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

Отчет по лабораторной работе №7
по курсу «Алгоритмы и структуры данных»
Тема: Динамическое программирование №1
Вариант 9

Выполнила:
Левчук С.А.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задание №6. Наибольшая возрастающая подпоследовательность

6 задача. Наибольшая возрастающая подпоследовательность

Дана последовательность, требуется найти ее наибольшую возрастающаю подпоследовательность.

- **Формат ввода / входного файла (input.txt).** В первой строке входных данных задано целое число n – длина последовательности ($1 \leq n \leq 300000$).

  Во второй строке задается сама последовательность. Числа разделяются пробелом.

  Элементы последовательности – целые числа, не превосходящие по модулю 10^9.

  - Подзадача 1 (полегче). $n \leq 5000$.
  - Общая подзадача. $n \leq 300000$.

- **Формат вывода / выходного файла (output.txt).** В первой строке выведите длину наибольшей возрастающей подпоследовательности, а во второй строке выведите через пробел саму наибольшую возрастающаю подпоследовательность данной последовательности. Если ответов несколько - выведите любой.

- Ограничение по времени. 2 сек.

- Ограничение по памяти. 256 мб.

- Пример:

input.txt	output.txt
6 3 29 5 5 28 6	3 3 5 28

Решение:

```
from utils import read, write
from bisect import bisect_left


def subsequence(n, sequence):
    lis = []
    parent = [-1] * n
    indexes = []

    for i in range(n):
        pos = bisect_left(lis, sequence[i])
        if pos == len(lis):
            lis.append(sequence[i])
            indexes.append(i)
        else:
            lis[pos] = sequence[i]
            indexes[pos] = i

        if pos > 0:
            parent[i] = indexes[pos - 1]
```

```
    lis_length = len(lis)
    lis_sequence = []
    current_index = indexes[-1]

    for _ in range(lis_length):
        lis_sequence.append(sequence[current_index])
        current_index = parent[current_index]

    lis_sequence.reverse()
    return [lis_length], lis_sequence


def main():
    write(end='')
    data = [line for line in read(type_convert=int)]
    n = data[0][0]
    array = data[1]
    result = subsequence(n, array)
    for line in result:
        write(*line, to_end=True)


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

Алгоритм:

1. Инициализация переменных
    1) lis: список для хранения элементов текущей наибольшей возрастающей подпоследовательности.
    2) parent: массив для хранения индекса родителя текущего элемента в lis.
    3) indexes: массив для хранения индексов элементов в sequence, соответствующих элементам в lis.
2. Цикл: используем бинарный поиск (bisect_left) для нахождения позиции, куда можно вставить текущий элемент sequence[i] в отсортированном списке lis.
    1) Если позиция равна длине lis, значит, элемент больше любого другого в lis, и мы просто добавляем его в конец.
    2) Иначе заменяем элемент на позицию pos новым значением.

    Обновляем массив indexes, чтобы отслеживать индекс элемента в исходной последовательности.

    Если позиция больше нуля, устанавливаем родительский индекс для текущего элемента как индекс элемента, стоящего перед ним в lis.
3. “Собираем” искомую последовательность:
    1) Идем назад по массиву parent от последнего элемента.

2) Переворачиваем полученную последовательность, так как она была построена в обратном порядке.
3) Возвращаем длину наибольшей возрастающей подпоследовательности и саму последовательность.

Результат работы программы:

Входные данные:

```
6
3 29 5 5 28 6
```

Выходные данные:

```
3
3 5 6
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab7.task6.src.subsequence import main, subsequence


class TestSubsequence(unittest.TestCase):

    def test_should_work_with_small_input(self):
        # given
        n = 6
        array = [3, 29, 5, 5, 28, 6]
        expected_result = ([3], [3, 5, 6])

        # when
        result = subsequence(n, array)

        # then
        self.assertEqual(result, expected_result)

    def test_should_work_when_all_equal_elements(self):
        # given
        n = 5
        sequence = [7, 7, 7, 7, 7]
        exp_length = [1]
        exp_sequence = [7]

        # when
        length, sequence = subsequence(n, sequence)

        # then
        self.assertEqual(length, exp_length)
        self.assertEqual(sequence, exp_sequence)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
```

```
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали функцию, которая находит наибольшую возрастающую подпоследовательность в заданной последовательности и возвращает ее длину и саму последовательность.

Задание №7. Шаблоны

7 задача. Шаблоны

Многие операционные системы используют шаблоны для ссылки на группы объектов: файлов, пользователей, и т. д. Ваша задача – реализовать простейший алгоритм проверки шаблонов для имен файлов.

В этой задаче алфавит состоит из маленьких букв английского алфавита и точки («.»). Шаблоны могут содержать произвольные символы алфавита, а также два специальных символа: «?» и «*». Знак вопроса («?») соответствует ровно одному произвольному символу. Звездочка «+» соответствует подстроке произвольной длины (возможно, нулевой). Символы алфавита, встречающиеся в шаблоне, отображаются на ровно один такой же символ в проверяемой строчке. Строка считается подходящей под шаблон, если символы шаблона можно последовательно отобразить на символы строки таким образом, как описано выше. Например, строчки «ab», «aab» и «beda.» подходят под шаблон «*a?», а строчки «bebe», «a» и «ba» –нет.

- **Формат ввода / входного файла (input.txt).** Первая строка входного файла определяет шаблон . Вторая строка S состоит только из символов алфавита. Ее необходимо проверить на соответствие шаблону. Длины обеих строк не превосходят 10 000. Строки могут быть пустыми – будьте внимательны!
- **Формат вывода / выходного файла (output.txt).** Если данная строка подходит под шаблон, выведите YES. Иначе выведите NO.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
k?t*n kitten	YES

input.txt	output.txt
k?t?n kitten	NO

Решение:

```
from utils import read, write


def patterns(pattern: str, string: str) -> str:
    n, m = len(pattern), len(string)
    dp = [[False] * (m + 1) for _ in range(n + 1)]

    dp[0][0] = True

    for i in range(1, n + 1):
        if pattern[i - 1] == '*':
            dp[i][0] = dp[i - 1][0]
        else:
            break

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if pattern[i - 1] == string[j - 1] or pattern[i - 1] == '?':
                dp[i][j] = dp[i - 1][j - 1]
            elif pattern[i - 1] == '*':
                dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
```

```
    return 'YES' if dp[n][m] else 'NO'


def main():
    write(end='')
    data = [line for line in read(type_convert=str)]
    pattern, string = data[0][0], data[1][0]
    write(patterns(pattern, string), to_end=True)


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

Алгоритм:

1. Инициализация переменных:
    1) n и m — длины шаблона и строки соответственно.
    2) dp — двумерный массив булевых значений размером (n+1) × (m+1), инициализированный значениями False.
2. Базовый случай: случай, когда и шаблон, и данная строка пусты, считается истинным.
3. Основная проверка: если первый символ шаблона — *, то этот символ может соответствовать нулевой длине строки. Поэтому заполняется первая колонка dp значениями, соответствующими предыдущему состоянию.
4. Основной цикл (заполнение динамического массива):
    1) Если текущий символ шаблона совпадает с текущим символом строки или является ?, то текущее состояние равно состоянию, смещенному на одну клетку вверх и влево.
    2) Если текущий символ шаблона — *, то текущее состояние равно логическому ИЛИ состояний, смещённых на одну клетку вверх или влево.
5. Если последнее состояние в матрице dp истинно, то шаблон соответствует строке, и возвращается "YES", иначе — "NO".

Результат работы программы:

Входные данные:

```
k?t*n
kitten
```

Выходные данные:

```
YES
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab7.task7.src.patterns import main, patterns
```

```
class TestPatternMatching(unittest.TestCase):

    def test_should_match_pattern(self):
        # given
        pattern = "ab*"
        string = "aba"
        expected_result = 'YES'

        # when
        result = patterns(pattern, string)

        # then
        self.assertEqual(result, expected_result)

    def test_should_not_match_pattern(self):
        # given
        pattern = "a*b*c"
        string = "abc"
        expected_result = 'YES'

        # when
        result = patterns(pattern, string)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм, который проверяет соответствие данных строки и шаблона, с использованием динамического массива.

Дополнительные задачи

Задание №1. Обмен монет

1 задача. Обмен монет

Как мы уже поняли из лекции, не всегда "жадное"решение задачи на обмен монет работает корректно для разных наборов номиналов монет. Например, если доступны номиналы 1, 3 и 4, жадный алгоритм поменяет 6 центов, используя три монеты (4 + 1 + 1), в то время как его можно изменить, используя всего две монеты (3 + 3). Теперь ваша цель - применить динамическое программирование для решения задачи про обмен монет для разных номиналов.

- **Формат ввода / входного файла (input.txt).** Целое число $money$ ($1 \leq money \leq 10^3$). Набор монет: количество возможных монет k и сам набор $coins = \{coin_1, ..., coin_k\}$. $1 \leq k \leq 100$, $1 \leq coin_i \leq 10^3$. Проверку можно сделать на наборе $\{1, 3, 4\}$. Формат ввода: первая строка содержит через пробел $money$ и k; вторая - $coin_1 coin_2 ... coin_k$.
  - ***Вариация 2:*** Количество монет в кассе ограничено. Для каждой монеты из набора $coins = \{coin_1, ..., coin_k\}$ есть соответствующее целое число - количество монет в кассе данного номинала $c = \{c_1, ..., c_k\}$. Если они закончились, то выдать данную монету невозможно.
- **Формат вывода / выходного файла (output.txt).** Вывести одно число – минимальное количество необходимых монет для размена $money$ доступным набором монет $coins$.
- Ограничение по времени. 1 сек.
- Примеры:

input.txt	output.txt
2 3 1 3 4	2

input.txt	output.txt
34 3 1 3 4	9

Решение:

```
from utils import read, write


def coin_exchange(money, coins):
    dp = [float('inf')] * (money + 1)

    dp[0] = 0

    for i in range(1, money + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[money]


def main():
    write(end='')
```

```
    data = [list(line) for line in read(type_convert=int)]
    money, k = data[0]
    coins = data[1]
    write(coin_exchange(money, coins), to_end=True)


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

Алгоритм:

1. Создаем массив dp длиной money + 1, заполненный значениями бесконечности (float('inf')), кроме первого элемента, который равен нулю. Значение dp[i] будет хранить минимальное количество монет, необходимое для размена суммы i.
2. Базовый случай: для суммы 0 требуется 0 монет.
3. Основной цикл:
    1) Внешний цикл перебирает все возможные суммы от 1 до money.
    2) Внутренний цикл перебирает все доступные монеты.
    3) Если текущая монета меньше или равна сумме i, то обновляем значение dp[i], выбирая минимум между текущим значением и количеством монет, необходимым для размена суммы i - coin, плюс одна монета.
4. Возвращаем минимальное количество монет, необходимых для размена всей суммы money.

Результат работы программы:

Входные данные:

```
2 3
1 3 4
```

Выходные данные:

```
2
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab7.task1.src.coin_exchange import main, coin_exchange


class TestCoinExchange(unittest.TestCase):

    def test_should_work_with_small_amount_of_money(self):
        # given
        money = 6
        coins = [1, 3, 4]
        expected_result = 2
```

```
        # when
        result = coin_exchange(money, coins)

        # then
        self.assertEqual(result, expected_result)

    def test_should_work_with_large_amount_of_money(self):
        # given
        money = 10000
        coins = [100, 200, 400]
        expected_result = 25

        # when
        result = coin_exchange(money, coins)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
        # given
        expected_time = 1

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали функцию для решения задачи про обмен монет, используя динамическое программирование.

Задание №2. Примитивный калькулятор

2 задача. Примитивный калькулятор

Дан примитивный калькулятор, который может выполнять следующие три операции с текущим числом x: умножить x на 2, умножить x на 3 или прибавить 1 к x. Дано положительное целое число n, найдите минимальное количество операций, необходимых для получения числа n, начиная с числа 1.

- **Формат ввода / входного файла (input.txt).** Дано одно целое число n, $1 \leq n \leq 10^6$. Посчитать минимальное количество операций, необходимых для получения n из числа 1.
- **Формат вывода / выходного файла (output.txt).** В первой строке вывести минимальное число k операций. Во второй – последовательность промежуточных чисел $a_0, a_1, ..., a_{k-1}$ таких, что $a_0 = 1, a_{k-1} = n$ и для всех $0 \leq i < k - a_{i+1}$ равно или $a_i + 1$, $2 \cdot a_i$, или $3 \cdot a_i$. Если есть несколько вариантов, выведите любой из них.

- Ограничение по времени. 1 сек.
- Примеры:

input.txt	output.txt
1	0 1

input.txt	output.txt
5	3 1 2 4 5

input.txt	output.txt
96234	14 1 3 9 10 11 22 66 198 594 1782 5346 16038 16039 32078 96234

Решение:

```
from utils import write, read


def primitive_calculator(n):
    dp = [float('inf')] * (n + 1)
    prev = [-1] * (n + 1)

    dp[1] = 0

    for i in range(1, n + 1):
        if i + 1 <= n:
            if dp[i + 1] > dp[i] + 1:
                dp[i + 1] = dp[i] + 1
                prev[i + 1] = i

        if i * 2 <= n:
            if dp[i * 2] > dp[i] + 1:
                dp[i * 2] = dp[i] + 1
                prev[i * 2] = i

        if i * 3 <= n:
            if dp[i * 3] > dp[i] + 1:
                dp[i * 3] = dp[i] + 1
                prev[i * 3] = i
```

```
    path = []
    current = n
    while current != 1:
        path.append(current)
        current = prev[current]
    path.append(1)
    path.reverse()

    return [dp[n]], path


def main():
    write(end='')
    n, = read(type_convert=int)
    n = n[0]
    result = primitive_calculator(n)
    for line in result:
        write(*line, to_end=True)


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

Алгоритм:

1. Инициализация переменных:
    1) dp: массив для хранения минимальных количеств операций для каждого числа от 1 до n.
    2) prev: массив для хранения предшествующих чисел в оптимальном пути.
2. Базовый случай: для числа 1 минимальное количество операций равно 0, так как оно уже достигнуто.
3. Для каждого числа i от 1 до n рассматриваются три операции:
    1) Увеличение на 1: i + 1.
    2) Умножение на 2: i * 2.
    3) Умножение на 3: i * 3. Если новое значение меньше текущего, оно обновляется вместе с соответствующим предшественником.
4. Находим путь от n до 1, следуя по массиву prev, и добавляем шаги в список path. Затем переворачиваем список, чтобы получить правильный порядок шагов.
5. Возвращаем минимальное количество операций и сам путь.

Результат работы программы:

Входные данные:

```
5
```

Выходные данные:

```
3
1 2 4 5
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab7.task2.src.primitive_calculator import main, primitive_calculator


class TestPrimitiveCalculator(unittest.TestCase):

    def test_should_work_for_small_input(self):
        # given
        n = 6
        expected_output = ([2], [1, 2, 6])

        # when
        result = primitive_calculator(n)

        # then
        self.assertEqual(result, expected_output)

    def test_should_work_for_large_input(self):
        # given
        n = 96234
          expected_output = ([14], [1, 3, 9, 10, 11, 22, 66, 198, 594, 1782,
5346, 16038, 16039, 32078, 96234])

        # when
        result = primitive_calculator(n)

        # then
        self.assertEqual(result, expected_output)

    def test_should_check_time_data(self):
        # given
        expected_time = 1

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self. assertLess(peak, expected_memory)
```

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

Вывод по задаче:

В ходе решения данной задачи мы с помощью динамического программирования реализовали алгоритм для нахождения минимального количества операций для получения числа n, начиная с 1.

Вывод

В ходе выполнения лабораторной работы №7 мы изучили изучили такой инструмент как динамическое программирование, а также решили несколько задач, применяя его.