Лекция 7. Динамическое программирование.

1. Введение в динамическое программирование

Динамическое программирование – это мощная техника проектирования алгоритмов, которая используется для решения задач оптимизации путём разбиения их на более мелкие перекрывающиеся подзадачи. Решения этих подзадач сохраняются (мемоизируются), чтобы избежать их повторного вычисления.

Ключевые признаки применимости ДП:

1. **Оптимальная подструктура.** Оптимальное решение всей задачи может быть построено из оптимальных решений её подзадач.
2. **Перекрывающиеся подзадачи.** При рекурсивном разбиении задачи мы сталкиваемся с одними и теми же подзадачами много раз.

Основные подходы:

1. **Нисходящее ДП (мемоизация, Top-Down).** Рекурсивное решение, в котором результаты подзадач сохраняются в кеше (словаре или массиве) при первом вычислении.
2. **Восходящее ДП (табуляция, Bottom-Up).** Итеративное решение, в котором мы начинаем с самых маленьких подзадач и последовательно заполняем таблицу (массив) до решения исходной задачи.

2. Классическая задача о рюкзаке (0/1 Knapsack)

Постановка задачи:

У нас есть рюкзак вместимостью W (целое число) и набор из n предметов. Каждый предмет i имеет:

- weight[i] – вес (целое число > 0).
- value[i] – стоимость (целое число > 0).

Цель. Выбрать подмножество предметов такое, чтобы их суммарный вес не превосходил W, а суммарная стоимость была **максимально возможной**.

Важное ограничение. Каждый предмет можно взять **не более одного раза** (0 – не берём, 1 – берём, отсюда и название 0/1).

3. Рекуррентное соотношение и наивная реализация

Идея. Будем перебирать предметы по одному. Для каждого предмета у нас есть два выбора: взять его или не взять.

Определение состояния ДП:

Пусть dp[i][w] – это **максимальная суммарная стоимость**, которую можно получить, используя первые i предметов и имея рюкзак вместимостью w.

Нам нужно найти: dp[n][W].

Базовые случаи:

Если количество предметов i = 0 или вместимость w = 0, то максимальная стоимость равна 0.

dp[0][w] = 0 для всех w

dp[i][0] = 0 для всех i

Рекуррентное соотношение:

1. Если вес текущего предмета weight[i-1] больше текущей вместимости w, то мы **не можем** его взять. Мы просто наследуем ответ от предыдущих предметов:

$$dp[i][w] = dp[i-1][w]$$

2. Если вес текущего предмета weight[i-1] меньше или равен w, то у нас **есть выбор**:

- **Не брать** предмет: dp[i][w] = dp[i-1][w]
- **Брать** предмет: dp[i][w] = dp[i-1][w - weight[i-1]] + value[i-1]

Мы выбираем вариант с максимальной стоимостью:

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1])$$

Пояснение. Когда мы берём предмет i, мы добавляем его стоимость value[i-1], но также мы должны учесть, что вместимость нашего рюкзака уменьшилась на weight[i-1]. Максимальная стоимость, которую мы можем набрать на этом уменьшенном рюкзаке для первых i-1 предметов, уже посчитана в dp[i-1][w - weight[i-1]].

Наивная рекурсивная реализация (без ДП):

```
def knapSack_recursive(W, weights, values, n):
    """Рекурсивное решение без мемоизации. Экспоненциальная сложность."""
    # Базовый случай: если не осталось предметов или места в рюкзаке
    if n == 0 or W == 0:
        return 0

    # Если вес последнего рассматриваемого предмета больше оставшейся вместимости
    # Мы не можем его взять, поэтому пропускаем его
    if weights[n - 1] > W:
        return knapSack_recursive(W, weights, values, n - 1)

    else:
        # Возвращаем максимум из двух случаев:
        # 1. Не берем текущий предмет - рекурсивно вызываем для n-1 предметов
        # 2. Берем текущий предмет - добавляем его стоимость и рекурсивно вызываем
        #    для n-1 предметов с уменьшенной вместимостью
        return max(
            knapSack_recursive(W, weights, values, n - 1),  # Вариант 1: не берем предмет
            values[n - 1] + knapSack_recursive(W - weights[n - 1], weights, values, n - 1)  # Вариант 2: берем предмет
        )


# Пример использования
values = [60, 100, 120]
```

```
weights = [10, 20, 30]
W = 50
n = len(values)
print("Максимальная стоимость (рекурсия):", knapSack_recursive(W, weights,
values, n))
# Вывод: Максимальная стоимость (рекурсия): 220
```

Проблема. Временная сложность наивной рекурсии – $O(2^n)$, так как для каждого предмета мы делаем два рекурсивных вызова. Это неприемлемо для больших n.

4. Восходящее ДП (табуляция) с использованием 2D-массива

Создадим таблицу dp размером (n+1) x (W+1) и будем заполнять её последовательно, используя рекуррентное соотношение.

```
def knapSack_2d(W, weights, values, n):
    """Реализация с использованием 2D таблицы. Сложность O(n*W)."""
    # Создаем таблицу dp размером (n+1) x (W+1)
    # dp[i][w] будет хранить максимальную стоимость для первых i предметов и
рюкзака вместимостью w
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Заполняем таблицу построчно
    for i in range(1, n + 1):  # i - количество рассмотренных предметов (от 1
до n)
        for w in range(1, W + 1):  # w - текущая вместимость рюкзака (от 1 до
W)

            # Проверяем, можем ли мы взять текущий предмет (i-1 по индексу в
массивах)
            if weights[i - 1] <= w:
                # Если можем взять, выбираем максимум из:
                dp[i][w] = max(
                    dp[i - 1][w],  # Вариант 1: не берем предмет
                    dp[i - 1][w - weights[i - 1]] + values[i - 1]  # Вариант
2: берем предмет
                    # dp[i-1][w - weights[i-1]] - максимальная стоимость для
оставшегося места
                    # values[i-1] - стоимость текущего предмета
                )
            else:
                # Если не можем взять предмет (слишком тяжелый), наследуем
значение сверху
                dp[i][w] = dp[i - 1][w]

    # Возвращаем ответ - максимальную стоимость для всех n предметов и полной
вместимости W
    return dp[n][W]


# Пример использования
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
```

```
n = len(values)
print("Максимальная стоимость (2D ДП):", knapSack_2d(W, weights, values, n))
# Вывод: Максимальная стоимость (2D ДП): 220
```

Анализ сложности:

- **Время:** O(n*W). Мы имеем два вложенных цикла.
- **Память:** O(n*W). Мы используем двумерный массив.

5. Оптимизация памяти (1D ДП)

Заметим, что для вычисления dp[i][w] нам нужна только строка dp[i-1]. Поэтому всю логику можно уместить в **одномерном массиве** dp размером W+1.

Важный нюанс. Чтобы не перезаписывать значения из предыдущей строки, которые могут понадобиться для вычислений справа, внутренний цикл должен идти **в обратном порядке** – от W до 0.

```
def knapSack_1d(W, weights, values, n):
    """Оптимизированная реализация с использованием 1D массива. Сложность
O(n*W), память O(W)."""
    # Создаем одномерный массив, где dp[w] хранит максимальную стоимость для
рюкзака вместимостью w
    dp = [0] * (W + 1)

    # Перебираем все предметы по очереди
    for i in range(n):          # i - индекс текущего предмета
        # ОБРАТНЫЙ порядок от W до веса текущего предмета
        # Это важно чтобы не использовать один предмет несколько раз
        for w in range(W, weights[i] - 1, -1):
            # Для каждой вместимости проверяем, выгодно ли добавить текущий
предмет
            # Сравниваем текущее значение с вариантом, когда берем текущий
предмет
            if dp[w] < dp[w - weights[i]] + values[i]:
                dp[w] = dp[w - weights[i]] + values[i]

    return dp[W]  # Возвращаем максимальную стоимость для полной вместимости


# Пример использования
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
n = len(values)
print("Максимальная стоимость (1D ДП):", knapSack_1d(W, weights, values, n))
# Вывод: Максимальная стоимость (1D ДП): 220
```

Почему обратный порядок?

Если бы мы шли от 0 до W, то при обновлении dp[w] мы могли бы использовать уже обновлённое значение dp[w - weights[i]], которое уже содержит текущий предмет i. Это привело бы к тому, что предмет был бы учтён несколько раз. Обратный порядок гарантирует, что при расчёте dp[w] мы используем состояние, соответствующее предыдущим предметам (до i).

6. Восстановление ответа

Часто недостаточно знать только максимальную стоимость. Нужно также вывести, **какие именно предметы были выбраны**.

Алгоритм восстановления:

1. Начинаем с ячейки dp[n][W] (или анализируем одномерный массив, но проще использовать 2D версию).
2. Двигаемся в обратном направлении от i = n, w = W до i = 0.
3. Если dp[i][w] != dp[i-1][w], это означает, что i-й предмет был включён в оптимальное решение.
   - Добавляем предмет i-1 в ответ.
   - Уменьшаем текущую вместимость w на weight[i-1].
4. Если dp[i][w] == dp[i-1][w], предмет не был взят, просто уменьшаем i.

```
def knapSack_with_items(W, weights, values, n):
    """Решает задачу о рюкзаке и возвращает стоимость и список выбранных
предметов."""
    # Создаем стандартную 2D таблицу как в предыдущем примере
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Заполняем таблицу стандартным способом
    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    # ВОССТАНОВЛЕНИЕ ОТВЕТА:
    res = dp[n][W]  # Максимальная стоимость
    selected_items = []  # Список для хранения выбранных предметов
    w = W  # Начинаем с полной вместимости

    # Двигаемся в обратном порядке: от последнего предмета к первому
    for i in range(n, 0, -1):
        if res <= 0:  # Если стоимость стала нулевой, выходим
            break

        # Если текущая стоимость НЕ РАВНА стоимости из строки выше,
        # значит текущий предмет был взят в оптимальное решение
        if res != dp[i - 1][w]:
            # Добавляем индекс предмета в список выбранных
            selected_items.append(i - 1)
            # Вычитаем стоимость и вес этого предмета
            res -= values[i - 1]
            w -= weights[i - 1]

    return dp[n][W], selected_items


# Пример использования
values = [60, 100, 120]
```

```
weights = [10, 20, 30]
W = 50
n = len(values)
max_value, items = knapSack_with_items(W, weights, values, n)
print("Максимальная стоимость:", max_value)
print("Индексы выбранных предметов:", items)
print("Выбранные предметы (индекс, вес, стоимость):")
for i in items:
    print(f"  [{i}]: вес={weights[i]}, стоимость={values[i]}")
# Вывод:
# Максимальная стоимость: 220
# Индексы выбранных предметов: [2, 1]
# Выбранные предметы (индекс, вес, стоимость):
#   [2]: вес=30, стоимость=120
#   [1]: вес=20, стоимость=100
```

7. Рюкзак с неограниченным количеством предметов (Unbounded Knapsack)

Постановка задачи. Теперь каждый предмет можно брать **неограниченное количество раз**.

Рекуррентное соотношение (для 1D массива) меняется:

dp[w] = max(dp[w], dp[w - weight[i]] + value[i])

Ключевое отличие. При расчёте dp[w] мы можем использовать уже обновлённое значение dp[w - weight[i]], которое могло уже включить текущий предмет i. Поэтому внутренний цикл должен идти **в прямом порядке** – от weight[i] до W.

```
def unboundedKnapSack(W, weights, values, n):
    """Решает задачу о рюкзаке с неограниченным количеством предметов."""
    dp = [0] * (W + 1)  # Одномерный массив для хранения максимальных
стоимостей

    # ПРЯМОЙ порядок обхода - это ключевое отличие!
    for w in range(1, W + 1):  # Для каждой возможной вместимости
        for i in range(n):  # Перебираем все предметы
            if weights[i] <= w:
                # Используем ПРЯМОЙ доступ - можем многократно использовать
предметы
                # dp[w - weights[i]] мог уже содержать текущий предмет i
                dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

    return dp[W]


# Пример использования
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
n = len(values)
print("Максимальная стоимость (Unbounded):", unboundedKnapSack(W, weights,
values, n))
```

```
# Вывод: Максимальная стоимость (Unbounded): 300 (предмет с весом 10 взят 5 раз)
```

8. Задача о сумме подмножеств (Subset Sum)

Постановка задачи. Дан массив натуральных чисел arr и целевое число target. Нужно определить, существует ли подмножество массива, сумма элементов которого равна target.

Модификация рюкзака:

- «Стоимость» предмета равна его «весу» (values[i] = arr[i]).
- Нас интересует не максимальная стоимость, а **достижимость** суммы.

Определение состояния ДП:

Пусть dp[i][s] – булево значение, которое равно True, если среди первых i элементов существует подмножество с суммой s.

Рекуррентное соотношение:

dp[i][s] = dp[i-1][s] or dp[i-1][s - arr[i-1]] (если s >= arr[i-1])

Базовый случай: dp[0][0] = True, dp[i][0] = True (пустое подмножество).

1D Реализация (оптимизированная по памяти):

```
def canPartitionSubsetSum(arr, target):
    """Проверяет, существует ли подмножество с заданной суммой."""
    n = len(arr)
    # dp[s] = True, если сумму s можно набрать из элементов массива
    dp = [False] * (target + 1)
    dp[0] = True  # Сумму 0 всегда можно набрать - пустое подмножество

    # Перебираем все числа из массива
    for num in arr:
        # ОБРАТНЫЙ порядок как в 0/1 рюкзаке (каждый элемент используется один раз)
        for s in range(target, num - 1, -1):
            # Если сумму (s - num) можно набрать, то и сумму s можно набрать, добавив num
            if dp[s - num]:
                dp[s] = True

    return dp[target]  # Возвращаем, можно ли набрать целевую сумму


# Пример использования
arr = [3, 34, 4, 12, 5, 2]
target = 9
print(f"Существует ли подмножество с суммой {target}? {canPartitionSubsetSum(arr, target)}")
# Вывод: Существует ли подмножество с суммой 9? True (4+5)
```

9. Сравнение сложности и применимости

Вариант задачи	**Временная сложность**	**Пространственная сложность**	**Ключевая особенность**
0/1 Рюкзак (2D)	O(n*W)	O(n*W)	Основа, легко восстановить ответ
0/1 Рюкзак (1D)	O(n*W)	O(W)	Оптимизация памяти, обратный порядок цикла
Unbounded Рюкзак (1D)	O(n*W)	O(W)	Прямой порядок цикла
Сумма подмножеств (1D)	O(n*target)	O(target)	Булев массив, обратный порядок цикла

Важное замечание. Сложность O(n*W) является **псевдополиномиальной**, а не полиномиальной, так как W – это число, а не размер входа. Если W очень велико (например, 2^n), алгоритм становится непрактичным. Задача о рюкзаке является **NP-трудной** в общем случае.

10. Типичные применения задачи о рюкзаке

- Оптимизация загрузки транспорта.
- Формирование инвестиционного портфеля.
- Распределение рекламного бюджета.
- Выбор проектов для реализации с ограниченными ресурсами.
- В криптографии и теории шифрования.
- Задача разбиения множества (Partition Problem).