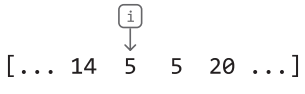
Глава 1
ДВА УКАЗАТЕЛЯ

Основные понятия

Как очевидно следует из названия, метод двух указателей представляет собой алгоритм, в котором используются два указателя. Но что такое указатель в данном контексте? Это переменная, представляющая индекс или позицию в структуре данных, такой как массив или связный список. Многие алгоритмы используют только один указатель, чтобы получить элемент или отслеживать текущую позицию:



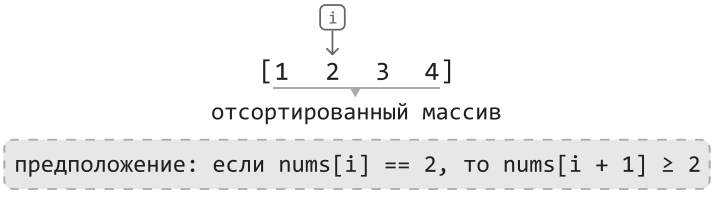
Добавление второго указателя открывает новые возможности. Самая важная из них — это сравнивать значения. Когда указатели находятся в двух разных позициях, можно сравнивать элементы и принимать на основе этого решения:

Во многих случаях такие сравнения реализуются с помощью вложенных циклов `for`, что дает временну́ю сложность $O(n^2)$, где n — длина структуры данных. В приведенном ниже фрагменте кода `i` и `j` — это два указателя, используемые для сравнения каждой пары элементов массива:

```
for i in range(n):
    for j in range(i + 1, n):
        compare(nums[i], nums[j])_
```

Часто такой подход не использует **потенциально предсказуемые** свойства структуры данных. Например, в отсортированном массиве можно заранее предсказать, увеличится или уменьшится значение при смещении указателя. Если мы двигаем указатель вправо в массиве, отсортированном по возрас-

танию, то гарантированно переходим к значению, которое больше или равно текущему:



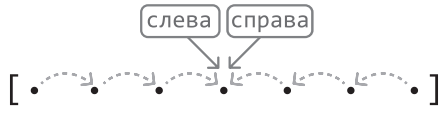
Таким образом, в структурах с предсказуемым поведением указатели можно перемещать логично и эффективно. Использование этой предсказуемости позволяет улучшать временную и пространственную сложность алгоритма — что мы и рассмотрим на реальных задачах собеседований в этой главе.

Стратегии с двумя указателями

Алгоритмы, использующие два указателя, как правило, имеют временную сложность $O(n)$, поскольку позволяют избежать вложенных циклов. Существует три основные стратегии применения этой техники.

Движение навстречу

В этой стратегии указатели начинают с противоположных концов структуры данных и постепенно двигаются друг к другу:



Указатели сдвигаются к центру, пока не будет выполнено условие задачи или пока они не пересекутся. Такой подход идеально подходит, если нужно сравнивать элементы с разных концов массива или строки.

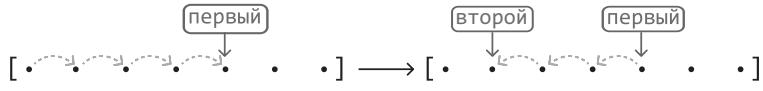
Однонаправленное движение

В этой стратегии перемещение обоих указателей начинается с одного конца структуры данных (обычно с начала) и происходит в одном направлении:

Как правило, указатели выполняют разные, но дополняющие друг друга функции. Например, один (чаще правый) ищет нужную информацию, а другой (чаще левый) — отслеживает текущую позицию или состояние.

Поэтапное движение

Здесь один указатель проходит по структуре данных, и, как только он встречает элемент, удовлетворяющий определенному условию, второй указатель начинает двигаться от этой позиции для дальнейшего анализа:



Как и в однонаправленном проходе, указатели выполняют разные функции: первый ищет определенные значения, а второй находит дополнительную информацию о найденном значении или диапазоне.

Мы подробно разберем все эти подходы на практических задачах в этой главе.

Когда использовать два указателя?

Алгоритмы с двумя указателями, как правило, применяются к линейным структурам данных, таким как массивы или связные списки. Один из признаков того, что задачу можно решить с помощью этой техники — предсказуемые изменения во входных данных, например **сортировка массива**.

Предсказуемые изменения могут проявляться по-разному. Например, палиндромная строка имеет симметричную структуру, что позволяет логично перемещать два указателя навстречу друг другу от концов к центру. По мере решения задач в этой главе вы научитесь легко распознавать такие свойства.

Другой признак, указывающий на применимость подхода двух указателей, — если задача требует найти пару значений или результат, зависящий от двух элементов.

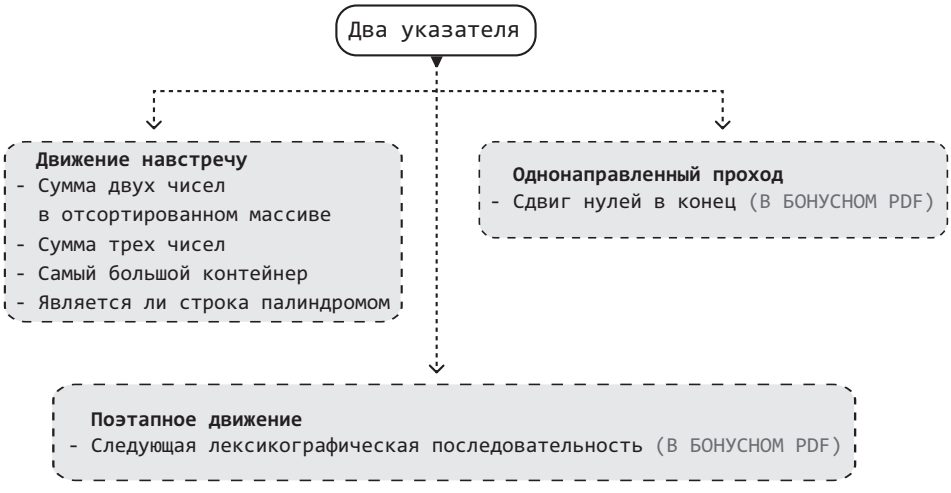
Пример из реальной жизни

Алгоритмы сборки мусора. При уплотнении памяти — важнейшем этапе сборки мусора — цель заключается в том, чтобы освободить непрерывный участок памяти, избавившись от пробелов, оставленных удаленными (то есть мертвыми) объектами. Для этого эффективно использовать подход с двумя указателями:

- Указатель `scan` проходит по куче, чтобы определить живые объекты.
- Указатель `free` отслеживает ближайшее доступное место, куда нужно переместить живой объект.

По мере продвижения `scan` пропускает удаленные элементы и перемещает живые объекты в позицию, указываемую `free`. Таким образом, живые объекты группируются, а неиспользуемая память освобождается в виде сплошного блока.

Содержание главы



</>

Решайте эти задачи в нашем онлайн-редакторе кода:
⟶ bit.ly/run-code

Метод двух указателей — универсальный и имеет множество специализированных вариантов, которые мы выделили в отдельные главы, такие как «Быстрый и медленный указатели» и «Скользящие окна».

Сумма двух чисел в отсортированном массиве

Дан массив целых чисел, отсортированный по возрастанию, и целевое значение (target). Необходимо вернуть индексы любой пары чисел в массиве, сумма которых равна целевому значению. Порядок индексов в ответе не важен. Если подходящей пары не найдено, вернуть пустой массив.

Пример 1:

Ввод: `nums = [-5, -2, 3, 4, 6]`, `target = 7`

Вывод: `[2, 3]`

Пояснение: `nums[2] + nums[3] = 3 + 4 = 7`

Пример 2:

Ввод: `nums = [1, 1, 1]`, `target = 2`

Вывод: `[0, 1]`

Также допустимы ответы: `[1, 0]`, `[0, 2]`, `[2, 0]`, `[1, 2]`, `[2, 1]`

Разбор решения

Решение методом полного перебора для этой задачи заключается в проверке всех возможных пар. Это достигается двумя вложенными циклами: внешний перебирает первый элемент пары, а внутренний — оставшиеся элементы, чтобы найти второй. Ниже приведен фрагмент кода для этого метода:

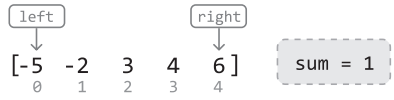
```
def pair_sum_sorted_brute_force(nums: List[int], target: int) -> List[int]:
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

Временная сложность такого подхода — $O(n^2)$, где n — длина массива. Однако мы не используем тот факт, что массив уже отсортирован. Поможет ли это создать более эффективное решение?

Здесь стоит рассмотреть **подход с двумя указателями**, поскольку отсортированный массив позволяет перемещать указатели логично. Давайте посмотрим, как это работает, на примере ниже:

```
[-5  -2   3   4   6],   target = 7
  0   1   2   3   4
```

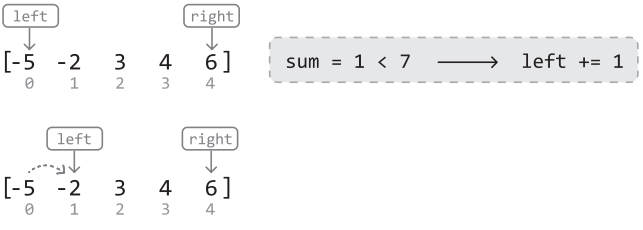
Хорошей отправной точкой будет рассмотреть наименьшее и наибольшее значения: первый и последний элементы массива соответственно. Их сумма равна 1.



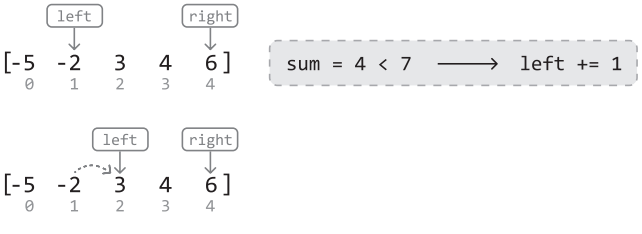
Поскольку 1 меньше целевого значения, нам нужно **сдвинуть один из указателей**, чтобы найти новую пару с большей суммой.

- **Левый указатель (left):** всегда указывает на значение, меньшее или равное значению, на которое указывает правый указатель, так как массив отсортирован. Если левый указатель переместить вправо, сумма станет больше или равна текущей (1).
- **Правый указатель (right):** перемещение влево правого указателя даст сумму, которая будет меньше или равна 1.

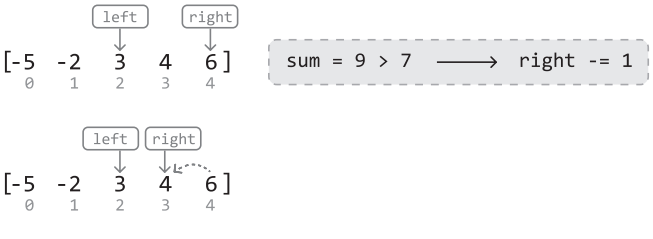
Следовательно, мы должны переместить вправо левый указатель, чтобы получить бóльшую сумму:



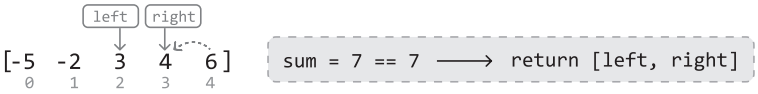
Снова сумма значений по текущим указателям (4) слишком мала. Поэтому еще раз переместим вправо левый указатель:



Теперь сумма (9) слишком велика. Поэтому переместим влево правый указатель, чтобы найти пару с меньшей суммой:



Наконец, мы нашли два числа, сумма которых равна целевому значению. Вернем их индексы:



Мы продемонстрировали работу алгоритма с двумя указателями, двигающимися навстречу. Давайте подведем итог. Для любой пары значений `left` и `right`:

- Если их сумма меньше цели, переместим вправо `left`, чтобы увеличить ее и приблизиться к цели.
- Если их сумма больше цели, переместим влево `right`, чтобы уменьшить ее и приблизиться к цели.
- Если сумма равна целевому значению, вернем индексы: `[left, right]`.

Мы можем прекратить движение указателей `left` и `right`, когда они встретятся: это означает, что ни одна пара не дает нужную сумму.

Реализация

```
def pair_sum_sorted(nums: List[int], target: int) -> List[int]:
    left, right = 0, len(nums) - 1
    while left < right:
        sum = nums[left] + nums[right]
        # Если сумма меньше, сдвигаем вправо левый указатель,
        # чтобы приблизиться к целевому значению.
        if sum < target:
            left += 1
        # Если сумма больше, сдвигаем влево правый указатель,
        # чтобы приблизиться к целевому значению.
        elif sum > target:
            right -= 1
        # Если нашли нужную пару — возвращаем ее индексы.
        else:
            return [left, right]
    return []
```

Анализ сложности

Временна́я сложность функции `pair_sum_sorted` — $O(n)$, поскольку в худшем случае мы выполняем примерно n итераций, используя метод двух указателей.

Пространственная сложность — $O(1)$, поскольку используется только фиксированное количество переменных.

Тестовые сценарии

В дополнение к уже рассмотренным примерам ниже приведены другие тесты, которые вы можете использовать. Эти дополнительные кейсы охватывают разные сценарии и позволяют убедиться, что ваш код работает корректно при разных входных данных.

Тестирование важно, потому что:

- помогает выявить ошибки в коде;
- гарантирует, что решение работает с редкими или граничными случаями;
- помогает учесть важные детали, которые иначе можно было бы упустить из виду.

Входные данные	Ожидаемый результат	Описание
`nums = [], target = 0`	`[]`	Тест на пустой массив
`nums = [1], target = 1`	`[]`	Массив из одного элемента — невозможно составить пару
`nums = [2, 3], target = 5`	`[0, 1]`	Пара из двух элементов, сумма которых равна цели
`nums = [2, 4], target = 5`	`[]`	Пара из двух элементов, не дающая нужную сумму
`nums = [2, 2, 3], target = 5`	`[0, 2]` или `[1, 2]`	Тест с повторяющимися значениями
`nums = [-1, 2, 3], target = 2`	`[0, 2]`	Проверка работы с отрицательными числами
`nums = [-3, -2, -1], target = -5`	`[0, 1]`	Оба числа в паре отрицательные

Совет: используйте всю предоставленную информацию.

Интервьюеры часто дают лишь минимально необходимое количество данных, чтобы вы могли приступить к решению задачи. Поэтому крайне важно внимательно проанализировать всю доступную информацию и определить, какие детали критичны для эффективного решения. В этой задаче ключ к оптимальному решению — понять, что входной массив отсортирован.

Сумма трех чисел

> Дан массив целых чисел. Необходимо вернуть все уникальные тройки чисел `[a, b, c]`, так чтобы `a + b + c = 0`. Решение не должно содержать повторяющиеся тройки (например, `[1, 2, 3]` и `[2, 3, 1]` считаются одинаковыми). Если подходящих троек чисел нет — вернуть пустой массив.
>
> Числа в каждой тройке могут быть возвращены в любом порядке, и порядок самих троек в выходном массиве также неважен.
>
> **Пример:**
>
> Ввод: `nums = [0, -1, 2, -3, 1]`
>
> Вывод: `[[-3, 1, 2], [-1, 0, 1]]`

Разбор решения

Решение методом полного перебора предполагает проверку всех возможных трех чисел в массиве, чтобы определить, дают ли они в сумме 0. Это можно реализовать тремя вложенными циклами, перебирающими все комбинации трех элементов.

Чтобы избежать повторяющихся троек чисел, можно отсортировать каждую найденную тройку — в результате одинаковые тройки с разным порядком (например, `[1, 3, 2]` и `[3, 2, 1]`) будут приведены к одному виду (`[1, 2, 3]`). Затем можно добавить такие отсортированные тройки в множество, чтобы убрать дубликаты. Вот пример реализации такого подхода:

```
def triplet_sum_brute_force(nums: List[int]) -> List[List[int]]:
    n = len(nums)
    # Используем множество, чтобы избежать повторений.
    triplets = set()
    # Перебираем индексы всех возможных троек.
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if nums[i] + nums[j] + nums[k] == 0:
                    # Сортируем тройку перед добавлением в
                    # множество.
                    triplet = tuple(
                        sorted([nums[i], nums[j], nums[k]])
                    )
                    triplets.add(triplet)
    return [list(triplet) for triplet in triplets]
```

Это решение очень неэффективно, его временная сложность — $O(n^3)$, где n — длина входного массива. Как можно его улучшить?

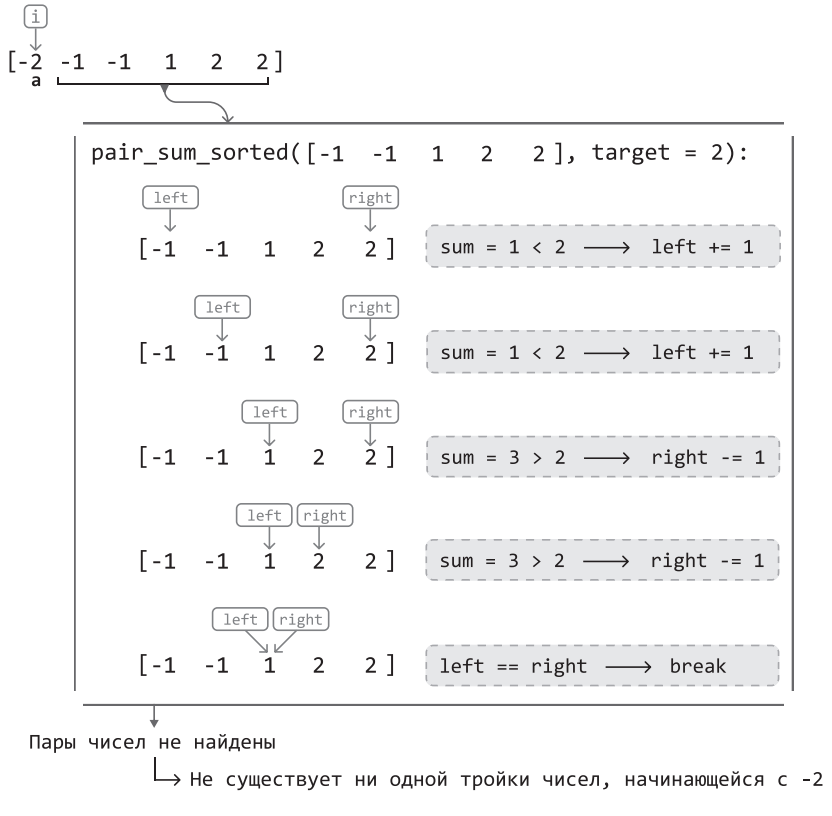
Попробуем найти хотя бы одну тройку чисел, сумма которых равна нулю. Обратите внимание: если мы зафиксируем один из элементов, задача сводится к поиску пары, сумма которой равна противоположному значению этого элемента.

> Для любой тройки `[a, b, c]`, если мы **зафиксируем** `a`, задача превращается в **поиск пары** `[b, c]`, **так что** `b + c = -a` (так как `a + b + c = 0 → b + c = -a`).

Похоже на знакомую задачу? Это потому, что задача поиска пары чисел, сумма которых равна цели, уже рассматривалась в разделе «Сумма двух чисел в отсортированном массиве». Однако мы можем использовать этот алгоритм только на отсортированном массиве. Поэтому первое, что нужно сделать — **отсортировать входные данные**. Рассмотрим следующий пример:

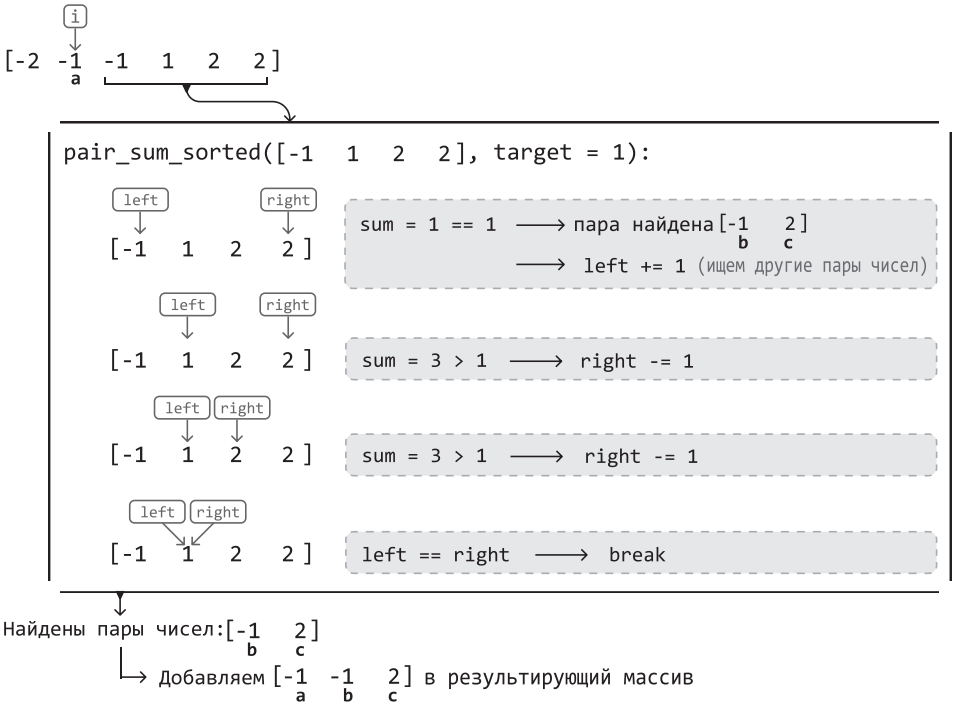
```
[-1  2 -2  1 -1  2] --сортировка--> [-2 -1 -1  1  2  2]
```

Теперь берем первый элемент `-2` (то есть с `a`) и используем метод `pair_sum_sorted` для оставшейся части массива, чтобы найти пару, сумма которой равна `2` (то есть `-a`):



Как видно, при вызове `pair_sum_sorted` не удалось найти пару с суммой `2`. Это означает, что троек чисел, начинающихся с `-2` и дающих сумму `0`, не существует.

Значит, сдвинем вправо основной указатель `i` и попробуем снова.



На этот раз мы нашли одну пару, которая дала корректную тройку.

Если продолжить процесс для оставшихся элементов массива, окажется, что `[-1, -1, 2]` — единственная тройка чисел, сумма которой равна 0.

Существует важное отличие между реализациями `pair_sum_sorted` в задаче «Сумма двух чисел в отсортированном массиве» и в данной задаче: здесь мы не останавливаемся после нахождения одной пары чисел, а продолжаем поиск, пока не будут найдены все подходящие варианты.

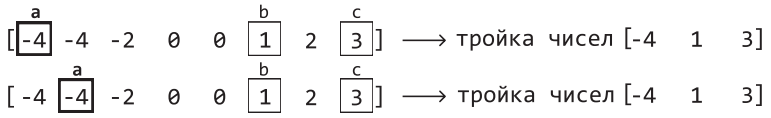
Обработка повторяющихся троек чисел

Мы ранее не акцентировали внимание на том, как избежать добавления в результат дубликатов троек чисел. Такая ситуация может возникнуть в двух случаях. Рассмотрим пример:

```
[-4 -4 -2 0 0 1 2 3]
```

Случай 1: повторяющиеся значения a

Первый случай, при котором могут появиться дубликаты, — это поиск пар чисел для троек, начинающихся с одного и того же значения a:



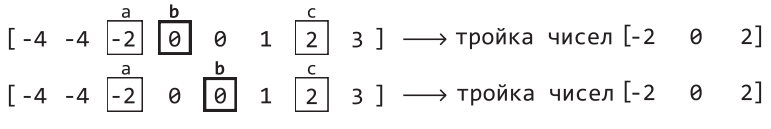
Поскольку pair_sum_sorted будет искать пары, сумма которых равна -a, в обоих случаях, мы, естественно, получим одни и те же пары чисел, а значит — одинаковые тройки.

Чтобы избежать выбора одного и того же значения a, мы продолжаем увеличивать i (где nums[i] — это текущее значение a), пока не встретим число, отличное от предыдущего. Мы делаем это до того, как начнем искать пары с помощью метода pair_sum_sorted. Эта логика работает, потому что массив отсортирован и равные значения идут друг за другом. Вот пример кода, проверяющего дубликаты значений a:

```
# Чтобы избежать дубликатов троек чисел, убедитесь, что текущее значение "a"
# не повторяет предыдущее в отсортированном массиве.
if i > 0 and nums[i] == nums[i - 1]:
    continue
... Ищем тройки чисел ...
```

Случай 2: повторяющиеся значения b

Во втором случае дубликаты могут возникать в процессе выполнения pair_sum_sorted. Для фиксированного целевого значения (-a) пары, начинающиеся с одного и того же b, всегда будут одинаковыми:



Чтобы избежать этого, нужно, как и раньше, пропускать повторяющиеся b.

Важно отметить, что нет необходимости явно обрабатывать дубликаты c. Если мы избегаем повторов в a и b, то каждая пара [a, b] будет уникальной, а значит и значение c (вычисляется как c = -(a + b)) тоже будет уникальным. Таким образом, избегая дубликатов в a и b, мы автоматически исключаем дубликаты в тройках [a, b, c].

Оптимизация

Интересное наблюдение: тройки чисел с суммой 0 не могут состоять только из положительных чисел. Поэтому можно прекратить поиск троек, как только

значение `a` становится положительным, так как это означает, что и `b`, и `c` тоже будут положительными, а значит, сумма будет больше `0`.

Реализация

Исходя из приведенной выше логики, нужно немного изменить функцию `pair_sum_sorted`, чтобы избежать дублирующихся троек чисел. Также необходимо передать параметр `start`, чтобы указать, с какой позиции подмассива начинать выполнение алгоритма поиска пар чисел. В остальном логика двух указателей остается почти идентичной той, что использовалась в задаче «Сумма двух чисел в отсортированном массиве».

```
def triplet_sum(nums: List[int]) -> List[List[int]]:
    triplets = []
    nums.sort()
    for i in range(len(nums)):
        # Оптимизация: тройки, состоящие только из
        # положительных чисел, не могут в сумме давать 0.
        if nums[i] > 0:
            break
        # Чтобы избежать повторяющихся троек, пропускаем "a",
        # если оно совпадает с предыдущим числом.
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        # Находим все пары, сумма которых равна -a (-nums[i]).
        pairs = pair_sum_sorted_all_pairs(nums, i + 1, -nums[i])
        for pair in pairs:
            triplets.append([nums[i]] + pair)
    return triplets

def pair_sum_sorted_all_pairs(nums: List[int],
                              start: int, target: int) -> List[int]:
    pairs = []
    left, right = start, len(nums) - 1
    while left < right:
        sum = nums[left] + nums[right]
        if sum == target:
            pairs.append([nums[left], nums[right]])
            left += 1
            # Чтобы избежать дубликатов пар [b, c],
            # пропускаем "b", если оно такое же, как предыдущее.
            while left < right and nums[left] == nums[left - 1]:
                left += 1
        elif sum < target:
            left += 1
        else:
            right -= 1
    return pairs
```