

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

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

Выполнил:
Чан Тхи Лиен
К3140

Проверил:
Петросян.А.М

Санкт-Петербург
2025 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	
Задание 2. Заправки	4
Задание 6. Максимальная зарплата	7
Задание 10. Яблоки	9
Задание 15. Удаление скобок	12
Задание 17. Ход конем	14
Дополнительные задачи	
Задание 13. Сувениры	16
Задание 14. Максимальное значение арифметического выражения	18
Задание 19. Произведение матриц	21
Вывод	23

Цель работы

В данной лабораторной работе изучается еще один мощный инструмент, жадные алгоритмы. Динамическое программирование тоже продолжаем изучать.

Функция для чтения и записи файлов.

```
1  def read_file(file_path): 16 usages
2      with open(file_path, 'r', encoding='utf-8') as infile:
3          lines = infile.readlines()
4
5          lines = [line.strip() for line in lines]
6          return lines
7
8  def write_file(file_path, result): 16 usages
9      with open(file_path, 'w', encoding='utf-8') as outfile:
10         outfile.write(result)
```

- В файле “input.txt” каждая строка - это элемент списка.

Задачи по варианту

Задание 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 $stop_1, stop_2, \dots, stop_n$ из вашего родного города. Какое минимальное количество заправок необходимо?

1. Функция `min_refills`:

```
def min_refills(d, m, stops): 10 usages
    stops.append(d) # Добавляем конечный пункт
    current_position = 0
    refills = 0
    i = 0 # Индекс текущей заправочной станции

    while current_position < d:
        # Поиск самой дальней станции, до которой можно доехать
        last_refill = current_position
        while i < len(stops) and stops[i] - current_position <= m:
            last_refill = stops[i]
            i += 1

        # Если не удалось доехать ни до одной станции
        if last_refill == current_position:
            return -1 # Невозможно доехать

        # Обновляем текущую позицию
        current_position = last_refill
        refills += 1

    return refills - 1 # уменьшаем на 1, так как последняя остановка не считается
```

- Функция принимает три аргумента:
 - d — общее расстояние до пункта назначения.
 - m — максимальное расстояние, которое можно проехать на полном баке.
 - `stops` — список координат заправочных станций.
- Добавляем конечный пункт d в список `stops`, чтобы рассматривать его как последнюю остановку.
- Цикл выполняется, пока не достигнута конечная точка.
 - Внутренний цикл ищет **самую дальнюю станцию**, до которой можно доехать **без дозаправки**.
 - `last_refill` обновляется на эту станцию.
 - `i` увеличивается, чтобы проверить следующую станцию.

- Если **last_refill** не изменился, значит, доехать дальше **нельзя**, и возвращаем **-1**.
 - Обновляем текущую позицию (**current_position**).
 - Увеличиваем счетчик заправок (**refills**).
 - Возвращаем **refills - 1**, так как последняя "остановка" в **d** не требует заправки.
2. Функция для выполнения задач.

```
def task2(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    d = int(data[0])
    m = int(data[1])
    stops = [int(i) for i in data[3].split(' ')]

    result = min_refills(d, m, stops)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

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

- data - это данные в файле **input.txt**
- **d, m** - первая и вторая строка.
- **stop** - список в четвёртой строке.

3. Проведенные тесты.

- Первый

input.txt:

```
950
400
4
200 375 550 750
```

output.txt:

```
2
```

```
Время работы: 0.00048399996012449265 секунд
память использовать: 0.011719 MB
```

- Второй

input.txt:

```
200
250
2
100 150
```

output.txt:

```
0
```

```
Время работы: 0.0004805999342352152 секунд
память использовать: 0.011719 МВ
```

4. Запись теста.

Алгоритм эффективно выбирает минимальное количество заправок, чтобы гарантированно доехать до **d**, или возвращает **-1**, если это невозможно.

Задание 6. Максимальная зарплата

Постановка задачи. Составить наибольшее число из набора целых чисел.

1. Функция `compare`:

```
def compare(x, y): 2 usages

    if x + y > y + x:
        return -1
    else:
        return 1
```

- Эта функция **определяет порядок сортировки** двух строк **x** и **y**.
 - Вместо обычного числового сравнения используется **конкатенация строк**.
 - **x + y** и **y + x** представляют два возможных порядка соединения чисел.
 - Возвращаем **-1**, если **x + y** больше (то есть **x** должно стоять перед **y**).
 - Возвращаем **1**, если **y + x** больше (значит, **y** должно стоять перед **x**).
- ### 2. Функция `largest_salary_number`:

```
def largest_salary_number(numbers): 8 usages

    numbers = list(map(str, numbers))

    numbers.sort(key=cmp_to_key(compare))

    return "".join(numbers) if numbers[0] != "0" else "0"
```

- Числа преобразуются в строки, так как мы будем их **сортировать по правилам строк**.
- **cmp_to_key(compare)** превращает нашу функцию сравнения **compare** в ключ для сортировки.
- Числа сортируются в таком порядке, чтобы при конкатенации они давали **максимально возможное значение**.
- **Объединяем отсортированные числа** в одну строку.
- **Проверяем на ведущие нули**:
 - Если первый элемент **"0"**, значит, все числа были нулями (**["0", "0", "0"]**), и возвращаем просто **"0"** (чтобы избежать **"000"**).

3. Функция для выполнения задач.

```
def task6(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    numbers = [int(i) for i in data[1].split(' ')]

    result = largest_salary_number(numbers)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task6()
```

4. Проведенные тесты.

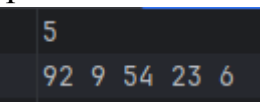
- Первый

input.txt: 

output.txt: 

Время работы: 0.007181999972090125 секунд
память использовать: 0.019531 MB

- Второй

input.txt: 

output.txt: 

Время работы: 0.000560399959795177 секунд
память использовать: 0.011719 MB

5. Запись теста.

Этот алгоритм формирует наибольшее возможное число, просто **умно сортируя элементы**.

Использует **кастомную функцию сравнения** для правильного порядка.

Задание 10. Яблоки

Постановка задачи. Алисе в стране чудес попались n волшебных яблок. Про каждое яблоко известно, что после того, как его съешь, твой рост сначала уменьшится на a_i сантиметров, а потом увеличится на b_i сантиметров. Алиса очень голодная и хочет съесть все n яблок, но боится, что в какой-то момент ее рост s станет равным нулю или еще меньше, и она пропадет совсем. Помогите ей узнать, можно ли съесть яблоки в таком порядке, чтобы в любой момент времени рост Алисы был больше нуля.

1. Функция `can_eat_apples`:

```
def can_eat_apples(n, s, apples): 10 usages

    apples_sorted = sorted(enumerate(apples, 1), key=lambda x: (min(x[1][0], x[1][1]), x[1][0]))

    order = []
    current_height = s

    for index, (a, b) in apples_sorted:
        if current_height <= a:
            return -1
        current_height -= a
        current_height += b
        order.append(index)

    return " ".join(map(str, order))
```

- Функция принимает три аргумента:
 - n — количество яблок.
 - s — изначальный размер желудка.
 - **apples** — список пар (a, b) , где:
 - a — насколько уменьшается размер желудка после поедания яблока.
 - b — насколько увеличивается размер желудка после переваривания яблока.
- `enumerate(apples, 1)` создаёт список вида $[(1, (a_1, b_1)), (2, (a_2, b_2)), \dots]$, где $1, 2, \dots$ — номера яблок.
- Сортировка по $(\min(a, b), a)$
- Сначала выбираем яблоки, где $\min(a, b)$ минимален (чтобы минимизировать риск).
- Если $\min(a, b)$ равны, сортируем по a (чтобы сначала съесть менее "опасные" яблоки).
- **order** — список порядка съедания.
- **current_height** — текущий размер желудка (s).
- Проходимся по отсортированному списку яблок.
- Если a (уменьшение размера) больше текущего размера желудка — мы не можем съесть яблоко, и возвращаем -1 .

- Если можем съесть яблоко, обновляем **current_height** и добавляем его номер в **order**.
 - Возвращаем порядок поедания яблок в виде строки.
2. Функция для выполнения задач.

```
def task10(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    array = [int(i) for i in data[0].split(' ')]
    n = array[0]
    s = array[1]
    apples = []
    for i in range(1, n + 1):
        x, y = map(int, data[i].split())
        apples.append((x, y))

    result = can_eat_apples(n, s, apples)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task10()
```

- Преобразовать первую строку в список и **n** - первый элемент, **s** - второй.
 - Добавить пары (**x**, **y**) в список **apples**.
3. Проведенные тесты.
- Первый

input.txt:

```
3 5
2 3
10 5
5 10
```

output.txt:

```
1 3 2
```

```
Время работы: 0.007914099958725274 секунд  
память использовать: 0.015625 MB
```

- Второй

```
3 5  
2 3  
10 5  
5 4
```

input.txt:

output.txt:

```
-1
```

```
Время работы: 0.0005136000690981746 секунд  
память использовать: 0.019531 MB
```

4. Запись теста.

Грамотная сортировка яблок помогает выбрать наилучший порядок
Жадный алгоритм с проходом по отсортированному списку.

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 образовывали правильную скобочную последовательность.

1. Функция `longest_valid_bracket_sequence`:

```
def longest_valid_bracket_sequence(s: str) -> str: 6 usages
    stack = []
    valid = [False] * len(s)

    matching = {')': '(', ']': '[', '}': '{'}
    opening = '(', '[', '{'

    for i, char in enumerate(s):
        if char in opening:
            stack.append((char, i))
        elif char in matching:
            if stack and stack[-1][0] == matching[char]:
                _, idx = stack.pop()
                valid[i] = valid[idx] = True

    return ''.join(s[i] for i in range(len(s)) if valid[i])
```

- Функция принимает **строку s**, содержащую скобки.
- **stack** — стек для хранения открывающих скобок и их индексов.
- **valid** — массив флагов, где **valid[i] = True**, если **s[i]** входит в корректную последовательность.
- **matching** — соответствие закрывающих скобок к открывающим.
- **opening** — множество открывающих скобок.
- Проходим по каждому символу строки.
- Если это открывающая скобка добавляем её в стек вместе с индексом.
- Если это закрывающая скобка проверяем, есть ли в стеке открывающая скобка, которая ей соответствует:
 - ❖ Если верхний элемент стека соответствует **char** → это правильная пара.
 - ❖ Устанавливаем **valid[i] = True** и **valid[idx] = True** → эти индексы являются частью правильной последовательности.
- Собираем только корректные скобки, пропуская **False** индексы.

2. Функция для выполнения задач.

```
def task15(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    s = data[0].strip()

    result = longest_valid_bracket_sequence(s)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

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

3. Проведенные тесты.

- Первый

input.txt:

output.txt:

```
Время работы: 0.006607200019061565 секунд
память использовать: 0.015625 MB
```

- Второй

input.txt:

output.txt:

```
Время работы: 0.0004923999076709151 секунд
память использовать: 0.015625 MB
```

4. Запись теста.

Жадный алгоритм с использованием стека.

Игнорирует некорректные скобки и не-скобочные символы.

Задание 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.

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

1. Определение ходов коня

```
MOD = 10 ** 9

# Возможные ходы коня
moves = {
    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],
    0: [4, 6]
}
```

2. Функция knight_dialer:

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

    dp = [1] * 10

    for _ in range(N - 1):
        new_dp = [0] * 10
        for digit in range(10):
            for next_digit in moves[digit]:
                new_dp[next_digit] = (new_dp[next_digit] + dp[digit]) % MOD
        dp = new_dp

    return sum(dp[d] for d in range(10) if d != 0 and d != 8) % MOD
```

- Функция принимает N — длину набираемого числа.
- Если $N == 1$, возможны любые цифры (1-9, но без 0 и 8).
- $dp[i]$ — количество способов попасть на цифру i в текущий момент.
- Для каждого N обновляем dp , считая количество способов попасть в $next_digit$, начиная с $digit$.
- Считаем только числа, которые не начинаются с 0 и 8.

3. Функция для выполнения задач.

```
def task17(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    N = int(data[0])

    result = knight_dialer(N)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

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

4. Проведенные тесты.

- Первый

input.txt:

output.txt:

```
Время работы: 0.0007565000560134649 секунд
память использовать: 0.019531 MB
```

- Второй

input.txt:

output.txt:

```
Время работы: 0.0004935999168083072 секунд
память использовать: 0.011719 MB
```

5. Запись теста.

Динамическое программирование с массивом **dp**.

Использует остаток по **MOD** для защиты от переполнения.

Дополнительные задачи

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 накопили.

1. Функция `can_partition`:

```
def can_partition(v): 6 usages
    total_sum = sum(v)

    # Если сумма не делится на 3, то разделить на 3 части невозможно
    if total_sum % 3 != 0:
        return 0

    target = total_sum // 3
    n = len(v)

    # Динамическое программирование для поиска возможных подмножеств
    # dp[i] будет хранить информацию, можно ли получить сумму i с помощью подмножеств
    dp = [False] * (target + 1)
    dp[0] = True # Сумма 0 всегда достижима

    for value in v:
        for j in range(target, value - 1, -1):
            if dp[j - value]:
                dp[j] = True

    # Если сумма target достижима, то проверяем, можно ли разделить на 3 равные части
    if dp[target]:
        return 1

    return 0
```

- Функция принимает `v` — список чисел.
- Вычисляем общую сумму `total_sum`.
- Если `total_sum` не делится на 3, сразу возвращаем 0 (разделить нельзя).
- `target` — это сумма, которую должно иметь каждое из трёх подмножеств.
- `dp[i]` — можно ли получить сумму `i` из элементов `v`.
- Проходим по массиву, пытаясь собрать подмножество с суммой `target`.
- Обновляем `dp[j]`, если `dp[j - value]` уже достижимо.
- Если можно собрать одно подмножество с суммой `target`, возвращаем 1.
- Иначе 0.

2. Функция для выполнения задач.

```
def task13(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = data[0]
    v = [int(i) for i in data[1].split(' ')]

    result = can_partition(v)

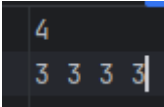
    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

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

3. Проведенные тесты.

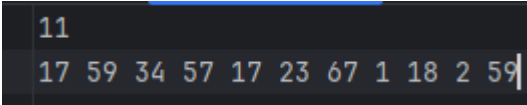
- Первый

input.txt: 

output.txt: 

Время работы: 0.006059799925424159 секунд
память использовать: 0.011719 MB

- Второй

input.txt: 

output.txt: 

Время работы: 0.0005151999648660421 секунд
память использовать: 0.011719 MB

4. Запись теста.

Использует динамическое программирование.

Проверяет только одно подмножество, но на практике этого может быть недостаточно.

Задание 14. Максимальное значение арифметического выражения

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

1. Функция `calculate`:

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

- Обычная функция вычисления результата для двух чисел **a** и **b** с оператором **op**.

2. Функция `max_expression_value`:

```
def max_expression_value(expr):  
    # Преобразуем выражение в два списка: операнды и операторы  
    numbers = [int(expr[i]) for i in range(0, len(expr), 2)]  
    operators = [expr[i] for i in range(1, len(expr), 2)]  
  
    n = len(numbers)  
  
    # Массивы для хранения максимальных и минимальных значений  
    max_val = [[float('-inf')] * n for _ in range(n)]  
    min_val = [[float('inf')] * n for _ in range(n)]  
  
    # Инициализация для одноэлементных подстрок  
    for i in range(n):  
        max_val[i][i] = numbers[i]  
        min_val[i][i] = numbers[i]  
  
    # Заполнение динамических таблиц  
    for length in range(2, n + 1): # Длина подстроки от 2 до n  
        for i in range(n - length + 1): # Начало подстроки  
            j = i + length - 1 # Конец подстроки  
            for k in range(i, j): # Разделяем подстроку по операторам  
                op = operators[k] # Оператор между числами  
                # Для всех возможных комбинаций применения операции  
                a, b = max_val[i][k], max_val[k + 1][j]  
                min_val[i][j] = min(min_val[i][j], calculate(min_val[i][k], min_val[k + 1][j], op),  
                                     calculate(min_val[i][k], max_val[k + 1][j], op),  
                                     calculate(max_val[i][k], min_val[k + 1][j], op),  
                                     calculate(max_val[i][k], max_val[k + 1][j], op))  
                max_val[i][j] = max(max_val[i][j], calculate(min_val[i][k], min_val[k + 1][j], op),  
                                     calculate(min_val[i][k], max_val[k + 1][j], op),  
                                     calculate(max_val[i][k], min_val[k + 1][j], op),  
                                     calculate(max_val[i][k], max_val[k + 1][j], op))  
  
    return max_val[0][n - 1]
```

- Принимает **expr** — строку, содержащую числа и операции.
 - Разбиваем строку на числа и операторы
 - Числа (**0, 2, 4, ...** индексы) сохраняются в **numbers**.
 - Операторы (**1, 3, 5, ...** индексы) сохраняются в **operators**.
 - Создаём таблицы для динамического программирования
 - **max_val[i][j]** — максимальное значение выражения от **i** до **j**.
 - **min_val[i][j]** — минимальное значение выражения от **i** до **j**.
 - Заполняем таблицы для одноэлементных подстрок
 - Если **i == j**, то значение подстроки — это само число.
 - Перебираем длины подстрок от **2** до **n**.
 - Перебираем все возможные подстроки длины **length**, начиная с **i** и заканчивая **j**.
 - Разделяем подстроку по операторам
 - Вычисляем максимальное и минимальное значение
 - Пробуем все возможные варианты вычисления: (**мин, мин**), (**мин, макс**), (**макс, мин**), (**макс, макс**)
 - Аналогично вычисляем **max_val[i][j]**, но берем максимум.
 - Возвращаем максимальное значение выражения от **0** до **n-1**.
3. Функция для выполнения задач.

```
def task14(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    expr = data[0].strip()

    result = max_expression_value(expr)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

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

4. Проведенные тесты.

- Первый

input.txt: 1+5

output.txt: 6

```
Время работы: 0.0006796000525355339 секунд  
память использовать: 0.023438 МВ
```

- Второй

input.txt: 5-8+7*4-8+9

output.txt: 200

```
Время работы: 0.0005033999914303422 секунд  
память использовать: 0.023438 МВ
```

5. Запись теста.

Алгоритм динамического программирования (**матрица $O(N^2)$**).

Перебирает все возможные расстановки скобок.

Оптимальное вычисление выражения для максимального результата.

Задание 19. Произведение матриц

Постановка 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.

1. Функция `matrix_chain_order`:

```
def matrix_chain_order(p): 5 usages
    n = len(p) - 1
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]

    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k + 1][j] + p[i] * p[k + 1] * p[j + 1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k

    return m, s
```

- Принимает **p** — список размеров матриц.
- Инициализация таблиц.
- Проходим по различным длинам цепочек (**l** от 2 до **n**).
- Перебираем возможные начала **i** и окончания **j**.
- **k** — индекс разбиения цепочки.
- Формула рекурсии - Это учитывает стоимость разбиения на две части.

2. Функция `optimal_parenthesis`:

```
def optimal_parenthesis(s, i, j): 6 usages
    if i == j:
        return f"A"
    else:
        return f"({optimal_parenthesis(s, i, s[i][j])}{optimal_parenthesis(s, s[i][j] + 1, j)})"
```

- Восстанавливает оптимальный порядок умножения матриц.

- Если одна матрица — возвращаем "A" (матрица без скобок).
 - Оптимальное разбиение определяется по $s[i][j]$.
 - Рекурсивно оборачиваем в скобки.
3. Функция для выполнения задач.

```
def task19(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    matrices = [tuple(map(int, data[i].split())) for i in range(1, n+1)]

    p = [matrices[0][0]] + [b for _, b in matrices]

    m, s = matrix_chain_order(p)

    result = optimal_parenthesis(s, i: 0, n-1)

    write_file(PATH_OUTPUT, str(result))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task19()
```

4. Проведенные тесты.

- Первый

input.txt:

3
10 50
50 90
90 20

output.txt:

((AA)A)

```
Время работы: 0.008399499929510057 секунд
память использовать: 0.050781 MB
```

- Второй

```
5
10 20
20 30
30 40
40 50
50 60
```

input.txt:

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

output.txt:

```
Время работы: 0.0007193000055849552 секунд
память использовать: 0.023438 MB
```

5. Запись теста.

Решает задачу умножения цепочки матриц с минимальным числом скалярных операций.

Использует динамическое программирование ($O(n^3)$).

Восстанавливает оптимальный порядок умножения.

Вывод:

- Решение заданий
- Использование мощных инструментов: грамотной сортировки, жадного алгоритма и динамического программирования