

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

Краткий справочник  
April 2, 2025

## Contents

1	Паттерны для Массивов/Строк	1
2	Базовые Алгоритмы и Структуры	1
3	Деревья и Рекурсия/ДП	2

### 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)

```
1 left = 0
2 result = ...
3 current_window_state = ... # e.g., dict, set
4
5 for right in range(len(data)):
6     # Расширяем окно вправо
7     add_element(data[right], current_window_state)
8
9     # Сжимаем окно слева, пока условие не
10    # выполнится
11    while not
12        is_window_valid(current_window_state):
13            remove_element(data[left],
14                            current_window_state)
15            left += 1
16
17    # Обновляем результат, если нужно
18    update_result(result, current_window_state)
19
20 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"). ● **Сложность:**

- 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.)
1			
2			
3	Merge Sort	$O(N \log N)$	$O(N \log N)$
4	Quick Sort	$O(N \log N)$	$O(N^2)$
5	Встроенная sort	$O(N \log N)$	$O(N \log N)$
6	* Обычно гибридные (TimSort в Python, IntroSort в C++)		
7	** 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, "Черепаша 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 (Черепаша и Заяц)

```
1 slow = head
2 fast = head
3 while fast is not None and fast.next is not None:
4     slow = slow.next
5     fast = fast.next.next
6     if slow == fast:
7         return True # Цикл найден
8 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)

```
1 from collections import deque
2 if not root: return []
3 queue = deque([root])
4 result = []
5 while queue:
6     level_size = len(queue)
7     current_level = []
8     for _ in range(level_size):
9         node = queue.popleft()
10        current_level.append(node.val)
11        if node.left: queue.append(node.left)
12        if node.right: queue.append(node.right)
13    result.append(current_level) # или другая
14    return result
```

### Структура DFS (рекурсивно, pre-order)

```
1 result = []
2 def dfs(node):
3     if not node: return
4     result.append(node.val) # Pre-order: Visit
5     dfs(node.left)         node first
6     dfs(node.right)
7     # In-order: dfs(left), visit, dfs(right)
8     # Post-order: dfs(left), dfs(right), visit
9 dfs(root)
10 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)

```
1 memo = {}
2 def fib(n):
3     if n in memo: return memo[n]
4     if n <= 1: return n
5     result = fib(n-1) + fib(n-2)
6     memo[n] = result
7     return result
```