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

Отчет по лабораторной работе №6
по курсу «Алгоритмы и структуры данных»
Тема: Динамическое программирование №1
Вариант 6

Выполнил:
Данилова А.В.
К3141 (номер группы)

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

6 задача. Наибольшая возрастающая подпоследовательность

Дана последовательность, требуется найти ее наибольшую возрастающаю подпоследовательность.

Листинг кода:

```
from pathlib import Path
from lab_7.utils.utils import work


def lis(n, arr):
    if not arr:
        return []

    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    max_length = max(dp)
    max_index = dp.index(max_length)

    result = [arr[max_index]]
    for i in range(max_index - 1, -1, -1):
        if arr[i] < arr[max_index] and dp[i] == max_length - 1:
            result.append(arr[i])
            max_length -= 1
            max_index = i

    return len(result), ' '.join(map(str, result[::-1]))


if __name__ == "__main__":
    work(Path(__file__), lis)
```

Этот код реализует функцию для нахождения длины и элементов самой длинной возрастающей подпоследовательности в заданном массиве чисел. Функция lis принимает два аргумента: n (количество элементов в массиве) и arr (сам массив). Она использует динамическое программирование для вычисления длины самой длинной возрастающей подпоследовательности. В начале создается массив dp, где каждый элемент инициализируется значением 1, так как минимальная длина возрастающей подпоследовательности для любого элемента — это он сам. Затем, с помощью двух вложенных циклов, функция сравнивает элементы массива. Если текущий элемент больше предыдущего, она обновляет значение в

массиве dp, чтобы отразить максимальную длину возрастающей подпоследовательности, заканчивающейся на текущем элементе. После завершения циклов функция находит максимальную длину возрастающей подпоследовательности и ее индекс. Затем, начиная с этого индекса, она собирает элементы самой длинной подпоследовательности, проверяя условия для добавления элементов в результирующий список. В конце функция возвращает длину найденной подпоследовательности и строку, содержащую её элементы в порядке возрастания.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 7
            TASK NUMBER: 6
                    INPUT DATA: (6, [3, 29, 5, 5, 28, 1])
                    OUTPUT DATA: (3, '3 5 28')
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.6700010746717453e-05 секунд	0.0006017684936523438 Б
Верхняя граница диапазона значений входных данных из текста задачи	1.5082141000311822 секунд	0.5500526428222656 Б

Вывод по задаче:
Код не только вычисляет длину наибольшей возрастающей подпоследовательности, но и восстанавливает саму последовательность. Это демонстрирует, что динамическое программирование может быть использовано не только для нахождения оптимального значения, но и для получения самого решения.

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.

Листинг кода:

```
from pathlib import Path
import sys
from lab_7.utils.utils import work

sys.setrecursionlimit(10 ** 6)


def is_match(p, w, m, n, exist):
    key = (m, n)

    if m == len(p):
        exist[key] = (n == len(w))
        return m == len(w)

    if n == len(w):
        for x in p[m:]:
            if x is False:
                exist[key] = False
                return False
        exist[key] = True
        return True

    if p[m] == '?' or p[m] == w[n]:
        exist[key] = is_match(p, w, m + 1, n + 1, exist)
    elif p[m] == '*':
        exist[key] = is_match(p, w, m, n + 1, exist) or is_match(p, w, m +
1, n, exist)
    else:
        exist[key] = False

    return exist.get(key)
```

```
def main(pattern, word):
    exist = {}
    if is_match(pattern[0], word[0], 0, 0, exist):
        return 'YES'
    return 'NO'


if __name__ == "__main__":
    work(Path(__file__), main)
```

Этот код реализует алгоритм для проверки, соответствует ли данное слово определённому шаблону, который может содержать символы подстановки. Шаблон может включать символ ?, который соответствует любому одному символу, и символ *, который соответствует любому количеству символов, включая ноль. В коде используется рекурсивная функция, которая принимает шаблон и слово, а также текущие позиции в обоих, и словарь для хранения уже вычисленных результатов (мемоизация). Функция проверяет различные условия, такие как достижение конца шаблона или слова, а также соответствие текущих символов. Если текущий символ шаблона — *, функция проверяет два варианта: игнорирование * или использование его для совпадения с одним или несколькими символами слова. Основная функция запускает процесс проверки, и если слово соответствует шаблону, возвращает "YES", в противном случае — "NO". Код также содержит блок, который позволяет ему работать как самостоятельный скрипт, обрабатывая входные данные и передавая их в основную функцию.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 7
            TASK NUMBER: 7
                    INPUT DATA: (['k?t*n'], ['kitten'])
                    OUTPUT DATA: YES
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.2400036212056875e-05 секунд	0.00037384033203125 Б

Верхняя граница диапазона значений входных данных из текста задачи	0.00040770001942291856 секунд	0.0756988525390625 Б

Вывод по задаче:
Использование массива dp для хранения длины наибольшей возрастающей подпоследовательности, заканчивающейся на каждом элементе, является типичным подходом в динамическом программировании. Это позволяет избежать повторного вычисления одних и тех же значений.
.

Дополнительные задачи

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.

Листинг кода:

```
from pathlib import Path
from collections import defaultdict
from lab_7.utils.utils import work


def coin_exchange(money, k, coins):
    mcn = defaultdict(int)
    for coin in coins:
        mcn[coin] = 1
    for i in range(min(coins), money + 1):
        if i not in mcn:
            mcn[i] = 10 ** 9
            for coin in coins:
                if coin < i:
                    mcn[i] = min(mcn[i], mcn[i - coin] + 1)
    return mcn[money]


def coin_exchange_2(money, k, coins, counts):
    mcn = defaultdict(int)
```

```
    for coin in coins:
        mcn[coin] = 1
    for i in range(min(coins), money + 1):
        if i not in mcn:
            mcn[i] = 10 ** 9
            free = counts
            for coin in coins:
                if coin < i and free[coin - 1] > 0:
                    mcn[i] = min(mcn[i], mcn[i - coin] + 1)
    return mcn[money]


if __name__ == "__main__":
    work(Path(__file__), coin_exchange)
```

Функция coin_exchange:

•Принимает на вход сумму денег (money), количество различных монет (k) и список доступных монет (coins).

• Создаёт словарь mcn, который будет хранить минимальное количество монет, необходимых для каждой суммы от 0 до money.

• Инициализирует количество монет для каждого номинала, равным 1.

• Проходит по всем возможным суммам от минимального номинала до money и для каждой суммы проверяет, можно ли её составить из ранее вычисленных значений с помощью имеющихся монет. Если да, обновляет минимальное количество монет.

Функция coin_exchange_2:

• Расширяет первую функцию, добавляя возможность учитывать ограниченное количество монет каждого номинала (параметр counts).

• Логика аналогична первой функции, но дополнительно проверяется наличие доступных монет перед обновлением значений.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 7
            TASK NUMBER: 1
                         INPUT DATA: (40, 4, [25, 20, 10, 5])
                         OUTPUT DATA: 2
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из	2.989999484270811e-05 секунд	0.001861572265625 Б

текста задачи		
Верхняя граница диапазона значений входных данных из текста задачи	0.007050699961837381 секунд	0.06402587890625 Б

Вывод по задаче:
Динамическое программирование позволяет легко адаптировать алгоритм под различные условия, как в случае с ограниченным количеством монет в coin_exchange_2.

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.

Листинг кода:

```
from pathlib import Path
from collections import defaultdict
from lab_7.utils.utils import work


def simple_calculator(n):
    if n == 1:
        return 0, 1
    md = defaultdict(list)
    md[1] = [1]
    md[2] = [1, 2]
    md[3] = [1, 3]
    for i in range(4, n + 1):
        if i not in md:
            md[i] = md[i - 1] + [i]
            if i % 6 == 0:
                md[i] = sorted([md[i // 3] + [i], md[i // 2] + [i], md[i]],
key=len)[0]
            elif i % 3 == 0:
                md[i] = sorted([md[i // 3] + [i], md[i]], key=len)[0]
            elif i % 2 == 0:
                md[i] = sorted([md[i // 2] + [i], md[i]], key=len)[0]
    return len(md[n]) - 1, ' '.join(map(str, md[n]))
```

```
if __name__ == "__main__":
    work(Path(__file__), simple_calculator)
```

Для каждого числа от 4 до n функция проверяет, существует ли уже последовательность для этого числа. Если нет, то она создаёт новую последовательность, добавляя текущее число к последовательности предыдущего числа (i-1). Затем выполняется проверка на делимость: • Если число делится на 6, выбирается наименьшая по длине последовательность из трёх вариантов: последовательность числа, делённого на 3, число, делённое на 2, и текущее число. • Если число делится на 3 или 2, аналогично выбирается наименьшая по длине последовательность из двух вариантов.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 7
            TASK NUMBER: 2
                    INPUT DATA: (96234,)
                    OUTPUT DATA: (14, '1 3 9 10 11 22 66 198 594 1782 5346 16038 16039 32078 96234')
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	5.700043402612209e-06 секунд	0.0002288818359375 Б
Верхняя граница диапазона значений входных данных из текста задачи	0.0939059000229463 секунд	26.10659408569336 Б

Вывод по задаче:
Динамическое программирование предоставляет возможность легко модифицировать алгоритм для учёта дополнительных условий или правил, что делает его мощным инструментом для решения сложных задач оптимизации.

Вывод

Данная лабораторная работа демонстрирует эффективность динамического программирования в решении задач с многими состояниями и зависимостями, обеспечивая как скорость выполнения, так и простоту модификации алгоритма.