Лекция 6. Динамическое программирование.

1. Введение в динамическое программирование

Динамическое программирование (ДП) – это метод решения сложных задач путём разбиения их на более простые подзадачи. Основная идея заключается в том, что мы решаем каждую подзадачу только один раз, сохраняя результат, и используем его для решения более крупных задач.

Ключевые принципы ДП:

1. **Оптимальная подструктура** – оптимальное решение задачи может быть построено из оптимальных решений её подзадач.
2. **Перекрывающиеся подзадачи** – подзадачи повторяются многократно, и мы можем сохранять их решения в таблице (мемоизация или табуляция).

Типичные применения:

- Редакционное расстояние (расстояние Левенштейна).
- Оптимальная расстановка переносов в тексте.
- Наибольшая общая подпоследовательность (НОП).
- Наибольшая возрастающая подпоследовательность (НВП).

2. Редакционное расстояние (расстояние Левенштейна)

Задача. Найти минимальное количество операций (вставка, удаление, замена символа), необходимых для преобразования одной строки в другую.

Рекуррентная формула:

Пусть dp[i][j] – расстояние между префиксами s1[:i] и s2[:j].

- Если s1[i-1] == s2[j-1]:

dp[i][j] = dp[i-1][j-1]

- Иначе:

dp[i][j] = 1 + min(dp[i-1][j], # Удаление dp[i][j-1], # Вставка dp[i-1][j-1]) # Замена

Реализация:

```
def levenshtein_distance(s1: str, s2: str) -> int:
    # Получаем длины обеих строк
    m, n = len(s1), len(s2)

    # Создаем матрицу (таблицу) для динамического программирования размером
(m+1) x (n+1)
    # dp[i][j] будет хранить минимальное количество операций для
преобразования
    # первых i символов s1 в первые j символов s2
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Инициализация базовых случаев:
    # Если вторая строка пустая, нужно удалить все символы из первой строки
    for i in range(m + 1):
        dp[i][0] = i  # i операций удаления
```

```
    # Если первая строка пустая, нужно вставить все символы второй строки
    for j in range(n + 1):
        dp[0][j] = j  # j операций вставки

    # Заполняем таблицу построчно
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # Проверяем, совпадают ли текущие символы
            if s1[i - 1] == s2[j - 1]:
                # Если символы одинаковые, не нужно выполнять операцию
                # Просто берем значение из диагональной ячейки
                dp[i][j] = dp[i - 1][j - 1]
            else:
                # Если символы разные, выбираем минимальную стоимость из трех
операций:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],  # Удаление: удаляем символ из s1
                    dp[i][j - 1],  # Вставка: вставляем символ в s1
                    dp[i - 1][j - 1]  # Замена: заменяем символ в s1
                )

    # Возвращаем результат - расстояние между полными строками
    return dp[m][n]


# Пример
s1, s2 = "kitten", "sitting"
print(f"Редакционное расстояние: {levenshtein_distance(s1, s2)}")  # Вывод: 3
```

3. Оптимальное расположение текста (перенос слов)

Задача. Разбить текст на строки так, чтобы минимизировать суммарный штраф за «некрасивые» пробелы в конце строк.

Формулировка:

Пусть у нас есть список длин слов lengths и ширина строки W. Штраф для строки с словами от i до j вычисляется как:

penalty = (W - (sum(lengths[i:j]) + (j - i - 1))) ** 3,

если это значение неотрицательно.

Рекуррентное соотношение:

dp[i] – минимальный штраф для первых i слов.

dp[i] = min(dp[j] + penalty(j, i)) для всех j < i,

где penalty(j, i) – штраф для строки, содержащей слова с j по i-1.

Реализация:

```
def word_wrap(words: list[str], width: int) -> int:
    n = len(words)
    # Преобразуем слова в их длины
    lengths = [len(word) for word in words]

    # dp[i] будет хранить минимальный штраф для первых i слов
    # Инициализируем большими значениями
```

```
    dp = [float('inf')] * (n + 1)
    dp[0] = 0  # Штраф для 0 слов равен 0

    # Перебираем все возможные конечные позиции
    for i in range(1, n + 1):
        total_length = 0

        # Перебираем возможные начала последней строки
        # Идем в обратном порядке для эффективности
        for j in range(i, 0, -1):
            # Добавляем длину текущего слова к общей длине
            total_length += lengths[j - 1]

            # Количество пробелов между словами в текущей строке
            spaces = i - j

            # Проверяем, помещается ли строка в заданную ширину
            if total_length + spaces > width:
                break  # Если не помещается, выходим из цикла

            # Вычисляем штраф для текущей строки
            # Штраф = куб оставшегося свободного места
            penalty = (width - total_length - spaces) ** 3

            # Обновляем минимальный штраф
            # dp[j-1] - штраф для слов до текущей строки
            dp[i] = min(dp[i], dp[j - 1] + penalty)

    return dp[n]  # Возвращаем штраф для всех слов


# Пример
words = ["cat", "dog", "elephant", "bird"]
width = 10
print(f"Минимальный штраф: {word_wrap(words, width)}")
```

4. Наибольшая общая подпоследовательность (НОП)

Задача. Найти длину самой длинной подпоследовательности, которая присутствует в обеих строках. Подпоследовательность – это последовательность символов, которые идут в исходном порядке, но не обязательно подряд.

Рекуррентная формула:

dp[i][j] – длина НОП для префиксов s1[:i] и s2[:j].

- Если s1[i-1] == s2[j-1]:

$$dp[i][j] = 1 + dp[i-1][j-1]$$

- Иначе:

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$$

Восстановление ответа:

Проходим из dp[m][n] назад, выбирая символы, которые совпадают.

Реализация:

```python
def longest_common_subsequence(s1: str, s2: str) -> tuple[int, str]:
    m, n = len(s1), len(s2)

    # Создаем таблицу ДП: dp[i][j] - длина НОП для s1[0:i] и s2[0:j]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Заполняем таблицу
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                # Если символы совпадают, увеличиваем длину НОП на 1
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                # Если символы разные, берем максимум из двух вариантов:
                # - без текущего символа s1
                # - без текущего символа s2
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # Восстановление самой подпоследовательности
    lcs = []  # Список для хранения символов НОП (в обратном порядке)
    i, j = m, n  # Начинаем с правого нижнего угла таблицы

    # Проходим таблицу в обратном направлении
    while i > 0 and j > 0:
        if s1[i - 1] == s2[j - 1]:
            # Если символы совпадают, добавляем в результат
            lcs.append(s1[i - 1])
            # Переходим к предыдущим символам в обеих строках
            i -= 1
            j -= 1
        elif dp[i - 1][j] > dp[i][j - 1]:
            # Переходим к ячейке сверху (игнорируем символ из s1)
            i -= 1
        else:
            # Переходим к ячейке слева (игнорируем символ из s2)
            j -= 1

    # Разворачиваем список, так как мы собирали символы с конца
    return dp[m][n], ''.join(reversed(lcs))


# Пример
s1, s2 = "ABCBDAB", "BDCAB"
length, subsequence = longest_common_subsequence(s1, s2)
print(f"Длина НОП: {length}, Подпоследовательность: {subsequence}")  # Вывод:
4, BCAB
```

5. Наибольшая возрастающая подпоследовательность (НВП)

Задача. Найти длину самой длинной строго возрастающей подпоследовательности в массиве.

Метод ДП за $O(n^2)$:

dp[i] – длина НВП, заканчивающейся на i-й элемент.

dp[i] = 1 + max(dp[j]) для всех j < i, таких что arr[j] < arr[i].

Метод за O(n log n) с бинарным поиском:

Используем список tails, где tails[k] – минимальный возможный последний элемент возрастающей подпоследовательности длины k+1.

Восстановление ответа:

Сохраняем предков для каждого элемента.

Реализация (O(n log n)):

```
def longest_increasing_subsequence(arr: list[int]) -> tuple[int, list[int]]:
    n = len(arr)

    # tails - список, где tails[k] хранит индекс наименьшего возможного
    # последнего элемента возрастающей подпоследовательности длины k+1
    tails = []

    # prev - массив для восстановления подпоследовательности
    # prev[i] хранит индекс предыдущего элемента в НВП для arr[i]
    prev = [-1] * n  # -1 означает отсутствие предыдущего элемента

    # Обрабатываем каждый элемент массива
    for i, x in enumerate(arr):
        # Бинарный поиск позиции для вставки текущего элемента
        left, right = 0, len(tails)

        while left < right:
            mid = (left + right) // 2
            # Сравниваем с элементом, хранящимся в tails[mid]
            if arr[tails[mid]] < x:
                left = mid + 1
            else:
                right = mid

        # Обновляем tails
        if left < len(tails):
            # Заменяем существующий элемент
            tails[left] = i
        else:
            # Добавляем новый элемент в конец (увеличиваем длину НВП)
            tails.append(i)

        # Запоминаем предыдущий элемент для восстановления
подпоследовательности
        if left > 0:
            prev[i] = tails[left - 1]

    # Восстановление самой подпоследовательности
    seq = []  # Список для хранения элементов НВП
    idx = tails[-1]  # Начинаем с последнего элемента самой длинной НВП

    # Проходим по цепочке предков в обратном порядке
    while idx != -1:
        seq.append(arr[idx])  # Добавляем элемент
```

```
            idx = prev[idx]  # Переходим к предыдущему элементу

        # Разворачиваем, так как мы собрали элементы с конца
        return len(tails), seq[::-1]


    # Пример
    arr = [10, 9, 2, 5, 3, 7, 101, 18]
    length, subsequence = longest_increasing_subsequence(arr)
    print(f"Длина НВП: {length}, Подпоследовательность: {subsequence}")  # Вывод:
    4, [2, 3, 7, 101]
```