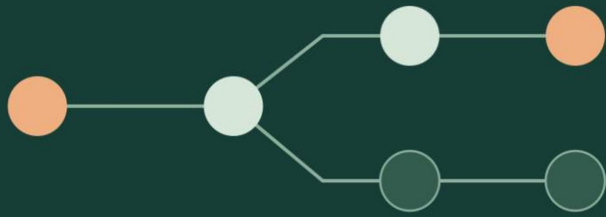
ГЛЕБ ЗАЙЦЕВ

Алгоритмы на Python



ОТ УСЛОВИЯ К ПРОВЕРЕННОМУ КОДУ

24 **задачи**
с решениями
и проверками

Поиск · структуры данных · графы
Динамическое программирование

ПРАКТИКУМ ПОСЛЕ ОСНОВ PYTHON

Глеб Зайцев

Алгоритмы на Python. 24 задачи с решениями и проверками

https://litres.ru/74417087
SelfPub; 2026

Аннотация

Практикум для тех, кто уже знает списки, словари, циклы и функции Python, но хочет научиться выбирать алгоритм под задачу. 24 самостоятельные работы: повторы и серии, поиск и интервалы, стек и очередь, пути в графах, зависимости, динамическое программирование и перебор. Для каждой работы заданы допустимые данные и правила выбора ответа, есть подсказка, полное решение, пример, дополнительные тесты и разбор сложности. Все данные приведены в книге; сторонние пакеты и платные сервисы не нужны. Отдельная инструкция помогает перенести код из электронной читалки с сохранением отступов. Это учебный практикум, а не обещание трудоустройства или готовые промышленные компоненты.

Содержание

Глеб Зайцев
Алгоритмы на Python. 24 задачи с решениями и проверками

От работающего примера к обоснованному решению

Найти короткую программу несложно. Сложнее объяснить, почему она решает именно вашу задачу: что произойдёт на пустом списке, какой из двух допустимых ответов вернётся и не исчезнут ли повторяющиеся элементы. Эти вопросы не украшение готового решения. Они определяют, какую информацию придётся хранить и какие действия программа имеет право выполнять.

В этой книге 24 самостоятельные задачи. Это не курс синтаксиса Python с нуля. Предполагается, что вы можете определить функцию, написать цикл, обратиться к списку и словарю и понять логическое условие. Новые структуры и приёмы появляются через конкретную работу: сначала формулируем контракт, затем выбираем состояние, которое будет поддерживать алгоритм.

Первая половина посвящена последовательностям, поиску, интервалам, стеку и очереди. Во второй появляются графы и задачи выбора: кратчайшие пути, порядок зависимостей, динамическое программирование и перебор. Последовательность позволяет наращивать инструменты, но каждый блок решения содержит всё необходимое для своего запуска. Предыдущие программы в память загружать не требуется.

Условия, учебные данные и объяснения подготовлены для этого практикума. Сами алгоритмические идеи общеизвестны: мы не выдаём двоичный поиск или поиск в ширину за авторские открытия. Книга подготовлена с помощью ИИ; решения выполняются и проверяются на примерах и дополнительных случаях. Автоматические тесты помогают найти ошибки, но не заменяют понимания условий и не доказывают пригодность программы для любых производственных данных.

Результат работы с задачей — не только зелёные проверки. Попробуйте без кода сказать, что хранится в каждой структуре, почему очередной шаг не теряет нужный ответ и когда алгоритм останавливается. Если объяснение не получается, вернитесь к небольшому примеру и выпишите состояния на бумаге. Пять аккуратно разобранных элементов полезнее большого случайного списка без понятного ответа.

Среда и безопасный учебный запуск

Решения проверяются на Python 3.12. Используются только встроенные возможности и стандартная библиотека; установка пакетов не нужна. Интерпретатор берите из официального источника python.org. Команда запуска зависит от системы: часто это python3 task.py или py task.py. Важно, чтобы выбранная команда действительно запускала Python 3, а не другую установленную программу.

Работайте в отдельной новой учебной папке. Все входные данные синтетические и приведены прямо в тексте. Функции задач не обращаются в сеть, не читают документы с компьютера и не изменяют переданные им входные списки. Не подставляйте реальные персональные данные ради эксперимента: они здесь ничего не добавят к пониманию алгоритма.

Данные вне явно оговорённого контракта не обязаны обрабатываться красиво. Например, двоичный поиск здесь получает уже отсортированный список, а алгоритм Дейкстры — неотрицательные веса. Отдельная промышленная оболочка могла бы проверять и отклонять неверный ввод. Такие проверки стоят времени и места; отсутствие оболочки не превращает ограниченный учебный алгоритм в универсальный сервис.

Функция называется solve во всех заданиях, но это разные функции. Не складывайте решения подряд в один файл: последнее определение заменит предыдущие. Для очередной задачи используйте отдельную пустую папку или новое имя файла и переносите полный блок решения вместе с блоком проверок этой же задачи. Условия и ожидаемый вывод копировать как Python-код не нужно.

Перенос кода из электронной книги

Отступы Python имеют смысл, но электронная читалка может схлопывать несколько пробелов в один. Поэтому перед каждой строкой решения и проверок стоит служебная вертикальная черта `|`, а каждый пробел ОБЯЗАТЕЛЬНОГО НАЧАЛЬНОГО отступа показан средней точкой ·. Например, `|····return result` означает четыре обычных пробела перед return result. Восемь точек означают восемь пробелов. Слева от черты читалка может добавлять собственные поля; они не относятся к программе.

Черта и средние точки — обозначения оформления, а не синтаксис Python и не знаки умножения. При переносе удалите черту вместе с полями слева от неё и замените каждую среднюю точку одним обычным пробелом. Обычные точки в именах методов, например list.append, не трогайте: это другой символ. Можно перепечатать короткое решение вручную или воспользоваться подготовительным скриптом ниже. Он снимает оформление, не меняя вычислений алгоритма. Средние точки используются только для начальных отступов, не внутри строковых данных задач.

Для автоматического переноса создайте новую пустую папку. Скопируйте из ОДНОЙ задачи весь блок решения,

затем её блок «Проверки для запуска» в обычный текстовый файл copied.txt, сохранённый как UTF-8. Создайте в той же папке clean.py и введите следующие десять строк без служебных черт. У всех этих десяти строк левый край одинаковый: обязательных отступов в самом подготовительном скрипте нет. Запустите python3 clean.py, затем python3 task.py.

```
from pathlib import Path
text = Path("copied.txt").read_text(encoding="utf
lines = text.splitlines()
marked = [s for s in lines if "|" in s]
stripped = [s.split("|", 1)[1] for s in marked]
clean = [s.replace("·", " ") for s in stripped]
result = "\n".join(clean).replace("\u00a0", " ") + "\n"
f = open("task.py", "x", encoding="utf-8")
f.write(result)
f.close()
```

Подготовительный скрипт создаёт task.py в режиме x: если такой файл уже существует, выполнение остановится с FileExistsError и старый файл останется нетронутым. Для следующей задачи безопаснее взять новую папку. Не меняй-

те режим на w только ради обхода ошибки, если в папке лежит нужная работа. Скрипт ничего не скачивает и не запускает автоматически; просмотрите получившийся task.py перед запуском.

Если читалка визуально переносит длинную строку, это ещё не новая строка исходника. Настоящая следующая строка начинается с очередной служебной черты. После копирования проверьте, что каждая такая строка сохранилась целиком. Если перенос превратился в настоящий разрыв, соедините части в редакторе до следующей черты. Не заменяйте обычные прямые кавычки типографскими и не меняйте обратную косую черту на похожий знак.

Контракт, инвариант и контрпример

Контракт — соглашение о том, какой ввод допустим и что именно должно вернуться. «Найти пару» недостаточно: нужно сказать, можно ли использовать один элемент дважды, считать ли разные позиции с одинаковыми значениями разными элементами и что делать, если пар несколько. В задачах этой книги такие детали входят в условие, а не оставляются на догадку читателю.

Инвариант — утверждение, которое сохраняется после каждого шага. В задаче с повторениями это может быть смысл множества уже встреченных значений; в поиске — границы области, где ещё может находиться ответ. Чтобы обосновать алгоритм, проверьте три вещи: утверждение верно до первого шага, сохраняется при очередном шаге и вместе с условием остановки даёт требуемый результат.

Контрпример показывает, где привлекательная догадка ломается. Если хочется заменить список множеством, сначала попробуйте ввод с повторами и проверьте, важен ли их порядок. Если хочется взять самый крупный номинал монеты, придумайте набор номиналов, на котором остаток требует лишних монет. Контрпример не опровергает полезность идеи вообще; он показывает, что ей нужны дополнительные

предпосылки.

Не все правильные ответы одинаково удобны для проверки. Иногда задача разрешает много путей, но книга фиксирует порядок обхода соседей. Иногда мы возвращаем только расстояние или число, чтобы не отвлекаться на восстановление самого объекта. Это осознанные разные контракты. Нельзя незаметно заменить один другим и затем обвинять тест в том, что он не принимает новый формат.

Как читать оценки времени и памяти

Буква n обычно обозначает число входных элементов; для графа отдельно используются число вершин V и число рёбер E. Запись O(n) описывает порядок роста числа операций при увеличении размера задачи, а не время в миллисекундах. Два линейных алгоритма могут заметно различаться на конкретной машине. Измерение помогает выбирать реализацию, но не заменяет оценку поведения на растущем вводе.

В задачах со словарями и множествами средняя постоянная стоимость операции — предположение стандартной модели хеширования, не обещание для любого набора ключей. Сравнение длинных строк тоже не бесплатно. Когда элементы являются небольшими числами или короткими строками, эти издержки удобно вынести за рамки первого разбора; на больших объектах их нужно учитывать отдельно.

Дополнительная память — то, что алгоритм создаёт сверх входа. Если возвращаемый результат сам может содержать n элементов, мы отдельно говорим, включён ли он в оценку. Копия входного списка занимает память, даже если сортировка этой копии называется «на месте». Рекурсивные вызовы также используют стек. Экономия одной таблицы не означает отсутствия остальных затрат.

Для динамического программирования размер числового параметра может быть важнее количества входных элементов. Таблица до суммы T содержит T+1 ячейку, хотя само число T записывается гораздо короче. Поэтому решение, быстрое для небольшого учебного T, не обязательно подходит для огромных значений. В этой книге нет обещаний, что любые большие входы уложатся в память.

Пять шагов на одну задачу

Сначала прочитайте условие и вручную получите ответ для основного примера. Затем придумайте один крайний случай, которого нет в условии. После этого воспользуйтесь подсказкой и напишите своё решение. Только теперь сравните его с приведённым кодом: важно сопоставлять не внешний вид строк, а контракт и сохраняемую информацию.

В блоке проверок строка assert сравнивает фактический ответ с ожидаемым. Если сравнение ложно, Python выдаёт AssertionError и не доходит до заключительного сообщения OK. Отсутствие ошибки означает лишь прохождение именно этих проверок. Не запускайте учебные проверки с флагом -O: в таком режиме assert может быть отключён.

Один хороший тест меняет одну смысловую деталь: добавляет дубликат, делает цель недостижимой, замыкает цикл, ставит границу на конец массива. Случайные данные полезны после этого, когда уже есть независимый способ получить эталон. Сравнивать программу с её собственной слегка переписанной копией недостаточно: обе могут повторять одну ошибку.

После решения ответьте на дополнительный вопрос. Он обычно изменяет контракт, а не просит механически увеличить размер входа. Если условия изменились, прежняя оценка сложности и прежнее доказательство требуют пересмот-

ра. Не обязательно сразу писать новый код: точное объяснение того, какая часть подхода перестала работать, уже является результатом.

Задача 1. Первое появление в журнале

Редактор выгрузил названия рубрик в порядке открытия вкладок. Ему нужен список рубрик без повторов, но именно в порядке первого появления: алфавитная сортировка скроет последовательность работы. Функция solve(items) получает конечный список строк и возвращает новый список. Равенство строк обычное, чувствительное к регистру и пробелам: 'Код', 'код' и 'код ' — разные значения. Пустая строка допустима, пустой вход даёт пустой результат. Вход не изменяется; каждое различное значение должно остаться ровно один раз.

Пример

Аргументы solve: (['карты', 'почта', 'карты', 'архив', 'почта'],)

Ожидаемый ответ: ['карты', 'почта', 'архив']

Подсказка

Ответу нужен порядок, а проверке повтора — быстрое членство. Не заставляйте одну структуру выполнять обе ро-

ли: заведите список результата и множество уже встреченных строк.

Решение

```
def solve(items):
    seen = set()
    result = []
    for item in items:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result
```

Проверки для запуска

```
cases = [((['карты', 'почта', 'карты', 'архив', 'почта'],),
  ['карты', 'почта', 'архив']),
 ((([],),), []),
 ((([''', '', 'a', ''],),), ['', 'a']),
 ((['Код', 'код', 'Код', 'код '],),), ['Код', 'код', 'код ']),
 ((['x', 'x', 'x'],),), ['x'])]
for args, expected in cases:
```

```
    assert solve(*args) == expected
print('OK: задача 1')
```

После успешных проверок: OK: задача 1

Почему это работает

На примере сначала обе структуры пусты. 'карты' отсутствует в seen, поэтому попадает и в множество, и в ответ. С 'почта' происходит то же самое. Второе появление 'карты' не меняет ни одну структуру. Затем добавляется 'архив', а последняя 'почта' пропускается. Таким образом, позиция добавления строки зависит только от её первого появления во входном журнале.

Инвариант после обработки любого префикса таков: seen содержит все различные строки этого префикса, а result — их первые появления в исходном порядке. Если новая строка уже известна, префикс не приобрёл нового значения и менять ответ нельзя. Если строка неизвестна, её текущая позиция неизбежно первая; добавление в конец сохраняет порядок всех предыдущих значений.

Когда обход завершён, инвариант описывает весь вход, а значит, доказывает и полноту результата, и отсутствие повторов, и устойчивость порядка. Само множество не задаёт порядок ответа: его перебор здесь вообще не используется. Строки хешируемы, поэтому подходят для множества; списки внутри items потребовали бы другого контракта. Мы не

исправляем регистр и пробелы незаметно для вызывающего кода: это отдельная задача.

Время и память

n — число строк, u — число различных строк. В среднем O(n) хеш-операций и O(u) дополнительной памяти без ответа. Это не гарантия худшего случая: при коллизиях возможно $O(n^2)$ сравнений. Хеширование и сравнение длинных строк оплачиваются отдельно по числу просмотренных символов.

Типичная ошибка

Вернуть list(set(items)): повторы исчезнут, но требуемый порядок первого появления не гарантирован. Проверять членство только в result корректно, однако на разных строках даёт $O(n^2)$ сравнений.

Измените условие

Можно ли сохранить порядок первых появлений, но вывести каждую строку в верхнем регистре?

Разбор вопроса

Да: проверяйте исходную строку в seen, а в result добавляйте item.upper(). Но разные исходные строки могут дать одинаковый вывод. Если уникальность нужна уже после преобразования, в множество следует класть преобразованное значение.

Задача 2. Свернуть соседние отсчёты

Датчик передаёт целочисленный режим каждую секунду. Для короткого отчёта нужно свернуть только соседние одинаковые отсчёты: возвращение к прежнему режиму позже начинает новую серию. Функция solve(values) принимает список целых чисел и возвращает список пар (значение, длина_серии). Пары следуют в порядке серий входа, длина всегда положительна. Пустой вход даёт пустой список. Сравнение — обычное равенство целых чисел; отрицательные числа и ноль допустимы. Входной список должен остаться без изменений.

Пример

Аргументы solve: ([4, 4, 1, 1, 1, 4, 0, 0],)
Ожидаемый ответ: [(4, 2), (1, 3), (4, 1), (0, 2)]

Подсказка

Держите отдельно значение текущей незаконченной серии и её длину. Закрывайте серию при смене значения. Подумайте, кто закроет последнюю серию, если следующего от-

счёта уже нет.

Решение

```
def solve(values):
    if not values:
        return []
    result = []
    current = values[0]
    count = 1
    for index in range(1, len(values)):
        value = values[index]
        if value == current:
            count += 1
        else:
            result.append((current,
count))
            current = value
            count = 1
    result.append((current, count))
    return result
```

Проверки для запуска

```
cases = ((([4, 4, 1, 1, 1, 4, 0, 0],),
```

```
[(4, 2), (1, 3), (4, 1), (0, 2)]),
  |·(([],), []),
  |·(([7],), [(7, 1)]),
  |·(([-2, -2, -2],), [(-2, 3)]),
  |·(([1, 2, 1],), [(1, 1), (2, 1), (1,
1)])]
  |for args, expected in cases:
  |····assert solve(*args) == expected
  |print('OK: задача 2')
```

После успешных проверок: OK: задача 2

Почему это работает

В примере первые два отсчёта накапливают серию (4, 2). Первая единица закрывает её, затем ещё две единицы увеличивают счётчик до трёх. Следующая четвёрка не объединяется с первой серией: между ними уже есть законченная серия единиц. При переходе к нулю записывается (4, 1), а после цикла явно добавляется последняя пара (0, 2).

Перед каждой итерацией result содержит все завершённые серии обработанного префикса. Переменные current и count описывают единственную его последнюю серию, которую ещё нельзя закрыть: следующий элемент способен её продолжить. Равное значение увеличивает только count. Неравное значение доказывает, что прежняя серия закончилась, и начинает новую длины один.

Каждый входной элемент относится ровно к одной серии, потому что границу проводим тогда и только тогда, когда соседние значения различаются. После заключительного append не остаётся незаписанного хвоста. Отдельная проверка пустого входа нужна не ради скорости: без неё нельзя прочитать values[0] и корректно создать первую серию. Код обходит индексы и не делает копию хвоста списка через values[1:].

Время и память

n — число отсчётов, r — число серий. Время O(n), дополнительная память O(1) без результата и O(r) с ним. Оценка считает сравнение целых чисел операцией O(1); хеш-таблицы не используются.

Типичная ошибка

Подсчитать общую частоту каждого значения: тогда две разные серии четвёрок превратятся в одну, и восстановить порядок отсчётов будет нельзя. Другая ошибка — забыть дописать хвост.

Измените условие

Как проверить, что сжатие не потеряло отсчёты?

Разбор вопроса

Разверните каждую пару (value, count) в count копий value и сравните со входом. Дополнительно сумма длин должна равняться n, а соседние пары результата должны иметь разные значения. Одного равенства суммарной длины для проверки порядка мало.

Задача 3. Самая ранняя подходящая пара

В журнале корректировок записаны целые суммы; требуется найти две записи с заданным итогом. Функция solve(values, target) принимает список целых чисел и целое target. Верните пару индексов (i, j), где 0 <= i < j < len(values) и сумма равна target, либо None, если пары нет. Неоднозначность разрешается строго: сначала минимизируйте j, а среди пар с этим j — i. Индексы нулевые, одинаковые значения на разных позициях разрешены; использовать одну позицию дважды нельзя. Вход не меняйте и не сортируйте.

Пример

Аргументы solve: ([4, 4, 9, 6, 1], 10)
Ожидаемый ответ: (0, 3)

Подсказка

Идите слева направо по кандидату на второй индекс. Храните для каждого уже увиденного значения только самый первый индекс. Проверять дополнение надо до добавления текущей позиции.

Решение

```
def solve(values, target):
    first = {}
    for j, value in enumerate(values):
        needed = target - value
        if needed in first:
            return (first[needed], j)
        if value not in first:
            first[value] = j
    return None
```

Проверки для запуска

```
cases = [(([4, 4, 9, 6, 1], 10), (0, 3)),
 (([], 0), None),
 (([5], 10), None),
 (([5, 5, 5], 10), (0, 1)),
 (([1, 4, 3, 6], 7), (1, 2)),
 (([-3, 8, 2, 5], 5), (0, 1))]
for args, expected in cases:
    assert solve(*args) == expected
print('OK: задача 3')
```

После успешных проверок: OK: задача 3

Почему это работает

Для примера словарь после первой четвёрки содержит 4: 0. Вторая четвёрка не находит дополнение 6 и не заменяет индекс на единицу. Девятке нужна единица, которой ещё не было. У шестёрки с индексом 3 дополнение 4 уже есть: возвращается (0, 3). Пара (2, 4) тоже имеет сумму 10, но проигрывает по второму индексу; пара (1, 3) проигрывает по первому.

Перед обработкой j словарь first хранит минимальный индекс каждого значения среди позиций строго левее j. Поэтому любая найденная в нём позиция отличается от текущей и образует допустимую пару. Условное добавление нового значения сохраняет минимальный индекс: более поздние повторы не должны затирать раннюю запись. Проверка до добавления также защищает случай target == 2 * value от использования одной записи дважды.

Алгоритм рассматривает вторые индексы по возрастанию и сразу останавливается при успехе. Следовательно, раньше подходящей пары быть не могло: иначе остановка уже произошла бы. Для фиксированного j нужно ровно одно значение target - value, а словарь выдаёт его самый ранний индекс. Это доказывает оба уровня правила выбора. Если обход кончился без ответа, для каждого возможного второго индекса проверены все подходящие первые позиции через их значе-

ния.

Время и память

n — длина списка, u — число разных просмотренных значений. В среднем O(n) времени при O(1) арифметике целых и хеш-доступе, O(u) дополнительной памяти. При патологических коллизиях худшее время $O(n^2)$; стоимость больших целых учитывается отдельно.

Типичная ошибка

Безусловно присваивать first[value] = j: повтор перезапишет первый индекс и нарушит правило выбора. Сортировка также не сохраняет нужный порядок вторых индексов исходного журнала.

Измените условие

Совпадает ли это правило с выбором лексикографически минимальной пары (i, j)?

Разбор вопроса

Нет. Для [1, 4, 3, 6] и цели 7 возвращается (1, 2), поскольк-

ку j = 2 встречается раньше j = 3. Лексикографическое сравнение сначала минимизирует i и выбрало бы (0, 3). Порядок критериев является частью задачи.

Задача 4. Сверка повторяющихся этикеток

Два склада передали списки строковых этикеток. Одна этикетка может обозначать несколько одинаковых единиц товара, поэтому повторы нельзя просто убрать. Функция solve(left, right) возвращает пересечение мультимножеств: каждая строка появляется столько раз, сколько составляет минимум её частот во входах. Порядок результата совпадает с порядком выбранных элементов left; для одной строки выбираются самые ранние доступные появления в left. Оба входа — списки строк, регистр и пробелы значимы, пустые списки допустимы. Ни один вход не изменяется.

Пример

Аргументы solve: (['b', 'a', 'b', 'c', 'a', 'b'], ['a', 'b', 'b', 'd'])
Ожидаемый ответ: ['b', 'a', 'b']

Подсказка

Превратите правый список в запас разрешений: сколько раз ещё можно взять каждую строку. Затем пройдите левый список в естественном порядке, расходуя по одному разре-

шению.

Решение

```
def solve(left, right):
    from collections import Counter

    remaining = Counter(right)
    result = []
    for item in left:
        if remaining[item] > 0:
            result.append(item)
            remaining[item] -= 1
    return result
```

Проверки для запуска

```
cases = [((['b', 'a', 'b', 'c', 'a',
'b'], ['a', 'b', 'b', 'd']),
  ['b', 'a', 'b']),
  (([], ['a']), []),
  ((['a', 'b'], []), []),
  ((['x', 'x', 'x'], ['x', 'x']), ['x',
'x']),
  ((['B', 'b', 'a'], ['a', 'b']), ['b',
```

```
'a'])]
    for args, expected in cases:
    ····assert solve(*args) == expected
    print('OK: задача 4')
```

После успешных проверок: OK: задача 4

Почему это работает

В примере правый склад даёт два разрешения для b и по одному для a и d. Первый b забирает одно, затем a расходует своё. Второй b расходует последнее разрешение b. Для c запаса нет; второй a и третий b тоже пропускаются. Неиспользованное разрешение d не создаёт элемент ответа, потому что в left нет соответствующей позиции. Получается [b, a, b].

Инвариант: remaining[x] равно частоте x в right минус число уже выбранных x из обработанного префикса left. Остаток не бывает отрицательным, поскольку уменьшение разрешено только после проверки на положительность. Одновременно result является подпоследовательностью просмотренной части left. Значит, ни допустимое количество, ни исходный порядок не могут нарушиться на очередном шаге.

Для каждой строки алгоритм берёт её появления, пока не закончится либо левый вход, либо запас правого. Число взятых равно минимуму частот, что и требуется от пересечения мультимножеств. Выбор ранних появлений следует из единственного левого обхода: при наличии разрешения от-

каза нет. Counter возвращает ноль для отсутствующего ключа, поэтому отдельная ветка для неизвестной этикетки не нужна. Чтение такого значения не расходует память на все строки из left.

Время и память

n = len(left), m = len(right), u — число разных строк right. В среднем O(n + m) хеш-операций, O(u) дополнительной памяти без ответа. Это средняя, не худшая оценка хеш-таблицы; при коллизиях возможно $O((n + m)^2)$ сравнений. Стоимость обработки символов длинных строк учитывается отдельно.

Типичная ошибка

Использовать пересечение set(left) & set(right): оно теряет кратность и не задаёт нужного порядка. Проверка item in right без уменьшения запаса, наоборот, пропустит слишком много копий.

Измените условие

Можно ли переставить left и right для экономии памяти?

Разбор вопроса

Не без дополнительной работы: частоты пересечения останутся теми же, но порядок ответа изменится. Наш контракт привязан именно к left. Более экономичная схема должна всё равно сохранить возможность пройти исходный left и выбрать его ранние допустимые появления.

Конец ознакомительного фрагмента.