собеседования ваш начальник да- ет вам несколько листков бумаги с цифрами и просит составить из этих цифр наибольшее число. Полученное число будет вашей зарплатой, поэтому вы очень заинтересованы в максимизации этого числа. Как вы можете это сделать?

На лекциях мы рассмотрели следующий алгоритм составления наибольшего числа из заданных *однозначных* чисел.

```
def LargestNumber(Digits):
    answer = ''

    while Digits:

        maxDigit = float(-inf)
        for digit in Digits:

            if digit >= maxDigit:
                maxDigit = digit

        answer += str(maxDigit)
```

К сожалению, этот алгоритм работает только в том случае, если вход состоит из однозначных чисел. Например, для ввода, состоящего из двух целых чисел 23 и 3 (23 не однозначное число!) возвращается 233, в то время как наибольшее число на самом деле равно 323. Другими словами, использование наибольшего числа из входных данных в качестве первого числа *не является безопасным ходом*.

Ваша цель в этой задаче – настроить описанный выше алгоритм так, чтобы он работал не только с однозначными числами, но и с произвольными положитель- ными целыми числами.

- **Постановка задачи.** Составить наибольшее число из набора целых чисел.
- **Формат ввода / входного файла (input.txt).** Первая строка входных данных содержит целое число n. Во второй строке даны целые числа $a_1, a_2, ..., a_n$.
- **Ограничения на входные данные.** $1 \le n \le 10^2$, $1 \le a_i \le 10^3$ для всех $1 \le i \le n$.
- **Формат вывода / выходного файла (output.txt).** Выведите наибольшее число, которое можно составить из $a_1, a_2, ..., a_n$.
- Ограничение по времени. 2 сек.
- Пример:

input.txt	output.txt	input.txt	output.txt
2 21 2	221	3 23 39 92	923923

```python
import os

def largest_number(digits):
    digits.sort(key=lambda x: x * 3, reverse=True)
    return ''.join(digits)

def main():
    txtf_directory = os.path.join(os.path.dirname(__file__), '..', 'txtf')

    if not os.path.exists(txtf_directory):
        os.makedirs(txtf_directory)

    input_file_path = os.path.join(txtf_directory, 'input.txt')

    if not os.path.exists(input_file_path):
        print(f"File {input_file_path} не найдено. Создайте файл с входными данными.")
        return

    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
        numbers = f.readline().strip().split()

    result = largest_number(numbers)

    with open(os.path.join(txtf_directory, 'output.txt'), 'w') as f:
        f.write(result)

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

input.txt:

```
3
23 39 92
```

output.txt:

```
923923
```

Объяснение кода задачи о максимальной зарплате:
- В данной задаче нам нужно составить максимальное число из заданного набора целых чисел. Для этого я использую подход, позволяющий корректно сравнивать числа, чтобы находить наилучший порядок для их конкатенации.

Структура кода:

1. **Импортирование необходимых библиотек**:
   - В начале кода мы импортируем библиотеку `os` для работы с файловой системой.
2. **Определение функции `largest_number`**:
   - Эта функция принимает список строк `digits`, представляющих числа.
   - Мы сортируем эти числа с помощью метода sort, используя ключ `key=lambda x: x * 3`. Это позволяет нам сравнивать числа по их возможной конкатенации. Умножение на 3 нужно для обеспечения корректного сравнения, поскольку максимальная длина числа в нашем случае — 3 (числа до 1000).
   - После сортировки мы соединяем отсортированные строки в одно большое число с помощью `''.join(digits)`.
3. **Определение функции `main`**:
   - В этой функции мы создаем директорию для хранения файлов, если она не существует.
   - Затем мы определяем путь к файлу `input.txt` и проверяем его наличие. Если файл не найден, выводится сообщение об ошибке.
   - Если файл существует, мы считываем количество чисел n и сами числа, сохраняя их в списке `numbers`.
4. **Запись результата в файл**:
   - После получения результата с помощью функции `largest_number`, мы записываем полученное максимальное число в файл `output.txt`.
5. **Запуск программы**:
   - В конце кода проверяем, запущен ли скрипт напрямую, и если да, то вызываем функцию `main`.

Примеры работы:

- Для входных данных:

```
2
21 2
```

- Функция сортирует числа так, чтобы на выходе получилось `221`.
- Для входных данных:

```
3
23 39 92
```

- Правильный порядок для максимальной конкатенации — `923923`.

Заключение:
- Данный код эффективно решает задачу по составлению максимального числа из заданных целых чисел, используя сортировку с учетом специфики конкатенации. Это позволяет получить правильный результат даже в случаях, когда числа имеют разную длину.

Задание 12 : Последовательность

- **Постановка задачи.** Дана последовательность натуральных чисел $a_1, a_2, \ldots, a_n$, и известно, что $a_i \le i$ для любого $1 \le i \le n$. Требуется определить, можно ли разбить элементы последовательности на две части таким образом, что сумма элементов в каждой из частей будет

равна половине суммы всех элементов последовательности.

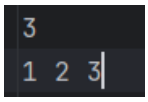
- **Формат ввода / входного файла (input.txt).** В первой строке входного файла находится одно целое число n. Во второй строке находится n целых чисел $a_1, a_2, ..., a_n$.
- **Ограничения на входные данные.** $1 \le n \le 40000$, $1 \le a_i \le i$.
- **Формат вывода / выходного файла (output.txt).** В первую строку выход- ного файла выведите количество элементов посделовательности в любой из получившихся двух частей, а во вторую строку через пробел номера этих элементов. Если построить такое разбиение невозможно, выведите -1.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

input.txt	output.txt
3 1 2 3	1 3

```
import os

def can partition(numbers):
    total_sum = sum(numbers)
    if total_sum % 2 != 0:
        return -1, []

    target sum = total sum // 2
    n = len(numbers)

    dp = [False] * (target_sum + 1)
    dp[0] = True

    for num in numbers:
        for j in range(target_sum, num - 1, -1):
            dp[j] = dp[j] or dp[j - num]

    if not dp[target_sum]:
        return -1, []

    subset = []
    w = target_sum
    for i in range(n - 1, -1, -1):
        if w >= numbers[i] and dp[w - numbers[i]]:
            subset.append(i + 1)
            w -= numbers[i]

    return len(subset), subset

def main():
    txtf_directory = os.path.join(os.path.dirname(__file__), '..', 'txtf')

    if not os.path.exists(txtf_directory):
        os.makedirs(txtf_directory)

    input_file_path = os.path.join(txtf_directory, 'input.txt')

    if not os.path.exists(input_file_path):
        print(f"File {input_file_path} не найдено. Создайте файл с входными данными.")
```

```
        return

    with open(input_file_path, 'r') as f:
        n = int(f.readline().strip())
        numbers = list(map(int, f.readline().strip().split()))

    count, indices = can_partition(numbers)

    output_file_path = os.path.join(txtf_directory, 'output.txt')
    with open(output_file_path, 'w') as f:
        if count == -1:
            f.write("-1\n")
        else:
            f.write(f"{count}\n")
            f.write(" ".join(map(str, indices)))

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

input.txt:



output.txt:

Объяснение кода задачи о разбиении последовательности:
В этой задаче требуется определить, можно ли разбить заданную последовательность натуральных чисел на две части так, чтобы сумма элементов в каждой части была равна половине суммы всех элементов. Я использую динамическое программирование для решения этой задачи.

Структура кода:

1. **Импортирование необходимых библиотек**:
   - В начале кода я импортирую библиотеку `os` для работы с файловой системой.
2. **Определение функции `can_partition`**:
   - Эта функция принимает список натуральных чисел `numbers`.
   - Я вычисляю общую сумму элементов `total_sum`. Если эта сумма нечетная, разбиение на две равные части невозможно, и функция возвращает `-1` и пустой список.
3. **Целевая сумма**:
   - Устанавливаем целевую сумму `target_sum`, равную половине общей суммы.
4. **Инициализация массива динамического программирования**:
   - Создаем массив `dp` длиной `target_sum + 1`, где `dp[j]` будет означать, можно ли получить сумму `j` из подмножества чисел. Инициализируем `dp[0]` как `True`, так как сумма 0 всегда возможна.
5. **Заполнение массива `dp`**:
   - Проходим по каждому числу из `numbers` и обновляем массив `dp` таким образом, чтобы определить, можно ли получить каждую сумму от `target_sum` до текущего числа.
6. **Проверка возможности достижения целевой суммы**:
   - Если `dp[target_sum]` равен `False`, это значит, что разбиение невозможно, и я возвращаю `-1`.

7. **Восстановление подмножества**:
    - Если разбиение возможно, я восстанавливаю подмножество, которое дает целевую сумму. Для этого проходим по массиву `dp` в обратном порядке и добавляем индексы чисел в подмножество.
8. **Определение функции `main`**:
    - В этой функции мы создаем директорию для хранения файлов, если она не существует.
    - Затем определяем путь к файлу `input.txt` и проверяем его наличие. Если файл не найден, выводится сообщение об ошибке.
    - Если файл существует, считываем количество чисел `n` и сами числа, сохраняя их в списке `numbers`.
9. **Запись результата в файл**:
    - После получения результата с помощью функции `can_partition`, я записываю количество элементов и их индексы в файл `output.txt`. Если разбиение невозможно, записываем `-1`.
10. **Запуск программы**:
    - В конце кода проверяем, запущен ли скрипт напрямую, и если да, то вызываем функцию `main`.

Примеры работы:

- Для входных данных:

```
3
1 2 3
```

- Общая сумма равна 6, и целевая сумма — 3. Вариант разбиения — {3} и {1, 2}, поэтому выводим `1` и `3`.
- Если, например, входные данные не позволяют разбить последовательность, возвращается `-1`.

Заключение:

- Данный код эффективно решает задачу разбиения последовательности на две равные части с помощью динамического программирования, что позволяет обрабатывать большие объемы данных за разумное время.

Задание 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 про- белами.

• **Ограничения на входные данные.** $1 \le n \le 20$, $1 \le v_i \le 30$ 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.

• Примеры:

input.txt	output.txt	input.txt	output.txt
4 3 3 3 3	0	1 40	0

input.txt	output.txt
11 17 59 34 57 17 23 67 1 18 2 59	1

Здесь 34 + 67 + 17 = 23 + 59 + 1 + 17 + 18 = 59 + 2 + 57.

input.txt	output.txt
13 1 2 3 4 5 5 7 7 8 10 12 19 25	1

Здесь 1 + 3 + 7 + 25 = 2 + 4 + 5 + 7 + 8 + 10 = 5 + 12 + 19.

```python
import os

def can_partition(nums):
    total_sum = sum(nums)

    if total_sum % 3 != 0:
        return 0

    target_sum = total_sum // 3
    n = len(nums)

    sums = {0}

    for num in nums:
        new_sums = set()
        for s in sums:
            new_sum = s + num
            if new_sum <= target_sum:
                new_sums.add(new_sum)
        sums.update(new_sums)

    return 1 if target_sum in sums else 0

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as file:
        n = int(file.readline().strip())
        nums = list(map(int, file.readline().strip().split()))

    result = can_partition(nums)

    with open(output_path, 'w') as file:
        file.write(str(result) + '\n')

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

input.txt:

```
11
17 59 34 57 17 23 67 1 18 2 59
```

output.txt:

```
1
```

Объяснение кода задачи о сувенирах:

- В этой задаче необходимо определить, можно ли разбить набор сувениров на три подмножества с равными суммами. Мы используем метод динамического программирования с помощью множества для решения этой задачи.

Структура кода:

1. **Импортирование необходимых библиотек**:
   - В начале кода мы импортируем библиотеку `os` для работы с файловой системой.
2. **Определение функции `can_partition`**:
   - Эта функция принимает список целых чисел `nums`, представляющих стоимости сувениров.
   - Мы вычисляем общую сумму элементов `total_sum`. Если эта сумма не делится на 3, невозможно разбить сувениры на три равные части, и функция возвращает `0`.
3. **Целевая сумма**:
   - Устанавливаем целевую сумму `target_sum`, равную `total_sum // 3`.
4. **Использование множества для хранения возможных сумм**:
   - Инициализируем множество `sums`, содержащее только ноль, так как сумма 0 всегда возможна.
   - Проходим по каждому числу в `nums` и обновляем множество `sums`, добавляя новые суммы, которые можно получить, добавляя текущее число к уже существующим суммам.
5. **Проверка возможности достижения целевой суммы**:
   - Если целевая сумма `target_sum` находится в множестве `sums`, это значит, что есть подмножество, сумма которого равна целевой. В таком случае функция возвращает `1`.
   - Если целевая сумма не найдена, возвращаем `0`.
6. **Определение функции `task_scheduler`**:
   - В этой функции мы определяем пути к файлам `input.txt` и `output.txt`.
   - Считываем данные из входного файла: количество сувениров `n` и их стоимости, сохраняя их в списке `nums`.
7. **Запись результата в файл**:
   - После вычисления результата с помощью функции `can_partition`, мы записываем его в файл `output.txt`.
8. **Запуск программы**:
   - В конце кода проверяем, запущен ли скрипт напрямую, и если да, вызываем функцию `task_scheduler`.

Примеры работы:

- Для входных данных:

```
11
17 59 34 57 17 23 67 1 18 2 59
```

- В этом случае возможно разбить набор на три подмножества с равными суммами, и функция также вернет `1`.

Заключение:

- Данный код эффективно решает задачу разбиения сувениров на три равные части с помощью динамического программирования и использования множеств, что позволяет обрабатывать большие объемы данных за разумное время.

Задание 15 : Удаление скобок

- **Постановка задачи.** Дана строка, составленная из круглых, квадратных и фигурных скобок. Определите, какое наименьшее количество символов необходимо удалить из этой строки,

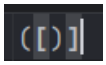
чтобы оставшиеся символы образовы- вали правильную скобочную последовательность.

- **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.** $1 \le s \le 100$.
- **Формат вывода / выходного файла (output.txt).** Выведите строку макси- мальной длины, являющейся правильной скобочной последовательностью, которую можно получить из исходной строки удалением некоторых симво- лов.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

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

```
import os

def is_matching_pair(opening, closing):
    return (opening == '(' and closing == ')') or \
           (opening == '[' and closing == ']') or \
           (opening == '{' and closing == '}')

def find_max_valid_subsequence(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

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

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if is_matching_pair(s[i], s[j]):
                dp[i][j] = dp[i + 1][j - 1] + 2
            else:
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

    result = []
    i, j = 0, n - 1
    while i <= j:
        if is_matching_pair(s[i], s[j]):
            result.append(s[i])
            result.append(s[j])
            i += 1
            j -= 1
        elif dp[i][j] == dp[i + 1][j]:
            i += 1
        else:
            j -= 1

    return ''.join(result)

def task_scheduler(input_file='input.txt', output_file='output.txt'):
```

```
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as file:
        s = file.read().strip()

    result = find_max_valid_subsequence(s)

    with open(output_path, 'w') as file:
        file.write(result)

task_scheduler()
```

input.txt: 

output.txt: 

Объяснение кода задачи об удалении скобок:

- В данной задаче требуется определить, какое наименьшее количество символов необходимо удалить из строки, состоящей из различных типов скобок, чтобы оставшиеся символы образовывали правильную скобочную последовательность. Мы используем динамическое программирование для решения этой задачи.

Структура кода:

1. **Импортирование необходимых библиотек**:
   - В начале кода мы импортируем библиотеку `os` для работы с файловой системой.
2. **Определение функции `is_matching_pair`**:
   - Эта функция проверяет, являются ли две скобки парными. Она принимает две строки `opening` и `closing` и возвращает `True`, если они составляют правильную пару (например, (и)).
3. **Определение функции `find_max_valid_subsequence`**:
   - Эта функция находит максимальную правильную скобочную последовательность в строке `s`.
   - Создаем двумерный массив `dp`, где `dp[i][j]` будет хранить длину максимальной правильной скобочной последовательности, которую можно получить из подстроки `s[i:j+1]`.
4. **Инициализация массива `dp`**:
   - Для каждой отдельной скобки (то есть для всех `i`, где `i` равно индексу) мы устанавливаем `dp[i][i] = 1`, так как одиночная скобка считается правильной последовательностью длиной 1.
5. **Заполнение массива `dp`**:
   - Используем два вложенных цикла для перебора всех подстрок длиной от 2 до `n`.
   - Если первая и последняя скобки в текущем диапазоне пары, обновляем значение `dp[i][j]` как `dp[i + 1][j - 1] + 2`.
   - Если они не парные, берем максимум между `dp[i + 1][j]` и `dp[i][j - 1]`, чтобы определить максимальную длину последовательности.
6. **Восстановление результата**:
   - Создаем список `result`, в который будем добавлять символы, формирующие максимальную последовательность.
   - Используем два указателя `i` и `j`, чтобы пройти по строке и восстановить правильную последовательность, основываясь на значениях в массиве `dp`.

7. **Определение функции `task_scheduler`**:
    - В этой функции мы определяем пути к файлам `input.txt` и `output.txt`.
    - Считываем строку из входного файла и передаем её в функцию `find_max_valid_subsequence`.
8. **Запись результата в файл**:
    - После вычисления результата записываем его в файл `output.txt`.
9. **Запуск программы**:
    - В конце кода проверяем, запущен ли скрипт напрямую, и если да, вызываем функцию `task_scheduler`.

Примеры работы:

- Для входных данных:

```
([)]
```

- Правильная скобочная последовательность, которую можно получить, — это `[]`, поэтому вывод будет `[]`.

Заключение:

- Данный код эффективно решает задачу об удалении лишних скобок, используя динамическое программирование. Это позволяет обрабатывать строки длиной до 100 символов, находя максимальную правильную скобочную последовательность.

Задание 18 : Кафе

- **Постановка задачи.** Около университета недавно открылось новое кафе, в котором действует следующая система скидок: при каждой покупке бо- лее чем на 100 рублей покупатель получает купон, дающий право на один бесплатный обед (при покупке на сумму 100 рублей и меньше такой купон покупатель не получает). Однажды вам на глаза попался прейскурант на ближайшие n дней. Внимательно его изучив, вы решили, что будете обедать в этом кафе все n дней, причем каждый день вы будете покупать в кафе ровно один обед. Однако стипендия у вас небольшая, и поэтому вы хотите по максимуму использовать предоставляемую систему скидок так, чтобы ваши суммарные затраты были минимальны. Требуется найти минималь- но возможную суммарную стоимость обедов и номера дней, в которые вам следует воспользоваться купонами.
- **Формат ввода / входного файла (input.txt).** В первой строке входного файла дается целое число n - количество дней. В каждой из последующих n строк дано одно неотрицательное целое число s_i – стоимость обеда в рублях на соответствующий день i.
- **Ограничения на входные данные.** $0 \le n \le 100$, $0 \le s_i \le 300$ для всех $0 \le i \le n$.
- **Формат вывода / выходного файла (output.txt).** В первой строке выдайте минимальную возможную суммарную стоимость обедов. Во второй строке выдайте два числа k_1 и k_2 – количество купонов, которые останутся у вас неиспользованными после этих n дней и количество использованных вами купонов соответственно. В последующих k_2 строках выдайте в возраста- ющем порядке номера дней, когда вам следует воспользоваться купонами. Если существует несколько решений с минимальной суммарной стоимо- стью, то выдайте то из них, в котором значение k_1 максимально (на случай, если вы когда-нибудь ещё решите заглянуть в это кафе). Если таких решений несколько, выведите любое из них.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 64 мб.

• Пример:

input.txt	output.txt
5 110 40 120 110 60	260 0 2 3 5

input.txt	output.txt
3 110 110 110	220 1 1 2

```
import os

def read_input(file_path):
    with open(file_path, 'r') as file:
        n = int(file.readline())
        costs = [int(file.readline()) for _ in range(n)]
    return n, costs

def minimize_cost(n, costs):
    dp = [[float('inf')] * (n + 2) for _ in range(n + 1)]
    dp[0][0] = 0

    for i in range(1, n + 1):
        for j in range(n + 1):
            if dp[i - 1][j] != float('inf'):
                dp[i][j] = min(dp[i][j], dp[i - 1][j] + costs[i - 1])
                if costs[i - 1] > 100:
                    dp[i][j + 1] = min(dp[i][j + 1], dp[i - 1][j] + costs[i - 1])

            if j > 0 and dp[i - 1][j] != float('inf'):
                dp[i][j - 1] = min(dp[i][j - 1], dp[i - 1][j])

    min_cost = float('inf')
    remaining_coupons = 0
    for j in range(n + 1):
        if dp[n][j] < min_cost:
            min_cost = dp[n][j]
            remaining_coupons = j

    used_coupons = []
    j = remaining_coupons
    for i in range(n, 0, -1):
        if j < n and dp[i][j] == dp[i - 1][j + 1]:
            used_coupons.append(i)
            j += 1
        elif dp[i][j] == dp[i - 1][j] + costs[i - 1]:
            pass
        elif costs[i - 1] > 100 and dp[i][j] == dp[i - 1][j - 1] + costs[i - 1]:
            j -= 1

    used_coupons.reverse()
    return min_cost, remaining_coupons, len(used_coupons), used_coupons

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    n, costs = read input(input path)

    min_cost, remaining_coupons, used_coupons_count, used_coupons = minimize_cost(n,
costs)
```

```
    with open(output_path, 'w') as file:
        file.write(f"{min_cost}\n")
        file.write(f"{remaining_coupons} {used_coupons_count}\n")
        for day in used_coupons:
            file.write(f"{day}\n")

if __name__ == '__main__':
    task_scheduler()
```

input.txt:

```
5
110
40
120
110
60
```

output.txt:

```
260
0 2
3
5
```

Объяснение кода задачи о кафе:

- В данной задаче необходимо минимизировать стоимость обедов, используя систему скидок в кафе. При каждой покупке на сумму более 100 рублей покупатель получает купон на бесплатный обед. Мы будем использовать динамическое программирование для решения этой задачи.

Структура кода:

1. **Чтение входных данных**:
   - Функция `read_input` считывает данные из файла. Она получает путь к файлу, считывает количество дней `n` и стоимости обедов на каждый день, сохраняя их в списке `costs`.
2. **Минимизация стоимости**:
   - Функция `minimize_cost` принимает количество дней `n` и список `costs`.
   - Создаем двумерный массив `dp`, где `dp[i][j]` будет хранить минимальную стоимость обедов за `i` дней с `j` купонами. Инициализируем `dp[0][0]` как `0`, так как стоимость нулевая, если мы не покупаем обеды.
3. **Заполнение массива `dp`**:
   - Внешний цикл проходит по всем дням, а внутренний — по доступному количеству купонов.
   - Если в предыдущий день купон не использовался, то текущая стоимость будет равна стоимости обеда на текущий день плюс стоимость обедов предыдущих дней.
   - Если стоимость обеда превышает 100 рублей, то мы можем получить купон, и в этом случае обновляем `dp[i][j + 1]`.
   - Если купон используется, проверяем, что он доступен, и обновляем `dp[i][j - 1]`.
4. **Определение минимальной стоимости и оставшихся купонов**:

- После заполнения массива `dp` ищем минимальную стоимость для всех возможных купонов на последний день. Сохраняем минимальную стоимость и количество оставшихся купонов.

5. **Восстановление использованных купонов**:
   - Создаем список `used_coupons`, чтобы отслеживать, в какие дни были использованы купоны. Мы проходим по массиву `dp` в обратном порядке, чтобы восстановить дни использования купонов.
6. **Определение функции `task_scheduler`**:
   - В этой функции определяются пути к файлам `input.txt` и `output.txt`.
   - Вызываем функцию `read_input` для считывания данных и `minimize_cost` для вычисления минимальной стоимости.
7. **Запись результата в файл**:
   - После вычислений записываем минимальную стоимость, количество оставшихся купонов и использованных купонов в файл `output.txt`, а также дни, когда были использованы купоны.
8. **Запуск программы**:
   - В конце кода проверяем, запущен ли скрипт напрямую, и если да, вызываем функцию `task_scheduler`.

Примеры работы:

- Для входных данных:

```
5
110
40
120
110
60
```

- Минимальная стоимость составит 260 рублей, и были использованы 2 купона, например, на дни 3 и 5.

Заключение:

- Данный код эффективно решает задачу минимизации затрат на обеды, используя динамическое программирование для учета системы скидок. Это позволяет оптимально распределить купоны и минимизировать общую стоимость обедов за заданный период.

Задание 19 : Произведение матриц

- **Постановка задачи.** В произведении последовательности матриц полно- стью расставлены скобки, если выполняется один из следующих пунктов:

– Произведение состоит из одной матрицы.

– Оно является заключенным в скобки произведением двух произведе- ний с полностью расставленными скобками.

Полная расстановка скобок называется оптимальной, если количество опе- раций, требуемых для вычисления произведения, минимально.

Требуется найти оптимальную расстановку скобок в произведении последовательности матриц.

- **Формат ввода / входного файла (input.txt).** В первой строке входных дан- ных содержится целое число n - количество матриц. В n следующих строк содержится по два целых числа a_i и

b_i - количество строк и столбцов в i-той матрице соответственно. Гарантируется, что $b_i = a_{i+1}$ для любого $1 \le i \le n - 1$.

- **Ограничения на входные данные.** $1 \le n \le 400$, $1 \le a_i, b_i \le 100$ для всех $1 \le i \le n$.
- **Формат вывода / выходного файла (output.txt).** Выведите оптимальную расстановку скобок. Если таких расстановок несколько, выведите любую.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 64 мб.
- Пример:

input.txt	output.txt
3 10 50 50 90 90 20	((AA)A)

- В данном примере возможно всего две расстановки скобок: $((AA)A)$ и $(A(AA))$. При первой количество операций будет равно $10 \cdot 50 \cdot 90 + 10 \cdot 90 \cdot 20 = 63000$, а при второй - $10 \cdot 50 \cdot 20 + 50 \cdot 90 \cdot 20 = 100000$.

```
import os

def read_input(file_path):
    with open(file_path, 'r') as file:
        n = int(file.readline())
        dimensions = [tuple(map(int, file.readline().split())) for _ in range(n)]
    return n, dimensions

def matrix_chain_order(dimensions):
    n = len(dimensions)
    dp = [[0] * n for _ in range(n)]
    split = [[0] * n for _ in range(n)]

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k + 1][j] + dimensions[i][0] * dimensions[k][1] *
dimensions[j][1]
                if cost < dp[i][j]:
                    dp[i][j] = cost
                    split[i][j] = k

    return dp, split

def construct_parenthesis(split, i, j):
    if i == j:
        return "A"
    else:
        return f"({construct_parenthesis(split, i,
split[i][j])}{construct_parenthesis(split, split[i][j] + 1, j)})"

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)
```

```
    n, dimensions = read_input(input_path)

    dp, split = matrix_chain_order(dimensions)

    optimal_parenthesis = construct_parenthesis(split, 0, n - 1)

    with open(output_path, 'w') as file:
        file.write(optimal_parenthesis + "\n")

if __name__ == '__main__':
    task_scheduler()
```

input.txt:

```
3
10 50
50 90
90 20
```

output.txt:

```
((AA)A)
```

Объяснение кода задачи о произведении матриц:

- В данной задаче требуется найти оптимальную расстановку скобок для произведения последовательности матриц, чтобы минимизировать количество операций умножения. Мы используем метод динамического программирования для решения этой задачи.

Структура кода:

1. **Чтение входных данных**:
   - Функция `read_input` считывает данные из файла. Она получает путь к файлу, считывает количество матриц `n` и их размеры (число строк и столбцов), сохраняя размеры в списке `dimensions`.
2. **Определение функции `matrix_chain_order`**:
   - Эта функция находит оптимальную расстановку скобок и количество операций для произведения матриц.
   - Мы создаем два двумерных массива:
     - `dp[i][j]` хранит минимальное количество операций для умножения матриц от `i` до `j`.
     - `split[i][j]` хранит индекс, на котором происходит разбиение матриц для оптимального умножения.
3. **Заполнение массива `dp`**:
   - Внешний цикл перебирает длину подцепочек матриц от 2 до `n`.
   - Внутренние циклы перебирают возможные начала и концы подцепочек.
   - Для каждой пары матриц `(i, j)` мы перебираем все возможные разбиения `k`, вычисляя стоимость умножения и обновляя `dp[i][j]` и `split[i][j]`, если находим более низкую стоимость.
4. **Конструкция скобок**:
   - Функция `construct_parenthesis` использует массив `split` для рекурсивного построения строки, представляющей оптимальную расстановку скобок.
   - Если текущий диапазон включает только одну матрицу, возвращается символ "A". В противном случае формируется строка с добавлением скобок для подцепочек.
5. **Определение функции `task_scheduler`**:

- В этой функции определяются пути к файлам `input.txt` и `output.txt`.
- Считываются данные и вызываются функции для вычисления оптимальной расстановки скобок.

6. **Запись результата в файл**:
   - После получения оптимальной расстановки скобок записываем результат в файл `output.txt`.
7. **Запуск программы**:
   - В конце кода проверяем, запущен ли скрипт напрямую, и если да, вызываем функцию `task_scheduler`.

Примеры работы:

- Для входных данных:

```
3
10 50
50 90
90 20
```

- Оптимальная расстановка скобок может быть `((AA)A)`.

Заключение:

- Данный код эффективно решает задачу о произведении матриц, используя динамическое программирование для минимизации операций умножения и построения оптимальной расстановки скобок. Это позволяет находить решение для больших последовательностей матриц, обеспечивая высокую производительность и точность.