Шпаргалка по частым задачам на leetcode / Паттерны и примеры Cheatsheet (XeLaTeX)

Краткий справочник

Содержание

Зачем нужны паттерны?

Собеседования по алгоритмам часто проверяют не знание сотен задач, а умение распознавать **основные подходы (паттерны)** к их решению. Знание паттернов позволяет быстро выбрать правильную структуру данных и алгоритм, даже если ты видишь задачу впервые. Это как иметь набор универсальных инструментов для разных типов винтов. Фокусируемся на паттернах, часто встречающихся в задачах уровня Medium на LeetCode.

1 Паттерны для Массивов/Строк

Паттерн: Два Указателя (Two Pointers)

Идея: Использование двух указателей (обычно индексов), которые движутся по массиву (или строке, или связанному списку) для выполнения какой-либо операции. Они могут двигаться навстречу друг другу, в одном направлении с разной скоростью или один может "ждать" другого. ● **Аналогия:** Два человека идут по тропинке. Либо навстречу (например, найти место встречи посередине), либо один догоняет другого (проверить, есть ли цикл), либо идут вместе с фиксированным расстоянием (скользящее окно, но без изменения размера). ● **Типовые задачи:**

- Найти два числа в ***отсортированном*** массиве, сумма которых равна X. (Указатели с разных концов).
- Проверить, является ли строка палиндромом. (Указатели с разных концов).
- Удалить дубликаты из *отсортированного* массива "на месте". (Один указатель пишет, другой читает).
- Найти цикл в связанном списке (см. раздел "Связанные списки").

● **Сложность:** Обычно $O(N)$ по времени, $O(1)$ по памяти (если модификация "на месте").

Паттерн: Скользящее Окно (Sliding Window)

Идея: Поддержание "окна" (подмассива или подстроки) определенного размера (или с определенными свойствами), которое "скользит" по основной структуре данных. Часто используется для задач, связанных с поиском оптимального непрерывного подмассива/подстроки. Размер окна может быть фиксированным или динамическим. ● **Аналогия:** Лупа, которую двигают вдоль длинного текста, чтобы рассмотреть фрагмент (окно). Размер лупы (окна) может быть постоянным или меняться. ● **Типовые задачи:**

- Найти максимальную/минимальную сумму подмассива *фиксированного* размера K.
- Найти длину самой длинной подстроки *без повторяющихся символов*. (Размер окна динамический).
- Найти все анаграммы подстроки S1 в строке S2. (Окно размера S1, проверяем совпадение частот символов).

● **Сложность:** Обычно $O(N)$ по времени (каждый элемент посещается 1-2 раза), $O(k)$ или $O(1)$ по памяти (k - размер алфавита или окна, если храним его содержимое). ● *(Часто внутри окна используется `dict` или `set` для отслеживания состояния, например, частот символов.)*

Общая структура (динамическое окно)

```
left = 0
result = ...
current_window_state = ... # e.g., dict, set

for right in range(len(data)):
    # Расширяем окно вправо
    add_element(data[right], current_window_state)

    # Сжимаем окно слева, пока условие не
    выполнится
    while not
    is_window_valid(current_window_state):
        remove_element(data[left],
        current_window_state)
        left += 1

    # Обновляем результат, если нужно
    update_result(result, current_window_state)

return result
```

Паттерн: Хеш-таблица / Словарь (Hash Table / Dictionary)

Идея: Использование структуры данных "ключ-значение" для **сверхбыстрого** ($\approx O(1)$ в среднем) поиска, добавления или проверки наличия элемента. Незаменимо, когда нужно часто проверять наличие элемента или считать частоты. В Python это 'dict' и 'set'. ● **Аналогия:** Гардероб с номерками. По номерку (ключу) ты мгновенно находишь свою куртку (значение). Или телефонная книга: по имени (ключу) находишь номер (значение). ● **Типовые задачи:**

- **Two Sum:** Найти два числа в массиве (не обязательно отсортированном), сумма которых равна X. (Храним 'value -> index' в 'dict').
- Группировка анаграмм. (Ключ - отсортированная строка или кортеж частот в 'dict', значение - список строк).
- Найти первый неповторяющийся символ в строке. (Храним 'char -> count' в 'dict').
- Проверить, содержит ли массив дубликаты. (Используем 'set' для проверки наличия; 'dict' – для хранения счетчиков или индексов).

● **Сложность:** В среднем $O(1)$ для 'get', 'insert', 'delete'. $O(N)$ по времени для обхода всей таблицы. $O(N)$ по памяти в худшем случае для хранения N элементов. Помните про возможные коллизии и $O(N)$ в худшем случае для операций, но на практике редко.

2 Базовые Алгоритмы и Структуры

Сортировки (Важна Идея и Сложность)

Идея: Упорядочивание элементов коллекции. Редко нужно реализовывать сортировку с нуля на собеседовании (если только не попросят), но важно знать *принцип работы* и *сложность* эффективных алгоритмов ('Merge Sort', 'Quick Sort'). ● **Аналогия:**

- **Merge Sort (Слиянием):** Разделяй и властвуй. Как сортировать большую колоду карт: раздели пополам, отсортируй каждую половину (рекурсивно), затем аккуратно слей две отсортированные стопки в одну.
- **Quick Sort (Быстрая):** Выбери "опорную" карту. Все карты меньше нее положи налево, все больше - направо. Повтори для левой и правой кучек.

● **Когда использовать?** Часто как **предварительный шаг** для других алгоритмов (например, для "Двух указателей"). ● **Слож-**

ность:

- Merge Sort: $O(N \log N)$ время (всегда), $O(N)$ память (для слияния).
- Quick Sort: $O(N \log N)$ время (в среднем), $O(N^2)$ (в худшем случае, редкий для хороших реализаций), $O(\log N)$ память (в среднем, для стека рекурсии).

Итог по сложности

```
Алгоритм       | Время (среднее) | Время (худшее)
  | Память (доп.)
---------------|-----------------|----------------|-------------
Merge Sort     | $O(N \log N)$   | $O(N \log N)$
  | $O(N)$
Quick Sort     | $O(N \log N)$   | $O(N^2)$
  | $O(\log N)$
Встроенная sort| $O(N \log N)$*  | $O(N \log N)$*
  | $O(N)$ или $O(\log N)$**
* Обычно гибридные (TimSort в Python, IntroSort в
  C++).
** Зависит от реализации (TimSort - O(N),
  IntroSort - O(log N)).
```

(Встроенные сортировки (`sort()`, `sorted()`) в Python (TimSort) и других языках часто являются гибридными и высокооптимизированными, поэтому их почти всегда предпочтительнее использовать на практике.)

Связанные Списки (Linked Lists)

Идея: Линейная структура, где каждый элемент (узел) содержит данные и *указатель* на следующий элемент. Нет индексов, доступ только последовательный. ● **Аналогия:** Квест "найди сокровище". Каждая записка (узел) говорит, где искать следующую. ● **Типовые задачи:**

- **Реверс списка:** Изменить указатели так, чтобы список шел в обратном порядке. (Классика: три указателя 'prev', 'current', 'next').
- **Обнаружение цикла:** Определить, есть ли в списке узел, указывающий на один из предыдущих узлов. (Паттерн "Два указателя": быстрый и медленный, "Черепаха и Заяц").
- Найти средний элемент. (Два указателя: один идет на 1 шаг, другой на 2).
- Слияние двух отсортированных списков.

● **Сложность:** Зависит от задачи. Реверс - $O(N)$ время, $O(1)$ память. Обнаружение цикла - $O(N)$ время, $O(1)$ память. Доступ к k-му элементу - $O(k)$ время. Вставка/удаление в начале - $O(1)$. ● *(Ключевой трейд-офф с массивами: быстрая $O(1)$ вставка/удаление **по известному узлу** (в начале/конце или если есть указатель), но медленный $O(N)$ доступ к элементу по индексу.)*

Обнаружение цикла (Черепаха и Заяц)

```python
slow = head
fast = head
while fast is not None and fast.next is not None:
    slow = slow.next
    fast = fast.next.next
    if slow == fast:
        return True # Цикл найден
return False # Цикла нет
```

3 Деревья и Рекурсия/ДП

Деревья: Обходы BFS и DFS

Идея: Иерархическая структура данных. Обходы - способ посетить все узлы.

- **BFS (Поиск в ширину / Breadth-First Search):** Исследование "уровень за уровнем". Использует **очередь (Queue)**.
- **DFS (Поиск в глубину / Depth-First Search):** Исследование "вглубь до упора, затем возврат". Использует **рекурсию** (неявный стек) или **явный стек (Stack)** для итеративной реализации. Варианты DFS: pre-order, in-order, post-order.

● **Аналогия:**

- BFS: Бросить камень в воду - волны расходятся по уровням. Или: поиск кратчайшего пути в лабиринте без весов - исследуем все соседние клетки, потом соседей соседей и т.д.
- DFS: Исследование лабиринта - идем по одному пути до тупика, возвращаемся, пробуем другой.

● **Типовые задачи:**

- Поуровневый обход дерева (Level Order Traversal) - Классический BFS.
- Найти максимальную глубину дерева - DFS (рекурсивно или итеративно).
- Проверить, является ли дерево бинарным деревом поиска (BST) - DFS (in-order обход).
- Найти путь от корня к узлу с заданной суммой - DFS.

● **Сложность:** $O(N)$ по времени (N - число узлов, посещаем каждый). $O(W)$ по памяти для BFS (W - макс. ширина дерева, в худшем случае $N/2$). $O(H)$ по памяти для DFS (H - высота дерева, для стека рекурсии/итерации, в худшем случае N). ● **Когда использовать?**

- **BFS:** Поиск кратчайшего пути в *невзвешенном* графе/дереве, поуровневый обход, задачи "ближайший сосед".
- **DFS:** Проверка связности, поиск цикла, поиск пути (не обязательно кратчайшего), задачи, требующие исследования ветки до конца (например, проверка валидности BST), многие задачи на бэктрекинг. Часто проще реализовать рекурсивно.

Структура BFS (итеративно)

```python
from collections import deque
if not root: return []
queue = deque([root])
result = []
while queue:
    level_size = len(queue)
    current_level = []
    for _ in range(level_size):
        node = queue.popleft()
        current_level.append(node.val)
        if node.left: queue.append(node.left)
        if node.right: queue.append(node.right)
    result.append(current_level) # или другая
    обработка
return result
```

Структура DFS (рекурсивно, pre-order)

```python
result = []
def dfs(node):
    if not node: return
    result.append(node.val) # Pre-order: Visit
    node first
    dfs(node.left)
    dfs(node.right)
    # In-order: dfs(left), visit, dfs(right)
    # Post-order: dfs(left), dfs(right), visit
dfs(root)
return result
```

Рекурсия / Бэктрекинг / ДП

Эти концепции тесно связаны и часто используются вместе, особенно в задачах на деревьях, графах и комбинаторике. ● **Рекурсия:**

- **Идея:** Функция вызывает саму себя для решения подзадачи меньшего размера. Обязательны *базовый случай* (условие остановки) и *рекурсивный шаг*.
- **Аналогия:** Матрешка. Чтобы открыть самую маленькую, нужно открыть все большие над ней.
- **Примеры:** Обходы деревьев (DFS), вычисление факториала.

● **Бэктрекинг (Backtracking):**

- **Идея:** Систематический перебор всех возможных кандидатов в решении. Строим решение по шагам. Если текущий шаг ведет в тупик (не может привести к решению), "откатываемся" (backtrack) и пробуем другой вариант. Часто реализуется через **рекурсивный DFS**, где на каждом шаге пробуются варианты, и происходит 'откат' при неудаче.
- **Аналогия:** Поиск выхода из лабиринта: идешь по пути, если тупик - возвращаешься к развилке и пробуешь другой путь.
- **Примеры:** Генерация всех перестановок/комбинаций/подмножеств, решение Судоку, задача N ферзей.

● **Динамическое Программирование (ДП / DP):**

- **Идея:** Решение задачи путем разбиения ее на *перекрывающиеся подзадачи* и решения каждой подзадачи только один раз, сохраняя ее результат (мемоизация или табуляция) для будущего использования. Требует наличия *оптимальной подструктуры*. Ключевое отличие от 'наивной' рекурсии — эффективная обработка **перекрывающихся подзадач** за счет сохранения их решений.
- **Аналогия:** Строительство из Lego. Чтобы построить большой замок (решить задачу), ты сначала строишь башни и стены (решаешь подзадачи). Если тебе снова нужна точно такая же башня, ты берешь уже готовую (сохраненный результат), а не строишь заново.
- **Примеры:** Числа Фибоначчи, задача о рюкзаке, поиск самой длинной общей подпоследовательности, "Лесенка" (Climbing Stairs).
- **Подходы:**
  - *Мемоизация (Top-Down):* Рекурсивное решение + кеширование результатов (обычно через словарь/массив).
  - *Табуляция (Bottom-Up):* Итеративное заполнение таблицы (массива) результатами подзадач, начиная с самых маленьких.

● **Сложность:** Сильно зависит от задачи. Может быть от полиномиальной (ДП на Фибоначчи - $O(N)$) до экспоненциальной (бэктрекинг для перестановок - $O(N!)$).

Пример ДП (Фибоначчи с мемоизацией)

```python
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    result = fib(n-1) + fib(n-2)
    memo[n] = result
    return result
```