УДК 004.438Python
ББК 32.973.22
И32

И32 Изадха Х., Бехзадидуст Р.

Решение трудных и увлекательных задач на Python / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2024. – 240 с.: ил.

ISBN 978-5-93700-280-8

Цель данной книги – укрепить навыки логического рассуждения и развить творческое мышление, представив и решив 90 не самых простых задач на Python. Задачи изложены доходчиво и сжато, снабжены с алгоритмами и комментариями, что помогает читателям следить за процессом их решения и понимать его суть.

Издание предназначено читателям с базовыми знаниями языка Python, которые стремятся вывести свои способности на новый уровень. Книга будет полезна студентам, преподавателям, разработчикам, а также участникам соревнований по программированию.

ISBN 978-3-03139-998-5 (англ.)
ISBN 978-5-93700-280-8 (рус.)

Оглавление

Предисловие

Программирование – это увлекательнейшая область человеческой деятельности, требующая творческого подхода, навыков решения задач и любознательности. Python – это популярный и универсальный язык программирования, широко используемый в различных областях: от науки о данных и машинного обучения до веб-разработки и научных вычислений. Простой синтаксис Python, обширная экосистема библиотек и динамичный характер делают его идеальным языком для решения сложных задач. Программирование заставляет людей мыслить логично, потому что процесс достижения результата должен быть точно сформулирован. Поэтому всякому программисту очень важно иметь книги, описывающие приемы решения сложных задач. С другой стороны, книги также необходимы для совершенствования навыков мышления и рассуждения в повседневной жизни и работе. Творческое мышление и логическое рассуждение имеют решающее значение для решения задач, и эта книга направлена на достижение двух общих целей:

1) совершенствование навыков мышления и рассуждения путем исследования и программирования сложных задач;
2) улучшение навыков программирования на Python путем постановки сложных задач и их последовательного решения.

Эта книга адресована всем желающим поднять на новый уровень свои навыки владения языком Python и решения сложных задач. Она будет полезна всем, кто владеет навыками программирования на Python независимо от их уровня, а также всем, кто желает освоить язык программирования достаточно хорошо, чтобы решать сложные задачи. В этой книге вы найдете многочисленные примеры решения сложных задач на Python с алгоритмами и примечаниями. Мы преследовали две основные цели, представляя 90 задач из различных областей и их решения. Каждая глава посвящена конкретному типу задач, что, как нам кажется, должно способствовать росту интереса у читателя. Эта книга разделена на семь глав. В первой главе даются самые основы программирования на Python, а в последующих главах рассматриваются конкретные типы задач. Например, в главе 2 разбираются математические задачи, в главе 3 – сложные числовые задачи, в главе 4 – задачи, связанные с обработкой строк, в главе 5 – игровые задачи, в главе 6 – счетные задачи и в главе 7 – разные задачи, не попавшие в предыдущие главы.

Эта книга рекомендуется студентам всех специальностей независимо от их уровня владения навыками программирования, а также преподавателям и вообще всем, желающим совершенствовать свои навыки программирования на Python. Также книга будет полезна студентам, готовящимся к участию в соревнованиях по программированию. Изучив темы, представленные в нашей книге, учащийся сможет решать сложные задачи на Python.

Тебриз, Иран *Хабиб Изадха, Рашид Бехзадидуст*

Об авторах

Доктор Хабиб Изадха (Dr. Habib Izadkhah) – доцент кафедры информатики Тебризского университета, Иран. Прежде чем уйти в академическую науку, он десять лет проработал инженером-программистом. В круг его исследовательских интересов входят: алгоритмы и графы, разработка программного обеспечения и биоинформатика. Совсем недавно он занимался разработкой и применением глубокого обучения для решения различных задач, связанных с интерпретацией медицинских изображений, распознаванием речи и текста и генеративными моделями. Участвовал в различных исследовательских проектах, был автором ряда научных статей на международных конференциях, семинарах и в журналах, а также написал пять книг, в том числе «Source Code Modularization: Theory and Techniques» (Springer) и «Deep Learning in Bioinformatics» (Elsevier).

Рашид Бехзадидуст (Rashid Behzadidoost) – кандидат наук, работает на кафедре информатики Тебризского университета, Иран. В настоящее время пишет докторскую диссертацию, специализируясь на искусственном интеллекте и обработке естественного языка. Рашид страстно увлечен программированием и любит решать сложные задачи. Он приобрел свои навыки за годы учебы, практики и преподавания. Читал несколько курсов по информатике, включая продвинутое программирование, микропроцессорную технику и структуры данных, в университете Тебриза.

Глава 1

Введение

В этой главе обсуждаются цели и применение данной книги.

1.1. Почему Python?

Python – это язык программирования высокого уровня, имеющий более простой синтаксис, по сравнению со многими другими языками программирования. Кроме того, Python – это универсальный, кросс-платформенный, многопарадигмальный и объектно ориентированный язык, поддерживающий динамические типы данных. Python – очень простой язык, его быстро освоит любой, имеющий хоть какие-то знания в области программирования. Основная причина такой простоты заключается в том, что инструкции языка Python подобны словам в английском языке, что значительно упрощает процесс обучения. Python имеет интерактивную оболочку, которая позволяет экспериментировать и тестировать команды. Кросс-платформенность – важное преимущество, позволяющее пользоваться языком в различных операционных системах, таких как Mac, Windows, Linux и даже iOS и Android. Одно из наиболее значительных преимуществ Python – обширная экосистема библиотек, охватывающая самые разные сферы практического применения. Фактически библиотека предоставляет множество готовых фрагментов кода, которые программисты могут использовать в своей работе. Например, чтобы подключить Python к базе данных, необязательно писать свой код, реализующий все тонкости подключения. Вместо этого можно воспользоваться готовой библиотекой. Все перечисленные особенности делают Python отличным языком для изучения. Кроме того, Python имеет множество практических применений, таких как веб-разработка, разработка игр, наука о данных и искусственный интеллект. Помимо простоты и универсальности, Python также известен среди разработчиков своим активным и доброжелательным сообществом. Будучи языком с открытым исходным кодом, Python сплотил вокруг себя огромное сообщество, способствующее его росту и развитию.

Это сообщество предоставляет множество ресурсов, включая документацию, форумы и учебные пособия, что упрощает обучение новичков и поиск решений опытным разработчикам. Кроме того, сообщество постоянно совершенствует Python, разрабатывая новые пакеты, библиотеки и фреймворки, гарантируя актуальность и востребованность языка в постоянно меняющемся технологическом ландшафте. Это доброжелательное и динамичное сообщество – еще одна причина, почему Python считается замечательным языком для изучения.

1.2. Без использования библиотек

Большинство книг по Python либо описывают синтаксис языка с базовыми примерами, либо фокусируются на пакетах, предназначенных для решения конкретных задач. Описание библиотек и предоставление простых упражнений часто весьма полезно для изучения Python и обретения навыков решения сложных задач.

Однако мы считаем, что такие подходы не годятся для профессиональных программистов, разрабатывающих сложные программы. Поэтому в этой книге применяется другой подход: обсуждаются задачи, не связанные с библиотеками (хотя в некоторых программах мы действительно будем использовать стандартные библиотеки), и представлены подробные пошаговые алгоритмы решения с подсказками и примерами кода с комментариями. Следуя этому подходу, мы надеемся привить программистам навыки решения сложных задач и тем самым улучшить их способности к программированию.

1.3. Развитие навыков программирования и творческого мышления при решении сложных задач

Решение сложных задач – эффективный способ улучшения навыков программирования и развития творческого мышления. Программисты должны сначала определить задачу, разработать решение, затем преобразовать его в алгоритм и, наконец, реализовать его на конкретном языке программирования. Решая различные сложные задачи, программисты учатся использовать систематический подход и улучшают свои навыки программирования. Целью этой книги является повышение эффективности мышления, развитие умения рассуждать, а также углубление понимания языка Python на примере 90 задач, решение которых подробно рассматривается в главах. Задачи сгруппированы по таким темам, как математика, числа, строки, игры, счет и др. Каждая глава содержит набор задач с примерами, подсказками и реализациями на Python. Задачи распределены

по темам следующим образом: математика – 28, числа – 21, строки – 13, игры – 14, счет – 8 и прочие задачи – 6.

1.4. Предварительные условия

В книге присутствуют только главы, посвященные решению сложных задач. Мы решили не включать в нее главу, посвященную основам Python, так как в интернете можно найти много хороших учебников, где эта тема рассматривается очень подробно. Поэтому единственное предварительное условие, предполагаемое этой книгой,– знание Python на самом базовом уровне.

1.5. Целевая аудитория

В этой книге представлены решения 90 задач, которые могут быть интересны широкому кругу людей, включая студентов, изучающих информатику и инженерные специальности, а также всех, кто хочет улучшить свои навыки владения языком Python. Представленные задачи охватывают различные области, что делает их пригодными для широкого круга практических применений. Более того, эта книга расширяет навыки мышления и рассуждения, независимо от используемого вами языка программирования. Она послужит ценным ресурсом для преподавателей Python в университетах или школах. Кроме того, разработчики программного обеспечения и участники соревнований могут воспользоваться этой книгой для совершенствования своих навыков программирования.

Глава 2

Математика

В этой главе рассматриваются 28 математических задач, которые сопровождаются примерами решения на Python. Вот эти задачи:

1. Вычисление порядкового номера в задаче Иосифа Флавия.
2. Подсчет количества путей к точке (0,0) на координатной сетке.
3. Создание отсортированного списка целых чисел для задачи выбора Брюсселя.
4. Поиск решения обратной гипотезы Коллатца.
5. Подсчет правильных прямых углов.
6. Ближайшее *s*-угольное число.
7. Поиск точки опоры физических весов.
8. Вычисление общего количества блоков, необходимых для построения пирамиды сфер.
9. Группировка одинаковых монет по некоторым условиям.
10. Поиск медианы по тройкам чисел.
11. Наименьшее число из семерок и нулей.
12. Преобразование математических выражений из постфиксной нотации в инфиксную и их оценка.
13. Достижение стабильного состояния в болгарском пасьянсе.
14. Вычисление площади прямоугольных башен Манхэттена на линии горизонта.
15. Разрезание прямоугольника на квадраты.
16. Удаление правильных прямых углов в двумерной сетке.
17. Вычисление значений в нижней грани гармонического треугольника Лейбница.
18. Достижение цели на основе расстояния Коллатца.
19. Поиск суммы двух квадратов для числа *n*.
20. Поиск трех чисел, сумма которых равна целевому числу *n*.
21. Определение совершенной степени.
22. Лунное умножение целых чисел.
23. *n*-й член последовательности Рекамана.

24. *n*-й член последовательности Ван Эка.
25. Поиск суммы чисел Фибоначчи на основе теоремы Цекендорфа.
26. Поиск *k*-го слова Фибоначчи.
27. Поиск прямой в двумерной сетке, пересекающей наибольшее количество точек.
28. Проверка сбалансированности центрифуги.

2.1. Задача Иосифа Флавия

Задача Иосифа Флавия – широко известная в информатике теоретическая задача. Суть ее заключается в следующем: в круг выстраивается *n* человек. Из стоящих выбирается *k*-й человек, который покидает круг. Далее покинувшие круг не участвуют в процессе подсчета, а последний оставшийся считается победителем. Цель состоит в том, чтобы разработать функцию, которая принимает заданные значения *n* и *k* и возвращает порядок, в котором игроки будут покидать круг. Здесь *n* представляет общее количество людей, а *k* – количество пропусков. Важно отметить, что *k* может быть меньше, равно или даже больше *n*.

Например, рассмотрим рис. 2.1, где в круг встали пять человек. Для $k = 2$ и $n = 5$ определите, в каком порядке они будут покидать круг. Пусть *ex* – это массив людей, покинувших круг. Поскольку $k = 2$, то сразу после начала игры выбирается x_2 и исключается из круга; *ex* = [2]. Далее выполняется *k* шагов по кругу и выбывает 2 + 2 = 4-й участник, т. е. x_4; *ex* = [2, 4]. И снова выполняется *k* шагов, в результате выбывает участник x_1; *ex* = [2, 4, 1]. Обратите внимание, что при пересечении конца массива происходит переход в его начало (как по кругу), поэтому на предыдущем шаге выбор пал на игрока x_1. Теперь в круге остаются x_3 и x_5. После выполнения *k* шагов из круга выбывает x_5, соответственно, *ex* = [2, 4, 1, 5]. Последний игрок, оставшийся в круге, – x_3, поэтому искомый порядок исключения из круга *ex* = [2, 4, 1, 5, 3]. В табл. 2.1 показаны некоторые ожидаемые результаты (содержимое массива *ex*) для разных значений *k* и *n*.

Таблица 2.1. Некоторые ожидаемые результаты для разных значений *k* и *n*

n, k	Ожидаемый результат
6, 2	[2, 4, 6, 3, 1, 5]
5, 2	[2, 4, 1, 5, 3]
8, 8	[8, 1, 3, 6, 5, 2, 7, 4]
3, 9	[3, 1, 2]
4, 3	[3, 2, 4, 1]

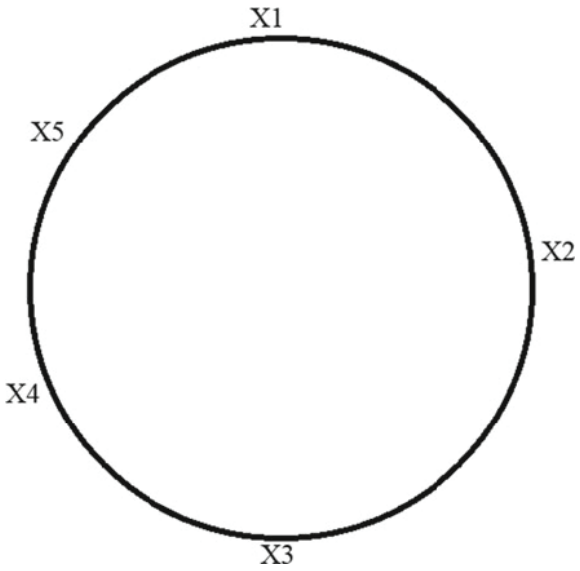

Рис. 2.1. Пример людей, вставших в круг

Алгоритм

Алгоритм принимает два аргумента: *n* – количество людей в круге и *k* – сколько людей нужно отсчитать, чтобы выбрать следующего для исключения из круга. В результате получается список чисел от 1 до *n*, представляющих людей в круге. Затем алгоритм последовательно исключает из круга каждого *k*-го человека, пока не останется только один, и возвращает список в том порядке, в котором люди исключались из круга. Чтобы гарантировать переход через правую границу массива к его началу, используется оператор деления по модулю. Вот подробное описание шагов алгоритма:

1. Принимаются два значения, `n` и `k`.
2. Создается список с именем `m`, содержащий все числа от `1` до `n`.
3. Создается пустой список с именем `ans` для хранения результата.
4. Устанавливается переменная `i = 0`.
5. Создается цикл `while`, который будет продолжать выполняться, пока список `m` не опустеет.
6. В цикле `while` обновляется индекс `i` по формуле 2.1: к значению `i` прибавляется `k` и вычитается `1`. Затем вычисляется остаток от деления индекса `i` на длину `m`, чтобы гарантировать, что `i` останется в пределах списка.
7. Вызовом метода `pop` из списка `m` исключается элемент с индексом `i` и добавляется в конец списка `ans`.

8. По завершении возвращается список `ans`.
9. В листинге 2.1 приводится код на Python, решающий задачу Иосифа Флавия.

$$(i + k - 1) \bmod m. \tag{2.1}$$

Листинг 2.1. Решение задачи Иосифа Флавия на Python

```
def josephus (n , k ) :
    '''
    Создать список чисел от 1 до n,
    представляющий людей в круге.
    '''
    m = l i s t ( range (1 , n + 1))

    '''
    Создать пустой список для хранения номеров людей
    в порядке их исключения из круга.
    '''
    ans = [ ]

    '''
    Инициализировать переменную i, хранящую
    текущую позицию в списке.
    '''
    i = 0

    '''
    Цикл, продолжающий исключать людей из круга,
    пока в нем никого не останется.
    '''
    while m:
        # Вычислить индекс следующего человека, исключаемого из круга.
        i = ( i + k - 1) % len (m)

        '''
        Исключить человека из списка
        и добавить его в список с результатами.
        '''
        ans.append(m.pop(i))

    # Вернуть результат.
    return ans
```

Функция `josephus` действует, как описано ниже.

1. Генерирует список `m`, включающий все числа от `1` до `n`.
2. Для хранения результата – списка людей, исключаемых из круга,– создает пустой список `ans`.
3. Создает переменную индекса `i` и присваивает ей начальное значение `0`.
4. Входит в цикл `while`, который продолжается до тех пор, пока из круга не будут исключены все игроки.
5. Внутри цикла функция вычисляет индекс следующего числа для исключения, прибавляя `k - 1` к текущему индексу `i` и определяя остаток от деления на длину оставшегося списка `m`.
6. Затем число, находящееся в списке `m` по вычисленному индексу, добавляется в конец списка `ans` и исключается из списка `m` с вызовом метода `pop`.
7. Наконец, функция возвращает список `ans`, с людьми, расположенными в порядке исключения из круга.

2.2. Подсчет количества путей к точке (0,0) на координатной сетке

В координатной сетке дана точка (*x, y*), представленная парой натуральных чисел, и нужно из нее добраться до точки (0,0). Задача состоит в том, чтобы подсчитать, сколькими возможными путями можно достичь начала координат (0,0), выполняя шаги влево или вниз. Напишите функцию, подсчитывающую наибольшее количество таких путей, не пересекающих точки с координатами в запретном списке. На входе даются *x* и *y* – координаты исходной точки, а *tabu* – список точек с запретными координатами. В табл. 2.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.2. Некоторые ожидаемые результаты для разных значений *x*, *y* и *tabu*

x, y, tabu	Ожидаемый результат
3, 2, []	10
1, 6, [(7, 1), (4, 4)]	7
8, 8, [(9, 10), (1, 4)]	11220
7, 5, [6,8]	792

Алгоритм

Для поиска решения используется следующий алгоритм.

1. Принимаются координаты (`row, col`) исходной точки и список `tabu`.
2. Создается двумерный массив `paths` с размером (*row* + 1) × (*col* + 1).
3. Элементу `paths [0] [0]` присваивается значение `1`.
4. Для всех `i` от `1` до `row`, если точка (`i,0`) не включена в `tabu`, присвоить элементу `paths [i] [0]` значение элемента `paths [i - 1] [0]`.
5. Для всех `j` от `1` до `col`, если точка (`0, j`) не включена в `tabu`, присвоить элементу `paths [0] [j]` значение `paths [0] [j - 1]`.
6. Для всех `i` от `1` до `row` и для всех `j` от `1` до `col`, если точка (`i, j`) не включена в `tabu`, присвоить элементу `paths [i] [j]` значение выражения `paths [i - 1] [j] + paths [i] [j - 1]`.
7. Вернуть `paths [row] [col]`.

В листинге 2.2 приводится код на Python, решающий задачу подсчета наибольшего числа переходов в точку (0,0).

Листинг 2.2. Решение задачи подсчета наибольшего числа переходов в точку (0,0)

```
def lattice_paths (row, col , tabu ) :
    '''Создать двумерный массив с размерами (row+1) x (col+1)
    и инициализировать все его элементы нулями
    '''
    paths = [ [0] * ( col+1) for _ in range(row+1)]

    '''Присвоить 1 элементу, соответствующему точке в левом нижнем
    углу, поскольку есть только один путь, чтобы достичь ее
    '''
    paths [0] [0] = 1

    '''Заполнить первый столбец в таблице,
    пропуская точки, присутствующие в списке 'tabu' '''
    for i in range(1 , row+1):
        if (i, 0) not in tabu:
            paths[i][0] = paths[i-1][0]

    '''Заполнить первую строку в таблице,
    пропуская точки, присутствующие в списке 'tabu'
    '''
    for j in range(1, col+1):
        if (0, j) not in tabu:
            paths[0][j] = paths[0][j-1]
```

```
24
25      # Заполнить остальные ячйки таблицы
26      for i in range(1, row+1):
27          for j in range(1, col+1):
28              # Пропустить ячейки в списке 'tabu'
29              if (i, j) not in tabu:
30                  '''Значение каждой незапрещенной ячейки
31                  определяется как сумма значений ячеек
32                  выше левее.'''
33                  paths[i][j] = paths[i-1][j] + paths[i][j-1]
34
35      return paths[row][col]
```

2.3. Создание отсортированного списка целых чисел для задачи выбора Брюсселя

В этой задаче необходимо написать функцию, которая принимает положительные целые числа *n*, *mink* и *maxk* и генерирует отсортированный список всех положительных целых чисел, образованных подмножеством цифр *m* исходного числа, которые либо в 2 раза больше *m*, либо в 2 раза меньше. Значения в получившемся списке должны следовать в порядке возрастания, при этом подмножество *m* должно попадать в диапазон значений от *mink* до *maxk*. В табл. 2.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.3. Некоторые ожидаемые результаты для разных входных значений в задаче выбора Брюсселя

n, mink, maxk	Ожидаемый результат
10, 2, 5	[5, 20]
9, 1, 4	18
47, 1, 1	[27, 87, 414]
100, 84, 99	[]

Алгоритм

Алгоритм решения задачи создает список, выполняя определенные операции с цифрами входного числа *n*. Указанные операции включают деление или умножение подмножества цифр на 2 в зависимости от того, представляет подмножество четное или нечетное целое число. Впоследствии полученный список чисел сортируется в порядке возрастания. Для поиска решения используется следующий алгоритм.

1. Алгоритм принимает три параметра: `n`, `mink` и `maxk`.
2. Создается пустой список с именем `result`.
3. Входное целое число `n` преобразуется в список его цифр с использованием генератора списков и сохраняется в переменной `digits`.
4. Далее выполняется цикл, перебирающий значения `k` в диапазоне от `mink` до `maxk+1`.
5. Для каждого значения `k` выполняется цикл по `i` от `0` до `len (digits)-k+1`.
6. Из списка `digits` извлекается срез от `i` до `i+k-1`, этот срез преобразуется в целое число, которое затем сохраняется в переменной `digit`.
7. Если `digit` – четное число, то оно делится на два, и результат сохраняется в переменной `half_digit`.
8. Создается новый список с именем `new_digits`, состоящий из первых `i` элементов списка `digits`, за которыми следуют цифры из `half_digit` и потом оставшиеся элементы из `digits`.
9. Список `new_digits` преобразуется в целое число и сохраняется в переменной `new_num`.
10. Число `new_num` добавляется в список `result`.
11. Значение `digit` умножается на `2`, и результат сохраняется в переменной с именем `nd`.
12. Создается новый список с именем `new_digits`, состоящий из первых `i` элементов списка `digits`, за которыми следуют цифры из `new_digit` и потом оставшиеся элементы из `digits`.
13. Список `new_digits` преобразуется в целое число и сохраняется в переменной `new_num`.
14. Значение `new_num` добавляется в конец списка `result`.
15. Список `result` сортируется в порядке возрастания с использованием алгоритма сортировки вставками.
16. И полученный отсортированный список `result` возвращается вызывающему коду.

В листинге 2.3 приводится код на Python, решающий задачу выбора Брюсселя.

Листинг 2.3. Решение задачи выбора Брюсселя

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        k = arr[i]
        j = i - 1
        while j >= 0 and k < arr[j]:
```

```
 6            arr[j+1] = arr[j]
 7            j -= 1
 8        arr[j + 1] = k
 9    return arr
10
11 def brussels_choice_problem(n, min_k, max_k) :
12    res = []
13    digits = [int(d) for d in str(n)]
14    for k in range(min_k, max_k + 1):
15        for i in range(len(digits) - k + 1):
16            d = int(''.join(str(d)
17                    for d in digits[i:i + k]))
18            if d % 2 == 0:
19                hd = d // 2
20                new_digits = digits[:i] + \
21                [int(d) for d in str(hd)] + digits[i + k:]
22                new_num = int(''.join(str(d)
23                              for d in new_digits))
24                res.append(new_num)
25            nd = d * 2
26            new_digits = digits[:i] + \
27            [int(d) for d in str(nd)] + digits[i + k:]
28            new_num = int(''.join(str(d)
29                          for d in new_digits))
30            res.append(new_num)
31    res = insertion_sort(res)
32    return res
```

2.4. Поиск решения обратной гипотезы Коллатца

Гипотеза Коллатца – это математическое утверждение, согласно которому любое целое число меньше 2^{68}, удовлетворяющее двум условиям, указанным в формуле 2.2, в конечном итоге достигнет числа 1 после конечного числа шагов. Гипотеза касается последовательностей и может быть выражена следующим образом: берется натуральное число *n,* и из него генерируются два числа в соответствии с формулой 2.2. Этот процесс повторяется для всех последующих чисел, пока последовательность не закончится единицей.

Ваша задача: написать функцию, которая вычисляет решение обратной гипотезы Коллатца. Если вычисление дает неверный результат (т.е. нецелое число), то функция должна вернуть значение `None`. Обратная гипотеза

Коллатца удваивает четные числа, а из нечетных чисел вычитает 1 и делит разность на 3. Формально гипотеза Коллатца представлена в формуле 2.2, а обратная гипотеза Коллатца – в формуле 2.3. В табл. 2.4 оказаны ожидаемые результаты для некоторых входных данных.

Таблица 2.4. Некоторые ожидаемые результаты для разных входных значений в обратной гипотезе Коллатца

shape	Ожидаемый результат
'dd'	4
'uudduuudd'	None
'ududududdddduddddd'	15
'uduuddduddduu'	None

Например, если входная строка *shape* = *ududddd*, то на первом этапе выбирается последний элемент в *shape* – символ *d* и применяется соответствующий ему первый случай в формуле 2.3, т. е. выполняется умножение на 2, в результате получается $x = x \times 2$ (первоначально $x = 1$). Далее, продолжая просматривать *shape* с конца, обнаруживаем три следующих подряд символа *d*, поэтому *x* обновляется следующим образом: $x = 2 \rightarrow 2 \times 2 = 4$, $x = 4 \rightarrow 4 \times 2 = 8$, $x = 8 \rightarrow 8 \times 2 = 16$. На следующем шаге обнаруживается символ *u*, поэтому применяется второй случай в формуле 2.3 и из числа *x* вычитается 1, а полученная разность делится на 3, соответственно, $x = 16 \rightarrow (16-1)/3 = 5$, далее снова следует символ *d*, поэтому $x = 5 \rightarrow 5 \times 2 = 10$. Последним следует символ *u*, соответственно, $x = 10 \rightarrow (10 \times 1)/3 = 3$. То есть решением обратной гипотезы Коллатца для *shape* = *ududddd* является число 3. Чтобы проверить правильность полученного значения, строку, что была передана в обратную гипотезу Коллатца, нужно сравнить со строкой, получаемой в ходе решения прямой гипотезы Коллатца, согласно формуле 2.2, которая применяется, пока *x* не достигнет 1. Итак, возьмем предыдущий полученный результат $x = 3$ и пустую строку *t*. На первом шаге имеем: $num = x$, $(num \bmod 2) = 1$, поэтому $num = (3 * num + 1) \rightarrow num = (3 * 3 + 1) = 10$ и $t = u$. Продолжим, $num = 10$, $10 \bmod 2 = 0$, поэтому $10/2 = 5$ и $t = ud$. На следующем шаге $num = 5$, и мы все еще не достигли единицы, поэтому $num = 5 * 3 + 1 = 16$ и $t = udu$. Далее выполняется такая последовательность шагов:

$$num = 16 \bmod 2 = 0 \rightarrow num = num/2 = 8 \rightarrow t = udud,$$
$$num = 8 \bmod 2 = 0 \rightarrow num = num/2 = 4 \rightarrow t = ududd,$$
$$num = 4 \bmod 2 = 0 \rightarrow num = num/2 = 2 \rightarrow t = ududd,$$
$$num = 2 \bmod 2 = 0 \rightarrow num = num/2 = 1 \rightarrow t = ududddd.$$

$$x = \begin{cases} x/2 & if\ x\%2 == 0 \\ 3x+1 & if\ x\%2 == 1 \end{cases}. \tag{2.2}$$

$$x = \begin{cases} 2x & if\ x\%2 == 0 \\ (x-1)/3 & if\ x\%2 == 1 \end{cases}. \tag{2.3}$$

Алгоритм

Алгоритм решения этой задачи заключается в преобразовании входной строки, состоящей только из символов `'u'` и `'d'`, в числовое значение, согласно обратной гипотезе Коллатца. Алгоритм инициализирует `x` числом `1.0` и приступает к обработке каждого символа во входной строке справа налево. Если текущий символ равен `'d'`, то алгоритм удваивает `x`. И наоборот, если текущий символ равен `'u'`, то уменьшает `x`, согласно обратной гипотезе Коллатца. Если предыдущее значение `x` не является целым числом, то алгоритм немедленно возвращает `None`. Как только все символы будут обработаны, алгоритм убеждается, что значение `x` не равно нулю. Далее алгоритм вычисляет ожидаемый результат, применяя гипотезу Коллатца к сгенерированному числу, и сравнивает его с исходной входной строкой. Если строки совпадают, то алгоритм возвращает вычисленное целое число. В противном случае он возвращает `None`. В листинге 2.4 приводится код на Python, решающий обратную гипотезу Коллатца.

Листинг 2.4. Поиск решения обратной гипотезы Коллатца

```
def check_if_integer(number):
    '''
    Проверить, является ли number целым числом,
    путем определения остатка от деления на 1
    '''
     if number % 1 == 0:
        return True
     else:
         return False

def pop_last_item(input_list):
     # Извлекает и удаляет последний символ из input_list
     list_length = len(input_list)
     last_item = input_list[list_length-1]
     del input_list[list_length-1]
     return last_item

def Inverse_collatz_conjecture(shape):
```

```
    '''
    Преобразовать входную строку в список символов
    и инициализировать x значением 1.0
    '''
    shape_list = list(shape)
    x = 1.0

    # Цикл по символам в списке
    while shape_list:
        item = pop_last_item(shape_list)
        if item == 'd':
            '''Удвоить значение x, если текущий
            символ 'd'
            '''
            x *= 2
        elif item == 'u':
            '''Уменьшить значение x, согласно
            обратной гипотезе Коллатца, если
            текущий символ 'u'
            '''
            prev = (x - 1) / 3
            is_integer = check_if_integer(prev)

            '''Если prev имеет целочисленное значение,
            то присвоить его переменной x
            '''
            if is_integer:
                x = prev
            else:
                # Если значение не является целым числом, то вернуть None
                return None

    # Если x равно 0 или входная строка пустая
    if x == 0 or not shape:
        return None

    true_answer = ''
    num = x

    '''
    Вычисление правильного ответа
    согласно прямой гипотезе Коллатца
    '''
```

```
    while num != 1:
        if num % 2 == 0:
            true_answer += 'd'
            num /= 2
        elif num % 2 == 1:
            true_answer += 'u '
            num = 3 * num + 1

    # Сравнить вычисленный и ожидаемый ответы
    if true_answer == shape:
        return int(x)
    else:
        return None
```

2.5. Подсчет правильных прямых углов

В двумерной сетке с целочисленными координатами узлов правильный прямой угол определяется тремя точками (*x, y*), (*x, y*+*h*) и (*x*+*h, y*) для некоторого значения *h* больше 0. Эти точки образуют фигуру, напоминающую столярный угольник или шеврон, направленный углом влево и вниз, причем точка (*x, y*) соответствует острию угла, а (*x, y*+*h*) и (*x*+*h, y*) определяют концы крыльев одинаковой длины и параллельные осям координат. Напишите функцию, которая принимает список точек, отсортированных по их координатам *x*, и возвращает количество правильных прямых углов. В табл. 2.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.5. Некоторые ожидаемые результаты для разных входных значений в задаче подсчета правильных прямых углов

n, mink, maxk	Ожидаемый результат
[(1, 1), (3, 5), (5, 2)]	0
[(0, 4), (0, 16), (2, 2), (2, 5), (5, 2), (9, 13)]	1
[(1, 3), (1, 7), (5, 3), (5, 5), (7, 3)]	2

Алгоритм

Алгоритм подсчета правильных прямых углов отыскивает все комбинации из трех точек в списке, которые образуют угол в форме столярного угольника или шеврона, как описано в постановке задачи. Для поиска решения используется следующий алгоритм.

1. Переменная `counter` инициализируется значением `0`.
2. Выполняется обход списка точек во вложенном цикле.

3. Для каждой точки проверяется, присутствует ли в списке другая точка с большей координатой `x` и такой же координатой `y`.
4. Если такая точка существует, то проверяется, присутствует ли в списке третья точка, необходимая для формирования угла (согласно определению задачи).
5. Если третья точка существует, то переменная `counter` увеличивается на `1`.
6. По завершении вернуть значение `counter`.

В листинге 2.5 приводится код на Python, решающий задачу подсчета правильных прямых углов.

Листинг 2.5. Подсчет количества углов

```
def counting_possible_corners(points):
    counter = 0
    for x, y in points:
        for x2, y2 in points:
            if x2 > x and y2 == y and \
                (x + y2 - y, y + x2 - x) in points:
                counter += 1
    return counter
```

2.6. Ближайшее *s*-угольное число

Пусть $s > 2$ – положительное целое число, определяющее бесконечную последовательность *s*-угольных чисел (их еще называют фигурными числами), где *i*-й элемент представлен формулой 2.4. Напишите функцию, которая принимает число сторон (или углов, если хотите) *s* и положительное целое число *n* и возвращает ближайшее *s*-угольное число. Если будет найдено два *s*-угольных числа, то функция должна вернуть наименьшее из них. В табл. 2.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.6. Некоторые ожидаемые результаты для разных входных значений в задаче поиска ближайшего *s*-угольного числа

n, sides	Ожидаемый результат
7, 8	8
1, 19	1
15, 18	18
87, 36	105

Алгоритм

Чтобы найти фигурное число с указанным числом сторон, ближайшее к заданному целому числу `n`, алгоритм выполняет вычисления в три этапа. Сначала он вычисляет фигурное число для конкретного индекса `i` по формуле 2.4. Затем использует двоичный поиск для определения индекса, соответствующего фигурному числу, ближайшему к `n`. И наконец, для трех индексов, окружающих средний индекс и полученных посредством двоичного поиска, вычисляет абсолютную разность между фигурными числами и `n` и возвращает фигурное число, ближайшее к `n`.

$$\frac{\left((s-2)\times i^2\right)-\left((s-4)\times i\right)}{2}. \tag{2.4}$$

В листинге 2.6 приводится код на Python, решающий задачу поиска ближайшего *s*-угольного числа.

Листинг 2.6. Поиск ближайшего *s*-угольного числа

```
def nearest_polygonal_number (n, sides ):
    # Вычисление фигурных чисел
    def calculate_polygonal_number(index):
       return((sides - 2) *
              index ** 2 - (sides - 4) * index) // 2

    # Определить верхнюю границу для двоичного поиска
    upper_bound = 1
    while calculate_polygonal_number(upper_bound) <= n:
        upper_bound *= 2

    # Выполнить двоичный поиск среднего индекса
    lower_bound, upper_bound = 0, upper_bound
    while lower_bound < upper_bound:
        middle_index = (lower_bound + upper_bound) // 2
        if calculate_polygonal_number (middle_index) < n:
            lower_bound = middle_index + 1
        else:
            upper_bound = middle_index

    '''
    В Python inf - это значение с плавающей
    точкой, представляющее положительную
    бесконечность. Это особое значение,
    представляющее любое число, которое больше
    любого другого конечного значения,
```

```
    включая целые числа и
    числа с плавающей точкой.
    '''
    closest_distance = float('inf')
    '''
    Вычислить абсолютную разность и найти
    ближайшее фигурное число
    '''
    nearest_polygonal = None
    for i in [-1,0, 1]:
        polygonal = \
            calculate_polygonal_number(lower_bound + i)
        distance = abs( polygonal - n)
        if distance < closest_distance:
            closest_distance = distance
            nearest_polygonal = polygonal

    # Вернуть ближайшее фигурное число
    return nearest_polygonal
```

2.7. Поиск точки опоры физических весов

Под «точкой опоры» подразумевается точка равновесия в списке весов, где общий вес на левой стороне равен общему весу на правой стороне. Эта задача требует определить позицию в непустом списке числовых значений, которая может служить опорой и сбалансировать вес обеих сторон. Согласно принципам физики, перекладина весов достигает равновесия, когда силы, действующие на оба ее конца, равны. Ваша задача: написать функцию, которая принимает список чисел и возвращает положение точки опоры, уравновешивающей веса. Если такая позиция не существует, то функция должна вернуть –1.

В табл. 2.7 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.7. Некоторые ожидаемые результаты для разных входных значений в задаче поиска точки опоры

Веса	Ожидаемый результат
[6, 6, 9]	–1
[43, 51, 35, 4]	1
[19, 25, 5, 42, 38, 8, 34, 16, 14, 8, 47, 42, 4, 20, 23]	7
[7, 24, 3, 38]	2

Алгоритм

Чтобы найти положение точки опоры в непустом списке числовых значений, уравновешивающей веса по обе стороны от нее, для каждого индекса в списке алгоритм определяет совокупный вес по обе стороны и возвращает индекс, в котором точка опоры сохраняет баланс (т. е. когда сумма весов по левую сторону равна сумме весов по правую сторону). Если такой позиции не существует, алгоритм возвращает –1.

Ниже подробно описаны шаги, выполняемые алгоритмом.

1. Получает список чисел `weights`.
2. Использует цикл `for` с функцией `range` для перебора индексов во входном списке.
3. Для каждого индекса `i` вычисляется сумма весов слева от точки опоры (сумма значений элементов с индексами меньше `i`), дополнительно при подсчете суммы значение каждого элемента умножается на его расстояние от точки опоры (т. е. `i - j`). Аналогично определяется сумма весов справа от точки опоры (сумма значений элементов с индексами больше `i`).
4. Если суммы весов слева и справа равны (т. е. точка опоры сбалансирована), то возвращается текущий индекс `i`.
5. Если позиция, удовлетворяющая условиям задачи, не найдена, то возвращается –1.

Следуя этим шагам, алгоритм может определить положение точки опоры, уравновешивающей веса во входном списке.

В листинге 2.7 приводится код на Python, решающий задачу поиска точки опоры.

Листинг 2.7. Поиск точки опоры физических весов

```
def find_fulcrum_position(weights):
    for i in range(len(weights)):
        left_weight_sum = sum(weights[j] * (i - j)
            for j in range(i))
        right_weight_sum = sum(weights[j] * (j - i)
            for j in range(i + 1, len(weights)))
        if left_weight_sum == right_weight_sum:
            return i
    return -1
```