Паттерн Топ К элементов: эффективный поиск лучших данных

Как не сортировать 10 млн строк, чтобы найти топ-10 самых частых запросов

Зачем нужен паттерн Топ K элементов?

В современной разработке часто возникает задача поиска K самых больших, маленьких или частотных элементов в массиве данных.

Вы приходите на собес в банк, дают задачу:

Найдите топ-5 самых частых транзакций за сутки (100 млн записей).

Вы: *sorted(transactions, key=count)[:5]*, **оценка** *O(n log(n))*

Интервьюер: А если у нас есть ограничения по памяти?

Вы: o_O

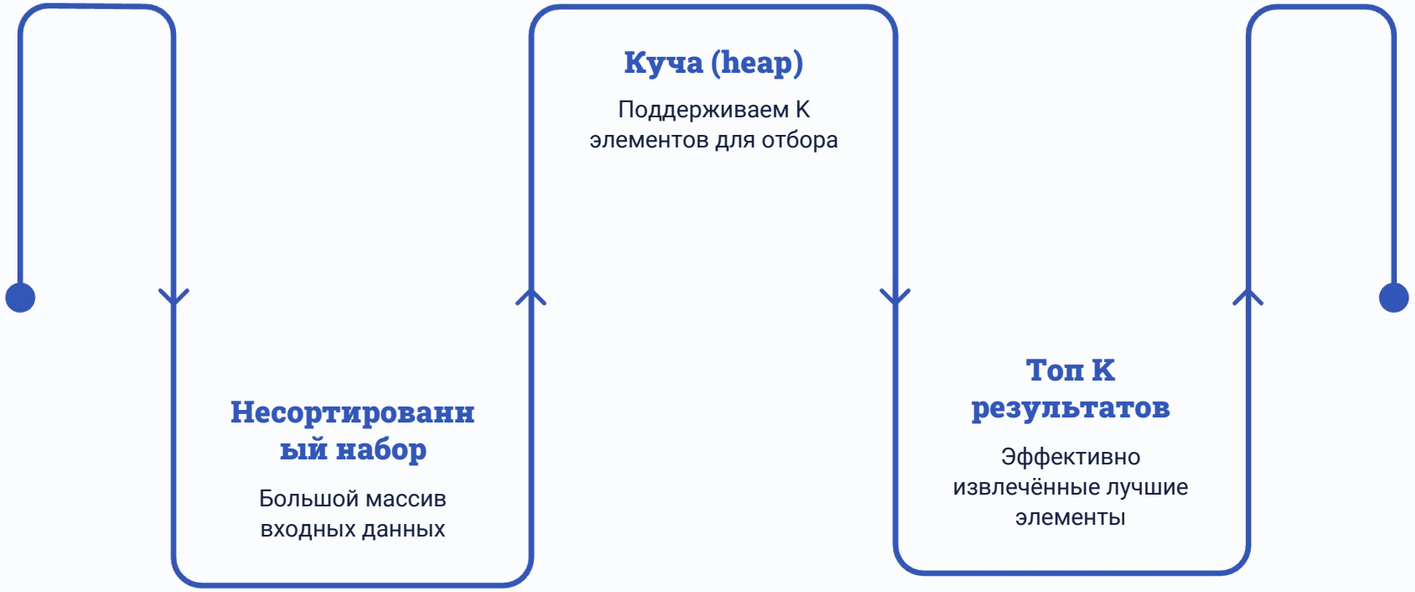
Реальные примеры

Топ-5 игроков в онлайн-игре, 3 самых частых слова в документе, 10 лучших товаров по рейтингу

Проблема наивного подхода

Сортировка всего массива требует O(n log n) времени, слишком дорого для больших данных

Ключевая идея паттерна



Используем специальную структуру данных, кучу (heap), для хранения только K элементов вместо работы со всем массивом.

1 Снижение сложности

Время работы уменьшается до O(n log k), значительный прирост производительности

2 Поддержание топа

Куча всегда содержит текущие топ K элементов в процессе обработки

Куча — структура с частичным порядком данных

Куча — это **НЕ сортированный массив**. Она знает только **одного лидера** — минимум или максимум.

Свойство	Мин-куча	Макс-куча
Корень	минимальный элемент	максимальный элемент
Инвариант	дети ≥ родителя	дети ≤ родителя
Что **НЕ** делает	не сортирует поддеревья	не сортирует поддеревья

Примеры:

```
Мин-куча:          Макс-куча:
     3                  9
    / \                / \
   5   9              6   4
  / \                / \
 7   8              2   3
```

Особенности кучи

Что умеет:

- Мгновенно отдавать минимум/максимум: `O(1)`
- Добавлять/удалять элемент с сохранением порядка: `O(log k)`

Что НЕ умеет:

- Хранить полностью отсортированный массив
- Быстро искать произвольный элемент (`O(k)`)

Ключевые операции (Python: heapq)

Операция	Сложность	
`heappush()`	*O(log k)*	Добавить элемент в кучу
`heappop()`	*O(log k)*	Извлечение и и удаление корня
`heapreplace()`	*O(log k)*	Удаляет корень и добавляет новый элемент

Разбор задачи: Топ-2 самых частых числа

Это задача **LeetCode 347**, классика для задач Топ Элементов

Условие задачи

Дано: [4, 6, 4, 6, 7, 4], k = 2
Найти: [4, 6] — два самых частых элемента

Ключевой инсайт

Куча размером ровно k — это эффективный фильтр:

Фиксированный размер

Никогда не растёт больше `k` элементов

Автоматическая очистка

Всегда выталкивает самого слабого кандидата

Оптимальная производительность

Работает за `O(n log k)` времени и `O(k)` памяти вместо `O(n log n)` времени и `O(n)` памяти при полной сортировке

Пошаговое решение

Дано: **[4, 6, 4, 6, 7, 4]**, k = 2, **Найти:** Два самых частых элемента

Шаг 0: Считаем частоты

4 : **3** раза
6 : **2** раза
7 : **1** раз

Важно: сначала строим частотный словарь, потом работаем с кучей

Шаг 1: Инициализация мин-кучи

Берём первые k пар (частота, элемент):

`Куча: [(2, 6), (3, 4)]`

Корень кучи = (2, 6), наименьший кандидат в корне

Почему мин-куча? Корень = минимальная частота среди кандидатов

Шаг 2: Обрабатываем оставшиеся элементы

Приходит элемент (1, 7), частота = 1

Сравниваем: 1 > корень.частота (2) ?: Нет

Решение: игнорируем, куча не меняется

Экономия: не вставляем лишний элемент: `O(1)` вместо `O(log k)`

Шаг 3: Получаем результат

Куча содержит: `[(2, 6), (3, 4)]`

Извлекаем числа: `[6, 4]`

Сортируем по убыванию частоты: `[4, 6]` **правильный ответ**

Важно: в условии LeetCode 347 допустим любой порядок элементов

Пример кода: топ-K самых частых элементов

Задача на LeetCode

LeetCode 347: Top K Frequent Elements

Ссылка: https://leetcode.com/problems/top-k-frequent-elements/

Формулировка:

Дано: массив целых чисел nums и число k

Найти: k самых частых элементов

Итоговая сложность:

- **Время:** O(n log k)
- **Память:** O(k)

```python
from collections import Counter
import heapq

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Шаг 1: считаем частоты
        freq = Counter(nums)

        # Защита от граничных случаев
        if k >= len(freq):
            return list(freq.keys())

        # Шаг 2: мин-куча размером k
        min_heap = []
        for num, count in freq.items():
            if len(min_heap) < k:
                heapq.heappush(min_heap, (count, num))
            elif count > min_heap[0][0]:
                heapq.heapreplace(min_heap, (count, num))

        # Шаг 3: извлекаем только элементы
        return [num for _, num in min_heap]
```

Шаг 0: Считаем частоты — два способа

Способ 1: collections.Counter (рекомендуется)

```
from collections import Counter
freq = Counter([1, 1, 1, 2, 2, 3])
# Результат: {1: 3, 2: 2, 3: 1}
```

Сложность: *O(n)*

Способ 2: ручной словарь (если нельзя использовать Counter)

```
freq = {}
for num in nums:
    freq[num] = freq.get(num, 0) + 1
```

Сложность: *O(n)*

Важно:

- Сначала строим частоты, потом кучу
- Куча работает с парами (частота, элемент), а не с исходным массивом

Классические задачи с паттерном Топ K

Задача 1

K-й наибольший элемент

LeetCode 215 (Medium)

Найти один элемент k-й по величине

Совет: мин-куча размером k, в конце вернуть корень

Задача 2

K ближайших точек

LeetCode 973 (Medium)

Точки с минимальным расстоянием до (0,0)

Совет: сравнивай квадраты расстояний, корень не извлекай

Задача 3

Топ-K частых слов

LeetCode 692 (Medium)

Найти K самых частых слов в массиве строк. При равных частотах — вернуть слова в лексикографическом порядке

Совет: инвертируй частоту

Типичные ловушки на собесе

1

Ловушка 1: Не проверили границы

```
# Ошибка:
heapq.nlargest(k,
freq.items())
```

```
# Правильно:
if k >= len(freq):
    return list(freq.keys())
```

Если k больше уникальных элементов, будет лишняя работа или ошибка

2

Ловушка 2: Вернули пары вместо элементов

```
# Ошибка:
return min_heap  # [(3, 1),
(2, 2)]
```

```
# Правильно:
return [num for _, num in
min_heap]  # [1, 2]
```

3

Ловушка 3: Забыли про мин-кучу в Python

```
# Для макс-кучи инвертируйте
знак:
heapq.heappush(heap, (-
priority, item))
```

В Python только мин-куча. Для поиска минимумов инвертируйте приоритет

Когда применять паттерн Топ K?

Ищите эти маркеры в условии:

Ключевые слова

- «топ-K», «K самых больших/маленьких»
- «самые частые», «самые редкие»
- «ближайшие K точек», «дальние K элементов»

Ограничения в условии

- «память ограничена» намёк: не можем хранить весь массив
- «не сортировать весь массив» прямой намёк на кучу

Размеры данных

- n очень большое (миллионы+), k маленькое (10–1000)
- Если бы сортировали — было бы слишком дорого: O(n log n) vs O(n log k)

Когда НЕ использовать паттерн Топ К

Нужна полная отсортированная последовательность

Если требуется весь массив в отсортированном виде, а не только топ K элементов

k близок к n (например, k = 90% от размера массива)

В этом случае куча не даст преимущества, проще отсосортировать весь массив

Нужен случайный доступ к любому элементу по индексу

Куча не поддерживает быстрый доступ по индексу, только для корня

Проверочный вопрос себе:

«Мне нужны только лучшие K элементов или весь отсортированный массив?»

Только K: Топ K + куча

Весь массив: обычная сортировка

ЗАКЛЮЧЕНИЕ

Подписывайся и оставайся на связи!

Сегодня ты узнал:

Сегодня ты узнал как решать задачу про K Top элементов, без повторного хранения элементов всего массива

Не откладывай практику:

- **Реши LeetCode 215 (Kth Largest Element)**
- **Реализуй кучу вручную — без heapq**

В следующем видео:

Как объединить 10 000 встреч в календаре за линейное время?

Все примеры и код — в описании.

Контакты и ресурсы:

Телеграм: t.me/marat_notes

Репозиторий с кодом: github.com/MaratNotes/marat_notes

YouTube: youtube.com/@marat_notes

VK Видео: vkvideo.ru/@club231048746