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 12

Выполнила:
Мкртчян.К.Г.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Оглавление

Задачи по варианту 3

Задача №1. Обмен монет. [N баллов] 3

Задача №5. НОП трех последовательностей. [N баллов] 5

Дополнительные задачи 7

Задача №4. НОП двух последовательностей. [N баллов] 7

Задача №6. Наибольшая возрастающая подпоследовательность. [N баллов] 8

Вывод по лабораторной работе 10

Задачи по варианту

Задача №1. Обмен монет. [N баллов]

Текст задачи:

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 = {coin1, ..., coink}. $1 \leq k \leq 100$, $1 \leq coin_i \leq 10^3$. Проверку можно сделать на наборе {1, 3, 4}. Формат ввода: первая строка содержит через пробел money и k; вторая - coin1coin2...coink.

– Вариация 2: Количество монет в кассе ограничено. Для каждой монеты из набора coins = {coin1, ..., coink} есть соответствующее целое число - количество монет в кассе данного номинала c = {c1, ..., ck}. 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.

Решение:

```
def exchange(money: int, coins: list[int]) -> int:
    dp = [0] + [float('inf')] * money
    for coin in coins:
        for j in range(coin, money+1):
            dp[j] = min(dp[j], dp[j - coin] + 1)
    return dp[money]


def exchange_with_amount(money: int, coins: dict[int: list[int]]) -> int:
    dp = [0] + [float('inf')] * money
    for c in coins.keys():
        denomination = c
        amount = coins[c]
        if amount > 0:
            for j in range(denomination, money+1):
                dp[j] = min(dp[j], dp[j - denomination] + 1)
    return dp[money]
```

Объяснение решения:

Функция exchange предназначена для решения задачи о минимальном количестве монет, необходимых для того, чтобы составить заданную сумму денег. Она использует метод динамического программирования для достижения этой цели. Вот основные шаги ее работы:

Инициализация массива: Функция начинает с создания массива dp, где каждый элемент представляет наименьшее количество монет, необходимое для составления определенной суммы. Первым элементом массива устанавливается 0 (поскольку для суммы 0 не требуется ни одной монеты), а все остальные элементы инициализируются значением infinity, чтобы обозначить невозможные комбинации на начальном этапе.

- Внешний цикл по монетам: Функция перебирает каждую монету из входящего списка coins. Для каждой монеты выполняется внутренний цикл.
- Внутренний цикл по суммам: Внутренний цикл перебирает все суммы от значения текущей монеты до целевой суммы money. Этот цикл проверяет, можно ли улучшить текущее количество монет для суммы j, добавив одну текущую монету coin.
- Обновление массива dp: Внутри внутреннего цикла функция обновляет значение dp[j] как минимум между текущим значением dp[j] и значением dp[j - coin] + 1. Это означает, что если можно составить сумму j - coin, то при добавлении одной монеты coin количество монет для составления суммы j может быть уменьшено.
- Возврат результата: после завершения всех итераций возвращается значение dp[money], которое содержит минимальное количество монет для составления суммы money. Если это значение по-прежнему равно infinity, это означает, что невозможно составить указанную сумму с использованием данных монет.
- Таким образом, функция эффективно находит оптимальное решение для задачи обмена монет путем последовательного анализа возможностей составления каждой промежуточной суммы.

Время выполнения программы, затраты памяти и результат:

Lab #7 |Task #1 - Test Table

	данные	время, сек.	память, МБ	результат
Значения из примера	2 3 1 3 4	0.0	31.66	2
Значения из примера	34 3 1 3 4	0.0	31.68	9
Значения из примера	50000 100 925 156 996 675	0.5	32.46	51
Значения из примера	50000 100 218 9 604 7 989 1 334 2	0.47	31.59	51

Вывод по задаче: Мы научились применять динамическое программирование для оптимизации решений, разбивая сложные задачи на более простые подзадачи.

Задача №5. НОП трех последовательностей. [N баллов]

Текст задачи:

Вычислить длину самой длинной общей подпоследовательности из трех последовательностей. Даны три последовательности A = (a1, a2, ..., an), B = (b1, b2, ..., bm) и C = (c1, c2, ..., cl), найти длину их самой длинной общей подпоследовательности, т.е. наибольшее неотрицательное целое число p такое, что существуют индексы $1 \le i1 < i2 < ... < ip \le n$, $1 \le j1 < j2 < ... < jp \le m$ и $1 \le k1 < k2 < ... < kp \le l$ такие, что ai1 = bj1 = ck1 , ..., aip = bjp = ckp .

• Формат ввода / входного файла (input.txt). – Первая строка: n - длина первой последовательности. – Вторая строка: a1, a2, ..., an через пробел. – Третья строка: m - длина второй последовательности. – Четвертая строка: b1, b2, ..., bm через пробел. – Пятая строка: l - длина второй последовательности. – Шестая строка: c1, c2, ..., cl через пробел.

• Ограничения: $1 \le n, m, l \le 100; -10^9 < ai , bi , ci < 10^9$.

• Формат вывода / выходного файла (output.txt). Выведите число p.

• Ограничение по времени. 1 сек.

Решение:

```
def longest_common_sub(n: int, arr_a: list[int], m: int, arr_b: list[int], l:
int, arr_c: list[int]) -> int:
    matrix = [[[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):
                matrix[i][j][k] = max(matrix[i - 1][j][k], matrix[i][j -
1][k], matrix[i][j][k - 1])
                if arr_a[i - 1] == arr_b[j - 1] == arr_c[k - 1]:
                    matrix[i][j][k] = max(matrix[i][j][k], matrix[i - 1][j -
1][k - 1] + 1)
    return matrix[n][m][l]
```

Объяснение решения:

Функция `longest_common_sub` предназначена для поиска длины наибольшей общей подпоследовательности (НОП) трех последовательностей чисел. Она применяет метод динамического программирования для эффективного решения данной задачи. Рассмотрим этапы работы функции без примеров:

- Инициализация трехмерной матрицы:
- В начале создается трехмерный массив `matrix`, размером `(n + 1) x (m + 1) x (l + 1)`, где `n`, `m`, и `l` - длины трех входных массивов. Этот массив будет хранить длины НОП для всех возможных комбинаций элементов этих массивов. Все элементы инициализируются нулем, что означает, что длина НГП для любых пустых последовательностей равна нулю.
- Тройной цикл по индексам:
- Функция использует три вложенных цикла для перебора всех возможных индексов `i`, `j`, и `k`, которые соответствуют элементам входных массивов `arr_a`, `arr_b`, и `arr_c` соответственно.
- Эти циклы начинаются с `1`, поскольку `matrix[0][][]`, `matrix[][0][]` и `matrix[][][0]` уже инициализированы нулями, что помогает избежать проверки пустых последовательностей.

- Обновление значений в матрице:
- На каждой итерации в циклах функция обновляет значение matrix[i][j][k]. Это значение определяется как максимум из трех возможных вариантов:
- Длина НОП, если элемент arr_a[i-1] не учитывается (то есть matrix[i - 1][j][k]).
- Длина НОП, если элемент arr_b[j-1] не учитывается (то есть matrix[i][j - 1][k]).
- Длина НОП, если элемент arr_c[k-1] не учитывается (то есть matrix[i][j][k - 1]).
- Проверка на совпадение элементов:
- Далее функция проверяет, совпадают ли текущие элементы всех трех массивов (arr_a[i-1], arr_b[j-1], и arr_c[k-1]). Если совпадение есть, то длина НОП также может увеличиться. В этом случае значение matrix[i][j][k] обновляется как максимум между текущим значением и matrix[i - 1][j - 1][k - 1] + 1, что учитывает найденное совпадение.
- Возврат результата:
- После завершения всех тройных циклов функция возвращает значение matrix[n][m][l], которое представляет длину наибольшей общей подпоследовательности для трех входных массивов.

Таким образом, функция использует динамическое программирование для нахождения длины НОП, эффективно обрабатывая все возможные комбинации трех последовательностей и обновляя результаты в матрице.

Время выполнения программы, затраты памяти и результат:

Lab #7 | Task #5 - Test Table

	данные	время, сек.	память, МБ	результат
Значения из примера	3 1 2 3 3 2 1 3 3 1 3 5	0.0	31.66	2
Значения из примера	5 8 3 2 1 7 7 8 2 1 3 8 10 7 6 6 8 3 1 4 7	0.0	31.68	3
Максимальные значения	100 664107257 497375511 -174973209 598948802 100 -25697344 -714941129 812141531 211767786 100 -25697344 872582899 382764821 906051543	0.19	32.33	7

Вывод по задаче: мы научились применять метод динамического программирования, который позволяет решать сложные задачи, разбивая их на более простые. Понимание принципа хранения промежуточных результатов для оптимизации времени выполнения алгоритма.

Дополнительные задачи

Задача №4. НОП двух последовательностей. [N баллов]

Текст задачи:

Вычислить длину самой длинной общей подпоследовательности из двух последовательностей. Даны две последовательности A = (a1, a2, ..., an) и B = (b1, b2, ..., bm), найти длину их самой длинной общей подпоследовательности, т.е. наибольшее неотрицательное целое число p такое, что существуют индексы $1 \le i1 < i2 < ... < ip \le n$ и $1 \le j1 < j2 < ... < jp \le m$ такие, что ai1 = bj1 , ..., aip = bjp .

• Формат ввода / входного файла (input.txt). – Первая строка: n - длина первой последовательности. – Вторая строка: a1, a2, ..., an через пробел. – Третья строка: m - длина второй последовательности. 4 – Четвертая строка: b1, b2, ..., bm через пробел.

• Ограничения: $1 \le n, m \le 100; -10^9 < ai , bi < 10^9$.

• Формат вывода / выходного файла (output.txt). Выведите число p.

• Ограничение по времени. 1 сек.

Решение:

```
def longest_common_sub(n, arr_a, m, arr_b):
    matrix = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            matrix[i][j] = max(matrix[i - 1][j], matrix[i][j - 1])
            if arr_a[i - 1] == arr_b[j - 1]:
                matrix[i][j] = max(matrix[i][j], matrix[i - 1][j - 1] + 1)
    return matrix[n][m]
```

Объяснение решения:

Функция `longest_common_sub` предназначена для нахождения длины наибольшей общей подпоследовательности (НГП) двух последовательностей. Она использует метод динамического программирования для эффективного решения этой задачи. Рассмотрим этапы работы функции без примеров:

- **Инициализация двумерной матрицы**:
- В начале создается двумерный массив `matrix` размером `(n + 1) x (m + 1)`, где `n` и `m` — длины двух входных массивов. Этот массив будет использоваться для хранения промежуточных результатов. Все элементы инициализируются нулями, что соответствует длине НГП для пустых последовательностей.
- **Двойной цикл по индексам**:
- Функция использует два вложенных цикла для перебора всех возможных пар индексов `i` и `j`, которые соответствуют элементам массивов `arr_a` и `arr_b`, начиная с индекса 1. Это делается для того, чтобы упростить логику обработки пустых подпоследовательностей, которые в данной матрице уже представлены нулями.
- **Обновление значений в матрице**:
- На каждой итерации циклов обновляется значение `matrix[i][j]`. Это значение определяется как максимум двух предыдущих значений:
- `matrix[i - 1][j]` — длина НГП без учета текущего элемента массива `arr_a`.

- `matrix[i][j - 1]` — длина НГП без учета текущего элемента массива `arr_b`.
- **Проверка на совпадение элементов**:
- Если элементы `arr_a[i - 1]` и `arr_b[j - 1]` совпадают, это значит, что текущие элементы соотносятся друг с другом. В этом случае значение `matrix[i][j]` может быть увеличено на 1 по сравнению с длиной НГП, найденной для предыдущих элементов обоих массивов, что соответствует `matrix[i - 1][j - 1] + 1`. Таким образом, происходит обновление значения `matrix[i][j]` с учетом найденного совпадения.
- **Возврат результата**:
- После завершения обеих итераций цикл возвращает значение `matrix[n][m]`, которое представляет длину наибольшей общей подпоследовательности для двух входных массивов.

Таким образом, функция использует динамическое программирование для нахождения длины НГП, эффективно обрабатывая все возможные комбинации двух последовательностей и обновляя результаты в матрице.

Время выполнения, затраты памяти и результат:

Lab #7 |Task #4 - Test Table

	данные	время, сек.	память, МБ	результат
Значения из примера	3 2 7 5 2 2 5	0.0	31.73	2
Значения из примера	1 7 4 1 2 3 4	0.0	31.75	0
Значения из примера	4 2 7 8 3 4 5 2 8 7	0.0	31.75	2
Максимальные значения	100 -963385600 -724606880 -679218640 752152392 100 -775054478 728770043 -449929443 -558938458	0.0	31.79	14

Вывод по задаче: мы научились применять метод динамического программирования, который позволяет решать сложные задачи, разбивая их на более простые. Понимание принципа хранения промежуточных результатов для оптимизации времени выполнения алгоритма.

Задача №6. Наибольшая возрастающая подпоследовательность. [N баллов]

Текст задачи:

Дана последовательность, требуется найти ее наибольшую возрастающую подпоследовательность.

- Формат ввода / входного файла (input.txt). В первой строке входных данных задано целое число n – длина последовательности ($1 \le n \le 300000$). 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 возрастающую подпоследовательность данной последовательности. Если ответов несколько - выведите
любой.

Решение:

```
def longest_sub(array_len: int, array: list[int]) -> tuple[int, list[int]]:
    inf = 10**10
    subsequence = [inf] * (array_len + 1)
    subsequence[0] = -inf
    for i in range(array_len):
        left = 0
        right = array_len
        while right - left > 1:
            middle = (left + right) // 2
            if subsequence[middle] >= array[i]:
                right = middle
            else:
                left = middle
        subsequence[right] = array[i]
    longest_subsequence = subsequence[1:subsequence.index(inf)]
    return len(longest_subsequence), longest_subsequence
```

Объяснение решения:

Функция `longest_sub` предназначена для нахождения длины и самой длины наибольшей возрастающей подпоследовательности в данном массиве. Она использует метод динамического программирования вместе с бинарным поиском для достижения более эффективного результата. Рассмотрим работу функции по этапам:

- **Инициализация**:
- Определяется некоторая большая константа `inf`, которая будет использоваться для инициализации массива `subsequence`.
- Создается массив `subsequence` размером `array_len + 1`, где все элементы инициализируются значением `inf`, кроме первого элемента, который инициализируется значением `-inf`. Это позволяет сохранить информацию о потенциальной минимальной подпоследовательности.
- **Основной цикл**:
- Функция проходит по каждому элементу входного массива `array`. На каждом шаге происходит следующее:
- Инициализируются переменные `left` и `right`, указывающие на границы текущего диапазона поиска в массиве `subsequence`.
- **Бинарный поиск**:
- Используется цикл для выполнения бинарного поиска по массиву `subsequence` для нахождения позиции, куда можно добавить текущий элемент массива `array[i]`.

- Цикл продолжается, пока `right - left > 1`. Это означает, что продолжается поиск, пока не будет найдено подходящее место (если такое есть) для вставки элемента в массив `subsequence`.
- На каждой итерации вычисляется положение `middle` и сравнивается значение в `subsequence` с текущим элементом. В зависимости от результата сравнения обновляются границы поиска (`left` и `right`).
- **Обновление массива**:
- После завершения поиска, в `subsequence[right]` помещается текущее значение `array[i]`, если оно больше предыдущих значений. Это формирует потенциальную возрастающую подпоследовательность.
- **Сбор результата**:
- После прохождения всех элементов массива, определяется наибольшая возрастающая подпоследовательность. Это делается путем извлечения элементов из массива `subsequence`, начиная с первого элемента до индекса, где находится `inf`. Это дает длину и саму возрастающую подпоследовательность.
- **Возврат результата**:
- Функция возвращает кортеж, содержащий длину наибольшей возрастающей подпоследовательности и саму последовательность.

Таким образом, функция использует комбинацию динамического программирования и бинарного поиска для нахождения возрастающей подпоследовательности более эффективно, чем простые алгоритмы, что делает её подходящей для работы с большими массивами.

Время выполнения, затраты памяти и результат:

Lab #7 | Task #6 - Test Table

	данные	время, сек.	память, МБ	результат
Значения из примера	6 3 29 5 5 28 6	0.0	31.71	3 3 5 6
Значения из примера	10 -5 3 2 3 -6 2 8 1 10 4	0.0	31.73	5 -6 1 3 4 10
Значения из примера	5000 -629782285 -933634303 116953379 -102443765	0.0	31.99	130 -999938035 -997550516 -996663839 -995747916
Максимальные значения	300000 -956162696 893895509 -488012814 -28289914	0.39	43.54	1067 -999998059 -999994165 -999993582 -999983885

Вывод по задаче: мы научились основам алгоритмов динамического программирования и бинарного поиска, а также тому, как их комбинировать для улучшения эффективности алгоритма.

Вывод по лабораторной работе

В ходе лабораторной работы мы познакомились с новым способом решения задач – динамическим программированием. Динамическое программирование разбивает задачу на подзадачи и решает каждую подзадачу только один раз, сохраняя результаты, чтобы избежать избыточных вычислений.