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

Выполнила:
Беляева В.А.
Группа:
К3139

Проверил:

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №4 Наибольшая общая подпоследовательность двух по следовательностей

Условие:

Вычислить длину самой длинной общей подпоследовательности из двух последовательностей.

Даны две последовательности $A = (a_1, a_2, ..., a_n)$ и $B = (b_1, b_2, ..., b_m)$, найти длину их самой длинной общей подпоследовательности, т.е. наибольшее неотрицательное целое число p такое, что существуют индексы $1 \leq i_1 < i_2 < ... < i_p \leq n$ и $1 \leq j_1 < j_2 < ... < j_p \leq m$ такие, что $a_{i_1} = b_{j_1}, ..., a_{i_p} = b_{j_p}$.

- **Формат ввода / входного файла (input.txt).**
  - Первая строка: n - длина первой последовательности.
  - Вторая строка: $a_1, a_2, ..., a_n$ через пробел.
  - Третья строка: m - длина второй последовательности.
  - Четвертая строка: $b_1, b_2, ..., b_m$ через пробел.
- Ограничения: $1 \leq n, m \leq 100; -10^9 < a_i, b_i < 10^9$.

Листинг кода:

```
def read_sequences(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        n_line = f.readline().strip()
        if not n_line:
            return [], []
        n = int(n_line)
        if n > 0:
            seqA = list(map(int, f.readline().strip().split()))
        else:
            seqA = []
        m_line = f.readline().strip()
        if not m_line:
            return seqA, []
        m = int(m_line)
        if m > 0:
            seqB = list(map(int, f.readline().strip().split()))
        else:
            seqB = []
    return seqA, seqB
def lcs_length(seqA, seqB):
    n = len(seqA)
    m = len(seqB)
    dp = [[0]*(m+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, m+1):
            if seqA[i-1] == seqB[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[n][m]
```

```
def task4():
    seqA, seqB = read_sequences(PATH)
    result = lcs_length(seqA, seqB)
    write_result(result, OUTPUT_PATH)
```

Объяснение: Код предназначен для нахождения длины наибольшей общей подпоследовательности (LCS) двух последовательностей. Функция `read_sequences` считывает данные из файла. Она получает количество элементов и сами элементы двух последовательностей, проверяя их наличие. Если последовательности пусты, возвращаются пустые списки. Функция `lcs_length` реализует вычисление длины LCS с помощью метода динамического программирования. Создаётся двумерная таблица `dp`, где каждая ячейка хранит длину LCS для соответствующих подстрок. Если элементы двух последовательностей совпадают, текущая ячейка обновляется значением из диагональной ячейки плюс единица. В противном случае берётся максимум значений из соседних ячеек. Итоговая длина LCS находится в правом нижнем углу таблицы. Функция `task4` объединяет эти процессы: считывает данные, вычисляет LCS и записывает результат в файл.

input	output
3 2 7 5 2 25	2
1 7 4 1 2 3 4	0

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница	0.001	0.002
Пример	0.003	0.005
Верхняя граница	01.10.00	01.01.00

Вывод : Код эффективно вычисляет длину наибольшей общей подпоследовательности с временной сложностью O(n·m)O(n \c*m), где n и m — длины последовательностей. Пространственная сложность также O(n·m)O(n \c*m)из-за использования таблицы. Подход на основе динамического программирования обеспечивает точное решение задачи, хотя может быть затратным для длинных последовательностей. Код читает и записывает данные из файлов, что делает его удобным для больших наборов данных.

Задача №5 Наибольшаяобщаяподпоследовательноститрехпо следовательностей

Условие:

Вычислить длину самой длинной общей подпоследовательности из *трех* последовательностей.

Даны три последовательности $A = (a_1, a_2, ..., a_n)$, $B = (b_1, b_2, ..., b_m)$ и $C = (c_1, c_2, ..., c_l)$, найти длину их самой длинной общей подпоследовательности, т.е. наибольшее неотрицательное целое число p такое, что существуют индексы $1 \leq i_1 < i_2 < ... < i_p \leq n$, $1 \leq j_1 < j_2 < ... < j_p \leq m$ и $1 \leq k_1 < k_2 < ... < k_p \leq l$ такие, что $a_{i_1} = b_{j_1} = c_{k_1}, ..., a_{i_p} = b_{j_p} = c_{k_p}$.

- **Формат ввода / входного файла (input.txt).**
  - Первая строка: n - длина первой последовательности.
  - Вторая строка: $a_1, a_2, ..., a_n$ через пробел.
  - Третья строка: m - длина второй последовательности.
  - Четвертая строка: $b_1, b_2, ..., b_m$ через пробел.
  - Пятая строка: l - длина второй последовательности.
  - Шестая строка: $c_1, c_2, ..., c_l$ через пробел.
- Ограничения: $1 \leq n, m, l \leq 100$; $-10^9 < a_i, b_i, c_i < 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите число p.
- Ограничение по времени. 1 сек.

Листинг кода:

```
def read_three_sequences(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        n_line = f.readline().strip()
        n = int(n_line) if n_line else 0
        A = list(map(int, f.readline().strip().split())) if n>0 else []
        m_line = f.readline().strip()
        m = int(m_line) if m_line else 0
        B = list(map(int, f.readline().strip().split())) if m>0 else []
        l_line = f.readline().strip()
        l = int(l_line) if l_line else 0
        C = list(map(int, f.readline().strip().split())) if l>0 else []
    return A, B, C
def lcs_three_length(A, B, C):
    n = len(A)
    m = len(B)
    l = len(C)
    dp = [[[0]*(l+1) for _ in range(m+1)] for __ in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, m+1):
            for k in range(1, l+1):
                if A[i-1] == B[j-1] == C[k-1]:
                    dp[i][j][k] = dp[i-1][j-1][k-1] + 1
                else:
                    dp[i][j][k] = max(
                        dp[i-1][j][k],
                        dp[i][j-1][k],
                        dp[i][j][k-1]
                    )
    return dp[n][m][l]
def task5():
    A, B, C = read_three_sequences(PATH)
    result = lcs_three_length(A, B, C)
    write_result(result, OUTPUT_PATH)
```

Объяснение: Код реализует задачу поиска длины наибольшей общей подпоследовательности (LCS) для трёх последовательностей. Функция `read_three_sequences` считывает из файла три последовательности чисел: первую с длиной n, вторую с длиной m, третью с длиной l. Каждая последовательность сохраняется как список. Основная функция `lcs_three_length` использует трёхмерный массив `dp`, где dp[i][j][k] хранит длину LCS для первых i, j, k элементов из последовательностей A, B, C. Алгоритм проходит по всем возможным комбинациям длин последовательностей. Если текущие элементы A[i−1], B[j−1], C[k−1] совпадают, то dp[i][j][k] увеличивается на 1. Иначе вычисляется максимум из возможных предыдущих состояний. Итоговая длина LCS возвращается из dp[n][m][l].

Функция `task5` объединяет всё: считывает данные, вызывает алгоритм и записывает результат в файл.

input	output
3 1 2 3 3 2 1 3 3 1 3 5	2

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница	0.005	0.010
Пример	0.020	0.050
Верхняя граница	50.000	01.05.00

Вывод : Алгоритм решает задачу наибольшей общей подпоследовательности для трёх последовательностей с использованием динамического программирования. Временная сложность составляет O(n·m·l)O(n \c*m \c*l), где n, m, l — длины входных последовательностей. Пространственная сложность также O(n·m·l)O(n \c*m \c*l) из-за трёхмерного массива. Код эффективно обрабатывает задачи средней сложности, но его производительность может снижаться для больших последовательностей.

Задача №6 Наибольшая возрастающая подпоследовательность

Условие:

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.

Листинг кода:

```
def lis(seq):
    if not seq:
        return (0, [])
    tail = []
    parent = [-1]*len(seq)
    pos = [0]*len(seq)
    for i in range(len(seq)):
        index = bisect.bisect_left(tail, seq[i])
        if index == len(tail):
            tail.append(seq[i])
        else:
            tail[index] = seq[i]
        pos[i] = index
        if index > 0:
            pass
        if i == 0:
            ends = [0]
        if index == len(ends):
            ends.append(i)
        else:
            ends[index] = i
        if index > 0:
            parent[i] = ends[index-1]
    length = len(tail)
    lis_index = ends[length-1]
    lis_sequence = []
    while lis_index != -1:
        lis_sequence.append(seq[lis_index])
        lis_index = parent[lis_index]
```

```
    lis_sequence.reverse()
    return (length, lis_sequence)
def task6():
    seq = read_sequence(PATH)
    length, subsequence = lis(seq)
    write_result(length, subsequence, OUTPUT_PATH)
```

Объяснение: Код реализует алгоритм нахождения наибольшей возрастающей подпоследовательности (LIS) в массиве с временной сложностью O(nlogn)O(n \log n). Функция `lis` принимает последовательность `seq` и возвращает её длину и саму подпоследовательность. Алгоритм использует массив `tail` для хранения последних элементов возрастающих подпоследовательностей каждой длины. Для восстановления LIS создаются вспомогательные массивы: `parent`, чтобы отслеживать предшественников элементов, и `ends`, указывающий на индексы последних элементов подпоследовательностей. В процессе обработки каждого элемента `seq[i]` определяется его место в `tail` через бинарный поиск. Если элемент расширяет текущую подпоследовательность, он добавляется в `tail`; иначе, заменяет ближайший больший элемент. После обработки восстанавливается подпоследовательность, начиная с последнего элемента LIS. Функция `task6` считывает входные данные, вызывает `lis`, и записывает длину и подпоследовательность в файл.

input	output
6 3 29 5 5 28 6	3 3 5 28

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница	0.002	0.004
Пример	0.008	0.020
Верхняя граница	25.000	31.01.00

Вывод : Код эффективно находит наибольшую возрастающую подпоследовательность с использованием бинарного поиска для вставки элементов, обеспечивая сложность O(nlogn)O(n \log n). Алгоритм поддерживает баланс между вычислительной сложностью и памятью, сохраняя вспомогательные данные для восстановления LIS. Результат включает длину и саму подпоследовательность, что делает алгоритм полезным для анализа последовательностей любых размеров.

Задача №7 Шаблоны

Условие:

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.

Листинг кода:

```
def match_pattern(pattern, text):
    p = len(pattern)
    t = len(text)
    dp = [[False]*(t+1) for _ in range(p+1)]
    dp[0][0] = True
    for i in range(1, p+1):
        if pattern[i-1] == '*':
            dp[i][0] = dp[i-1][0]
        else:
            break
    for i in range(1, p+1):
        for j in range(1, t+1):
            if pattern[i-1] == text[j-1] or (pattern[i-1] == '?' and text[j-1] !
= ''):
                dp[i][j] = dp[i-1][j-1]
            elif pattern[i-1] == '*':
```

```
                # '*' может соответствовать нулю или многим символам
                dp[i][j] = dp[i-1][j] or dp[i][j-1]
            else:
                dp[i][j] = False
    return dp[p][t]
def task7():
    with open(PATH, 'r', encoding='utf-8') as f:
        pattern = f.readline().rstrip('\n')
        text = f.readline().rstrip('\n')
    result = match_pattern(pattern, text)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out:
        out.write("YES\n" if result else "NO\n")
```

Объяснение:Код реализует алгоритм проверки строки на соответствие шаблону с использованием динамического программирования. В шаблоне символ ? заменяет ровно один символ, * — любую последовательность символов (включая пустую строку), а остальные символы требуют точного совпадения. Функция match_pattern создаёт таблицу dp размером (len(pattern)+1) x (len(text)+1), где dp[i][j] показывает, совпадают ли первые i символов шаблона и j символов текста. Пустой шаблон соответствует пустому тексту, а начальные * могут быть равны пустой строке. Далее перебираются все символы шаблона и текста. Если текущие символы совпадают или шаблон содержит ?, то значение берётся из предыдущего состояния. При * допускаются два случая: символ пропускается (dp[i-1][j]) или включается (dp[i][j-1]). Функция возвращает результат совпадения всей строки и шаблона. В task7 из файла считываются шаблон и текст, проверяется их совпадение, и результат записывается в выходной файл YES или NO.

input	output
k?t*n kitten	YES

	Затраты памяти (Мб)	Время выполнения (с)
Нижняя граница	0.001	0.001
Пример	0.005	0.010
Верхняя граница	05.00	0.500

Вывод : Алгоритм решает задачу проверки соответствия строки шаблону с использованием динамического программирования, обеспечивая временную сложность O(p x t), где p — длина шаблона, t — длина текста. Пространственная сложность также составляет O(p x t) из-за использования таблицы. Код стабилен и эффективен для обработки шаблонов с произвольной длиной и количеством * или ?, предоставляя точный результат.

Вывод по лабораторной: Отчет по лабораторной работе №7 по теме «Динамическое программирование». Реализованы задачи на нахождение наибольшей общей подпоследовательности, проверки соответствия строки шаблону, а также задачи на поиск наибольшей возрастающей подпоследовательности. Продемонстрирована эффективность алгоритмов с использованием динамического программирования. Задачи выполнены успешно, результаты подтверждены примерами.