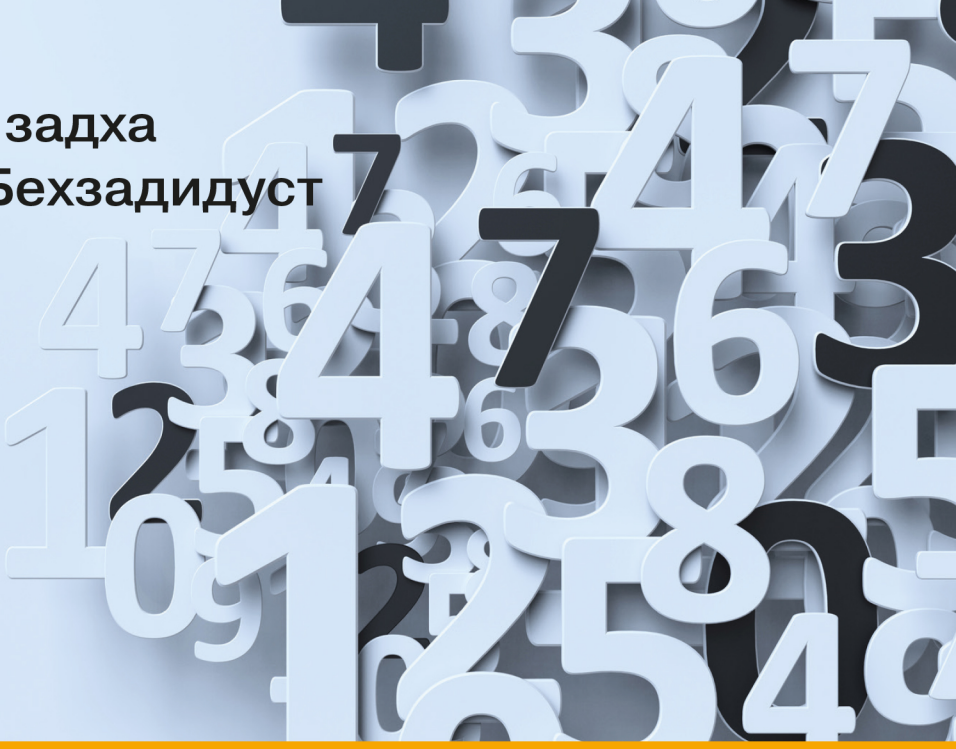
Хабиб Изадха
Рашид Бехзадидуст



Решение трудных и увлекательных задач на Python

ДМК
издательство

Хабиб Изадха, Рашид Бехзадидуст

Решение трудных и увлекательных задач на Python

Challenging Programming in Python: A Problem Solving Perspective

Habib Izadkhah · Rashid Behzadidoost

Springer

Решение трудных и увлекательных задач на Python

Хабиб Изадха, Рашид Бехзадидуст

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2024

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 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]

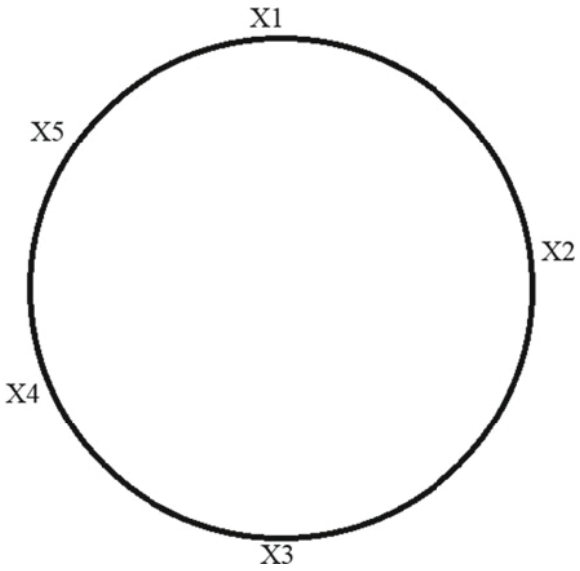


Рис. 2.1. Пример людей, вставших в круг

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
```

2.8. Вычисление общего количества блоков, необходимых для построения пирамиды из сфер

Дана пирамида, построенная из сфер. Основание пирамиды имеет размер $n \times m$, а каждый последующий уровень имеет ширину и длину на 1 больше по сравнению с предыдущим уровнем. Ваша задача: написать функцию, которая принимает размеры n и m, а также высоту h и вычисляет общее количество сфер, необходимых для построения такой пирамиды.

В табл. 2.8 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.8. Некоторые ожидаемые результаты для разных входных значений в задаче вычисления общего количества блоков, необходимых для построения пирамиды из сфер

n, m, h	Ожидаемый результат
4, 2, 2	23
28, 30, 3	2699
3, 3, 3	50
2, 7, 9	654

Алгоритм

Для вычисления общего количества необходимых сфер алгоритм использует набор формул, учитывающих ширину и длину каждого уровня пирамиды, а также высоту пирамиды. Ниже подробно описаны шаги алгоритма.

1. Из начальной высоты `h` вычитается `1`. Самый нижний слой считается первым слоем, а самый верхний – `(h + 1)`-м слоем. Однако формула, используемая в коде, предполагает, что самый нижний слой – это `0`-й слой, а самый верхний – это `h`-й слой. Поэтому чтобы высота `h` соответствовала используемой формуле, нужно вычесть `1` из входной высоты `h` (т. е. `h + 1-1`, или `h`).
2. Общее количество блоков для данного уровня вычисляется с использованием следующей формулы:

```
a = m * n * (h + 1).
```

3. К общей сумме прибавляется количество блоков, необходимых для каждого уровня пирамиды, согласно формуле

```
a += (m + n) * (h * (h + 1)//2),
```

сумма первых `h` треугольных чисел, умноженная на сумму `m` и `n`.

4. Наконец, прибавляются дополнительные блоки, необходимые для создания каждого слоя пирамиды, по следующей формуле:

```
a += (h * (h + 1) * (2 * h + 1))//6.
```

5. В завершение возвращается полученное значение `a`. Оно равно сумме первых `h` квадратных чисел, деленной на `6`.

В листинге 2.8 приводится код на Python, решающий задачу вычисления общего количества блоков, необходимых для построения пирамиды из сфер.

Листинг 2.8. Вычисление общего количества блоков, необходимых для построения пирамиды из сфер

```
def Counting_sphere_pyramid_blocks(n, m, h ):
    h -= 1
    '''
    Формула вычисления общего количества
    блоков, необходимых для построения
    пирамиды с основанием
    n на m и высотой h.
    '''

    a = m * n * (h + 1)
    a += (m + n) * (h * (h + 1) // 2)
    a += (h * (h + 1) * (2 * h + 1)) // 6
    return a
```

2.9. Группировка монет

В этой задаче дан массив из n одинаковых монет, который необходимо разделить на g групп, где g – положительное целое число, большее или равное единице. Если после формирования полных групп остаются какие-либо монеты, то оставшиеся монеты в массиве откладываются и сохраняются. Для вычисления количества монет в каждой группе используется формула 2.5. Процесс группировки повторяется, пока все монеты не будут учтены.

$$n//g * c. \tag{3.5}$$

Ваша задача – написать функцию, которая принимает три входных параметра: n (общее количество монет), g (количество групп) и c (количество монет в одной группе) – и возвращает массив, содержащий количество монет, оставшихся на каждом этапе группировки. Если $n = 0$, то функция должна вернуть пустой список. Например, пусть $n = 10$, $g = 3$ и $c = 2$. Следую-

щая последовательность показывает оставшиеся монеты на каждом этапе группировки: 10, 3, 2 → 1; 6, 3, 2 → 0; 4, 3, 2 → 1; 2, 3, 2 → 2; 0, 3, 2. Следовательно, [1, 0, 1, 2] представляет оставшиеся монеты на каждом шаге для $n = 10$, $g = 3$ и $c = 2$. На первом шаге в этой последовательности 1 получается из ($10 \bmod 3 = 1$). На втором шаге 6 получается из $10//3 \times 2 = 6$, а 3 и 2 остаются постоянными на протяжении всего процесса.

В табл. 2.9 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.9. Некоторые ожидаемые результаты для разных входных значений в задаче группировки монет

n, g, c	Ожидаемый результат
255, 128, 70	[127, 70]
301, 10, 1	[1, 0, 3]
49, 49, 2	[0, 2]
10, 2, 1	[0, 1, 0, 1]

Алгоритм

Для разделения монет на группы алгоритм использует рекурсию, которая выполняется до тех пор, пока не перестанут формироваться дальнейшие группы. Ниже подробно описаны шаги алгоритма.

1. Входной параметр `n` сравнивается с нулем. Если он равен нулю, то возвращается пустой список, так как нет монет для разделения на группы.
2. Подсчитывается количество монет, которые не могут составить полную группу, путем вычисления остатка от деления `n` на `g`. Результат сохраняется в переменной `remain`.
3. Вычисляется общее количество монет, необходимое для формирования всех полных групп, путем умножения числа полных групп (`n//g`) на размер каждой группы (`c`). Это значение сохраняется в переменной `coins_needed`.
4. Затем функция вызывает себя рекурсивно с входными параметрами `coin_needed`, `g` и `c`, пока не останется монет для группировки.
5. По завершении возвращается массив, содержащий количество оставшихся монет на каждом этапе группировки.
6. В листинге 2.9 приводится код на Python, решающий задачу группировки монет.

Листинг 2.9. Решение задачи группировки монет

```
def Grouping_Coins(n, g, c):
    if n == 0:
        return []
    # Сколько монет не уместилось в следующую группу
    remain = n % g

    '''
    (1) n // g:
    обозначает количество групп, которые можно сформировать.
    (2) n // g * c:
    обозначает количество монет, из которых можно сформировать эти группы.
    '''
    coins_needed = (n // g) * c
    remaining_coins = Grouping_Coins(coins_needed, g, c)
    return [remain] + remaining_coins
```

2.10. Поиск медианы по тройкам чисел

Эта задача требует многократного поиска медианы в списке положительных целых чисел, пока не останется максимум три числа. В частности, для каждой группы из трех чисел необходимо вычислить их медиану и использовать ее в качестве входного аргумента для следующей итерации. Этот процесс продолжается до тех пор, пока не останется не более трех чисел. Их медиана возвращается как окончательный результат.

Ваша задача: написать функцию Python, которая принимает список положительных целых чисел и возвращает медиану последней оставшейся группы чисел.

В табл. 2.10 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.10. Некоторые ожидаемые результаты для задачи вычисления медианы в тройке медиан

Список	Ожидаемый результат
[909, 4, 4, 7, 9, 12, 77, 45]	9
[1, 4, 90, 65, 3, 2]	4
[22, 40, 65, 80, 93, 21]	80
[100, 9, 12, 20, 3]	12

Алгоритм

Для вычисления медианы медиан алгоритм итеративно вычисляет медианы в сегментах списка, содержащих по три элемента, пока не останется только три числа. Процесс включает деление входного списка на группы по три элемента, вычисление их медиан и добавление их в список медиан. Затем функция рекурсивно вызывает себя со списком медиан, пока не останется не более трех элементов. Потом функция сортирует их в порядке возрастания и возвращает среднее значение в качестве результата.

1. Определяется длина входного списка.
2. Если длина равна 1, то единственный элемент такого списка возвращается как медиана.
3. В противном случае создается пустой список `medians` для хранения медиан.
4. Входной список делится на подсписки по три элемента в каждом с использованием цикла `for` и функции `range` с размером шага `3`. Каждый подсписок сортируется в порядке возрастания, и из него извлекается средний элемент в качестве медианы.
5. Каждая медиана добавляется в список `medians`.
6. Предыдущий шаг выполняется рекурсивно, пока не останется не более трех элементов.
7. Когда останется не более трех элементов, они сортируются в порядке возрастания, и извлекается средний элемент, который и будет конечной медианой.
8. Конечная медиана возвращается в качестве результата.

В листинге 2.10 приводится код на Python, решающий задачу поиска медианы по тройкам чисел.

Листинг 2.10. Решение задачи группировки монет

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

def median_of_medians(items):
```

```
    # Получить длину списка
    n = len(items)

    '''
    Если в списке только один элемент,
    вернуть его как медиану
    '''
    if n == 1:
        return items[0]
    else:
        # Создать пустой список для хранения медиан
        medians = []

        '''
        Разбить список на подсписки
        по три элемента, отсортировать их и
        взять средние элементы как медианы
        '''
        for i in range(0, n, 3):
            sublist = items[i:i + 3]
            if len(sublist) < 3:
                median = sublist[len(sublist) // 2]
            else:
                sorted_sublist = insertion_sort(sublist)
                median = sorted_sublist[1]
            medians.append(median)

        '''
        Рекурсивно вызывать функцию со
        списком medians, пока
        не останется не более трех элементов
        '''
        if len(medians) > 3:
            return median_of_medians(medians)
        else:
            '''
            Если осталось три элемента или меньше,
            то отсортировать их и взять средний
            элемент, который и будет конечной медианой
            '''
            sorted_items = insertion_sort(medians)
            return sorted_items[len(sorted_items) // 2]
```

2.11. Наименьшее число из семерок и нулей

В западной культуре число семь считается символом удачи, а число ноль, напротив, считается нежелательным. Числа, составленные из семерок и нулей, такие как 77777777777777777 или 77700, часто называют «семьноль». Одна замечательная теорема доказывает, что для любого положительного целого числа существует множество целых чисел, составленных из семерок и нулей, которые делятся на *n* без остатка.

Ваша задача: написать функцию, которая отыскивает наименьшее положительное целое число, состоящее из семерок и нулей, которое без остатка делится на заданное число *n*. При этом число, состоящее из одних нулей, не считается желаемым результатом.

В табл. 2.11 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.11. Некоторые ожидаемые результаты для задачи поиска наименьшего числа из семерок и нулей

n	Ожидаемый результат
25	700
49	777777
1023	77777777777777777777777777777
104	777777000

Алгоритм

Чтобы определить наименьшее целое число, состоящее только из 7 или из комбинации семерок и нулей, которое без остатка делится на заданное положительное целое число, используется алгоритм, который выполняет итерации по возрастающим степеням числа 10, пока не будет найдено число, кратное n, состоящее только из семерок (например, 777777) или семерок и нулей (например, 777777000). Начальное пространство поиска находится в диапазоне от 1 до 10.

В каждой итерации к произведению предыдущего остатка на 10 прибавляется 7. Если остаток числа появляется более одного раза, это указывает на то, что данная последовательность цифр из семерок и нулей представляет число, кратное n, которое начинается с 7 и заканчивается на 0. Если достигнут остаток 0, то это говорит о том, что число, начинающееся с цифры 7 и кратное числу n, найдено.

Если в текущем диапазоне поиска решение не обнаружено, то алгоритм расширяет диапазон поиска и продолжает итерации. Ниже подробно описаны шаги алгоритма.

1. Создается словарь с именем `remainder_positions` для отслеживания остатков и их позиций в каждой итерации. Создается переменная `current_remainder` с начальным значением `0`.
2. Переменные `digits_min` и `digits_max` инициализируются начальными значениями `1` и `10` соответственно. Они определяют пространство поиска числа из семерок и нулей, кратного `n`, и переменная `num_digits` изменяется в диапазоне от `digits_min` до `digits_max`.
3. Затем запускается цикл `while`, который выполняется до тех пор, пока не будет найдено число из семерок и нулей, кратное `n`.
4. В цикле `while` переменная `num_digits` последовательно получает значение от `digits_min` до `digits_max` включительно.
5. Для каждого значения `num_digits` вычисляется остаток по формуле

   `(current_remainder * 10 + 7)% n`.

6. Если текущий остаток равен нулю, возвращается целое число, состоящее из последовательности семерок.
7. Если текущий остаток уже появлялся раньше, то из предыдущего и текущего блоков формируется число, кратное числу `n`, которое начинается с `7` и заканчивается `0`. Для этого отыскивается позиция `i` предыдущего вхождения `current_remainder` в словаре `remainder_positions` и целое число, состоящее из последовательности `7`, за которой следуют `0` с использованием выражения

   `'7' * (num_digits - i) + '0' * i`.

8. Если текущий остаток ранее не появлялся, он добавляется в словарь `remainder_positions` со своей позицией `num_digits`.
9. После завершения всех итераций в текущем пространстве поиска оно расширяется путем установки `digits_min` равным `digits_max` и умножения `digits_max` на `10`.
10. Шаги 4–9 повторяются до тех пор, пока не будет найдено число из семерок и нулей, кратное заданному числу `n`. Если число из семерок и нулей не отыщется, то итерации будут продолжаться бесконечно.

В листинге 2.11 приводится код на Python, решающий задачу поиска числа из семерок и нулей, кратного заданному числу `n`.

Листинг 2.10. Поиск наименьшего числа из семерок и нулей

```
1 def Smallest_Seven_Zero(n):
2     # Словарь для хранения остатков и их позиций
```

```
    remainder_positions = {}
    current_remainder = 0

    # Определить начальное пространство поиска
    digits_min = 1
    digits_max = 10

    '''
    Выполнять итерации по степеням 10,
    пока не будет найдено число, кратное n
    '''
    while True:
        for num_digits in range(digits_min, digits_max ):
            current_remainder = \
                (current_remainder * 10 + 7) % n
            if current_remainder == 0:
                return int('7' * num_digits)
            elif current_remainder in remainder_positions:
                # объединить предыдущий блок с текущим
                i = remainder_positions[current_remainder]
                return int('7' * (num_digits - i) + '0' * i)
            else:
                remainder_positions[current_remainder ] =\
                    num_digits

        # Расширить пространство поиска, если необходимо
        digits_min = digits_max
        digits_max *= 10
```

2.12. Оценка математических выражений в постфиксной нотации

Целью этой задачи являются преобразование математических выражений из постфиксной нотации в инфиксную и их оценка. В инфиксной нотации операторы в математическом выражении записываются между парами операндов, например *a* + *b*. В постфиксной нотации оператор следует за парой операндов, например *a b* +. Соответственно, выражение 2 7 + 3 *, записанное в постфиксной нотации, эквивалентно выражению (2 + 7) * 3 в инфиксной нотации, которое дает в результате число 27.

Ваша задача: написать функцию, возвращающую числовой результат, но не эквивалент в инфиксной нотации.

В табл. 2.12 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.12. Некоторые ожидаемые результаты для задачи оценки математических выражений, записанных в постфиксной нотации

Выражение	Ожидаемый результат
[5, 6, '+', 7, '*']	77
[3, 7, 9, '*', '+']	66
[3, 7, 9, '/', '+']	3
[8, 2, '+']	10

Алгоритм

Алгоритм оценки выражения в постфиксной нотации выполняет следующие шаги.

1. Выбирает каждый элемент из заданного выражения в постфиксной нотации.
2. Если элемент является целым числом, то он добавляется в список `Storage`.
3. Если элемент является оператором (+, –, *, /), то из списка `Storage` извлекаются два последних элемента (операнда).
4. И затем с этой парой операндов выполняется соответствующая операция (+, –, *, /). Результат операции сохраняется в переменной `result` и затем добавляется в список `Storage`.
5. Шаги 3–4 повторяются до тех пор, пока не будут обработаны все элементы в выражении.
6. В завершение возвращается результат – последний элемент в списке `Storage`.

В листинге 2.12 приводится код на Python, решающий задачу оценки математических выражений в постфиксной нотации.

Листинг 2.12. Оценка математических выражений в постфиксной нотации

```
def pop_last_item(input_list):
    # Извлекает последний элемент из входного списка и возвращает его
    list_length = len(input_list)
    last_item = input_list[list_length - 1]
    del input_list[list_length - 1]
```

```
    return last_item
def postfix_evaluate(Postfix):
    '''
    Создать пустой список для
    хранения операндов и результатов
    '''
    Storage = []

    '''
    Выполнить обход элементов в списке Postfix
    с выражением, записанным в постфиксной нотации
    '''
    for item in Postfix:
        '''
        Если это целое число,
        то добавить его в конец списка Storage
        '''
        if isinstance(item, int):
            Storage.append(item)
        else:
            '''
            Если это оператор,
            то извлечь два последних операнда из списка Storage
            '''
            Number0, Number1 = \
                pop_last_item(Storage), pop_last_item(Storage)
            '''
            Выполнить соответствующую операцию
            с двумя операндами
            '''
            if item == '-':
                result = Number1 - Number0
            elif item == '+':
                result = Number1 + Number0
            elif item == '*':
                result = Number1 * Number0
            elif item == '/':
                result = \
                    Number1 // Number0 if Number0 != 0 else 0
            Storage.append(result)

    return pop_last_item(Storage)
```

2.13. Достижение стабильного состояния в болгарском пасьянсе

В этой задаче дается список из n карт, представленных положительными целыми числами. Цель состоит в том, чтобы продолжать генерировать новые списки чисел, отличающиеся от исходного, пока не будет достигнуто стабильное состояние. Под стабильным состоянием понимается ситуация, когда новый список содержит те же элементы, что и предыдущий список, но сами элементы могут быть упорядочены иначе.

При генерировании новых списков применяются два правила:

1) сумма чисел в списке должна быть равна треугольному числу (треугольные числа определяются согласно формуле 2.6);

$$\frac{k \times (k+1)}{2}; \tag{2.6}$$

2) при создании нового списка на каждом этапе из всех чисел вычитается одна единица, а последним элементом списка становится длина предыдущего списка. Если результат вычитания равен нулю, он удаляется.

Этот процесс повторяется до тех пор, пока не будет достигнуто стабильное состояние. Таким образом задача требует найти список, удовлетворяющий обоим условиям. Этот процесс называется болгарским пасьянсом. Например, подсчитаем количество шагов для достижения стабильного состояния для [3] – заданного списка чисел и $k = 2$. Сначала необходимо проверить, равно ли треугольное число сумме элементов данного списка:

$$k \times (k + 1)/2 \rightarrow 2 \times (2 + 1)/2 = 3, \text{ т.е. } 3 = 3.$$

Следующие шаги показывают, как достигается стабильное состояние:

$$[3] \rightarrow [2, 1] \rightarrow [1, 2, 0] \rightarrow [1, 2].$$

Поскольку элементы [2, 1] и [1, 2] равны, можно считать, что стабильное состояние достигнуто, даже притом что элементы в списках [2,1] и [1, 2] упорядочены по-разному. Поэтому стабильное состояние достигается уже после одного шага.

Ваша задача: написать функцию, которая принимает список из n положительных целых чисел и треугольное число и возвращает количество шагов, необходимых для достижения стабильного состояния.

В табл. 2.13 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.13. Некоторые ожидаемые результаты для задачи подсчета шагов, необходимых для достижения стабильного состояния

Points, k	Ожидаемый результат
[3], 2	1
[3, 7, 8, 14], 4	0
[10, 10, 10, 10, 10, 5],10	74
[3000, 2050], 100	7325

Один из важнейших шагов в решении этой задачи – правильное определение момента стабилизации. Для этого нужно отсортировать текущий список и сравнить его с предыдущим. Если два списка совпадают, то это указывает, что стабильное состояние достигнуто. Однако, несмотря на кажущуюся простоту, этот подход может быть очень сложным.

Для определения состояния (стабильного или нет) используются два алгоритма: `is_stable_state` и `Stable_State_in_Bulgarian_Solitaire`.

Алгоритм

Алгоритм `Stable_State_in_Bulgarian_Solitaire` выполняет следующие шаги.

1. Принимает два аргумента: `points` (список положительных целых чисел) и `k` (целое число). Инициализирует переменную `number_of _moves` значением `0`.
2. Вычисляет сумму чисел во входном списке по формуле

   ```
   k * (k + 1)//2
   ```

   и сохраняет ее в переменной `equation`.
3. Вызывает алгоритм `is_stable_state`, который принимает два аргумента: `points` (список положительных целых чисел) и `k` (целое число).
4. Сравнивает сумму чисел в списке `points` со значением `equation`. Если они не равны, то возвращает ноль, чтобы сообщить, что стабильное состояние пока не достигнуто. Пока стабильное состояние не будет достигнуто, выполняется следующий цикл:
   - из каждого элемента в списке `points` вычитается `1`;
   - из списка `points` удаляются все нулевые значения;
   - в конец списка `points` добавляется элемент с длиной текущего списка точек;
   - переменная `number_of _moves` увеличивается на 1;
   - проверяется стабильность текущего состояния с использованием алгоритма `is_stable_state`.
5. Возвращает конечное значение `number_of _moves`.

6. Алгоритм `is_stable_state` выполняет следующие шаги.
7. Принимает два аргумента: `points` (список целых положительных чисел) и `k` (целое число).
8. Создает массив `count` с `k+1` нулями.
9. Перебирает в цикле значения `p` в списке `points` и увеличивает соответствующий элемент в `count`, если `p` меньше или равно `k`.
10. Перебирает элементы в массиве `count` и сравнивает каждый с `1`. Если встречается элемент, не равный `1`, то возвращается `False`, указывающее, что состояние нестабильно.
11. Если цикл завершается без возврата `False`, то возвращается `True`, указывающее, что состояние стабильно.

В листинге 2.13 приводится код на Python, определяющий состояние в болгарском пасьянсе.

Листинг 2.13. Определение состояния в болгарском пасьянсе

```
def Stable_State_in_Bulgarian_Solitaire(points, k):
    # Инициализировать число ходов нулем
    number_of_moves = 0

    '''
    Формула вычисления
    суммы очков
    '''
    equation = k * (k + 1) // 2

    def is_stable_state(points, k):
        """
        Для отслеживания частоты каждого целого числа
        между 1 и k в списке points
        используется массив счетчиков count. Если какое-то
        целое число встречается несколько раз или вообще не встречается,
        то это значит, что стабильное состояние пока не достигнуто
        и поэтому функция должна вернуть False.
        И напротив, если каждое число от 1 до k
        встречается ровно один раз, то функция возвращает True,
        указывая, что стабильное состояние достигнуто.
        """
        count = [0] * (k + 1)
        for p in points:
            if p <= k:
                count [p] += 1
```

```
        for i in range(1, k+1):
            if count[i] != 1:
                return False
        return True

    '''
    Проверить соответствие входного списка points
    условиям болгарского пасьянса
    '''
    if sum(points) == equation:
    # Выполнять цикл до достижения стабильного состояния
    while is_stable_state(points, k) != True:
        # Уменьшить на 1 величину каждого элемента в points
        for i in range(len(points)):
            points[i] -= 1

        points_Length = len(points)
        # Удалить из points элементы, равные нулю
        points = [p for p in points if p > 0]

        '''
        Добавить новый элемент в points
        с размером списка после удаления нулевых элементов
        '''
        points.append(points_Length)

        # Увеличить счетчик ходов
        number_of_moves += 1

    # Вернуть общее число ходов
    return number_of_moves
```

2.14. Вычисление площади прямоугольных башен Манхэттена на линии горизонта

Есть список прямоугольных башен в виде (*s*, *e*, *h*), где *s*, *e* и *h* – это начало и конец на оси X и высота соответственно. Ваша задача: написать функцию, которая принимает кортеж (*s*, *e*, *h*) и возвращает площадь прямоугольных башен. Обратите внимание, что общая площадь башен не должна вычисляться дважды.

В табл. 2.14 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.14. Некоторые ожидаемые результаты для задачи вычисления площади прямоугольных башен Манхэттена на линии горизонта

Towers	Ожидаемый результат
[(2, 6, 98), (1, 0, 9)]	383
[(3, 7, 1)]	4
[(–8, 6, 3), (6, 14, 11), (0, 4,–5)]	130
[(2, 550, 222), (1, 0, 4)]	121652

Алгоритм

Для вычисления площади данных башен используются два алгоритма: `divide` и `Manhattan_Skyline`. Алгоритм `divide` выполняет следующие шаги.

1. Принимает два параметра: `start` и `end`. Если `start` равно `end`, то возвращается список, содержащий два элемента: `[towers [start] [0], towers [start] [2]]` и `[towers [start] [1],0]`.
2. В противном случае вычисляется средняя точка между началом и концом и алгоритм рекурсивно вызывается для левой и правой половин списка башен. Затем результаты для левой и правой половин объединяются в новый список. На этом шаге выполняются следующие действия:
   - вычисляется средняя позиция;
   - создается пустой список `merged` и инициализируются нулем две переменные `i` и `j`;
   - инициализируются значением `None` две переменные `h1` и `h2`;
   - в цикле выполняется обход элементов списка `merged` по диапазону `len (left) + len (right)`;
   - если значение `i` больше или равно длине списка `left`, то в `merged` добавляется список, содержащий `[right [j] [0], right [j] [1] if h1 is None else max (h1, right [j] [1])]`, затем `j` увеличивается на `1` и цикл продолжается;
   - если значение `j` больше или равно длине списка `right`, то в `merged` добавляется список, содержащий `[left [i] [0], left [i] [1] if h2 is None else max (h2, left [i] [1])]`, затем `i` увеличивается на `1` и цикл продолжается;
   - если координата X `i`-го элемента в `left` меньше или равна координате X `j`-го элемента, то в `merged` добавляется список, содержащий `[left [i] [0], left [i] [1] if h2 is None else max (h2, left [i] [1])]`, затем переменной `h1` присваивается координата Y `i`-го элемента в `left` и `i` увеличивается на `1`;

- в противном случае в `merged` добавляется список `[right [j] [0], right [j] [1] if h1 is None else max (h1, right [j] [1])]`, затем переменной `h2` присваивается координата Y `j`-го элемента в `right` и `j` увеличивается на `1`.

3. В завершение возвращается объединенный список `merged`.

Алгоритм `Manhattan_Skyline` выполняет следующие шаги.

1. Принимает список башен `towers`.
2. Вычисляет площадь, ограниченную линией горизонта, с использованием `divide`.
3. Возвращает вычисленную площадь.

В листинге 2.14 приводится код на Python, вычисляющий площади прямоугольных башен Манхэттена на линии горизонта.

Листинг 2.14. Вычисление площади прямоугольных башен Манхэттена на линии горизонта

```
def rectangular_towers_in_manhattan_skyline(towers):

    ''' divide_and_conquer вычисляет
    линию горизонта для данного диапазона башен
    '''
    def divide_and_conquer(start, end):
        '''
        Если в заданном диапазоне имеется только одна башня,
        то вернуть координаты X ее начала и конца
        '''
        if start == end:
            return [[towers[start][0],
                     towers[start][2]],
                    [towers[start][1],0]]

        '''
        Разделить диапазон на две половины и
        рекурсивно вычислить линии горизонта для каждой половины
        '''
        mid = (start + end) // 2
        left = divide_and_conquer(start, mid)
        right = divide_and_conquer(mid + 1, end)

        '''
        Объединить две линии горизонта,
```

```
        используя слияние с сортировкой
        '''
        merged = []
        i, j = 0, 0
        # Высоты левой и правой башен
        h1, h2 = None, None

        for x in range(len(left) + len(right)):

            if i >= len(left):
                merged.append([right[j][0], right[j][1]
                    if h1 is None else max(h1, right[j][1])])
                j += 1

            elif j >= len(right):
                merged.append([left[i][0],left[i][1]
                    if h2 is None else max(h2, left[i][1])])
                i += 1
                '''
                Если следующий элемент в left
                совпадает со следующим
                элементом в right, то добавить
                элемент из left и обновить его высоту
                '''

            elif left[i][0] <= right[j][0]:
                merged.append([left[i][0], left[i][1]
                    if h2 is None else max(h2, left[i][1])])
                h1 = left[i][1]
                i += 1
                '''Если следующий элемент в right
                совпадает со следующим элементом в left
                то добавить элемент из right
                и обновить его высоту
                '''
            else:
                merged.append([right[j][0], right[j][1]
                    if h1 is None else max(h1, right[j][1])])
                h2 = right[j][1]
                j += 1

        # Вернуть список merged
        return merged
```

```
69
70      # Получить длину списка towers
71      n = len(towers)
72
73      # Вычислить линию горизонта для всего диапазона towers
74      result = divide_and_conquer(0, n - 1)
75
76      '''
77      Вычислить площадь башен вдоль линии горизонта,
78      используя правило трапеций
79      '''
80      area = sum((result[i + 1][0] - result[i][0]) *
81                 result[i][1] for i in range(len(result) - 1))
82
83      # Вернуть вычисленную площадь
84      return area
```

2.15. Разрезание прямоугольника на квадраты

Дан кортеж (*a*, *b*), представляющий длину и ширину прямоугольника соответственно. Ваша задача: написать функцию, которая принимает кортеж (*a*, *b*) и возвращает минимальное количество резов, которые необходимо сделать, чтобы из прямоугольника получить несколько квадратов.

В табл. 2.15 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.15. Некоторые ожидаемые результаты для задачи разрезания прямоугольника на квадраты

a, b	Ожидаемый результат
(7, 7)	0
(17, 10)	6
(5, 3)	3
(7, 9)	5

Алгоритм

Лучше всего для решения этой задачи подходит рекурсивная функция. Цель алгоритма состоит в том, чтобы определить минимальное количество резов, необходимых для разрезания заданного прямоугольника на квадраты. Для достижения этой цели алгоритм начинается с проверки, является ли данный прямоугольник квадратом. Если заданный прямоугольник уже является квадратом, то алгоритм возвращает `0`. Однако если

прямоугольник не является квадратом, то алгоритм рекурсивно делит его на меньшие прямоугольники-квадраты, пока не достигнет желаемого результата. В этом процессе используется мемоизация для хранения ранее вычисленных результатов, чтобы избежать избыточных вычислений. После преобразования всех прямоугольников в квадраты алгоритм сравнивает результаты, полученные при разрезании конечного квадрата как по горизонтали, так и по вертикали. Затем выбирает решение с минимальным количеством резов, запоминает результат и возвращает его.

Алгоритм выполняет следующие шаги.

1. Принимает `a` (длина) и `b` (ширина).
2. Создается словарь `memo` для хранения значений.
3. Проверяется равенство `a` и `b`. Если они равны, то возвращается `0`, поскольку прямоугольник уже является квадратом и резать его не требуется.
4. Проверяется равенство `a` или `b` значению `1`. Если `a` или `b` равно `1`, то возвращается максимальное значение из `a` и `b` минус `1`, поскольку в этом случае может быть только один квадрат.
5. Создается ключ в форме кортежа `(a, b)`, и проверяется, присутствует ли этот ключ в словаре `memo`. Если ключ существует, то возвращается его значение.
6. Переменная `answer` инициализируется значением `(a - 1) * b`.
7. Выполняется цикл (с переменной цикла `i`) по диапазону значений от `1` до `a//2 + 1`.
8. Выбирается минимальное значение из `answer` и `(a - i, b, memo)` плюс `(i, b, memo)` плюс `1`.
9. Выполняется цикл (с переменной цикла `i`) по диапазону значений от `1` до `b//2 + 1`.
10. Выбирается минимальное значение из `answer` и `(a, b - i, memo)` плюс `(a, i, memo)` плюс `1`.
11. В словарь `memo` добавляется пара ключ-значение.
12. Возвращается ответ.

В листинге 2.15 приводится код на Python, вычисляющий минимальное количество резов прямоугольника для получения квадратов.

Листинг 2.15. Вычисление минимального количества резов прямоугольника для получения квадратов

```
1 def RectToSquares(a , b, memo={}):
2     '''
3     Проверить, является ли данный прямоугольник
```

```
    квадратом
    '''
    if a == b:
        return 0

    if a == 1 or b == 1:
        return (max(a, b) - 1)

    key = (a , b)
    '''Проверить, был ли ранее вычислен
    ответ для этой комбинации входных данных
    '''
    if key in memo:
        return memo[key]



    '''
    Инициализировать answer
    максимальным значением

    Представляет максимальное
    число резов, необходимых для
    разрезания прямоугольника на квадраты.
    '''
    answer = (a - 1) * b

    # Разрезать прямоугольник по горизонтали
    for i in range(1, a // 2 + 1):
        '''
        Рекурсивно вычислить
        минимальное число резов
        для получения двух меньших прямоугольников
        '''
        # Выполнить горизонтальный рез
        horizontal_cut = RectToSquares(a - i, b, memo)
        vertical_cut = RectToSquares(i, b, memo)
        # Прибавить 1, чтобы учесть начальный рез
        total_cuts = horizontal_cut + vertical_cut + 1
        # Обновить answer, если найдено лучшее решение
        answer = min(answer, total_cuts)

    # Разрезать прямоугольник по вертикали
```

```
47      for i in range(1, b // 2 + 1):
48          '''
34          Рекурсивно вычислить
35          минимальное число резов
36          для получения двух меньших прямоугольников
52          '''
53          # Выполнить вертикальный рез
54          horizontal_cut = RectToSquares(a, i, memo)
55          vertical_cut = RectToSquares(a, b - i, memo)
56          # Прибавить 1, чтобы учесть начальный рез
57          total_cuts = horizontal_cut + vertical_cut + 1
58          # Обновить answer, если найдено лучшее решение
59          answer = min(answer , total_cuts )
60
61      # Запомнить answer
62      memo[ key ] = answer
63      return answer
```

2.16. Удаление правильных прямых углов в двумерной сетке

Для данного набора точек в двумерной сетке под «углом» понимаются три точки в форме (x, y), $(x, y + h)$ и $(x + h, y)$ для некоторого значения $h > 0$, представляющие острие угла с двумя крыльями. Ваша задача: написать функцию, которая принимает список точек, упорядоченных по углам, и возвращает минимальное количество точек, которые необходимо удалить из списка, чтобы исключить все правильные прямые углы.

В табл. 2.16 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.16. Некоторые ожидаемые результаты для задачи удаления правильных прямых углов в двумерной сетке

Points	Ожидаемый результат
[(3, 3), (3, 8), (8, 3)]	1
[(0, 1), (4, 5), (3, 2)]	0
[(5, 0), (1, 3), (1, 4), (2, 0), (2, 2), (2, 3), (4, 0), (4, 0)]	2

Алгоритм

Чтобы удалить все углы, образованные тройками точек в заданном списке, путем удаления из списка минимального количества точек, алгоритм сначала выявляет все правильные прямые углы, которые образуют

тройки точек во входном списке, используя вложенные циклы. Затем он использует рекурсию с мемоизацией, чтобы определить минимальное количество точек, которые необходимо удалить из списка, дабы удалить все правильные прямые углы.

В листинге 2.16 приводится код на Python, вычисляющий минимальное количество точек, которые нужно удалить из входного списка, чтобы удалить все правильные прямые углы.

Листинг 2.16. Вычисление минимального количества точек, которые нужно удалить из входного списка, чтобы удалить все правильные прямые углы

```
'''
Рекурсивно удаляет минимальное количество точек для удаления углов
'''
def MinCornerRemover(corners, memo):
    '''
    Базовый случай: если список углов пуст
    и ничего не нужно удалять
    '''
    if len(corners) == 0:
        return 0

    '''
    Проверить, обрабатывался ли
    текущий список углов прежде
    '''
    key = tuple(corners)
    if key in memo:
        return memo[key]


    removals = len(corners)

    '''
    Инициализировать множество, в котором запоминаются
    уже обработанные точки
    '''
    points_set = set ( )
    # Цикл по углам в списке
    for corner in corners:
        '''
        Цикл по точкам
        текущего угла
```

```
        '''
        for point in corner:
            '''
            Если точка еще не обрабатывалась,
            то добавить ее в множество
            '''
            if point not in points_set:
                points_set.add(point)
                '''
                Вычислить углы, оставшиеся
                после исключения текущей точки
                '''
                remaining_corners = \
                    [c for c in corners if point not in c]
                '''
                Рекурсивно вычислить минимальное число
                удаляемых точек для удаления остальных углов
                '''
                '''и прибавить 1, чтобы учесть
                текущую удаленную точку
                '''
                removals = \
                    min(1 + MinCornerRemover \
                        (remaining_corners, memo), removals)

        '''
        Запомнить результат для текущего
        списка углов
        '''
        memo[key] = removals
        # Вернуть минимальное число удаленных точек
        return removals

'''
Удаляет все углы
из заданного списка точек
'''
# Принимает список кортежей, представляющих точки
def cut_corners(points):
    # Создать пустой список для хранения углов
    corners = [ ]
    # Цикл по точкам в списке
    for i in range(len(points)):
```

```
        x = points[i][0]
        y = points[i][1]
        '''
        Проверить, имеется ли другая точка
        с такой же координатой y
        '''
        # т. е. имеется ли другая точка правее текущей
        for j in range(i + 1, len(points)):
            if points[j][1] == y:
                h = points[j][0] - x
                '''Проверить, имеется ли точка на h единиц
                выше текущей
                '''
                '''и правее другой точки
                с той же координатой y
                '''
                if h > 0 and (x, y + h) in points:
                    '''
                    Добавить кортеж, представляющий угол
                    в список углов corners
                    '''
                    corners.append \
                        (((x, y), (x + h, y), (x, y + h)))

    '''
    Создать пустой словарь
    для хранения прежде вычисленных результатов
    '''
    memo = {}
    '''
    Вызвать функцию, которая рекурсивно удалит
    минимальное количество точек с использованием мемоизации
    '''
    return MinCornerRemover(corners, memo)
```

2.17. Треугольник Лейбница

В треугольнике Паскаля каждое число равно сумме двух чисел, находящихся непосредственно над ним, тогда как в треугольнике Лейбница каждое число равно сумме двух чисел под ним.

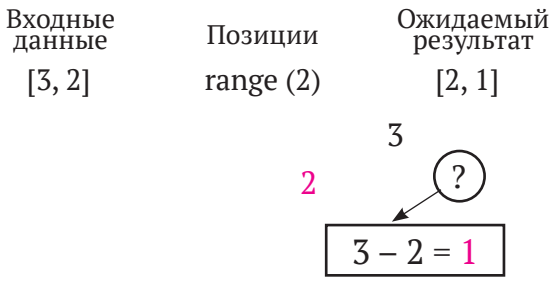
Ваша задача: написать функцию, которая принимает значения крайней левой грани треугольника и возвращает соответствующее значение нижнего ряда в указанной позиции.

В табл. 2.17 показаны ожидаемые результаты для некоторых входных данных.

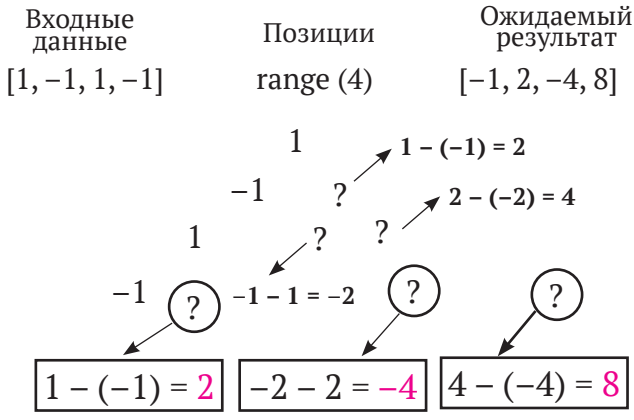
Таблица 2.17. Некоторые ожидаемые результаты для задачи вычисления значений в нижней грани гармонического треугольника Лейбница

LeftMost_Values, positions	Ожидаемый результат
[1, 2, 3], range (3)	[3,–1, 0]
[4, 7, 9, 3], range (4)	[3, 6,–8, 7]
[88, 90, 1, 0, 9], range (4)	[9,–9, 10, 78]
[20, 95], range (2)	[95,–75]

Например, на рис. 2.2 показаны два численных примера треугольника Лейбница. Входными данными для первого примера (A) являются 2 и 3, а позиция – *range* (2). Функция *range* возвращает последовательность чисел, начинающуюся с 0, поэтому нулевой и первой позициям отвечают 2 и 1 соответственно.



А) Числовой пример треугольника Лейбница



Б) Числовой пример треугольника Лейбница

Рис. 2.2. Два числовых примера треугольника Лейбница

Алгоритм

Для решения поставленной задачи воспользуемся алгоритмом, выполняющим следующие шаги.

1. Принимает два входных параметра: `LeftMost_Rows` – список значений, представляющих крайнюю левую грань треугольника Лейбница, и `positions` – список целых чисел, представляющих позиции в нижней грани, значения для которой нужно найти.
2. Создается пустой список `result` для хранения искомых значений.
3. Создается словарь `memo` для хранения промежуточных значений, вычисленных в ходе решения.
4. На основе представленных значений в левой грани вычисляются значения в первом столбце треугольника.
5. Полученные значения сохраняются в словаре `memo` с ключами, соответствующими индексам.
6. На основе значений из предыдущего ряда вычисляются оставшиеся значения в треугольнике.
7. Каждое значение в треугольнике вычисляется как разность между значением, стоящим выше текущего ряда, и значением в текущем ряду.
8. Полученные значения сохраняются в словаре `memo` с ключами, соответствующими индексам.
9. Значения, соответствующие нижней грани треугольника, хранящиеся в словаре, добавляются в список результатов `result`.
10. Полученный список возвращается.
11. В листинге 2.17 приводится код на Python, вычисляющий значения в нижней грани гармонического треугольника Лейбница.

Листинг 2.17. Вычисление значений в нижней грани гармонического треугольника Лейбница

```
 1 '''
 2 Эта функция принимает два аргумента:
 3 список значений, представляющих
 4 левую грань треугольника Лейбница,
 5 и список целых чисел, представляющих
 6 позиции в нижней грани тругольника,
 7 для которых нужно найти значения.
 8 '''
 9 def Leibniz_triangle(LeftMost_Rows, positions):
10     '''
```

```
    Создать пустой список для хранения
    искомых результатов
    '''
    result = []
    '''
    Создать словарь для хранения результатов
    промежуточных вычислений, выполняемых в ходе алгоритма
    '''
    memo = {}

    '''
    Вычислить значения в первом столбце треугольника,
    опираясь на представленные значения в левой грани.
    '''
    for i in range(len(LeftMost_Rows)):
        memo[(i, 0)] = LeftMost_Rows[i]

    '''
    На основе предыдущего столбца вычислить
    остальные значения в треугольнике.
    '''
    for i in range(1, len(LeftMost_Rows)):
        for j in range(1, i + 1):
            '''
            Каждое значение в треугольнике вычисляется
            как разность между вышестоящим и нижестоящим
            значениями
            '''
            value = memo[(i - 1, j - 1)] - memo[(i, j - 1)]
            memo[(i,j)] = value

    '''
    Извлечь значения для нижней грани
    треугольника из словаря
    memo и добавить их
    в список результатов result.
    '''
    for i in positions:
        result.append(memo[(len(LeftMost_Rows) - 1, i)])

    # Вернуть результаты.
    return result
```

2.18. Расстояние Коллатца

Имея заданное целое число *n*, требуется достичь некоторого положительного целого числа с применением формул $3 \times n + 1$ и $n//2$, взятых из гипотезы Коллатца. Пусть путь между начальным и конечным числами *start* и *goal* имеет несколько слоев, содержащих по нескольку положительных целых чисел. Первый слой инициализируется начальным числом *start*. Все положительные целые числа имеют одинаковое расстояние в каждом слое. Начиная с начального числа *start* для каждого следующего слоя генерируются числа однократным применением формул $3 \times n + 1$ и $n//2$ к каждому числу в предыдущем слое. Создание слоев продолжается, пока в каком-то из них не будет получено искомое число *goal*. Рассмотрим следующий пример. Пусть *start* = 10 и *goal* = 20. На каждом шаге генерируется несколько положительных целых чисел с одинаковым расстоянием, пока не будет получено число *goal*. Поскольку первый (будем считать его нулевым) слой инициализируется числом *start*, в нем имеется только число 10. Далее:

- слой 1: к числу 10 применяются формулы $3 \times n + 1$ и $n//2$, и получаются два числа $(3n + 1) = (3 \times 10 + 1) = 31$, $(n//2) = (10//2) = 5$;
- слой 2: к числам 31 и 5 применяются те же формулы, и получаются четыре числа $(3 \times 31 + 1) = 94$, $(3 \times 5 + 1) = 16$, $(31//2) = 15$, $(5//2) = 2$;
- слой 3: 283, 49, 46, 7, 47, 8, 7, 1;
- слой 4: 850, 148, 139, 22, 142, 25, 22, 4, 141, 24, 23, 3, 23, 4, 3, 0;
- слой 5: 2551, 445, 418, 67, 427, 76, 67, 13, 424, 73, 70, 10, 70, ...;
- слой 6: 7654, 1336, 1255, 202, 1282, 229, 202, 40, ...;
- слой 7: 22963, 4009, 3766, ..., **20**...

Как видите, в слое 7 обнаружилось целевое число **20.** Ваша задача: написать функцию, которая принимает начальное и конечное числа и возвращает номер слоя, в котором присутствует целевое число.

В табл. 2.18 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.18. Некоторые ожидаемые результаты для задачи поиска целевого числа на основе расстояния Коллатца

Start, goal	Ожидаемый результат
20, 321	9
22, 15	8
4, 7	2
10, 20	7

Алгоритм

Для достижения целевого числа используется метод поиска в ширину (Breadth-First Search, BFS). Алгоритм BFS сначала обходит все узлы на одном уровне и только потом переходит на следующий. В нашем случае каждый сгенерированный слой, возникающий в результате применения формул из гипотезы Коллатца, образует уровень, который нужно исследовать. Алгоритм начинает исследование с числа `start_num`, представляющего первый слой, и генерирует последующие слои, применяя одни и те же формулы. Вот подробное описание шагов алгоритма.

1. Принимаются два положительных целых числа `start_num` и `goal_num`.
2. Инициализируются переменные `num_layers` и `curr_layer`, где `num_layers = 0` и `curr_layer` – множество, содержащее только начальное число, т. е. `start_num`.
3. Пока `goal_num` не обнаружится в `curr_layer`, выполняются следующие действия:
   - создается пустое множество `next_layer` (поскольку повторяющиеся числа нам не нужны);
   - для каждого числа в `curr_layer` вычисляются значения по формулам `num//2` и `3 * num + 1`;
   - эти значения добавляются в `next_layer`;
   - значение `next_layer` присваивается переменной `curr_layer`;
   - `num_layers` увеличивается на 1.
4. Если `goal_num` присутствует в `curr_layer`, то возвращается `num_layers`.

В листинге 2.18 приводится код на Python, вычисляющий расстояние Коллатца.

Листинг 2.18. Вычисление расстояния Коллатца

```
def get_collatz_distance(start_num, goal_num):
    """
    Принимает начальное и целевое числа
    и генерирует все слои с положительными
    целыми числами, используя последовательность
    Коллатца для поиска кратчайшего пути от
    start_num до goal_num.
    """
    # Инициализировать счетчик слоев
    num_layers = 0
    '''
    Инициализировать текущий слой как
```

```
    множество с единственным числом start_num
    '''
    curr_layer = {start_num}
    '''
    Продолжать выполнять цикл, пока в текущем слое
    не обнаружится число goal_num
    '''
    while goal_num not in curr_layer:
        '''
        Создать пустое множество,
        представляющее следующий слой
        '''
        next_layer = set ( )

        for num in curr_layer:
            '''Сгенерировать следующие числа в
            последовательности Коллатца для каждого числа
            в текущем слое
            '''
            next_layer.update([
                num // 2,
                3 * num + 1
            ])
        '''
        Сделать текущим вновь
        сгенерированный слой
        '''
        curr_layer = next_layer
        '''
        Увеличить счетчик слоев
        '''
        num_layers += 1
    '''
    Вернуть количество слоев, которые потребовалось
    пересечь для достижения целевого числа
    '''
    return num_layers
```

2.19. Сумма двух квадратов

Дано положительное целое число *n*, и требуется найти два числа, сумма квадратов которых равна *n*. Например, если *n* равно 145, то ответом будет пара чисел (12, 1). Если есть несколько ответов, то следует выбрать ответ

с наибольшим максимальным числом. Например, для $n = 50$ есть два возможных ответа: (7, 1) и (5, 5). Поскольку 7 больше 5, правильный ответ: (7, 1). Ваша задача: написать функцию, которая принимает положительное целое число *n* и возвращает два числа, сумма квадратов которых равна *n*. Если таких чисел нет, функция должна вернуть *None*.

В табл. 2.19 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.19. Некоторые ожидаемые результаты для задачи суммы двух квадратов

n	Ожидаемый результат
50	(7,1)
145	(12, 1)
1480	(38, 6)
540	None

Алгоритм

Алгоритм находит основания второй степени, сначала выбирая максимальное значение для первого основания, а затем проверяя все значения-кандидаты для второго основания. Если значение-кандидат для второго основания дает пару оснований, удовлетворяющих условиям, то функция возвращает эту пару. Вот подробное описание шагов алгоритма.

1. Принимается целое положительное число `n`.
2. Вычисляется максимальное значение первого основания как квадратный корень из половины числа `n`.
3. Создается набор возможных значений для второго основания от `1` до значения первого основания.
4. Перебираются возможные значения второго основания.
5. Для каждого возможного значения второго основания вычисляется разность между `n` и квадратом второго основания.
6. Вычисляется квадратный корень разности.
7. Проверяется равенство квадрата корня разнице.
8. Если равно, то возвращается пара оснований, сумма квадратов которых равна `n`.
9. Если такой пары не существует, возвращается `None`.

В листинге 2.19 приводится код на Python, отыскивающий пару чисел, сумма квадратов которых равна заданному числу.

Листинг 2.19. Поиск пары чисел, сумма квадратов которых равна заданному числу

```
def get_squares_summing_to_n(n ) :
    """
    Эта функция возвращает кортеж с двумя
    целыми числами, сумма квадратов которых равна n.
    """
    # Максимальное значение первого основания
    max_first_base = int((n / 2) ** 0.5)
    # Множество значений-кандидатов для второго основания
    '''
    Второе основание не может быть больше первого,
    поэтому 1 + max_first_base
    '''
    candidates = set(range(1, 1 + max_first_base))
    # Цикл по значениям-кандидатам второго основания
    for second_base in candidates:
        '''
        Вычислить разность между n
        и квадратом второго основания
        '''
        difference = n - second_base ** 2
        # Вычислить квадратный корень разности
        root = int(difference ** 0.5)
        '''
        Сравнить квадрат корня
        с разностью
        '''
        if root ** 2 == difference:
            '''
            Вернуть пару оснований,
            сумма квадратов которых равна n
            '''
            return (root, second_base)
    # Если пара не найдена, вернуть None
    return None
```

2.20. Поиск трех чисел

Дан отсортированный список положительных целых чисел, в котором нужно отыскать три числа, сумма которых равна заданному.

Ваша задача: написать функцию, которая принимает отсортированный список положительных целых чисел и возвращает *True*, если в списке есть

три числа, сумма которых равна заданному числу *target*, в противном случае она должна возвращать *False*.

В табл. 2.20 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.20. Некоторые ожидаемые результаты для задачи поиска трех чисел

ListNumbers, target	Ожидаемый результат
[3, 5, 6, 8, 9, 21], 14	True
[2, 4, 8, 16, 32], 16	False
[550, 600, 2000, 3000, 4000], 900	False
[1, 2, 16, 79, 80, 340], 83	True

Алгоритм

В решении используются три вложенных цикла `for` для обхода всех чисел и сравнения их сумм с целевым числом `target`. Однако это простое решение имеет высокую временную сложность. Для достижения поставленной задачи алгоритм использует следующие шаги.

1. Принимается отсортированный список чисел и целевое число `target`.
2. Длина списка сравнивается с числом `3`. Если она меньше, то возвращается `False`.
3. Выполняется обход элементов списка.
4. Для каждого числа в списке вычисляется новое целевое число, вычитанием текущего числа из целевого.
5. На следующем этапе используется алгоритм поиска двух чисел. Алгоритм поиска двух чисел принимает список чисел и целевое число и возвращает `True`, если в списке есть два числа, сумма которых равна целевому числу. В противном случае возвращается `False`.
6. Алгоритм поиска двух чисел выполняет следующие шаги:
   а) длина списка сравнивается с числом `3`. Если она меньше, то возвращается `False`;
   б) инициализируются два указателя: один указывает на начало списка, а другой – на его конец;
   в) пока начальный указатель меньше конечного указателя:
      1) вычисляется сумма двух чисел в позициях начального и конечного указателей;
      2) если сумма равна целевому числу, вернуть `True`;
      3) если сумма меньше целевого числа, увеличивается начальный указатель, чтобы перейти к большему числу;

4) если сумма больше целевого числа, уменьшается конечный указатель, чтобы перейти к меньшему числу;

д) если совпадение не найдено, возвращается `False`.

7. Если такая пара существует, возвращается `True`.
8. Если в списке не найдено совпадений, возвращается `False`.

В листинге 2.20 приводится код на Python, отыскивающий три числа, сумма которых равна заданному числу.

Листинг 2.20. Поиск трех чисел, сумма которых равна заданному числу

```
def has_two_sum(numbers, target):
    '''
    Если длина списка
    меньше 2, то в нем
    невозможно найти два числа,
    сумма которых равна заданному числу
    '''
    if len(numbers) < 2:
        return False

    '''Инициализировать указатели, указывающие
    на начало и конец списка
    '''
    start = 0
    end = len(numbers) - 1

    '''Продолжать перемещать указатели,
    пока они не встретятся
    '''
    while start < end:
        two_sum = numbers[start] + numbers[end]

        '''Если сумма двух чисел равна
        целевому, значит, мы нашли ответ
        '''
        if two_sum == target:
            return True

            '''Если сумма меньше целевого числа,
            то увеличить указатель start,
            чтобы перейти к следующему большему числу
            '''
```

```
33          elif two_sum < target:
34              start += 1
35
36              # Если сумма больше целевого числа,
37              # то уменьшить указатель end,
38              # чтобы перейти к следующему меньшему числу
39          else:
40              end -= 1
41
42      '''Если искомой пары чисел не нашлось,
43      то вернуть False
44      '''
45      return False
46
47  '''
48  Главная функция, отыскивающая три числа
49  в сортированном списке целых чисел,
50  сумма которых
51  равна целевому числу
52  '''
53  def has_three_summers(numbers_list, goal):
54      '''Если длина списка
55      меньше 3, то в нем невозможно
56      найти три числа,
57      сумма которых равна целевому числу
58      '''
59      if len(numbers_list) < 3:
60          return False
61
62      # Цикл по элементам списка
63      for i in range(len(numbers_list)):
64          '''Получить очередное число
65          из списка
66          '''
67          x = numbers_list[i]
68          '''Вычислить новое целевое число
69          вычитанием
70          '''
71          new_goal = goal - x
72
73          '''
74          Проверить наличие в оставшейся части списка
75          пары чисел, сумма которых
```

```
76          равна новому целевому числу
77          '''
78          if has_two_sum(numbers_list[i + 1:], new_goal):
79              return True
80
81      '''
82      Если ничего не найдено,
83      вернуть False
84      '''
85      return False
```

2.21. Определение совершенной степени

Положительное целое число считается совершенной степенью, если его можно выразить в виде $base^{power}$, где *base* и *power* являются целыми числами больше 1. Ваша задача: написать функцию, которая принимает положительное целое число *n*, и если ей удастся найти $base^{power} = n$, то она возвращает *True*, иначе она должна возвращать *False*. Например, для $n = 64$ существует пара чисел $base = 2$ и $power = 6$, т. е. $2^6 = 64$.

В табл. 2.21 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.21. Некоторые ожидаемые результаты для задачи определения идеальной степени

n	Ожидаемый результат
2	False
27	True
729	True
1369	True

Алгоритм

Чтобы определить, является ли положительное целое число `n` совершенной степенью, используется алгоритм, выполняющий двоичный поиск основания совершенной степени. Алгоритм начинается с показателя степени, равного 2, и продолжает перебирать в цикле все возрастающие числа, пока не найдет идеальную степень или не выяснит, что таковой не существует. Вот подробное описание шагов алгоритма.

1. Принимается целое положительное число `n`.
2. Показателю степени `power` присваивается значение `2`.
3. Запускается цикл `while True:`.

4. Если `2 ** power > n`, возвращается `False`.
5. Инициализируются переменные `low = 2` и `high = n`.
6. Пока `low` меньше или равно `high`:
    1) переменной `mid` присваивается результат деления `(low + high)` на `2` с округлением вниз;
    2) переменной `guess` присваивается значение `mid`, возведенное в степень `power`;
    3) если `guess` равно `n`, верните True;
    4) иначе, если `guess` меньше `n`, переменной `low` присваивается значение `mid + 1`;
    5) иначе переменной `high` присваивается значение `mid - 1`.
7. Если `base ** power` равно `n`, то возвращается `True`.
8. Значение `power` увеличивается на 1.

В листинге 2.21 приводится код на Python, проверяющий, является ли заданное число совершенной степенью.

Листинг 2.21. Определение совершенной степени

```
def perfect_power(n):
    '''
    В языке Python оператор ** выполняет возведение в степень
    '''
    power = 2

    while True:
        if 2 ** power > n:
            return False

        # Найти наибольшую степень, используя двоичный поиск
        low = 2
        high = n
        while low <= high:
            mid = (low + high ) // 2
            guess = mid ** power

            if guess == n:
                return True
            elif guess < n:
                low = mid + 1
            else:
                high = mid - 1

```

```
25          # Увеличить power
26          power += 1
```

2.22. Лунное умножение целых чисел

Результатом лунного умножения двух чисел является наименьшее из них, а результатом лунного сложения – наибольшее. Например, 2 + 7 =7 + 2 = 7, $2 \times 7 = 7 \times 2 = 2$, поскольку лунные операции обладают свойством коммутативности. Напишите функцию, которая принимает два n-значных целых числа a и b и возвращает их лунное произведение. Как и при обычном сложении, единичным элементом для сложения является ноль, а для любого натурального числа n единичным элементом является 9, где $2 \times 9 = 2$, $7 \times 9 = 7$, $9 \times 63 = 63$. Рассмотрим следующий пример. Пусть $a = 10$ и $b = 21$, умножение выполняется поразрядно.

- $(1 \times 10) \rightarrow (1 \times 0 = 0) \rightarrow (1 \times 1 = 1) \rightarrow 10$;
- $(2 \times 10) \rightarrow (2 \times 0 = 0) \rightarrow (2 \times 1 = 1) \rightarrow 10$;
- $+\frac{10}{10} = 110.$

В табл. 2.22 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.22. Некоторые ожидаемые результаты для задачи лунного умножения

a, b	Ожидаемый результат
10, 21	110
170, 76	1760
45, 96	455
8, 7	7

Алгоритм

Алгоритм решения этой задачи прост и понятен. За один шаг он выполняет умножение между каждыми двумя разрядами, а затем складывает умноженные числа с учетом переносов.

В листинге 2.22 приводится код на Python, выполняющий лунное умножение.

Листинг 2.22. Лунное умножение

```
1 def multiply_by_lunar(a, b):
2     '''
```

```
    Функция лунного сложения
    двух чисел
    '''
    def add_in_lunar(x, y):
        # Преобразовать x и y в строки и перевернуть их
        '''
        Потому что вычисления начинаются с
        младшей значащей цифры.
        '''
        x_reversed = rev(x)
        y_reversed = rev(y)

        # Инициализировать сумму пустой строкой
        sum_reversed = ''

        '''Вычислить максимальную длину строковых представлений чисел
        и выполнить сложение
        '''
        max_len = max(len(x_reversed), len(y_reversed))

        i = 0
        while i < max_len:
            '''Выполнить лунное сложение цифр
            в перевернутых строковых представлениях
            '''
            sum_reversed += \
                max(x_reversed[i:i + 1], y_reversed[i:i + 1])
            i += 1

        '''
        Перевернуть строковое представление результата,
        преобразовать в целое число и вернуть
        '''
        return int ( sum_reversed[::-1])

    # Преобразовать a и b в строки и перевернуть их,
    '''
    потому что сложение чисел начинается
    с младшего значащего разряда.
    '''
    a_reversed = rev(a)
    b_reversed = rev(b)

```

```
    # Инициализировать массив numbers и переменную answer
    numbers = []
    answer = 0

    i = 0
    j = 0
    d = ''
    while i < len(b_reversed):
        if j < len(a_reversed):
            '''
            Добавить в конец d наименьшую цифру из a и b в
            текущих позициях перевернутых строковых представлений
            '''
            d += min(a_reversed[j], b_reversed[i])
            j += 1
        else:
            '''Если достигнут конец a, то просто добавить
            остальные цифры из b в d
            '''
            numbers.append('0' * i + d)
            i += 1
            j = 0
            d = ''

    if d:
        # Добавить остальные цифры из d в numbers
        numbers.append('0' * i + d)

    '''Сложить числа на каждом шаге, используя
    функцию add_in_lunar
    '''
    for num in numbers:
        answer = add_in_lunar(answer, num[::-1])

    return answer


# Функция, переворачивающая число
def rev(x):
    x_reversed = ''
    for i in range(len(str(x)) - 1, -1, -1):
        x_reversed += str(x)[i]
    return x_reversed
```

2.23. *n*-й член последовательности Рекамана

Последовательность Рекамана – это последовательность, демонстрирующая рекуррентное соотношение, когда вычисление нового элемента зависит от предшествующих. Для иллюстрации получим *n*-й член последовательности Рекамана (где $n = 1$), выполнив вычисления согласно рекурсивной формуле 2.7. Нулевой член a_0 всегда равен нулю. Приступая к вычислению a_1, мы обнаруживаем, что $0-1 = -1 < 0$, отвергаем второе условие и переходим к третьему, которое дает $a_1 = 0 + 1$. Ваша задача: написать функцию, которая принимает положительное целое число *n* и возвращает *n*-й член последовательности Рекамана.

$$x = \begin{cases} 0 & if\ n == 0 \\ a_{n-1} - 1 & if\ a_{n-1} - 1 > 0 \text{ и числа еще нет в последовательности.} \\ a_{n-1} + n & else \end{cases} \quad (2.7)$$

В табл. 2.23 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.23. Некоторые ожидаемые результаты для задачи поиска *n*-го члена последовательности Рекамана

n	Ожидаемый результат
17	25
83	72
919	756
632	308

Алгоритм

Чтобы найти *n*-й член, алгоритм инициализирует последовательность ее первым элементом и запоминает в ней все последующие элементы. Затем выполняет итерации по оставшимся элементам до *n* и для каждого вычисляет следующий элемент, используя формулу 2.7. Если следующий элемент отрицательный или уже присутствует в последовательности, то использует альтернативную формулу для расчета следующего элемента. Затем следующий вычисленный элемент добавляется в последовательность и в множество использованных элементов. Наконец, алгоритм возвращает *n*-й элемент последовательности.

В листинге 2.23 приводится код на Python, выполняющий поиск *n*-го члена последовательности Рекамана.

Листинг 2.23. Поиск n-го члена последовательности Рекамана

```
def Nth_term_of_recaman_sequence(n):
    '''
    Возвращает n-й член последовательности Рекамана
    '''
    # Инициализировать последовательность первым элементом
    seq=[0]
    # Инициализировать множество использовавшихся элементов
    seen = {0}
    # Обход остальных элементов до n-го
    for i in range(1, n + 1):
        # Получить предыдущий элемент в последовательности
        prev = seq[-1]
        '''Вычислить следующий элемент
        по формуле, как показано ниже
        '''
        next = prev - i
        '''
        Если следующий элемент отрицательный или уже использовался,
        то выполнить вычисления по альтернативной формуле
        '''
        if next < 0 or next in seen:
            next = prev + i
        # Добавить элемент в последовательность
        seq.append(next)
        # Поместить этот элемент в множество использовавшихся элементов
        seen.add(next)
    # Вернуть n-й элемент
    return seq[-1]
```

2.24. *n*-й член последовательности Ван Эка

Рассмотрим последовательность *S*, состоящую из положительных целых чисел. Первый член *S* равен 0. Есть ли ноль перед текущим нулем? Нет, поэтому последовательность равна (0, 0). Для последнего числа (в данном случае для второго нуля) задается тот же вопрос: есть ли ноль перед текущим (вторым) нулем? Да, поэтому последовательность равна (0, 0, 1). И снова для последнего числа (1) задается вопрос: есть ли «единица» перед текущей единицей? Нет, значит, последовательность равна (0, 0, 1, 0). И снова для последнего числа (0) задается вопрос: есть ли ноль перед текущим нулем? Да, поэтому последовательность равна (0, 0, 1, 0, 2)

и т. д. Проще говоря, последовательность подсчитывает количество шагов, необходимых для достижения первого запрошенного числа, если следовать начиная с хвоста последовательности до запрошенного числа, и подсчет прекращается, когда обнаруживается искомое число. Эта популярная последовательность называется последовательностью Ван Эка. Ее члены определяются рекурсивно, поскольку каждый следующий член вычисляется на основе предыдущих. Ваша задача: написать функцию, которая принимает положительное целое число *n* и возвращает *n*-й член последовательности Ван Эка.

В табл. 2.24 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.24. Некоторые ожидаемые результаты для задачи поиска *n*-го члена последовательности Ван Эка

n	Ожидаемый результат
258	3
25	4
2	1
2089	0

Алгоритм

Ниже перечислены шаги, выполняемые алгоритмом поиска `n`-го члена последовательности Ван Эка.

1. Последовательность инициализируется первым членом, равным `0`.
2. Создается пустой словарь для хранения каждого значения в последовательности с их индексами.
3. Для каждого индекса от `0` до `n - 1`:
    1) получить предыдущее значение в последовательности `seq [i]`;
    2) проверить, встречалось ли предыдущее значение раньше, поиском его в словаре;
    3) если встречалось, то вычислить разность между текущим и предыдущим значениями;
    4) добавить разность в последовательность;
    5) если предыдущее значение не встречалось, добавить в последовательность `0`;
    6) обновить словарь, указав текущий индекс `i` для предыдущего значения.
4. Вернуть `n`-й член последовательности, то есть `seq [n - 1]`.

В листинге 2.24 приводится код на Python, выполняющий поиск *n*-го члена последовательности Ван Эка.

Листинг 2.24. Поиск n-го члена последовательности Ван Эка

```
"""
Возвращает n-й член
последовательности Ван Эка.
"""
def find_nth_term_in_van_eck_sequence(n):
    '''
    Инициализировать последовательность
    первым членом
    '''
    sequence = [0]
    '''
    Создать словарь для хранения
    значений членов последовательности с их индексами
    '''
    value_to_latest_index = {}
    for i in range(n):
        # Проверить, встречалось ли предыдущее значение раньше
        previous_value = sequence[i]
        if previous_value in value_to_latest_index:
            '''
            Если встречалось, то вычислить разность между
            текущим индексом и предыдущим
            '''
            difference = i - \
                value_to_latest_index[previous_value]
            # Добавить разность в последовательность
            sequence.append(difference)
        else:
            # Если не встречалось, то добавить 0 в последовательность
            sequence.append(0)
        '''
        Обновить индекс текущего
        значения в словаре
        '''
        value_to_latest_index [ previous_value ] = i
    # Вернуть последнее значение в последовательности
    return sequence[-1]
```

2.25. Поиск суммы чисел Фибоначчи на основе теоремы Цекендорфа

Согласно теореме Цекендорфа, любое произвольное положительное целое можно выразить как сумму (необязательно последовательных) чисел Фибоначчи. Напишите функцию, которая принимает положительное целое число *n* и возвращает список непоследовательных чисел Фибоначчи, сумма которых равна *n*. Числа в списке должны следовать в порядке убывания.

В табл. 2.25 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.25. Некоторые ожидаемые результаты для задачи поиска суммы чисел Фибоначчи

n	Ожидаемый результат
53	[34, 13, 5, 1]
25	[21, 3,1]
31	[21, 8, 2]
3009	[2584, 377, 34, 13, 1]

Алгоритм

Алгоритм генерирует последовательность чисел Фибоначчи, сортирует ее в порядке убывания с использованием алгоритма сортировки вставками и выбирает элементы из отсортированной последовательности, руководствуясь определенными критериями. Ниже перечислены шаги, выполняемые алгоритмом.

1. Принимается целое положительное число `n`.
2. Инициализируются переменные: `previous_fibonacci_number` – значением `0`, `current_fibonacci_number` – значением `1`, `fibonacci_numbers` – списком, содержащим `previous_fibonacci_number` и `current_fibonacci_number`, `selected_fibonacci_numbers` – пустым списком и `current_sum` – суммой `previous_fibonacci_number + current_fibonacci_number`.
3. Используется цикл `while`, чтобы сгенерировать числа Фибоначчи до `n`:
   1) добавить `current_sum` в `fibonacci_numbers`;
   2) обновить `previous_fibonacci_number` значением `current_fibonacci_number`;
   3) обновить `current_fibonacci_number` значением `current_fibonacci_number + previous_fibonacci_number`;
   4) обновить `current_sum` значением `previous_fibonacci_number + current_fibonacci_number`;

5) продолжать выполнять цикл, пока `current_sum` не станет больше `n`;
6) отсортировать `fibonacci_numbers` в порядке убывания.

4. Используется цикл `for` для выбора чисел Фибоначчи, сумма которых дает в результате `n`, выполняющий следующие действия:
   1) переменная `sum_of_fibonacci_numbers` инициализируется значением `0`;
   2) для каждого числа в `fibonacci_numbers` выполняются следующие действия:
      а) если `sum_of_fibonacci_numbers + number` меньше или равно `n`, то `number` добавляется в `selected_fibonacci_numbers` и переменная `sum_of_fibonacci_numbers` обновляется значением `sum_of_fibonacci_numbers + number`;
      б) если `sum_of_fibonacci_numbers` равна `n`, то возвращается `selected_fibonacci_numbers`.

В листинге 2.25 приводится код на Python, выполняющий поиск суммы чисел Фибоначчи, равной заданному числу *n*.

Листинг 2.25. Поиск суммы чисел Фибоначчи, равной заданному числу n

```
def sort_desc(arr):
    for i in range(1, len(arr)):
        curr = arr[i]
        j= i - 1
        while j >= 0 and curr > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = curr
    return arr

def get_non_consecutive_fibonacci_numbers_summing_to_n(n):
    prev = 0
    curr = 1
    fib_nums = [prev, curr]
    selected_fib_nums = []
    curr_sum = prev + curr

    while curr_sum <= n:
        fib_nums.append(curr_sum)
        prev, curr = curr, curr + prev
        curr_sum = prev + curr

    fib_nums = sort_desc(fib_nums)
```

```
24
25      sum_of_fib_nums = 0
26      for num in fib_nums:
27          if sum_of_fib_nums + num <= n:
28              selected_fib_nums.append(num)
29              sum_of_fib_nums += num
30              if sum_of_fib_nums == n:
31                  return selected_fib_nums
```

2.26. Поиск *k*-го слова Фибоначчи

Слова Фибоначчи – это последовательность, напоминающая обычную последовательность чисел Фибоначчи, с той лишь разницей, что в роли суммы двух предыдущих чисел рассматривается их конкатенация. Первый член этой последовательности равен 0, второй – 01 (как строка). Каждый последующий член получается путем объединения двух предыдущих членов. Например, третий член получается путем объединения первого и второго членов, что дает в результате 010. Аналогично четвертый член получается путем объединения второго и третьего членов, что приводит к 01001, и т.д.

Ваша задача: написать функцию, которая принимает целое число *k* и возвращает *k*-е «слово» Фибоначчи – строку, состоящую из символов 0 и 1.

В табл. 2.26 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.26. Некоторые ожидаемые результаты для задачи поиска *k*-го слова Фибоначчи

k	Ожидаемый результат
65	0
170	0
98	1
2022	1

Алгоритм

Решение задачи заключается в создании последовательности до *k*-й позиции и извлечении указанного элемента. Однако этот метод имеет высокую временную и пространственную сложность. Для поиска *k*-го символа используются формулы 2.8 и 2.9, где *phi* обозначает золотое число.

$$phi = \frac{1+\sqrt{5}}{2}, \tag{2.8}$$

$$\lfloor (k+2) * phi/(1+(phi*2)) \rfloor - (k+1) * phi/(1+(phi*2)). \tag{2.9}$$

В листинге 2.26 приводится код на Python, выполняющий поиск *k*-го символа в последовательности слов Фибоначчи.

Листинг 2.26. Поиск k-го символа в последовательности слов Фибоначчи

```
import decimal
def sqrt(num):
    decimal.getcontext().prec = 165
    return decimal.Decimal(num).sqrt()
def floor(x):
    return int(x - 1) if x < 0 else int(x)
def fibonacci_word(k):
    # Получить корень из пяти
    root_5 = sqrt(5)

    # Вычислить золотое число
    phi = (1 + root_5)/2
    x = floor((k + 2) * phi / (1 + (phi * 2)))
    y = floor((k + 1) * phi / (1 + ( phi * 2)))
    # Вернуть k-й символ
    return x - y
```

2.27. Поиск прямой в двумерной сетке, пересекающей наибольшее количество точек

Точка в двумерной целочисленной сетке представлена кортежем с координатами *x* и *y*, например (2, 5) или (10, 3). Через две разные точки на плоскости проходит ровно одна прямая. Эта прямая простирается бесконечно по обе стороны и пересекает бесконечное число других точек на плоскости. Доказать это утверждение не просто, но оно верное, поэтому примем его за аксиому.

Ваша задача: написать функцию, которая принимает список точек в двумерной сетке и отыскивает прямую, заданную двумя точками, которая пересекает наибольшее количество точек из заданного списка. Функция должна возвращать не саму прямую, а только количество точек, лежащих на ней.

В табл. 2.27 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.27. Некоторые ожидаемые результаты для задачи поиска прямой, пересекающей наибольшее количество из заданных точек

k	Ожидаемый результат
[(4, 4), (6, 6), (3, 2), (1, 4)]	2
[(14, 4), (0, 0), (1, 2), (3, 0)]	2
[(3, 15), (1, 4), (2, 6), (8, 7), (3, 8)]	3

Алгоритм

Алгоритм принимает список двумерных точек и возвращает максимальное количество точек, лежащих на одной из найденных прямых. Сначала алгоритм проверяет длину входного списка, и если она меньше 3, то возвращает длину списка. Иначе перебирает все возможные пары различных точек в списке, вычисляет наклон прямой, проходящей через эти точки, и запоминает частоту каждого наклона в словаре. Он также запоминает количество повторяющихся точек. В завершение алгоритм возвращает максимальное количество точек на прямой, прибавляя текущее максимальное количество точек к числу повторяющихся точек и сравнивая его с предыдущим максимальным значением.

В листинге 2.27 приводится код на Python, выполняющий поиск прямой в двумерной сетке, пересекающей наибольшее количество точек.

Листинг 2.27. Поиск прямой в двумерной сетке, пересекающей наибольшее количество точек

```
def calculate_gcd(a, b) :
    '''
    Вычисляет наибольший общий
    делитель двух целых чисел a и b
    '''
    while b != 0:
        a, b = b, a % b
    return a

def count_points_on_line(points):
    n = len(points)
    '''
    Если в списке меньше трех точек,
    они всегда будут лежать на одной прямой.
```

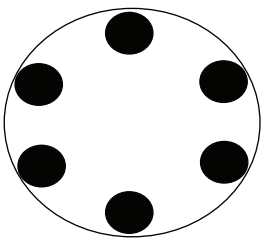
```
    '''
    if n < 3:
        return n

    max_points_on_line = 0
    i = 0
    '''
    Выполнить обход всех точек
    в списке
    '''
    while i < n:
        '''
        Создать словарь для хранения
        наклонов и их частот
        '''
        slope_count = {}
        num_duplicates = 1
        cur_max_points_on_line = 0
        j = i + 1
        # Обойти остальные точки и подсчитать наклоны прямых
        while j < n:
            if points[i] == points[j]:
                num_duplicates += 1
            else:
                dx = points[j][0] - points[i][0]
                dy = points[j][1] - points[i][1]
                if dx == 0:
                    slope = float('inf')
                else:
                    gcd = calculate_gcd(dy, dx)
                    slope = (dy // gcd, dx // gcd)

                '''
                Добавить наклон в словарь
                или увеличить его частоту
                '''
                slope_count[slope] = \
                    slope_count.get(slope, 0) + 1

                '''
                Обновить максимальное количество точек
                на прямой с текущим наклоном
                '''
```

```
58                  cur_max_points_on_line = \
59                      max(cur_max_points_on_line,
60                          slope_count[slope])
61
62              j += 1
63
64          '''
65          Обновить максимальное количество точек
66          на прямой, проходящей через текущую точку
67          '''
68          max_points_on_line =\
69              max(max_points_on_line,
70                  cur_max_points_on_line + num_duplicates)
71          i += 1
72
73      return max_points_on_line
```

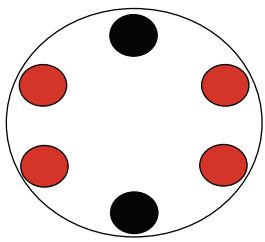
2.28. Проверка сбалансированности центрифуги

Ученые используют центрифуги для пробирок в своих опытах, и очень важно сбалансировать пробирки в центрифуге, чтобы не нарушить ее работу или не вызвать повреждения при высокой скорости вращения. Сбалансированности можно добиться, размещая пробирки в центрифуге, следуя определенной закономерности. Например, на рис. 2.3 (А) показана центрифуга с шестью отверстиями, а на рис. 2.3 (Б) – с четырьмя пробирками, вставленными в отверстия. Как можно видеть на рис. 2.3 (Б), при таком размещении пробирок центрифуга сбалансирована.

Ваша задача: написать функцию, которая принимает количество отверстий в центрифуге n и количество пробирок k и сообщает, можно ли разместить пробирки в центрифуге так, чтобы она оставалась сбалансированной.



А) Центрифуга с шестью отверстиями, $n = 6$



Б) Центрифуга с шестью отверстиями и четырьмя пробирками 4 tube (показаны красным цветом), $n = 6$, $k = 4$

Рис. 3.3. Пример сбалансированной центрифуги

В табл. 2.28 показаны ожидаемые результаты для некоторых входных данных.

Таблица 2.28. Некоторые ожидаемые результаты для задачи проверки сбалансированности центрифуги

n, k	Ожидаемый результат
76, 19	True
23, 3	False
113, 18	False
60, 2	True

Алгоритм

Первый шаг – определить простые множители n и сохранить их в списке L. Далее необходимо проверить, можно ли получить оба значения, n и $n - k$, используя простые множители в L. Например, рассмотрим случай, когда $n = 6$ и $k = 3$. Список простых множителей для n – [2, 3]. Оба значения, n и $n - k$, можно выразить, используя простые множители в L. В частности, $k = 3$ и присутствует в L, $n - k = 6–3 = 3$ тоже присутствует в L, а значит, при таких условиях пробирки можно разместить в центрифуге так, чтобы не нарушить ее сбалансированность. Еще один пример: пусть $n = 15$ и $k = 8$, список простых множителей n равен [5, 3]. Значение k можно получить сложением 5 и 3, но невозможно получить значение $n - k$, в данном случае равное 7, используя числа из списка. Следовательно, функция должна вернуть *False*.

Для решения задачи понадобятся два алгоритма. Первый алгоритм определяет список простых множителей для n, а второй проверяет возможность получить значения n и $n - k$ с использованием простых множителей, полученных первым алгоритмом.

Поиск простых множителей

Алгоритм создает пустой список для хранения простых множителей. Первое простое число равно 2. Пока n больше 1, выполняется деление n на 2. Если остаток равен нулю и число не было добавлено в список, оно добавляется в список. На каждом этапе n обновляется значением частного. Если n больше не делится на 2 без остатка, то рассматривается следующее простое число, и этот процесс повторяется до тех пор, пока $n < 1$.

Алгоритм принимает положительное целое число n и возвращает список простых множителей этого числа. Вот подробное описание этого алгоритма.

1. Создается пустой список `primes` для хранения простых множителей.
2. Переменная `i` инициализируется значением `2` – первым простым числом.
3. Запускается цикл `while`, который выполняется, пока `n > 1`.
4. В теле цикла проверяется остаток от деления `n` на `i`. Если `n` делится на `i` без остатка, это означает, что `i` является делителем `n`.
5. Если `i` – простой множитель `n` и его еще нет в списке `primes`, то он добавляется в список.
6. `n` делится на `i`, поэтому мы можем продолжать проверять, является ли `i` простым множителем оставшегося значения `n`.
7. Если `i` не является простым множителем `n`, то `i` увеличивается на `1` и выполняется переход к следующей итерации.
8. Цикл продолжается до тех пор, пока `n` не станет меньше или равно `1`.
9. В заключение алгоритм возвращает список `primes`, содержащий все простые множители числа `n`.

Размен монет

Алгоритм размена монет используется для определения возможности представить заданное число комбинацией значений из заданного списка. Этот алгоритм принимает число и массив и проверяет, можно ли сформировать число из элементов заданного списка. Идея решения данной задачи заключается в использовании динамического программирования. В частности, алгоритм создает список `ways` из `n + 1` элементов и инициализирует его нулями, за исключением первого элемента (`ways [0]`), которому присваивается значение `1`. Это отражает тот факт, что существует один способ составить общее количество `0`, не используя монет из списка. `i`-й элемент массива содержит количество раз, которыми число `i` можно сформировать из чисел в заданном массиве `primes`. Следовательно, последний элемент массива указывает количество способов образования числа `n` из элементов массива `primes`.

В листинге 2.28 приводится код на Python, выполняющий проверку возможности балансировки центрифуги.

Листинг 2.28. Задача проверки возможности балансировки центрифуги

```
def is_balanced_centrifuge(n, k):
    def get_primes(n):
        """
        Возвращает список простых множителей
        для заданного числа.
        """
```

```
 7         primes = []
 8         i= 2
 9         # Цикл по всем возможным множителям числа
10         while n > 1:
11             if n % i == 0:
12                 # Добавить множитель в список primes
13                 if i not in primes:
14                     primes.append(i)
15                 '''Разделить число на множитель,
16                 чтобы получить следущий множитель
17                 '''
18                 n /= i
19             else:
20                 '''
21                 Если не делится без остатка,
22                 то попробовать следующее число
23                 '''
24                 i += 1
25         return primes
26
27     def can_build_number(arr, n):
28         """
29         Проверяет, можно ли составить число
30         из элементов заданного списка.
31         """
32         '''
33         Создать список ways, чтобы
34         проверить каждое число, меньшее целевого значения
35         '''
36         ways = [0] * (n + 1)
37         # Получить 0 можно только одним способом
38         ways[0] = 1
39         # Цикл по всем элементам списка
40         for i in range(len(arr)):
41             # Цикл по всем возможным числам, участвующим в сборке
42             for j in range(arr[i], n + 1):
43                 '''
44                 Прибавить число способов и получить текущее
45                 число с использованием текущего элемента
46                 '''
47                 ways[j] += ways[j - arr[i]]
48         '''
49         Если есть хотя бы один способ сформировать
```

```
        целевое число, то вернуть True
        '''
        return ways[n] != 0

    # Получить список простых множителей для n
    primes = get_primes(n)
    # Проверить возможность балансировки
    # с использованием списка простых множителей
    return can_build_number(primes, k ) \
           and can_build_number(primes, n - k)
```

Глава 3

Числа

В этой главе рассматривается 21 задача с сопровождением примерами решения на Python. Вот список задач:

1. Проверка, является ли число числом Циклопа.
2. Проверка наличия цикла домино в списке чисел.
3. Извлечение возрастающих чисел из заданной строки.
4. Развертывание целочисленных интервалов.
5. Свертывание целочисленных интервалов.
6. Проверка левостороннего и правостороннего игрального кубика.
7. Очки за повторяющиеся числа.
8. Первое меньшее число.
9. Первый объект, предшествующий *k* меньшим объектам.
10. Поиск *n*-го члена последовательности Калкина–Уилфа.
11. Поиск и обращение восходящих подмассивов.
12. Наименьшие целые степени.
13. Сортировка циклов в графе.
14. Получение представления числа в сбалансированной троичной системе.
15. Строгое возрастание.
16. Сортировка по приоритетам.
17. Сортировка положительных чисел с сохранением порядка отрицательных чисел.
18. Сортировка: сначала числа, потом символы.
19. Сортировка дат.
20. Сортировка по алфавиту и длине.
21. Сортировка чисел по количеству цифр.

3.1. Число Циклопа

Неотрицательное целое число называется числом Циклопа, если оно удовлетворяет следующим условиям: количество цифр нечетно, средняя цифра (также известная как «глаз») равна нулю, а все остальные цифры отличны от нуля. Ваша задача: написать функцию, проверяющую, является ли заданное неотрицательное целое число числом Циклопа. Функция должна принимать неотрицательное целое число и проверить, удовлетворяет ли оно вышеперечисленным условиям. Если входное целое число является числом Циклопа, то функция должна вернуть значение *True*, иначе она должна вернуть *False*.

В табл. 3.1 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.1. Некоторые ожидаемые результаты для задачи проверки числа Циклопа

n	Ожидаемый результат
11000	False
709	True
11318	False
6022	False

Алгоритм

Чтобы проверить, является ли заданное число числом Циклопа, алгоритм выполняет следующие шаги.

1. Принимается целое число `n`.
2. Число `n` преобразуется в строку цифр и сохраняется в переменной `digits`.
3. Проверяется, четна ли длина строки `digits`. Если да, то возвращается `False`, потому что число Циклопа должно иметь нечетное количество цифр.
4. Выбирается цифра в середине, индекс которой определяется делением длины `digits` на 2 с использованием целочисленного деления (`//`), и сохраняется в переменной `middle_number`.
5. Проверяется, равна ли цифра с индексом `middle_number` нулю (`0`). Если нет, то возвращается `False`, потому что число Циклопа должно иметь `0` в середине.
6. Подсчитывается количество нулей в `digits` и результат сохраняется в переменной `count`. Если число больше `1`, возвращается `False`, пото-

му что число Циклопа может иметь только один ноль (`0`). Иначе возвращается `True`, потому что входное значение `n` удовлетворяет всем условиям числа Циклопа.

В листинге 3.1 приводится код на Python, выполняющий проверку числа Циклопа.

Листинг 3.1. Проверка числа Циклопа

```
 1 def cyclop_number(n):
 2     digits = str(n)
 3     if len(digits) % 2 == 0:
 4         return False
 5
 6     middle_number = len(digits) // 2
 7     if digits[middle_number] != '0':
 8         return False
 9
10     count = digits.count('0')
11     if count > 1:
12         return False
13
14     return True
```

3.2. Цикл домино

Костяшку домино можно представить кортежем (*x*, *y*), где *x* и *y* – положительные целые числа. Цель этой задачи – определить, имеется ли цикл домино в заданном наборе костяшек. Например, в наборе [(5, 2), (2, 3), (3, 4), (4, 5)] имеется цикл домино. Первая костяшка (5, 2) связана с (2, 3); $2 \rightarrow 2$. Костяшка (2, 3) связана с (3, 4); $3 \rightarrow 3$. Так же связаны остальные костяшки: $4 \rightarrow 4$, $5 \rightarrow 5$. Ваша задача: написать функцию, которая принимает список костяшек домино и возвращает *True*, если в последовательности есть цикл, иначе возвращает *False*.

В табл. 3.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.2. Некоторые ожидаемые результаты для задачи проверки цикла домино

Tiles	Ожидаемый результат
[(5, 2), (2, 3), (3, 4), (4, 5)]	True
[(3, 4), (4, 2), (2, 3), (3, 1), (4, 2)]	False
[(3, 4), (4, 2), (2, 3), (3, 1), (4, 2), (2, 4), (6, 3), (3, 2)]	False
[(6, 4), (4, 5), (5, 6)]	True

Алгоритм

Алгоритм должен проверить равенство компонентов *y* и *x* двух сравниваемых кортежей на каждом шаге. Если они не равны, он должен вернуть *False*, иначе он должен вернуть *True*.

В листинге 3.2 приводится код на Python, выполняющий проверку цикла домино.

Листинг 3.2. Проверка цикла домино

```
def Is_a_domino_cycle(tiles):
    index = 0
    while index < len(tiles):
        # Проверить условия
        if tiles[index][0] != tiles[(index - 1) % len(tiles)][1]:
            '''
            Если хотя бы одно условие не выполняется,
            то вернуть False.
            '''
            return False
        index += 1
    return True
```

3.3. Извлечение возрастающих чисел

В этой задаче дается строка цифр и гарантируется, что в строке присутствуют только цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Требуется разбить строку на последовательность возрастающих чисел. Например, число 457990 можно представить в виде последовательности возрастающих чисел 4, 5, 7, 9 и 90.

Ваша задача: написать функцию, которая принимает строку из цифр и возвращает список возрастающих чисел.

В табл. 3.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.3. Некоторые ожидаемые результаты для задачи извлечения возрастающих чисел

Digits	Ожидаемый результат
'457990'	[4, 5, 7, 9, 90]
'1'	[1]
'13900456'	[1, 3, 9, 45]
27811700	[2, 7, 8, 11, 70]

Алгоритм

Алгоритм состоит из двух блоков проверки возрастающих чисел. Сначала определяется пустая строка *concatnumbers* и переменная *b* со значением 1. Если *b* = 1, выполняется вход в первый блок, а если *b* = 0, выполняется вход во второй блок. Для каждой цифры в данной строке в первом блоке производится линейный поиск, пока следующее число больше предыдущего и *b* = 1, и каждое следующее такое число добавляется в массив *storage*, после чего оно становится предыдущим числом. Во втором блоке, если условия входа в первый блок не выполняются, переменной *b* присваивается 0, затем текущее число объединяется со следующей цифрой и помещается в *concatnumbers*. Если значение *concatnumbers* больше предыдущего числа, то оно сохраняется в *storage* и переменной *concatnumbers* присваивается пустая строка. Пока объединение следующих цифр меньше или равно предыдущему числу, объединение повторяется для следующих цифр. Когда *b* = 0, алгоритм ни за что не сможет войти в первый блок, поскольку ни одна цифра не будет больше предыдущих чисел и входить в первый блок бессмысленно.

В листинге 3.3 приводится код на Python, выполняющий извлечение возрастающих чисел.

Листинг 3.3. Извлечение возрастающих чисел

```
def extract_increasing_digits(digits):
    number=[]
    concatnumbers = ''
    for x in str(digits):
        number.append(int(x))
    # Предыдущее число
    prev = number[0]
    storage = []
    # Пока есть только одна цифра
    storage = [number[0]]
    number = number[1:]
    b=1
    for next_ in number:
        '''
        Этот блок извлекает возрастающие цифры,
        т. е. числа из одной цифры,
        такие как 2,3,4
        '''
        if next_ > prev and b == 1:
            storage.append(next_)
            prev = next_
```

```
        else:
            '''
            Этот блок извлекает возрастающие числа,
            состоящие из нескольких цифр,
            такие как 90, 990, 9990
            '''
            b = 0
            concatnumbers += str(next_)
            if int(concatnumbers) > prev:
                storage.append(int(concatnumbers))
                prev = int(concatnumbers)
                '''
                Очистить строку concatnumbers, чтобы подготовить ее
                для сборки следующего числа из цифр,
                которое больше текущего
                '''
                concatnumbers = ''
    return storage
```

3.4. Развертывание целочисленных интервалов

Диапазон последовательных положительных целых чисел можно представить как строку, включающую первое и последнее значения, разделенные дефисом (-). Вот пример допустимого интервала: «1,17–21,43,44».

Ваша задача: написать функцию, которая развертывает подобные интервалы в списки отдельных чисел. Функция должна принимать строку с числами и возвращать список чисел, разделенных запятыми.

Важное условие: числа во входной строке гарантированно располагаются в порядке возрастания.

В табл. 3.4 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.4. Некоторые ожидаемые результаты для задачи развертывания целочисленных интервалов

Intervals	Ожидаемый результат
'1, 7 – 16, 120 – 124, 568'	[1, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 120, 121, 122, 123, 124, 568]
'2 – 10'	[2, 3, 4, 5, 6, 7, 8, 9, 10]
'2, 3 – 5, 19'	[2, 3, 4, 5, 19]
17	[17]

Алгоритм

Сначала алгоритм должен разбить строку по символу запятой (,). Затем отделить отдельные числа от интервалов. Отдельные числа сохраняются в списке, а интервалы развертываются с помощью цикла `for`, который выполняет перебор от начала до конца интервала и включает в выходной список каждое следующее число. Вот подробное описание шагов алгоритма.

1. Принимается строка `intervals` с интервалами.
2. Если строка `intervals` пустая, то возвращается пустой список.
3. Создается пустой список с именем `result` для хранения чисел, получающихся после развертывания интервалов.
4. Входная строка разбивается на отдельные интервалы по запятым (,), которые сохраняются в списке `intervals_list`.
5. Для каждого интервала в списке `intervals_list` проверяется, содержит ли он одно число (точнее, отсутствие дефиса «-»). Если интервал – это одно число, то он преобразуется в целое число с помощью `int ()` и добавляется в список `result`.
6. Если интервал содержит диапазон чисел (т.е. присутствует дефис «-»), то он разбивается на отдельные элементы по дефису, после чего элементы преобразуются в числа `start` и `end` с помощью `int ()`.
7. Генерируются числа между `start` и `end` с помощью функции `range ()` и добавляются в список `result` посредством метода `extend ()`.
8. После обработки всех интервалов возвращается окончательный список `result`, содержащий все развернутые интервалы.

В листинге 3.4 приводится код на Python, выполняющий развертывание целочисленных интервалов.

Листинг 3.4. Развертывание целочисленных интервалов

```
def Expanding_Integer_Intervals(intervals):
    '''
    Если входная строка пустая,
    то вернуть пустой список
    '''
    if intervals == '':
        return []
    '''
    создать пустой список для хранения
    развернутых интервалов
```

```
    '''
    result = []
    '''
    разбить входную строку
    на интервалы
    '''
    intervals_list = intervals.split(',')

    for interval in intervals_list:
        '''
        Проверить, содержит ли интервал
        только одно число
        '''
        if '-' not in interval:
            '''
            преобразовать его в целое число
            и добавить в список result
            '''
            result.append(int(interval))

        else:
            # Если интервал содержит диапазон чисел,
            # то извлечь начало и конец диапазона
            start_end = interval.split('-')
            start = int(start_end[0])
            end = int(start_end[1])
            '''
            сгенерировать все числа между start
            и end и добавить их в список result
            '''
            '''
            range(x,y)---range(x,y-1),
            поэтому (x,y+1), чтобы не упустить последний элемент
            '''
            result.extend(range(start , end + 1))
    '''
    Вернуть получившийся список
    с развернутыми интервалами
    '''
    return result
```

3.5. Свертывание целочисленных интервалов

Эта задача является обратной к предыдущей задаче. Напишите функцию, которая принимает список чисел и возвращает строку интервалов в форме «первый–последний», разделенных запятыми.

В табл. 3.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.5. Некоторые ожидаемые результаты для задачи свертывания целочисленных интервалов

Items	Ожидаемый результат
[1, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 120, 121, 122, 123, 124, 568]	'1, 7–16, 120–124, 568'
[2, 3, 4, 5, 6, 7, 8, 9, 10]	'2–10'
[2, 3, 4, 5, 19]	'2, 3–5, 19'
[17]	17

Алгоритм

Алгоритм решения этой задачи основан на правилах. Фактически он перебирает список целых чисел и определяет диапазоны последовательных целых чисел.

1. Принимается список положительных целых чисел.
2. Инициализируются переменные `start` и `end` значением `None`.
3. Для каждого целочисленного элемента `item` во входном списке `items` выполняются следующие действия:
   а) если `start` имеет значение `None`, то обеим переменным, `start` и `end`, присваивается значение `item`;
   б) если `item` равен `end + 1`, то переменной `end` присваивается значение `item`;
   в) иначе текущий получившийся диапазон добавляется в список `ranges`, причем если значение `start` равно значению `end`, то добавляется `str (start)`, иначе добавляется `str (start) + '.' + str (end)`;
   г) затем начинается свертывание нового диапазона, для чего обеим переменным, `start` и `end`, присваивается значение `item`.
4. Если переменная `start` имеет значение, отличное от `None`, то заключительный диапазон добавляется в `ranges` с использованием той же логики, что и на предыдущем шаге.
5. Диапазоны объединяются через запятую, и полученная строка возвращается.

В листинге 3.5 приводится код на Python, выполняющий свертывание целочисленных интервалов.

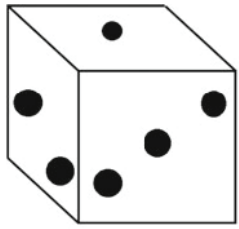
Листинг 3.5. Свертывание целочисленных интервалов

```
def collapse_integer_intervals(items):
    ranges = []
    start = end = None

    for item in items :
        if start is None:
            # Инициализировать новый диапазон
            start = end = item
        elif item == end + 1:
            # Продолжить текущий диапазон
            end = item
        else:
            # Добавить текущий диапазон в список
            if start == end:
                ranges.append(str(start))
            else:
                ranges.append(str(start) + '-' + str(end))

            # Инициализировать новый диапазон
            start = end = item

    # Добавить последний диапазон в список
    if start is not None:
        if start == end:
            ranges.append(str(start))
        else:
            ranges.append(str(start) + '-' + str(end))

    # Объединить диапазоны в строку через запятую
    return " ,".join(ranges)
```

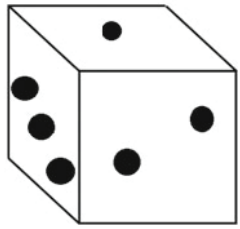
3.6. Левосторонний игральный кубик

Рассмотрим игральные кубики, имеющие шесть граней с точками от одной до шести и восемь углов, причем если смотреть с каждого угла, то видны три грани. Левосторонние и правосторонние кубики выглядят по-разному. Например, на рис. 3.1 (б) изображен левосторонний кубик, а на рис. 3.1 (а) – правосторонний. Чтение видимых граней производится слева направо (по часовой стрелке). Для левостороннего кубика числа

читаются как 1, 2, 3. Если повернуть левосторонний кубик против часовой стрелки, то, следуя тому же правилу, мы прочитаем 2, 3, 1, если повернуть еще раз, то 3, 1, 2. Следовательно, при взгляде на угол левостороннего кубика, объединяющий грани 1, 2, 3, мы имеем три перестановки 1, 2, 3; 2, 3, 1 и 3, 1, 2. Кубик имеет восемь разных углов, и для каждого возможны три перестановки, поэтому общее число перестановок равно $8 \times 3 = 24$. То же верно и для правостороннего кубика. Всего имеется $24 + 24 = 48$ перестановок для левостороннего и правостороннего кубиков вместе. Ваша задача: написать функцию на Python, которая принимает угол и определяет, является кубик левосторонним или правосторонним. Если кубик левосторонний, то функция должна вернуть *True*, в противном случае – *False*.



(а) Правосторонний



(б) Левосторонний

Рис. 3.1. Примеры прово-и левосторонних кубиков

В табл. 3.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.6. Некоторые ожидаемые результаты для задачи определения типа кубика, лево- или правостороннего

n	Ожидаемый результат
(4, 2, 1)	True
(6, 3, 2)	True
(6, 5, 4)	False

Алгоритм

Количество перестановок чисел в левостороннем (и правостороннем) кубике невелико, достаточно найти 24 левосторонних угла, и если входные данные совпадают с рассматриваемым углом, то следует вернуть `True`, в противном случае – `False`.

В листинге 3.6 приводится код на Python, проверяющий, является ли кубик левосторонним.

Листинг 3.6. Проверка, является ли кубик левосторонним

```
 1 def is_left_handed_dice(pips):
 2     Left_Handed = [(1, 2, 3), (3, 1, 2), (2, 3, 1),
 3                    (1, 4, 2), (2, 1, 4), (6, 3, 2),
 4                    (4, 2, 1), (5, 6, 4), (4, 5, 6),
 5                    (6, 4, 5), (3, 6, 5), (5, 3, 6),
 6                    (6, 5, 3), (5, 4, 1), (1, 5, 4),
 7                    (4, 1, 5), (2, 4, 6), (6, 2, 4),
 8                    (4, 6, 2), (1, 3, 5), (5, 1, 3),
 9                    (3, 5, 1), (2, 6, 3), (3, 2, 6)]
10
11     if pips in Left_Handed:
12         return True
13     else:
14         return False
```

3.7. Очки за повторяющиеся числа

Представьте, что вы идете по улице и встречаете два такси. Одно с номером 1729, другое с номером 6666. Очевидно, что число 6666 легче запомнить благодаря повторению цифр в последовательности. Человеческий разум легче запоминает числа с повторяющимися цифрами. Например, люди проще запоминают показания часов 12:12, чем показания, не имеющие повторяющихся цифр, например 10:23. Целью этого задания является привлечение внимания к последовательности чисел, в которых есть повторяющиеся цифры. Последовательности оцениваются по количеству повторяющихся цифр. Если в последовательности есть две повторяющиеся цифры, то ей выставляется оценка 1. Если в последовательности три повторяющиеся цифры, то она получает оценку 10. Если в последовательности четыре повторяющиеся цифры, то ей начисляется 100 очков и т.д. Более того, если последовательность заканчивается последовательностью повторяющихся цифр, то полученное количество очков удваивается. Например, число 12333 имеет повторяющиеся цифры и заканчивает-

ся последовательностью повторяющихся цифр. Его оценка вычисляется по формуле 3.1.

$$score = \begin{cases} 10^{k-z} & \text{Есть последовательность повторяющихся цифр} \\ 2 \times 10^{k-z} & \text{Число заканчивается последовательностью повторяющихся цифр} \end{cases} . \quad (3.1)$$

Напишите функцию, которая принимает положительное целое число n, и если оно содержит последовательность повторяющихся цифр, то за каждую такую последовательность число получает $10^{(k-2)}$ очков, а если n заканчивается последовательностью повторяющихся цифр, то оценка за эту последовательность умножается на 2, т. е. $2 \times 10^{(k-2)}$.

В табл. 3.7 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.7. Некоторые ожидаемые результаты для задачи начисления очков за последовательности повторяющихся цифр в числе

n	Ожидаемый результат
1233	2
1569	0
17777	200
588885515555	301

Алгоритм

Алгоритм принимает целое число, перебирает цифры во входном числе и проверяет, совпадает ли текущая цифра с предыдущей цифрой. Если совпадает, то алгоритм увеличивает счетчик на количество повторяющихся цифр. Если отличается, то алгоритм подсчитывает количество повторяющихся цифр на данный момент и сбрасывает счетчик для новой цифры. Наконец, алгоритм подсчитывает все оставшиеся повторяющиеся цифры в конце числа.

Вот подробное описание шагов алгоритма.

1. Принимается целое число `n`.
2. Переменная `score` инициализируется нулем.
3. Переменная `count` инициализируется единицей.
4. Переменная `prev_digit` инициализируется значением `None`.
5. Число `n` преобразуется в строку, и выполняется перебор цифр:

а) если текущая цифра совпадает с предыдущей (`prev_digit`), то увеличивается переменная `count`;

б) если текущая цифра отличается от предыдущей цифры, то:

в) если `count` больше `1`, то переменная `score` увеличивается на $10^{(count-2)}$. Получается оценка, основанная на количестве повторяющихся цифр;

г) переменной `count` присваивается `1` для новой цифры;

д) переменной `prev_digit` присваивается текущая цифра для проверки наличия повторяющейся последовательности.

6. Если повторяющиеся цифры последние в числе, то к оценке `score` прибавляется $2 \times 10^{(count-2)}$. Это дает дополнительные очки за повторяющиеся цифры.
7. Полученная оценка возвращается.

В листинге 3.7 приводится код на Python, вычисляющий оценку за повторяющиеся числа.

Листинг 3.7. Оценка за повторяющиеся числа

```
def duplicate_digit_score(n):
    score = 0
    # Инициализировать prev_digit значением None
    count = 1
    prev_digit = None

    '''
    обход цифр в
    числе n
    '''
    for digit in str(n):
        '''
        Сравнить текущую цифру
        с предыдущей
        '''
        if prev_digit == digit:
            # Увеличить счетчик повторяющихся цифр
            count += 1
        else:
            '''
            если предыдущая цифра повторялась несколько раз,
            то вычислить оценку
            '''
```

```
            if count > 1:
                # Вычислить оценку за повторяющиеся цифры
                score += 10 ** (count - 2)
                # Сбросить счетчик для новой цифры
                count = 1
        '''
        Обновить prev_digit
        для следующей итерации
        '''
        prev_digit = digit

    '''
    Вычислить оценку для повторяющихся цифр
    в конце числа
    '''
    if count > 1:
        '''
        Удвоить оценку
        за повторяющиеся цифры в конце
        '''
        score += 2 * 10 ** (count - 2)
    return score
```

3.8. Первое меньшее число

Пусть есть массив, заполненный положительными целыми числами. В этой задаче вы должны для каждого числа найти первое меньшее число в массиве, и если меньшего числа нет, то число должно остаться неизменным. Например, рассмотрим массив *array* = [2, 77, 13, 1]. Первое число равно 2, и слева от него нет никакого другого числа, поэтому рассматривается часть массива справа. Справа есть число 1, которое меньше 2, поэтому [2, 77, 13, 1] → [1]. Следующее число равно 77. Слева от него имеется число 2, а справа – число 1. Число 1 меньше, но 2 – это первое меньшее число, поэтому [2, 77, 13, 1] → [1, 2]. Следующее число – 13, и первое меньшее число – 1, поэтому [2, 77, 13, 1] → [1, 2, 1]. Следующее число – 1. Меньше его нет чисел в массиве, поэтому оно перекочевывает в результат: [1, 2, 1, 1].

Ваша задача: написать функцию, которая принимает массив положительных целых чисел и возвращает массив чисел, каждый элемент которого является первым числом, меньше соответствующего числа во входном массиве.

В табл. 3.8 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.8. Некоторые ожидаемые результаты для задачи выбора первых меньших чисел

Array	Ожидаемый результат
[8, 6, 16, 1921, 17]	[6, 6, 6, 16, 16]
[2, 3, 4, 871]	[2, 2, 3, 4]
[1,77,7, 770, 700, 11]	[1, 1, 1, 7, 11, 7]
[6, 8, 9, 888, 1401, 1402]	[6, 6, 8, 9, 888, 1401]

Алгоритм

В алгоритме предусмотрены вложенные циклы, перебирающие все элементы входного массива и окружающих его элементов. Алгоритм находит первый меньший элемент для каждого элемента входного массива, сравнивая элемент из входного массива с соседними элементами слева и справа от него. Если любой из элементов слева или справа меньше текущего, то он выбирается как первый меньший элемент. Алгоритм также обрабатывает случаи, когда для текущего элемента в массиве нет меньшего элемента.

В листинге 3.8 приводится код на Python, отыскивающий первые меньшие числа.

Листинг 3.8. Поиск первых меньших чисел

```
def Nearest_first_smaller_number(array):
    n = len(array)
    nearest_smaller = []

    # Цикл по элементам входного массива
    for x, current_element in enumerate(array):

        '''
        Цикл по элементам
        слева и справа от текущего
        '''
        for y in range(1, n):

            '''
            Получить элемент слева
            от текущего (или выбрать
            текущий элемент, если слева нет элементов)
```

```
            '''
            if x >= y:
                left = array[x - y]

            else:
                left = current_element

            '''
            Получить элемент справа
            от текущего (или выбрать
            текущий элемент, если справа нет элементов)
            '''

            if x + y < n:
                right = array[x + y]
            else:
                current_element

            '''
            Если слева или справа есть элемент,
            который меньше текущего,
            то добавить его
            в список nearest_smaller
            '''
            if left < current_element \
               or right < current_element:
                nearest_smaller.append \
                    (left if left < right else right)
                break

            '''
            Если нет элементов слева
            или справа меньше текущего,
            то добавить текущий элемент
            в список nearest_smaller
            '''
        else :
            nearest_smaller.append(current_element)

return nearest_smaller
```

3.9. Первый объект, предшествующий *k* меньшим объектам

В этой задаче дается список объектов, и вы должны найти и вернуть объект, которому предшествует как минимум *k* меньших объектов. Если такого элемента нет, то функция должна вернуть значение *None*. Например, рассмотрим *items* = [2, 1, 9, 14] и *k* = 2. Какому числу предшествуют как минимум *k* меньших чисел? Так как числу 9 предшествуют числа 1 и 2, то 9 является ответом. Если *items* = [4, 2, 1, 9, 14] и *k* = 2, то ответ снова равен 9, поскольку по условию задачи искомому числу должно предшествовать не меньше *k* меньших чисел. В качестве другого примера рассмотрим массив строк: *items* = [«cobol», «ruby», «c++», «python», «c++», «php»] и *k* = 2. Какой строке предшествуют хотя бы *k* меньших строк? Для начала нужно определить длины строк, а затем выполнить те же операции, что и в предыдущем примере. Ответом на вопрос во втором примере будет строка «python», так как ее длина равна шести и перед ней стоят более короткие строки «c++», «cobol» и «ruby».

В табл. 3.9 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.9. Некоторые ожидаемые результаты для задачи поиска первого объекта, предшествующего ***k*** меньшим объектам

Items, k	Ожидаемый результат
[4,2, 1, 9, 14], 3	9
[2, 3, 4, 871], 4	None
[700, 3, 900, 400, 1100], 2	900
['cobol','ruby','c++','python','c','php'], 2	'python'

Алгоритм

Алгоритм просматривает каждый элемент в списке `items` и проверяет, имеются ли перед ним хотя бы `k` меньших элементов. Для этого алгоритм создает подсписок всех элементов перед текущим элементом, которые меньше его. Это делается с использованием генератора списков. После создания подсписка его длина сравнивается с `k`. Если длина больше или равна `k`, то алгоритм возвращает текущий элемент как первый в списке, которому предшествует не менее `k` меньших элементов. Если такого элемента не существует, то алгоритм возвращает `None`.

В листинге 3.9 приводится код на Python, отыскивающий первый объект, которому предшествует не менее `k` меньших объектов.

Листинг 3.9. Поиск первого объекта, которому предшествует не менее k меньших объектов

```
def first_preceded_by_k_smaller_number(items, k=1):
    '''
    Цикл по элементам в списке items
    и их индексам.
    '''
    for i, current_number in enumerate(items):

        '''
        Создать список элементов слева от текущего,
        которые меньше его.
        '''
        smaller_numbers = \
            [n for n in items[:i] if n < current_number]

        '''
        Если имеется не меньше k меньших элементов,
        предшествующих текущему, то вернуть текущий элемент.
        '''
        if len(smaller_numbers) >= k:
            return current_number

    '''
    Если нет элемента, удовлетворяющего
    условиям, то вернуть None.
    '''
    return None
```

3.10. Поиск n-го члена последовательности Калкина–Уилфа

Дерево Калкина–Уилфа – это корневое двоичное дерево, вершины которого находятся во взаимно однозначном соответствии с положительными рациональными числами. Корню дерева соответствует число 1, и для любого рационального числа a/b его левый дочерний элемент соответствует числу $a/(a+b)$, а правый дочерний элемент – числу $(a+b)/b$. Каждое рациональное число встречается в дереве ровно один раз. При выполнении обхода дерева Калкина–Уилфа по уровням генерируется последовательность рациональных чисел, известная как последовательность Калкина–Уилфа. Ваша задача: определить функцию, которая принимает положительное

целое число *n* и возвращает *n*-й член последовательности Калкина–Уилфа, который является рациональным числом. Рассмотрим следующий пример. Если *n* = 10, то результатом функции должно быть число. Процесс поиска ответа описан ниже.

В табл. 3.10 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.10. Некоторые ожидаемые результаты для задачи поиска первого объекта, предшествующего k меньшим объектам

n	Ожидаемый результат
11	5/2
2022	32/45
1993	59/43
24	2/7
10	3/5

Алгоритм

Для решения задачи используется поиск в ширину (Breadth-First Search, BFS) – алгоритм обхода графа, генерирующий последовательность Калкина–Уилфа до *n*-го члена. Для хранения узлов последовательности используется структура данных, называемая очередью. Алгоритм генерирует каждый член последовательности, вычисляя левые и правые дочерние узлы текущего узла и добавляя их в очередь. В конце возвращается *n*-й член последовательности. Вот подробное описание шагов алгоритма.

1. Принимается целое положительное число `n`.
2. Значение `n` уменьшается на `1`.
3. Создается пустая очередь `queue`.
4. В очередь добавляется корневой узел `(1, 1)`.
5. Затем запускается цикл, выполняющий следующие шаги `n` раз:
   а) из очереди извлекается следующий узел, и его числитель и знаменатель присваиваются переменным `parent_num` и `parent_denom` соответственно;
   б) вычисляется левый дочерний элемент текущего узла установкой его числителя равным `parent_num`, а знаменателя – равным `parent_num + parent_denom`;
   в) вычисляется правый дочерний элемент текущего узла установкой его числителя равным `parent_num + parent_denom`, а знаменателя – равным `parent_num`;

г) левый и правый дочерние узлы добавляются в очередь queue.

6. Из очереди извлекается последний элемент, и его числитель и знаменатель присваиваются переменным final_num и final_denom соответственно.
7. Если знаменатель последнего узла равен 1, то возвращается его числитель как целое число.

В листинге 3.10 приводится код на Python, отыскивающий *n*-й член последовательности Калкина–Уилфа.

Листинг 3.10. Поиск n-го члена последовательности Калкина–Уилфа

```
def nth_term_calkin_wilf(n):
    # Функция, проверяющая отсутствие элементов в очереди
    def is_empty(q):
        return len(q) == 0

    # Функция, добавляющая элемент в очередь
    def enqueue(q, item):
        q.append(item)
        return q

    # Функция, извлекающая элемент из очереди
    def dequeue(q):
        if not is_empty(q):
            return q.pop(0)

    '''
    Создать очередь с корневым узлом
    (1/1) и уменьшить n на 1
    '''
    n -= 1
    queue = []
    queue = enqueue(queue, (1 , 1))

    '''
    Цикл генерирования последовательности Калкина-Уилфа
    до n-го члена с использованием алгоритма BFS
    '''
    for _ in range(n):
        # Извлечь следующий узел из очереди
```

```
30          parent_num, parent_denom = dequeue(queue)
31          '''
32          Вычислить левый и правый
33          дочерние узлы
34          '''
35          left_child_num = parent_num
36          left_child_denom = parent_num + parent_denom
37          right_child_num = parent_num + parent_denom
38          right_child_denom = parent_denom
39          '''
40          Добавить в очередь левый и правый
41          дочерние элементы
42          '''
43          queue = enqueue\
44              (queue, (left_child_num, left_child_denom))
45          queue = enqueue\
46              (queue, (right_child_num, right_child_denom))
47
48      # Извлечь из очереди последний элемент
49      final_num, final_denom = dequeue(queue)
50
51      # Вернуть n-й член последовательности Калкина-Уилфа
52      if final_denom == 1:
53          return final_num
54      else:
55          return str(final_num) + '/' + str(final_denom)
```

3.11. Поиск и обращение восходящих подмассивов

В этой задаче дается массив положительных целых чисел и требуется найти обратные возрастающие подмассивы. Каждый подсписок должен содержать только строго увеличивающиеся элементы. Например, для входного массива [5, 7, 220, 33, 2, 6, 8, 1, 45] на первом этапе образуются подмассивы: [5, 7, 220], [33], [2, 6, 8], [1, 45], а на следующем этапе содержимое подмассивов переворачивается в обратном порядке: [220, 7, 5], [33], [8, 6, 2], [45, 1]. Затем перевернутые подмассивы объединяются в исходном порядке. Следовательно, [5, 7, 220, 33, 2, 6, 8, 1, 45] → [220, 7, 5, 33, 8, 6, 2, 45, 1].

В табл. 3.11 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.11. Некоторые ожидаемые результаты для задачи обращения восходящих подмассивов

Arrays	Ожидаемый результат
[6, 3, 2]	[6, 3, 2]
[5, 7, 220, 33, 2, 6, 8, 1, 45]	[220, 7, 5, 33, 8, 6, 2, 45, 1]
[1, 5, 7, 9, 90, 13, 11, 23]	[90, 9, 7, 5, 1, 13, 23, 11]
[44, 33, 57, 36, 38, 900]	[44, 57, 33, 900, 38, 36]

Алгоритм

В алгоритме используется метод двух указателей, где указатель `i` указывает на начало подсписка, а указатель `j` используется для поиска конца восходящего подсписка. Отыскав восходящий подмассив, программа «вырезает» подмассив, располагает элементы подсписка в обратном порядке и добавляет его в итоговый список. Затем она передвигает указатель `i` в начало следующего подмассива, и процесс повторяется до тех пор, пока не будет достигнут конец списка.

В листинге 3.11 приводится код на Python, обращающий восходящие подмассивы.

Листинг 3.11. Поиск и обращение восходящих подмассивов

```
def reverse_ascending_subarrays(List_Numbers):
    '''
    Создать пустой список для хранения
    восходящих подсписков
    '''
    ascending_subarray = []
    # Установить начальный индекс в 0
    i=0
    # Цикл по списку
    while i < len(List_Numbers):
        # Установить j в текущий индекс
        j = i
        # Извлечь восходящий подмассив
        while j < len(List_Numbers) - 1 \
              and List_Numbers[j] < List_Numbers[j + 1]:
            j += 1
        '''
        Обратить и добавить подсписок
        в список с результатом
        '''
```

```
        ascending_subarray.extend\
            (List_Numbers[i:j + 1][::-1])
        # Установить начальный индекс в начало следующего подмассива
        i = j + 1
    return ascending_subarray
```

3.12. Наименьшие целые степени

Большие целые степени могут усложнить вычисления с точки зрения времени и потребляемого объема памяти. Цель этой задачи: найти наименьшие положительные целые числа *x* и *y*, такие, что a^x и b^y находятся в пределах определенного допуска друг друга. В данном контексте под «допуском» понимается максимально допустимая разность между двумя целочисленными степенями *a* и *b*. Ваша задача: написать функцию, которая принимает *a*, *b* и *tolerance* и возвращает наименьшие положительные целые числа *x* и *y*.

В табл. 3.12 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.12. Некоторые ожидаемые результаты для поиска наименьших целых степеней

a, b, tolerance	Ожидаемый результат
12, 16, 10	(19, 17)
32, 40, 99	(248, 233)
43, 33, 73	(66, 71)

Алгоритм

Для решения этой задачи мы используем «жадный» подход. Алгоритм пытается найти наименьшую пару положительных целых чисел *x* и *y*, соответствующих заданному допуску. Для этого он последовательно увеличивает значения *x* и *y*, пока разность между степенями a^x и b^x не станет меньше или равна заданному допуску. Этот не гарантирует получение оптимального решения, но находит достаточно хорошее решение за разумное время.

1. Принимаются целые числа `a` и `b`.
2. Переменным `x` и `y` присваивается `1`.
3. Переменным `a_pow` и `b_pow` присваиваются значения `a` и `b` соответственно.
4. Запускается бесконечный цикл:

а) если a_pow больше, чем b_pow, то допуск tolerance сравнивается с результатом деления b_pow на разность между a_pow и b_pow;

б) если допуск меньше, то возвращаются x и y;

в) иначе y увеличивается на 1 и b_pow умножается на b;

г) если a_pow меньше, чем b_pow, то допуск tolerance сравнивается с результатом деления a_pow на разность между b_pow и a_pow;

д) если допуск меньше, то возвращаются x и y;

е) в противном случае x увеличивается на 1 и a_pow умножается на a;

ж) если a_pow и b_pow равны, то возвращаются x и y.

В листинге 3.12 приводится код на Python, отыскивающий наименьшие целые степени.

Листинг 3.12. Поиск наименьших целых степеней

```
def smallest_integer_powers(a, b, tolerance=100):
    x, y = 1, 1
    a_pow, b_pow = a, b

    while True:
        if a_pow > b_pow:
            if tolerance <= b_pow / (a_pow - b_pow):
                return x, y
            y += 1
            b_pow *= b
        elif a_pow < b_pow:
            if tolerance <= a_pow / (b_pow - a_pow):
                return x, y
            x += 1
            a_pow *= a
        else:
            return x, y
```

3.13. Сортировка циклов в графе

Цикл в графе – это путь, который начинается и заканчивается в одной и той же вершине. Рассмотрим массив *G*, заполненный положительными целыми числами. Каждое число в *G* является вершиной графа. Цель этой задачи состоит в том, чтобы найти циклы в *G* и отсортировать их в некотором порядке. Например, пусть *G* = [2, 4, 6, 5, 3, 1, 0] и нужно найти циклы такие, в которых большее число образует первую вершину; вершины, позиции или индексы которых находятся после позиции с наибольшим

числом, вставляются после позиции с наибольшим числом; а уже за этими вершинами вставляются вершины, позиции или индексы которых находятся перед позицией с наибольшим числом. Далее необходимо отсортировать все циклы в порядке возрастания.

Рассмотрим порядок решения задачи для нашего примера. Первая вершина в *G* содержит число 2, какое число находится в индексе 2? Поскольку индексация в Python начинается с нуля, мы получаем число 6. Следующий шаг: какое число находится в индексе 6? Это число 0. Следующий шаг: какое число находится в индексе 0? Это число 2, следовательно, мы нашли цикл $2 \rightarrow 6 \rightarrow 0 \rightarrow 2$. Вставляем в список первый цикл, *C* = [[2, 6, 0]]. Первая вершина, которую мы не посещали,– это число 4. Она считается первой вершиной следующего цикла. Следующий шаг: какое число стоит в индексе 4? Это число 3. Следующий шаг: какое число находится в индексе 3? Это число 5. Следующий шаг: какое число находится в индексе 5? Это число 1. Следующий шаг: какое число находится в индексе 1? Это число 4. Найден еще один цикл, $4 \rightarrow 3 \rightarrow 5 \rightarrow 1 \rightarrow 4$. Добавляем его в список циклов *C* = [[2, 6, 0], [4, 3, 5, 1]]. Теперь в *G* не осталось вершин, которые мы не посетили, поэтому выполняем процесс сортировки. Для [2, 6, 0] наибольшее число (вершина) равно 6, поэтому [2, 6, 0] → [6], 0 помещаем после 6, поэтому [6] → [6, 0], а 2 предшествовало 6, поэтому [6, 0] → [6, 0,2]. Получаем обновленные циклы: *C* = [[6, 0, 2], [4, 3, 5, 1]]. Во втором цикле [4, 3, 5, 1] наибольшее число равно 5, поэтому [4, 3, 5, 1] → [5]. За числом 5 в цикле следует 1, поэтому 1 вставляется после 5, а перед 5 мы имеем 4 и 3, поэтому добавляем 4 и 3 после 5 и 1. В результате получаем циклы *C* = [[6, 0, 2], [5, 1, 4, 3]]. На следующем шаге циклы сортируются по возрастанию, и получается *C* = [[6, 0, 2], [5, 1, 4, 3]] → *C* = [[5, 1, 4, 3], [6, 0, 2]]. На последнем этапе вложенные массивы форматируются и преобразуются в единый плоский массив: *C* = [5, 1, 4, 3, 6, 0, 2].

Ваша задача: написать функцию, которая принимает массив *G* и возвращает отсортированные циклы.

В табл. 3.13 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.13. Некоторые ожидаемые результаты для задачи поиска и сортировки циклов в графе

G	Ожидаемый результат
[1, 2, 4, 5, 3, 0]	[5, 0, 1, 2, 4, 3]
[1, 2, 3, 4, 0]	[4, 0, 1, 2, 3]
[4, 3, 7, 0, 1, 5, 2, 6]	[4, 1, 3, 0, 5, 7, 6, 2]
[1, 3, 5, 0, 2, 4]	[3, 0, 1, 5, 4, 2]

Алгоритм

Общий алгоритм основан на использовании двух алгоритмов: поиска в глубину (Depth-First Search, DFS) и сортировки. DFS используется для поиска всех циклов в ориентированном графе. Этот алгоритм посещает каждую вершину в графе только один раз и исследует исходящие ребра из вершин, пока не достигнет конца графа или не найдет цикл. Как только цикл будет найден, он добавляется в список циклов, и алгоритм переходит к первой непосещавшейся вершине. Алгоритм сортировки сортирует каждый цикл в списке циклов в соответствии с определенными критериями. В нашем случае первым помещается наибольший элемент цикла, а затем остальные элементы в порядке возрастания. После сортировки циклов они объединяются в общий список. Наконец, отсортированный плоский список возвращается в виде результата.

В листинге 3.13 приводится код на Python, отыскивающий и сортирующий циклы в графе.

Листинг 3.13. Поиск и сортировка циклов в графе

```
def insertion_sort(arr):
    '''
    Сортирует массив с применением
    алгоритма сортировки вставками
    '''
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
def Sorting_Cycles(graph):
    # Отыскивает циклы в ориентированном графе

    flat_cycles = []
    cycles = []

    '''
    Алгоритм DFS для
    определения циклов
    '''
    def find_cycles(g):
```

```
        visited = set()
        for vertex in g:
            if vertex in visited:
                continue
            start = vertex
            current_cycle = [start]
            visited.add(start)
            neighbor = g[start]
            while neighbor != start:
                current_cycle.append(neighbor)
                visited.add(neighbor)
                neighbor = g[neighbor]
            cycles.append(current_cycle)
        return cycles

    '''
    Сортирует список циклов
    в соответствии с заданными критериями
    '''
    def sort_cycles(cycles):
        sorted_cycles = []
        for cycle in cycles:
            if len(cycle) == 1:
                sorted_cycles.append(cycle)
            else:
                highest = max(cycle)
                highest_index = cycle.index(highest)
                sorted_cycle = \
                    cycle[highest_index:] + cycle[:highest_index]
                sorted_cycles.append(sorted_cycle)
        sorted_cycles = insertion_sort(sorted_cycles)
        flat_sorted_cycles = \
            [vertex for cycle
                    in sorted_cycles for vertex in cycle]
        return flat_sorted_cycles

    # Вызов функций find_cycles и sort_cycles
    cycles = find_cycles(graph)
    flat_cycles = sort_cycles(cycles)

    # Вернуть плоский список циклов
    return flat_cycles
```

3.14. Получение представления числа в сбалансированной троичной системе

Мы знаем, что в любой системе счисления с основанием r используются цифры от 0 до $r - 1$. Пусть дано троичное число $n = 1022$. Преобразуем его в систему счисления с основанием 10: $2 \times 3^0 + 2 \times 3^1 + 0 \times 3^1 + 1 \times 3^3 = 35$. Это обычный способ преобразования троичного числа в десятичное. Существует другой способ преобразования числа из системы счисления с основанием 3 в систему с основанием 10. Рассмотрим следующий пример: $-1 \times 3^0 + 0 \times 3^1 + 1 \times 3^2 + 1 \times 3^3 = 35$. Второе представление называется представлением в сбалансированной троичной. В сбалансированной троичной системе счисления используются цифры 0, 1 и –1, тогда как в обычной троичной системе – цифры 0, 1 и 2. В этом задании нужно найти коэффициенты при цифрах в сбалансированной троичной системе счисления. Напишите функцию, которая принимает положительное целое число n и возвращает массив коэффициентов в сбалансированной тройной системе, таких, что их сумма равняется n. Обратите внимание, что коэффициенты в массиве должны следовать в порядке убывания их абсолютных значений, а знак перед элементом определяет цифру (1 или –1).

В табл. 3.14 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.14. Некоторые ожидаемые результаты для задачи получения чисел в сбалансированной троичной системе

n	Ожидаемый результат
25	[27,–3, 1]
28	[27, 1]
88	[81, 9,–3, 1]
1400	[2187,–729,–81, 27,–3,–1]

Алгоритм

Алгоритм принимает положительное целое число n. Чтобы получить коэффициент, алгоритм вычисляет и сохраняет остаток от деления n на 3. Затем он выбирает целую часть от $(n + 1)/3$ и снова вычисляет остаток. Этот процесс повторяется до тех пор, пока n не станет равно 0. На следующем шаге принимается, что p – это соответствующая позиция коэффициента. Для каждого p-го элемента в списке остатков алгоритм сравнивает его с 1. Если элемент равен 1, алгоритм вычисляет 1×3^p. Если элемент –1, алгоритм вычисляет -1×3^p. Если элемент равен 0, то никакие вычисления не выполняются.

В листинге 3.14 приводится код на Python, получающий представление числа в сбалансированной троичной системе.

Листинг 3.14. Получение представления числа в сбалансированной троичной системе

```
def Obtaining_Numbers_in_Balanced_Ternary_System(n):
    remainders = ""
    while n!=0:
        # Получение остатков
        '''
        В сбалансированной троичной системе используются цифры 0 ,1 и -1,
        но -1 - это два символа: '-' и '1'.
        Поэтому вместо '-1' используется '2'
        '''
        remainders = remainders + "012"[n % 3]
        '''
        Выражение [n % 3] дает число 0, 1 или 2,
        которое затем используется как индекс в строке "012".
        т. е.
        "012"[1] дает 1
        "012"[0] дает 0
        "012"[2] дает 2
        '''
        # Получение целой части
        n = ((n+1)//3)
    Coefficients = remainders
    Storage_Array = []
    '''
    Коэффициенты умножаются на
    соответствующие степени троек
    '''
    '''
    0 не учитывается, так как умножение на 0
    дает 0
    '''
    for i in range(len(Coefficients) - 1, -1, -1):
        '''
        Цикл выполняется в обратном порядке,
        потому что позиции перебираются
        от больших
        к меньшим.
        '''
        if Coefficients[i] == '1':
```

```
            Storage_Array.append(3 ** i)
            '''
            В сбалансированной троичной системе нет цифры 2, если коэффициент
            равен 2, то в нашем случае это означает, что он равен -1.
            '''
        elif Coefficients[i] == '2':
            Storage_Array.append(-1 * (3 ** i))
    return Storage_Array
```

3.15. Строгое возрастание

Цель этой задачи: проверить, расположены ли элементы в списке строго по возрастанию. Важно отметить, что список с повторяющимися числами не может считаться строго возрастающим. Чтобы реализовать решение, обратите внимание на следующие моменты:

- на вход задачи всегда будет передаваться список чисел;
- список, содержащий только один элемент, положительный или отрицательный, считается строго возрастающим;
- в строго возрастающем списке не может быть повторяющихся чисел.

В табл. 3.15 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.15. Некоторые ожидаемые результаты для задачи проверки строгого возрастания чисел в списке

Items	Ожидаемый результат
[−6]	True
[1, 2, 4, 7, 1890]	True
[−3,−2, 15, 16, 2000]	True
[−3,−2, 159, 18, 915]	False

Алгоритм

Алгоритм, реализованный в этой задаче, предполагает сверку каждого числа во входном списке с предыдущим числом. Если окажется, что какое-то число меньше или равно предыдущему, то алгоритм вернет *False*, сообщая, что список не является строго возрастающим. И наоборот, если все числа строго больше своих предшественников, то алгоритм вернет *True*, сообщая, что список действительно строго возрастающий.

В листинге 3.15 приводится код на Python для проверки строгого возрастания чисел в списке.

Листинг 3.15. Проверка строгого возрастания чисел в списке

```
def Is_Strictly_Ascending(items):
    '''
    Просто сравнивает каждый элемент с предшествующим
    '''
    '''
    Первое число рассматривать как
    первое предшествующее
    '''
    previous = items[0]
    templist = []
    for num in items:
        # Если условие "строго больше" не выполняется, то вернуть False
        if num < previous:
            return False
        # Если условие "не повторяются" не выполняется, то вернуть False
        if num in templist:
            return False
        # обновить предыдущее число
        previous = num
        '''
        Если элемент присутствует в templist,
        значит, это дубликат
        и нужно вернуть False
        '''
        templist.append(num)
    return True
```

3.16. Сортировка по приоритетам

В этом задании даются список и множество. Оба содержат положительные и отрицательные целые числа. Ваша задача: написать функцию, которая принимает эти две структуры данных и сортирует список с учетом приоритетных элементов, указанных в множестве. В частности, функция должна сначала выявить элементы, присутствующие как в списке, так и в множестве, и добавить их в новый список в порядке возрастания. Затем она должна добавить оставшиеся элементы из исходного списка так же в порядке возрастания.

В табл. 3.16 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.16. Некоторые ожидаемые результаты для задачи сортировки по приоритетам

List, set	Ожидаемый результат
[5, 2,7, 3, 2, 1,29], {2, 7}	[2, 2, 7, 1, 3, 5, 29]
[1, 4, 8, 9, 13, 11, 17], {11, 12}	[11, 1, 4, 8, 9, 13, 17]
[4, 5, 7, 13, 290], {3, 4, 1, 8}	[4, 5, 7, 13, 290]
[4,–2, 5, 7, 13,–8, 9, 8], {2, 4, 1, 8}	[4, 8,–8,–2, 5, 7, 9, 13]

Алгоритм

Алгоритм выявляет минимальный элемент во входном списке и проверяет, присутствует ли он в множестве. Если присутствует, то алгоритм добавляет минимальный элемент в выходной список столько раз, сколько он появляется во входном списке, и удаляет его из входного списка, а также из множества, чтобы избежать дублирования. Этот процесс повторяется для всех элементов в множестве, что гарантирует сортировку списка в порядке приоритетов, определяемых множеством. После обработки всех элементов множества все оставшиеся элементы входного списка сортируются по возрастанию и добавляются в выходной список.

В листинге 3.16 приводится код на Python, выполняющий сортировку по приоритетам.

Листинг 3.16. Сортировка по приоритетам

```
def priority_sort(list, set):
    '''
    Сортирует входной список,
    основываясь на элементах множества.
    Если список пустой, возвращает пустой список.
    '''

    # Если список пустой, то вернуть пустой список.
    if not list:
        return list

    # Создать пустой список для хранения отсортированных элементов.
    sorted_list = []

    '''Создать объект диапазона
    от 0 до размера множества.
    '''
```

```
    iterate_count = range(len(set))

    # Цикл по диапазону.
    for i in iterate_count:
        '''Проверить присутствие во входном списке
        минимального элемента из множества.
        '''
        if min(set) in list:
            '''
            Если присутствует, выполнить цикл по элементам входного списка
            и добавить минимальный элемент в sorted_list столько раз,
            '''
            # Сколько он встретится во входном списке.
            for k in range(list.count(min(set))):
                sorted_list.append(min(set))
                '''Удалить все вхождения минимального
                элемента из входного списка.
                '''
                list.remove(min(set))
            '''
            Удалить минимальный элемент
            из множества.
            '''
            set.remove(min(set))
        else:
            '''
            Если минимальный элемент
            отсутствует во входном списке,
            то просто удалить его из множества.
            '''
            set.remove(min(set))

    '''
    Создать объект диапазона от 0
    до размера входного списка.
    '''
    iterate_count = range(len(list))

    # Цикл по диапазону.
    for i in iterate_count:
        # Отыскать минимальный элемент во входном списке.
        min_elem = min(list)
        # Добавить минимальный элемент в sorted_list.
```

```
61          sorted_list.append(min_elem)
62          # Удалить минимальный элемент из входного списка.
63          list.remove(min_elem)
64
65      # Вернуть отсортированный список.
66      return sorted_list
```

3.17. Сортировка положительных чисел с сохранением порядка отрицательных чисел

Это задание состоит в том, чтобы, получив массив целых чисел, отсортировать положительные числа и сохранить порядок отрицательных чисел.

Ваша задача: написать функцию, которая принимает массив положительных и отрицательных целых чисел и возвращает массив с той же длиной, в котором положительные числа отсортированы, а порядок отрицательных чисел сохраняется.

В табл. 3.17 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.17. Некоторые ожидаемые результаты для задачи сортировки положительных чисел с сохранением порядка отрицательных чисел

Array	Ожидаемый результат
[79, 7,–3,–4, 6,–1]	[6, 7,–3,–4, 79,–1]
[451, 419,–1001, 2, 3,–2]	[2, 3,–1001, 419, 451,–2]
[–39,–456, 0,–7,–3,–599]	[–39,–456,0,–7,–3,–599]
[]	[]

Алгоритм

Для сортировки положительных чисел во входном массиве в порядке возрастания используется пузырьковая сортировка, а отрицательные числа остаются на исходных позициях. Сначала из исходного массива удаляются все отрицательные числа и помещаются в словарь вместе со своими индексами. Оставшиеся положительные числа сортируются, а затем отрицательные числа вставляются обратно на исходные позиции с использованием индексов, хранящихся в словаре.

В листинге 3.17 приводится код на Python, выполняющий сортировки положительных чисел с сохранением порядка отрицательных чисел.

Листинг 3.17. Сортировка положительных чисел с сохранением порядка отрицательных чисел

```
def sort_positives_keep_negatives(array):
    # Функция, реализующая пузырьковую сортировку
    def bubblesort(array:list):
        for i in range(len(array)):
            flag = True
            # Цикл по элементам массива
            for j in range(len(array) - i - 1):
                '''
                Сравнить два соседних числа
                и поменять их местами при необходимости
                '''
                if array[j] > array[j + 1]:
                    array[j], array[j + 1] = \
                    array[j + 1], array[j]
                    flag = False
            if flag:
                return array
    # Словарь для хранения отрицательных элементов
    neg = {}
    for i in range(len(array)):
        # если текущий элемент отрицательный
        if array[i] < 0:
            '''
            добавить индекс:значение в
            словарь neg
            '''
            neg[i] = array[i]
    # Временно удалить отрицательные элементы
    array = [i for i in array if i >= 0]
    # Отсортировать положительные числа
    array = bubblesort(array)
    # Цикл по словарю
    for i, v in neg.items():
        '''
        Вставить отрицательные числа
        в соответствующие индексы
        '''
        array.insert(i , v)
    return array
```

3.18. Сортировка: сначала числа, потом символы

В этом задании дается список списков. Структура исходного списка должна сохраняться, но содержимое вложенных списков должно быть отсортировано так, чтобы первыми следовали числа, а затем символы. И числа, и символы должны сортироваться в порядке возрастания.

Ваша задача: написать функцию, которая принимает список списков, состоящих из чисел и символов, и возвращает список с той же структурой, но сортирует содержимое вложенных списков так, что сначала следуют числа, а затем символы – и те, и другие в порядке возрастания.

В табл. 3.18 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.18. Некоторые ожидаемые результаты для задачи сортировки чисел и символов

NestedList	Ожидаемый результат
[[0, 0, 0.5, 1], [3], [5, 5, 'X', 'Y'], ['Z', 'a'], ['b', 'er', 'f'], ['p', 's']]	[[0, 0, 0.5, 1], [3], [5, 5, 'X', 'Y'], ['Z', 'a'], ['b', 'er', 'f'], ['p', 's']]
[[2, 74, 3, 0, 0.5], [4, 7, 'a', 'b', 'e', 99]]	[[0, 0.5, 2, 3, 4], [7, 74, 99, 'a', 'b', 'e']]
[[419, 419, 't', 'r', 'pol'], ['x', 'y',88, 'r']]	[[88, 419, 419, 'pol', 'r'], ['r', 't', 'x', 'y']]
[[2, 3, 'u'], ['w', 100, 4, 5], ['r', 'd', 't', 1]]	[[1, 2, 3], [4, 5, 100, 'd'], ['r', 't', 'u', 'w']]

Алгоритм

Алгоритм сначала объединяет вложенные списки в один плоский список, затем отделяет числа от символов, сортирует числа и символы с помощью пузырьковой сортировки и, наконец, объединяет их обратно в отсортированный список списков с исходной структурой.

В листинге 3.18 приводится код на Python, сортирующий числа и символы.

Листинг 3.18. Сортировка чисел и символов

```
def number_then_character(NestedList):
    # Функция, реализующая пузырьковую сортировку
    def bubblesort(lst):
        for i in range(len(lst)):
            flag = True
            # Цикл по элементам массива
            for j in range(len(lst) - i - 1):
                if lst[j] > lst[j + 1]:
```

```
                    '''
                    Сравнить два соседних числа
                    и поменять их местами при необходимости
                    '''
                    lst[j], lst[j + 1] = lst[j + 1], lst[j]
                    flag = False
        if flag:
            # Если перестановка не выполнялась, значит, список отсортирован
            return lst
    # Объединить вложенные списки в один плоский список
    flattedlist = [i for lst in NestedList for i in lst]
    # Сохранить размеры списков
    sizes = []
    # Выполнить обход коллекции и сохранить размеры
    for lst in NestedList:
        sizes.append(len(lst))
    # числа: Извлечь числа (int/float)
    numbers = \
        [n for n in flattedlist if isinstance(n, (int, float))]
    # символы: Извлечь символы
    characters = \
        [str(l) for l in flattedlist if str(l).isalpha()]
    # Отсортировать числа
    numbers = bubblesort(numbers)
    # Отсортировать символы
    characters = bubblesort(characters)
    # Объединить два отсортированных массива
    flattedlist = numbers + characters
    # Для хранения конечного результата
    sortedlist = []
    for s in sizes:
        # Создать массив с размером s
        sortedlist.append(flattedlist[:s])
        '''
        Обновить плоский список после вставки
        предыдущих элементов в отсортированный список'''
        flattedlist = flattedlist[s:]
    return sortedlist
```

3.19. Сортировка дат

Целью этого задания является сортировка списка дат в формате DD-MM-YYYY_HH: MM в порядке возрастания или убывания. Критериями сорти-

ровки являются год, месяц, день, час и минута. Ваша задача: написать функцию, которая принимает список дат и времени (в виде строк) и направление сортировки и возвращает отсортированный список дат и времени.

В табл. 3.19 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.19. Некоторые ожидаемые результаты для задачи сортировки дат

times, sort_types	Ожидаемый результат
['09-02-2001_10:03', '10-02-2000_18:29', '01-01-1999_00:55'],'ASC'	['01-01-1999_00:55', '10-02-2000_18:29', '09-02-2001_10:03']
['01-04-2004_10:03', '10-02-2006_03:29', '01-01-2022_00:55'],'ASC'	['01-04-2004_10:03', '10-02-2006_03:29', '01-01-2022_00:55']
['01-04-2004_10:03', '10-02-2006_03:29', '01-01-2022_00:55'],'DSC'	['01-01-2022_00:55', '10-02-2006_03:29', '01-04-2004_10:03']
['09-02-2001_10:03', '10-02-2000_18:29', '01-01-1999_00:55'],'DSC'	['09-02-2001_10:03', '10-02-2000_18:29', '01-01-1999_00:55']

Алгоритм

Алгоритм преобразует строки с датами и временем в объекты даты и времени и сортирует их с использованием алгоритма пузырьковой сортировки. Затем объекты даты и времени преобразуются в строки.

В листинге 3.19 приводится код на Python, сортирующий даты.

Листинг 3.19. Сортировка дат

```
 1 import datetime
 2 def bubble_sort(arr):
 3     for i in range(len(arr) - 1):
 4         for j in range(0, (len(arr) - i) - 1):
 5             if arr[j] > arr[j + 1]:
 6                 # Поменять местами, если необходимо
 7                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
 8     return arr
 9 def sort_dates(times, sort_types):
10     time_objects = []
11     sorted_dates=[]
12     # Цикл по элементам списка
13     wanted_date = "%d-%m-%Y_%H:%M"
14     for t in times:
15
```

```
        time_objects.append(
            datetime.datetime.strptime(t, wanted_date))
    # Сортировать даты с помощью алгоритма пузырьковой сортировки
    time_objects = bubble_sort(time_objects)

    if sort_types.lower() == 'asc':
        for t in time_objects:
            # Преобразовать даты в строки
            sorted_dates.append(
                (datetime.datetime.strftime(t, wanted_date)))
        return sorted_dates
    elif sort_types.lower() == 'dsc':
        '''
        Так как изначально даты отсортированы в порядке возрастания,
        выполнить обход списка в обратном направлении.
        '''
        for t in time_objects[::-1]:
            # Преобразовать даты в строки
            sorted_dates.append(
                (datetime.datetime.strftime(t, wanted_date)))
    return sorted_dates
```

3.20. Сортировка по алфавиту и длине

Целью этого задания является сортировка слов в заданной строке сначала по алфавиту, а затем по длине.

Ваша задача: написать функцию, которая принимает строку и возвращает строку, в которой слова отсортированы по алфавиту и длине.

В табл. 3.20 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.20. Некоторые ожидаемые результаты для задачи сортировки слов в строке по алфавиту и длине

Sentence	Ожидаемый результат
'Year of the Tiger, is it fog'	'Tiger, Year fog the it of is'
'Python is one of the most used languages'	'languages Python most used the one of is'
'FIFA World Cup Qatar'	'Qatar World FIFA Cup'
'for if while def range else set'	'range while else def set for if'

Алгоритм

Алгоритм принимает строку с предложением, разбивает ее на слова, сортирует их в алфавитном порядке, используя алгоритм пузырьковой сортировки, а затем повторно сортирует их по длине. Для этого выполняется обход слов в отсортированном списке, и длина каждого слова сравнивается с длинами предшествующих слов. Если слово длиннее предшествующего, они меняются местами. Наконец, отсортированные слова объединяются в одну строку и возвращаются.

В листинге 3.20 приводится код на Python, сортирующий слова по алфавиту и длине.

Листинг 3.20. Сортировка слов по алфавиту и длине

```
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, (len(arr) - i) - 1):
            if arr[j] > arr[j + 1]:
                # Поменять местами, если необходимо
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
def sorted_by_alphabetical_and_length(sentence):
    words = sentence.split()
    # Отсортировать в алфавитном порядке
    words = bubble_sort(words)
    for word in range(0, len(words)):
        for i in range(0, word):
            # Поменять местами, если необходимо
            if len(words[word]) > len(words[i]):
                temp = words[i]
                words[i] = words[word]
                words[word] = temp

    '''
    Добавить отсортированные слова в sorted_sentence
    (в строку)
    '''
    sorted_sentence = ""
    for item in range(0, len(words)):
        sorted_sentence += words[item] + ' '
    return sorted_sentence
```

3.21. Сортировка чисел по количеству цифр

Стандартная операция сравнения чисел говорит нам, что число 101 больше числа 9. Однако в контексте сортировки по значениям цифр число 9 больше, чем 101. Целью этой задачи является сортировка массива чисел по количеству составляющих их цифр. Например, рассмотрим массив *array* = [81181972, 8111972], и пусть *a* = 81181972 и *b* = 8111972. Сравнение начинается с наибольшей цифры, в данном случае это 9. Оба числа имеют по одной цифре 9, поэтому сравнивается следующая по величине цифра – цифра 8. В *a* две восьмерки, а в *b* – одна, поэтому считается, что *a* больше. В общем случае сравнение выполняется следующим образом: если в *a* и *b* имеется одинаковое количество цифр с одним и тем же значением, то проверяется следующая цифра, меньшая по величине. Так продолжается до тех пор, пока не будет найдено отличие или пока не будут исчерпаны все цифры в числах.

Ваша задача: написать функцию, которая принимает массив положительных целых чисел и возвращает массив, отсортированный по количеству цифр.

В табл. 3.21 показаны ожидаемые результаты для некоторых входных данных.

Таблица 3.21. Некоторые ожидаемые результаты для задачи сортировки чисел по количеству цифр

Array	Ожидаемый результат
[81181972, 9, 123198776, 23, 456, 1]	[1, 23, 456, 9, 123198776, 81181972]
[1, 2, 3, 11, 9, 77, 66, 87, 111]	[1, 11, 111, 2, 3, 66, 77, 87, 9]
[39, 456, 0, 7, 3, 599]	[0, 3, 456, 7, 39, 599]
[]	[]

Алгоритм

Стратегия «разделяй и властвуй» – это метод решения задач, предполагающий деление сложной задачи на более мелкие, более управляемые подзадачи и решение каждой подзадачи отдельно. Алгоритм сортировки слиянием как раз основан на стратегии «разделяй и властвуй». Он многократно делит заданный список на подсписки, пока в каждом подсписке не останется только один элемент. Эти подсписки затем объединяются вместе, образуя отсортированный список. Алгоритм (сортировки слиянием) сначала рекурсивно делит входной массив на две половины, пока не будет достигнут базовый вариант с одним элементом или пустой массив. Затем он определяет, какой из двух элементов должен следовать пер-

вым, основываясь на количестве их цифр. Наконец, два отсортированных подмассива объединяются в порядке сортировки.

В листинге 3.21 приводится код на Python, сортирующий числа по количеству цифр.

Листинг 3.21. Сортировка чисел по количеству цифр

```
def sort_by_digit_count(arr):
    """
    Сортирует массив положительных целых чисел
    по количеству цифр.
    """
    # Применить алгоритм сортировки слиянием
    if len(arr) > 1:
        # Поиск середины массива
        mid = len(arr) // 2

        # Деление массива на две половины
        L = arr[:mid]
        R = arr[mid:]

        # Сортировка первой половины
        L = sort_by_digit_count(L)

        # Сортировка второй половины
        R = sort_by_digit_count(R)

        # Объединить две отсортированные половины
        i, j, k = 0, 0, 0
        while i < len(L) and j < len(R):
            if max_(L[i], R[j]) == R[j]:
                arr [k] = L[ i ]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        '''Проверить, есть ли в первой половине
        элемент, который должен стоять левее
        '''
        while i < len(L):
            arr[k] = L[i]
```

```
37              i += 1
38              k += 1
39
40          '''Проверить, есть ли в правой половине
41          элемент, который должен стоять левее
42          '''
43          while j < len(R):
44              arr[k] = R[j]
45              j += 1
46              k += 1
47
48      return arr
49
50
51  def max_(a, b):
52      """
53      Принимает два положительных целых числа, a и b,
54      и возвращает то, в котором присутствует больше цифр
55      с бóльшим значением. В случае равенства
56      возвращается большее число.
57
58      """
59      a, b = str(a), str(b)
60      for i in range(9, -1, -1):
61          if a.count(str(i)) > b.count(str(i)):
62              return int(a)
63          elif a.count(str(i)) < b.count(str(i)):
64              return int (b)
65      return max(int(a), int(b))
```

Глава 4

Строки

В этой главе рассматриваются 13 задач, которые сопровождаются примерами решения на Python. Вот эти задачи:

1. Шифрование текста блинчиком.
2. Перестановка гласных в обратном порядке.
3. Форма слова из текстового корпуса.
4. Высота слова из текстового корпуса.
5. Объединение соседних цветов по заданным правилам.
6. Вторая машина Маккаллоха.
7. Слово Чампернауна.
8. Объединение заданных строк в новую строку на основе некоторых правил.
9. Расшифровка слов.
10. Автокорректор слов.
11. Правильная форма глагола в испанском языке.
12. Выбор слов с одним и тем же набором букв.
13. Выбор слов из текстового корпуса, соответствующих шаблону.

4.1. Шифрование текста блинчиком

Задача шифрования текста блинчиком заключается в переворачивании подстрок в заданном тексте t в порядке индексов 2, 3, …, n, где n – длина t. Например, для текста t = «python» шифрование производится следующим образом: сначала переворачивается подстрока, заканчивающаяся вторым символом (y), «python» → «ypthon». Затем переворачивается подстрока, заканчивающаяся третьим символом (t): «ypthon» → «tpyhon». Далее переворачивается подстрока, заканчивающаяся четвертым символом (h): «tpyhon» → «hypton». И так далее до n-го символа: «hypton» → «otpyhn» → «nhypto».

Ваша задача: написать функцию, которая принимает строку и применяет к ней шифрование блинчиком.

В табл. 4.1 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.1. Некоторые ожидаемые результаты для задачи шифрования текста блинчиком

t	Ожидаемый результат
'python'	'nhypto'
'Deep Learning has revolutionized Pattern Recognition'	'niigcRnetPdzniuoe a nnaLpeDe erighsrvltoie atr eonto'
'Challenging Programming'	'gimroPginlaChlegn rgamn'
'x = 2x + 1 (9 * 56y)'	')6 (2=x x+19*5y'

Алгоритм

Алгоритм принимает строку с исходным текстом и возвращает новую строку – зашифрованную версию исходной строки. Он перебирает каждый индекс строки и переворачивает подстроку в исходной строке до этого индекса, а затем объединяет перевернутую подстроку с остальной частью исходной строки (т. е. с подстрокой после индекса). Этот процесс повторяется для каждого индекса строки, и в конце получается полностью зашифрованная строка.

В листинге 4.1 приводится код на Python, выполняющий шифрование текста блинчиком.

Листинг 4.1. Шифрование текста блинчиком

```
# Функция, переворачивающая строку с помощью цикла for
def reverse_string(s):
    reversed_s = ''
    # Цикл по индексам в s в обратном порядке
    for i in range(len(s)-1, -1, -1):
        '''
        Добавить символ с текущим индексом
        в reversed_s
        '''
        reversed_s += s[i]
    # Вернуть перевернутую строку
    return reversed_s

# Функция шифрования блинчиком
```

```
15 def pancake_scramble_into_texts(t):
16     for i in range(len(t)):
17         # Перевернуть подстроку до индекса i включительно
18         Reversed_String = reverse_string(t[:i + 1])
19         # Взять остаток исходной строки
20         Orginal_String = t[i +1:]
21         # Объединить перевернутую подстроку с остатком исходной строки
22         t = Reversed_String + Orginal_String
23     # Вернуть зашифрованную строку
24     return t
```

4.2. Перестановка гласных в обратном порядке

В этом задании дается текст и нужно создать новый текст, в котором гласные переставлены в обратном порядке. Кроме того, если символ с определенным индексом в исходном тексте имеет верхний регистр, соответствующий символ в новой строке тоже должен быть в верхнем регистре. Например, если исходный текст *s* = «Other», тогда в результате должен получиться текст *o* = «Ethor». В этой задаче рассматриваются гласные «aeiouAEIOU».

Для некоторых входов ожидаемые результаты показаны в табл. 5.2.

В табл. 4.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.2. Некоторые ожидаемые результаты для задачи перестановки гласных в обратном порядке

t	Ожидаемый результат
'Other'	'Ethor'
'Deep Learning has revolutionized Pattern Recognition'	'Doip Liorneng hes ravelitoinuzod Pettarn Ricagneteen'
'Challenging Programming'	'Chillangong Prigremmang'

Алгоритм

Для перестановки гласных в строке алгоритм выполняет обход символов входной строки с дополнительными итерациями по индексам гласных букв. В частности, алгоритм сначала извлекает все гласные буквы и их индексы из входной строки, используя генератор списков, который фильтрует символы по принадлежности к набору гласных. Затем он перебирает индексы в обратном порядке и собирает гласные в список. Наконец, про-

грамма перебирает исходные позиции гласных во входной строке и заменяет их соответствующими гласными из перевернутого списка, сохраняя при этом формат прописных/строчных букв в каждой позиции.

В листинге 4.2 приводится код на Python, выполняющий перестановку гласных в обратном порядке.

Листинг 4.2. Перестановка гласных в обратном порядке

```
def reverse_vowels_into_texts(s ) :
    # Множество гласных
    vowels = set ( "aeiouAEIOU" )
    chars = list(s)
    vowel_indices = \
        [i for i in range(len(chars)) if chars[i] in vowels]

    # Извлечь все гласные по их индексам
    reversed_vowels = []
    for i in range(len(vowel_indices) - 1, -1, -1):
        reversed_vowels.append(chars[vowel_indices[i]])

    '''
    Заменить гласные в исходном тексте,
    выполнив перестановку в обратном порядке
    '''
    j = 0
    for i in vowel_indices:
        if chars[i].isupper():
            chars[i] = reversed_vowels[j].upper()
        else:
            chars[i] = reversed_vowels[j].lower()
        j += 1

    return ''.join(chars)
```

4.3. Форма слова из текстового корпуса

Форма заданного слова *w* длиной *m* – это список *l*, состоящий из *m* – 1 значений +1, –1 и 0. Число, добавляемое в результирующий список *l*, зависит от алфавитного порядка: '*a*' < '*b*' < '*c*' < '*d*' < '*e*' < '*f*' < '*g*' < '*h*' < '*i*' < '*j*' < '*k*' < '*l*' < '*m*' < '*n*' < '*o*' < '*p*' < '*q*'' < '*s*' < '*t*' < '*u*' < '*v*' < '*w*' < '*x*' < '*y*' < '*z*'. Для каждой пары α и β если $\alpha < \beta$, то *shape* = 1, если $\alpha > \beta$, то *shape* = –1 и если $\alpha = \beta$, то *shape* = 0. Например, пусть дано слово *word* = *optimum*, для определения его формы выполняются следующие шаги: *o* < *p*, соответственно, *shape* = [1] → *p* < *t*,

соответственно, *shape* = [1, 1] → *t* > *i*, соответственно, *shape* = [1, 1,–1] → *i* < *m*, соответственно, *shape* = [1, 1,–1, 1] → *m* < *u*, соответственно, *shape* = [1, 1,–1, 1, 1] → *u* > *m*, соответственно, *shape* = [1, 1,–1, 1, 1,–1]. В этом задании требуется найти все слова, формы которых совпадают с заданной. Напишите функцию, которая принимает список слов (корпус текста) и эталонную форму и возвращает список слов, форма которых совпадает с заданной.

В табл. 4.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.3. Некоторые ожидаемые результаты для задачи формы слова из текстового корпуса

Words, shape	Ожидаемый результат
['congeed', 'outfeed', 'strolld', 'mail', 'stopped'], [1,–1,–1,–1, 0,–1]	['congeed', 'outfeed', 'strolld']
['eeten', 'good', 'aare', 'oozes', 'sstor'], [0, 1,–1, 1]	['eeten', 'oozes', 'sstor']

Алгоритм

Алгоритм определяет форму слова на основе алфавитного порядка составляющих его символов и использует эту форму для фильтрации списка слов. Правила реализуются с помощью операторов. Для этого используется оператор `if-else`.

В листинге 4.3 приводится код на Python, вычисляющий и возвращающий все слова с формой, совпадающей с заданной.

Листинг 4.3. Вычисление форм слов и возврат тех из них, формы которых совпадают с заданной

```
def Word_Shape_from_a_Text_Corpus(words, shape):
    output = []
    # Выполнить обход списка слов
    for i in words:
        '''
        Выбрать слова, форма которых
        совпадает с заданной
        '''
        if len(i) == len(shape)+1:
            if word_shape(i) == shape:
                output.append(i)
    return output
```

```
13 def word_shape(word) :
14     # Создать массив
15     shape = ([None] * (len(word) - 1))
16     for i in range((len(shape))):
17         '''
18         Функция ord принимает символ и
19         возвращает целое число (код юникода), на основе
20         которого определяется алфавитный порядок.
21         '''
22         '''
23         Определить форму данного слова,
24         используя заданные критерии
25         '''
26         a = ord(word[i])
27         b = ord(word[i + 1])
28         if a < b:
29             shape[i] = 1
30         elif a == b:
31             shape[i] = 0
32         elif a > b:
33             shape[ i ] = -1
34     return shape
```

4.4. Высота слова из текстового корпуса

Под длиной слова понимается количество содержащихся в нем символов, а под высотой – количество значимых подслов, которые можно получить из него. Слово, не имеющее собственного значения, имеет нулевую высоту, тогда как слово, которое имеет смысл и не может быть разделено на два значимых подслова, имеет высоту, равную единице.

Высота слов, которые можно разбить на подслова, равна наибольшей высоте его подслов плюс один. Например, слово «roqm» не имеет собственного значения и, следовательно, имеет нулевую высоту. С другой стороны, слово «chukker» нельзя разделить на два значимых подслова, поэтому его высота равна единице. Наконец, такое слово, как «enterprise», можно рекурсивно разбить на подслова до получения бессмысленных слов, и его высота будет определяться количеством его подслов. Ваша задача: написать функцию, которая принимает список слов *words* и одно слово *word* и возвращает высоту слова *word*, определяемую поиском подслов в *words*.

В табл. 4.4 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.4. Некоторые ожидаемые результаты для задачи определения высоты слова по словам в текстовом корпусе

Words, word	Ожидаемый результат
['A', 'chukker', 'corpus', 'text'], wjobnv	0
['A', 'chukker', 'corpus', 'text'], chukker	1

Важно отметить, что полная высота слова определяется по словам в текстовом корпусе.

Алгоритм

Алгоритм рекурсивно вычисляет высоту слова, разбивая его на более мелкие части, рекурсивно вычисляя высоты этих частей и комбинируя высоты частей, получает высоту исходного слова. Чтобы избежать избыточных вычислений и повысить производительность, используется мемоизация. Ниже подробно описаны шаги, выполняемые алгоритмом.

1. Принимается три входных параметра: `words` – список слов в текстовом корпусе, `word` – слово, высоту которого необходимо вычислить, и `memo` – словарь, используемый для запоминания ранее вычисленных высот.
2. Если параметр `memo` не указан, то создается новый пустой словарь.
3. Если высота текущего слова уже вычислена и хранится в `memo`, то возвращается запомненное значение.
4. Для поиска данного слова в списке слов используется алгоритм двоичного поиска. Если слово найдено в списке, возвращается его индекс; в противном случае возвращается –1.
5. Если слово отсутствует в списке `words`, его высота принимается равной `0`, возвращается `0` и алгоритм завершается.
6. Для хранения всех возможных способов деления слова на две части и проверки наличия обеих частей в списке слов используется список `validList`.
7. Цикл перебирает все возможные позиции разделения слова `word`, от позиции 1 до длины слова минус 1. Для каждой позиции извлекаются левая и правая части слова.
8. Если в списке `words` обнаруживаются левая и правая части, то они добавляются в список `validList`.
9. Если допустимых разбиений не обнаружено, высота слова принимается равной `1` и возвращается `1`.
10. Для хранения высот всех допустимых разделений создается список `hs`.

11. Для каждого допустимого разделения (`ls, rs`) высота левой и правой частей вычисляется рекурсивно. Если высота какой-то части уже вычислена и хранится в `memo`, то используется запомненное значение; в противном случае выполняется рекурсивный вызов для вычисления высоты. Высоты обеих частей складываются, и прибавляется 1 (исходное слово имеет смысл), чтобы получить высоту текущего разделения.
12. По завершении возвращается максимальная высота из всех допустимых разделений, хранящихся в `memo`.

В листинге 4.4 приводится код на Python, вычисляющий высоту слова.

Листинг 4.4. Вычисление высоты слова

```
'''
Эта функция реализует двоичный поиск
значения в отсортированном списке
'''
'''
Двоичный поиск - популярный алгоритм, используемый для
поиска элементов в упорядоченном списке.
'''
'''
В процессе работы функция последовательно делит интервал поиска
пополам, пока не будет найдено искомое значение или
пока интервал поиска не опустеет.
'''
def binary_search(lst, x):
    left = 0
    right = len(lst) - 1

    while left <= right:
        mid = (left + right) // 2
        if lst[mid] < x:
            left = mid + 1
        else:
            right = mid - 1

    return left

'''
Эта функция вычисляет высоту заданного слова,
используя список возможных подслов
```

```
30 и рекурсивно вычисляя высоты
31 этих подслов
32 '''
33 def Word_Height_From_a_Text_Corpus(words, word, memo=None):
34
35     # Создать пустой словарь memo, если он не существует
36     if memo is None:
37         memo = {}
38
39     '''
40     Проверить, была ли вычислена и сохранена
41     в memo высота текущего слова
42     '''
43     if word in memo:
44         return memo[word]
45
46     '''
47     выполняет поиск слова в списке
48     слов с использованием алгоритма двоичного поиска
49     '''
50     def Search(l, x):
51         i = binary_search(l, x)
52         return i if i != len(l) and l[i] == x else -1
53
54     '''
55     Если слово не найдено в списке
56     слов, то его высота равна 0
57     '''
58     if Search(words, word) == -1:
59         return 0
60
61     validList = []
62     '''
63     Отыскать все возможные разбиения слова
64     на две части и проверить присутствие
65     обеих частей в списке слов
66     '''
67     for s in range(1, len(word)):
68         ls = word[0:s]
69         rs = word[s:]
70
71         if Search(words, ls) != -1 \
72            and Search(words, rs) != -1:
```

```
            validList.append((ls, rs))

    '''
    Если допустимое разбиение не найдено,
    то высота слова принимается равной 1
    '''
    if not validList:
        return 1

    hs = []
    '''
    Вычислить высоту каждой допустимой части
    и взять максимальное значение
    '''
    for (ls, rs) in validList:
        left = memo.get(ls, None) or \
            Word_Height_From_a_Text_Corpus(words, ls, memo)
        right = memo.get(rs, None) or \
            Word_Height_From_a_Text_Corpus(words, rs, memo)
        hs.append(max(left, right) + 1)

    '''
    Сохранить вычисленную высоту текущего слова
    в словаре и вернуть ее
    '''
    memo[word] = max(hs)
    return max(hs)
```

4.5. Объединение соседних цветов по заданным правилам

В этом задании дана строка, состоящая из символов, соответствующих трем цветам: *yellow* (желтый), *red* (красный) и *blue* (синий). Требуется попарно скомбинировать цвета, составляющие эту строку, и получить ответ, применяя следующие правила.

1. Если складываются два одинаковых цвета, то в результате получается тот же самый цвет.
2. Если складываются два разных цвета, то в результате получается третий цвет.
3. Например, для строки *s* = *'rybyr'* результат определяется, как описано далее. Начиная с крайнего левого индекса рассматривается пара элементов с индексами 0 и 0 + 1 (*r* и *y*), комбинирование которых дает

цвет *b* (*r* и *y* – разные цвета, поэтому получается третий цвет, в этом примере *b*, согласно второму правилу), и *b* вставляется в массив *A*, следующая пара, 1 и 1 + 1 (*y* и *b*), дает *r*, и *r* вставляется в массив *A*. Следующая пара, 2 и 2 + 1 (*b* and *y*), дает *r*, и *r* вставляется в массив *A*, следующая пара, 3 и 3 + 1 (*y* и *r*), дает *b*, и *b* вставляется в массив *A*. В результате получается массив *A*= *brrb*.

Далее анализируется массив *A*. Начиная с крайнего левого индекса рассматривается пара элементов с индексами 0 и 0 + 1 (*b* и *r*), комбинирование которых дает цвет *y*, и *y* вставляется в массив *B*. Следующая пара, 1 и 1 + 1 (*r* и *r*), дает *r*, и *r* вставляется в массив *B*, следующая пара, 2 и 2 + 1 (*r* и *b*), дает *y*, и *y* вставляется в массив *B*. В результате получается массив *B* = *yry*.

Далее анализируется массив *B*. Начиная с крайнего левого индекса рассматривается пара элементов с индексами 0 и 0 + 1 (*y* и *r*), комбинирование которых дает цвет *b*, и *b* вставляется в массив *C*. Следующая пара, 1 и 1 + 1 (*r* и *y*), дает *b*, и *b* вставляется в массив *C*. В результате получается массив *C* = *bb*.

Аналогично анализируется массив *C*, и получается массив *D* = *b*. Длина *D* равна единице, и его единственный элемент (в данном примере *b*) является возвращаемым значением.

Ваша задача: написать функцию, которая принимает строку, состоящую из символов, соответствующих трем цветам, и возвращает один цвет, полученный в результате комбинирования.

В табл. 4.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.5. Некоторые ожидаемые результаты для задачи объединения соседних цветов по заданным правилам

Colors	Ожидаемый результат
'rybyr'	'b'
'rrrryybby'	'y'
'rby'	'b'
'bbbryryrybrrbyr'	'y'

Важно отметить, что полная высота слова определяется по словам в текстовом корпусе.

Алгоритм

Алгоритм принимает строку, состоящую из символов, соответствующих трем цветам, и возвращает один цвет. Для этого он многократно объединяет попарно соседние цвета, пока не останется только один цвет. В част-

ности, он делит входной список цветов на более мелкие списки цветов, а затем объединяет каждую пару независимо. Получившийся цвет возвращается в качестве результата. Чтобы избежать повторного вычисления одних и тех же комбинаций снова и снова, функция запоминает предыдущие результаты в словаре, чтобы использовать их, если понадобится.

В листинге 4.5 приводится код на Python, решающий задачу объединения соседних цветов по заданным правилам.

Листинг 4.5. Объединение соседних цветов по заданным правилам

```
def combine(a, b):
    # Создать словарь для поиска возвращаемых значений.
    ColorFinder = {"b":{"y" : "r" , "r" : "y"},
                   "y":{"b" : "r" , "r" : "b"},
                   "r":{"b" : "y" , "y" : "b"}}
    '''Если комбинируемые цвета совпадают,
    то вернуть этот же цвет, иначе
    использовать словарь для выбора соответствующего результата.
    '''
    if a == b:
        return a
    else:
        return ColorFinder[a][b]
'''
Эта функция принимает строку со списком цветов,
комбинирует соседние цвета согласно правилам
и возвращает единственный цвет.
'''
def combine_adjacent_colors(colors):
    '''
    Создать словарь для хранения результатов of
    предыдущих вызовов combine().
    В роли ключей используются кортежи из пар цветов,
    а в роли значений – результат комбинирования двух цветов.
    '''
    cache = {}

    '''
    Выполнять цикл, пока не останется единственный
    цвет в списке.
    '''
    while len(colors) > 1:
        '''
```

```
        Создать новый список для хранения результатов
        комбинирования соседних элементов.
        '''
        TemparrayColors = ([None] * (len(colors) - 1))

        '''
        Цикл по парам соседних цветов
        во входном списке.
        '''
        for i in range(len(TemparrayColors)):
            '''
            Проверить, присутствует ли результат комбинирования
            текущей пары цветов в словаре cache.
            '''
            if (colors[i], colors[i + 1]) in cache:
                # Если присутствует, вернуть его.
                TemparrayColors[i] = \
                    cache[(colors[i], colors[i + 1])]
            else:
                '''
                Если нет, то вызвать функцию combine(),
                чтобы получить результат, сохранить его в cache
                и использовать.
                '''
                result = combine(colors[i], colors[i + 1])
                cache[(colors[i], colors[i + 1])] = result
                TemparrayColors[i] = result

        '''
        Заменить входной список новым
        списком с комбинированными цветами.
        '''
        colors = TemparrayColors
    # Вернуть окончательный результат.
    return colors[0]
```

4.6. Вторая машина Маккаллоха

В этом задании рассматривается пример эзотерического языка программирования, который поможет вам улучшить ваши навыки программирования на Python. Эзотерический язык программирования, также известный как эзоланг,– это язык, предназначенный для проверки сложных и нетрадиционных идей. Он не предназначен для практического исполь-

зования и служит исключительно развлекательным целям. Система перезаписи строк Маккаллоха использует цифры от 0 до 9 и применяет к ним определенные правила. В этом контексте массив X представляет строки со значениями в диапазоне от 0 до $n - 1$, а массив Y вычисляется на основе X и состоит из значений в диапазоне от 1 до $n - 1$.

1. Если $X[0] = 2$, то в Y возвращается оставшаяся часть массива X.
2. Если $X[0] = 3$, то в Y возвращается $X + 2 + X$.
3. Если $X[0] = 4$, то в Y возвращается $X[:: -1]$.
4. Если $X[0] = 5$, то в Y возвращается $X + X$.

Ваша задача: написать функцию, которая принимает строку X и возвращает результат, полученный на основе упомянутых правил.

В табл. 4.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.6. Некоторые ожидаемые результаты для задачи перезаписи строк

X	Ожидаемый результат
'2999'	'999'
'329'	'929'
'101'	None
'322097845'	'209784522097845'

Алгоритм

Алгоритм перезаписи заданной строки цифр в соответствии с определенными правилами принимает строку X и на основе ее первой цифры выбирает правило для применения к цифрам. Затем этот процесс рекурсивно повторяется до тех пор, пока в строке не останется цифр или пока не встретится цифра, для которой нет правила.

В листинге 4.6 приводится код на Python, решающий задачу перезаписи заданной строки цифр.

Листинг 4.6. Перезапись заданной строки цифр

```
# Функция, удаляющая первую цифру
def remove_first_digit(X):
    return X[1:]

'''
Функция, разрезающая строку и
```

```
 7 вставляющая '2' в середину
 8 '''
 9 def slice_and_append_2(X):
10     return mcculloch(X[1:]) + '2' + mcculloch(X[1:])
11
12 # Функция, обращающая строку
13 def reverse(X):
14     return mcculloch(X[1:])[::-1]
15
16 # Функция, удваивающая строку
17 def double(X):
18     return mcculloch(X[1:]) + mcculloch(X[1:])
19
20 def mcculloch(X) :
21     '''
22     Словарь, отображающий первую
23     цифру в соответствующую функцию
24     '''
25     functions = {
26         '2' : remove_first_digit,
27         '3' : slice_and_append_2,
28         '4' : reverse,
29         '5' : double,
30     }
31     # Если первая цифра в X имеется в словаре
32     if X[0] in functions:
33         '''
34         Вызвать соответствующую функцию
35         для оставшихся цифр в X
36         '''
37         return functions[X[0]](X)
```

4.7. Слово Чампернауна

Слово Чампернауна – это длинная строка, не содержащая разделителей-запятых. Оно состоит из последовательности чисел начиная с единицы и увеличивающихся на единицу. Цель этой задачи – вернуть цифру, соответствующую данному индексу. Важно отметить, что эту задачу невозможно решить с помощью обычного метода, такого как *list.index*, из-за ограничений во времени и пространстве. Для решения данной задачи следует написать функцию, которая принимает положительное целое число *n*, представляющее желаемый индекс, и возвращает соответствующую цифру. Например, в седьмой позиции (счет начинается с 0) в последова-

тельности «1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...» находится цифра 8.

В табл. 4.7 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.7. Некоторые ожидаемые результаты для задачи получения слова Чампернауна

X	Ожидаемый результат
12**214	'9'
7	'8'
717897987691852588770047	'2'
3111111198765431001	'1'

При решении задачи количество цифр в каждой последовательности определяется по следующей таблице.

Номер последовательности	Диапазон	Число слов	Число цифр в каждой последовательности
1	1–9	9	9
2	10–99	90	180
3	100–999	900	2700
4	1000–9999	9000	36 000

Алгоритм

1. Принимается один входной параметр: `n` (номер искомой цифры `n`-я цифра).
2. Инициализируются переменные `d` (цифры), `p_i` (передаваемые индексы), `p_n` (передаваемые числа) и `w` (количество слов) значениями `0` и `9` соответственно.
3. Пока `n` больше или равно `p_i + w * (d + 1)`, увеличивать `d` на `1`, обновлять `p_i` до `p_i + w * (d + 1)`, `p_n` до `p_n + w` и `w` до `w * 10`.
4. Вычисляется индекс `p` как `(n - p_i)//(d + 1)`.
5. Вычисляется искомое число как `p_n + p + 1`.
6. Полученное число преобразуется в строку, и выбирается символ по индексу `(n - p_i)% (d + 1)`.
7. Возвращается `n`-я цифра как результат работы алгоритма.

В листинге 4.7 приводится код на Python, решающий задачу получения слова Чампернауна.

Листинг 4.7. Получение слова Чампернауна

```
def Champernowne_Word(n):
    d = 0
    p_i = 0
    p_n = 0
    w = 9

    while n >= p_i + w * (d + 1):
        p_i += w * (d + 1)
        p_n += w
        d += 1
        # Переход к очередной последовательности
        w *= 10

    p = (n - p_i) // (d + 1)

    '''
    Преобразовать число в строку и получить
    цифру по искомому индексу
    '''

    num = str (p_n + p + 1)
    return num[(n - p_i) % (d + 1)]
```

4.8. Объединение строк

Пусть *v* = {*aeiou*} – множество гласных, а *X* и *Y* – первое и второе слова соответственно. Если в слове *X* имеется хотя бы одна гласная из множества *v*, то считается, что оно имеет как минимум одну группу гласных. Если в *X* имеется две гласные, разделенные согласной, то они образуют две группы гласных, а если гласные следуют друг за другом, то все они относятся к одной группе. Например, в слове *pyth***o***n***i***st* две группы гласных (обратите внимание, что *y* не принадлежит множеству *v*, поэтому она не учитывается при подсчете групп), а в слове *k***ee***p* – одна. Задача заключается в создании новой строки на основе двух имеющихся. Вот правила создания новой строки:

1. Если в *X* одна группа гласных, то сохраняются только символы перед гласной, и эта часть *X* объединяется с *Y* так, что первые согласные в *Y* перед первой гласной удаляются. Например, если *X* = *'go'* и *Y* = *'meaning'*, то на выходе получится строка *'geaning'*.

2. Если в *X* имеется более одной группы гласных, то сохраняются все символы, предшествующие второй гласной, и эта часть *X* объединяется с *Y* так, что первые согласные в *Y* перед первой гласной удаляются. Например, если *X* = *'python'* и *Y* =*'visual'*, то на выходе получится *'pythisual'*.

Напишите функцию, которая принимает две строки, *X* и *Y*, и возвращает новую строку, составленную из *X* и *Y*, следуя приведенным выше правилам.

В табл. 4.8 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.8. Некоторые ожидаемые результаты для задачи объединения заданных строк в новую строку

X, Y	Ожидаемый результат
'go', 'meaning'	'geaning'
'python', 'visual'	'pythisual'
'elliot', 'bill'	'ill'
'ross', 'jules'	'rules'

Алгоритм

Алгоритм принимает две входные строки, `first_word` и `second_word`, и объединяет их в одну строку. Объединение осуществляется путем выбора соответствующих сегментов из каждой входной строки. На первом шаге алгоритм определяет последовательные гласные в каждой входной строке. Для этого он просматривает каждый символ в обеих входных строках и проверяет, является ли он гласной. Если символ является гласной, то алгоритм проверяет, был ли и предыдущий символ гласной. Если да, то индекс текущей гласной добавляется в подсписок предыдущих гласных. В противном случае для текущей гласной создается новый подсписок. После того как для каждой входной строки будут определены группы последовательных гласных, алгоритм определяет, какие сегменты входных строк следует сохранить в объединенной строке. Если первая входная строка не содержит последовательных гласных, то сохраняется вся строка. Если первая входная строка содержит только одну группу последовательных гласных, то алгоритм сохраняет все ее содержимое до первой гласной. В противном случае алгоритм сохраняет все в первой входной строке до предпоследней группы гласных. Сегмент второй входной строки, который необходимо сохранить в объединенной строке, всегда начинается с первой гласной второй входной строки. Наконец, алгоритм объединяет выбранные сегменты двух входных строк и возвращает результат.

В листинге 4.8 приводится код на Python, решающий задачу объединения строк.

Листинг 4.8. Объединение строк

```
def combine_strings_into_one(first_word, second_word):
    vowels = 'aeiou'

    '''
    Извлечь последовательные гласные из первого слова
    '''
    first_vowel_groups = []
    for i in range(len(first_word)):
        if first_word[i] in vowels:
            '''
            Если предыдущий символ – гласная,
            то добавить индекс в предыдущий подсписок
            '''
            if i > 0 and first_word[i - 1] in vowels:
                first_vowel_groups[-1].append(i)
                '''
                Иначе создать новый подсписок
                для текущей гласной
                '''
            else:
                first_vowel_groups.append([i])

    '''
    Извлечь последовательные гласные
    из второго слова
    '''
    second_vowel_groups = []
    for i in range(len(second_word)):
        if second_word[i] in vowels:
            '''
            Если предыдущий символ – гласная,
            то добавить индекс в
            предыдущий подсписок
            '''
            if i > 0 and second_word[i - 1] in vowels:
                second_vowel_groups[-1].append(i)
                '''
                Иначе создать новый подсписок
```

```
39                  для текущей гласной
40                  '''
41              else:
42                  second_vowel_groups.append([i])
43
44      # Определить сохраняемый сегмент в первом слове
45      '''
46      Если в первом слове нет последовательных гласных,
47      то сохранить все слово целиком
48      '''
49      if len(first_vowel_groups) == 0:
50          fw_segment = first_word
51          '''
52          Если в слове только одна группа гласных,
53          то сохранить все буквы
54          до первой гласной
55          '''
56      elif len(first_vowel_groups) == 1:
57          fw_segment = first_word[:first_vowel_groups[0][0]]
58          '''
59          Иначе сохранить все буквы
60          до второй с конца
61          группы гласных
62          '''
63      else:
64          fw_segment = first_word[:first_vowel_groups[-2][0]]
65
66      '''
67      Определить сохраняемый сегмент
68      во втором слове, который начинается
69      с первой гласной во втором слове
70      '''
71      sw_segment = second_word[second_vowel_groups[0][0]:]
72
73      return fw_segment + sw_segment
```

4.9. Расшифровка слов

Английские слова можно произвольно зашифровать, не влияя на их читабельность, при условии что первая и последняя буквы остаются одинаковыми, а длина слова остается неизменной. Цель этого задания – расшифровать заданное слово и определить похожие слова из предоставленного списка. Ваша задача: написать функцию, которая принимает строку *word*

и список слов *words* и возвращает список слов из *words*, которые соответствуют следующим критериям:

1. Слово имеет ту же длину, что и *word*, и первая и последняя буквы в слове совпадают с первой и последней буквами в *word*.
2. Порядок сортировки остается прежним.

В табл. 4.9 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.9. Некоторые ожидаемые результаты для задачи расшифровки слов

Words, word	Ожидаемый результат
['pycorn', 'pipline', 'python', 'ceo', 'we'], 'pohytn'	['python']
['camerier', 'academic', 'company', 'creamier'], 'ceamierr'	['camerier', 'creamier']

Алгоритм

Пусть дана строка `word`. Чтобы расшифровать слово в этой строке, выполняются следующие шаги.

1. Создается пустой массив `res` для хранения похожих слов.
2. Из списка `words` последовательно выбирается каждое слово `w`.
3. Для каждого слова `w` проверяется совпадение его длины с длиной `word`.
4. Если проверка прошла успешно, то проверяется совпадение первых и последних букв в `w` и `word`.
5. Если и эта проверка прошла успешно, то `w` добавляется в список `res`.
6. Шаги 3–5 повторяются для всех слов в `words`.
7. Возвращается полученный список `res`.

В листинге 4.9 приводится код на Python, решающий задачу расшифровки слов.

Листинг 4.9. Расшифровка слов

```
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, (len(arr) - i) - 1):
            if arr[j] > arr[j + 1]:
                # Поменять элементы местами
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

```
 8 def Unscrambling_the_given_words(words, word):
 9     res=[]
10     for w in words:
11         '''
12         С помощью инструкции continue текущее слово w
13         пропускается, и выполняется переход к следующему слову.
14         '''
15         if len(w) != len(word):
16             continue
17         elif w[0] != word[0] or w[-1] != word[-1]:
18             continue
19         # При желании можно использовать любой другой алгоритм сортировки
20         if bubble_sort(list(word)) == bubble_sort(list(w)):
21             res .append(w)
22     return res
```

4.10. Автокорректор слов

Существует несколько методов автоматического исправления орфографических ошибок в словах. Один из наиболее распространенных основан на вычислении расстояния между словами. Этот метод предполагает замену букв в слове ближайшими на клавиатуре. Например, если предполагаемое слово «car», а входное слово «csr», то букву «s», следует заменить буквой «a», которая находится рядом с «s» на клавиатуре, поэтому «csr» → «car». Эта задача требует вычисления всех расстояний между ключевыми словами в списке и данным словом с последующей заменой входного слова словом из списка, имеющим минимальное расстояние до заданного. Ваша задача: написать функцию, которая принимает проверяемое слово *word* и список слов *words* и возвращает слово *w* из списка *words*, имеющее минимальное расстояние от заданного.

В табл. 4.10 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.10. Некоторые ожидаемые результаты для задачи автокоррекции слов

Words, word	Ожидаемый результат
['pycorn', 'pipline', 'python', 'ceo', 'we'], 'pohytn'	'python'
['camerier', 'academic', 'company', 'creamier'], 'ceamierr'	'creamier'

Алгоритм

Пусть даны строка `word` и список слов `words`. Для каждого слова `w` в `words` алгоритм проверяет, имеют ли `w` и `word` одинаковую длину. Если это так,

то алгоритм вычисляет расстояние на клавиатуре *QWERTY* для каждой пары символов в `w` и `word`, сохраняет расстояния в массив `storage` и вычисляет сумму расстояний. Эта сумма присваивается слову `w`. Процесс повторяется для всех слов в `words`. Наконец, алгоритм возвращает слово из `words`, которое имеет минимальное расстояние до данного слова `word`.

В листинге 4.10 приводится код на Python, решающий задачу автокоррекции слов.

Листинг 4.10. Автокоррекция слов

```
def keyword_distance():
    # Порядок букв на типичной клавиатуре
    top = {c: (0, i) for (i, c) in enumerate("qwertyuiop")}
    mid = {c: (1, i) for (i, c) in enumerate("asdfghjkl")}
    bot = {c: (2, i) for (i, c) in enumerate( "zxcvbnm")}
    # Обновить словарь, передав несколько аргументов
    keys = dict(top, **mid, **bot)
    dist = dict()
    lows = "abcdefghijklmnopqrstuvwxyz"
    # Вычислить расстояние
    for cc1 in lows:
        for cc2 in lows:
            (r1, c1) = keys[cc1]
            (r2, c2) = keys[cc2]
            dist[(cc1, cc2)] = \
                (abs(r2 - r1) + abs(c2 - c1)) ** 2
    return dist
dist = keyword_distance()
# Вызов keyword_distance
'''
При использовании этой функции нет нужды
каждый раз вычислять расстояние между парами.
'''
def ds(c1, c2):
    return dist [(c1, c2)]
def autocorrectr_word(words,word):
    '''
    Извлечь слово той же
    длины, что и входное слово
    '''
    '''
    strip: удаляет начальные пробелы
    '''
```

```
    '''
    dict.fromkeys: присваивает значение каждому
    объекту в последовательности
    '''
    dists = dict.fromkeys\
        ([x.strip() for x in words if len(x) == len(word)], 0)
    for w in dists.keys():
        '''
        Сохранить расстояние между
        текущим и заданным словами
        '''
        storage = []
        for i in range(len(word)):
            storage.append(ds(w[i], word[i]))
        dists[w] = sum(storage)
    return min(dists, key=dists.get)
```

4.11. Правильная форма глагола в испанском языке

В этом задании требуется найти правильные формы глагола в испанском языке. Ваша задача: написать функцию, которая принимает глагол, лицо и время и возвращает правильную форму глагола в испанском языке.

В табл. 4.11 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.11. Некоторые ожидаемые результаты для задачи определения правильной формы глагола в испанском языке

Verb, subject, tense	Ожидаемый результат
'ganar', 'ustedes', 'pretérito'	'ganaron'
'escribir', 'ellos', 'imperfecto'	'escribían'
'tomar', 'nosotros', 'futuro'	'tomaremos'

Агоритм

В испанском языке есть три типа правильных глаголов. Следовательно, необходимо рассмотреть три условия и определить правильную форму глагола с учетом изменения по лицам и временам. Для этой цели используется словарь Python. Ключами в словаре служат лица, а значениями – окончания, которые следует добавить к данному глаголу. Проще говоря, отыскав лицо, остается только добавить окончание к заданному слову.

В листинге 4.11 приводится код на Python, решающий задачу определения правильной формы глагола в испанском языке.

Листинг 4.11. Определение правильной формы глагола в испанском языке

```
def Correct_verb_form_in_Spanish(verb, subject, tense):
    '''
    Словарь для хранения окончаний,
    которые должны добавляться к глаголам.
    Роль ключей в di играют лица, а роль
    значений – окончания для добавления
    к глаголам
    '''
    di = {}
    # Правильные глаголы первой формы
    if verb[-2::] == 'ar':
        # Если указано настоящее время
        if tense == 'presente':
            di = {'yo': 'o', 'tú': 'as',
                  'él': 'a', 'ella': 'a',
                  'usted': 'a',
                  'nosotros': 'amos',
                  'nosotras': 'amos',
                  'vosotros': 'áis',
                  'vosotras': 'áis',
                  'ellos': 'an',
                  'ellas': 'an',
                  'ustedes': 'an'
            }

        elif tense == 'pretérito' :
            di = {'yo': 'é', 'tú': 'aste',
                  'él': 'ó', 'ella': 'ó',
                  'usted': 'ó',
                  'nosotros': 'amos',
                  'nosotras': 'amos',
                  'vosotros': 'asteis',
                  'vosotras': 'asteis',
                  'ellos': 'aron',
                  'ellas': 'aron',
                  'ustedes': 'aron'
            }
```

```
38          elif tense == 'imperfecto':
39              di = {'yo': 'aba', 'tú': 'abas',
40                    'él': 'aba', 'ella': 'aba',
41                    'usted': 'aba',
42                    'nosotros': 'ábamos',
43                    'nosotras': 'ábamos',
44                    'vosotros': 'abais' ,
45                    'vosotras': 'abais',
46                    'ellos': 'aban',
47                    'ellas': 'aban',
48                    'ustedes': 'aban'
49              }
50          '''
51          Повторить то же самое для
52          правильных глаголов второй формы
53          '''
54      elif verb[-2::] == 'er':
55          if tense == 'presente':
56              di = {'yo': 'o', 'tú': 'es',
57                    'él': 'e', 'ella': 'e',
58                    'usted': 'e',
59                    'nosotros': 'emos',
60                    'nosotras': 'emos',
61                    'vosotros': 'éis',
62                    'vosotras': 'éis',
63                    'ellos': 'en',
64                    'ellas': 'en', 'ustedes': 'en'
65              }
66          elif tense == 'pretérito':
67              di = {'yo': 'í', 'tú': 'iste',
68                    'él': 'ió', 'ella': 'ió',
69                    'usted': 'ió',
70                    'nosotros': 'imos',
71                    'nosotras': 'imos',
72                    'vosotros': 'isteis',
73                    'vosotras': 'isteis',
74                    'ellos': 'ieron',
75                    'ellas': 'ieron',
76                    'ustedes': 'ieron'
77              }
78          elif tense == 'imperfecto':
79              di = {'yo': 'ía', 'tú': 'ías',
80                    'él': 'ía', 'ella': 'ía',
```

```
                'usted': 'ía',
                'nosotros': 'íamos',
                'nosotras': 'íamos',
                'vosotros': 'íais',
                'vosotras': 'íais',
                'ellos': 'ían',
                'ellas': 'ían', 'ustedes': 'ían'
            }
        '''
        Повторить то же самое для
        правильных глаголов третьей формы
        '''
    elif verb[-2::] == 'ir':
        if tense == 'presente':
            di = {'yo': 'o', 'tú': 'es',
                'él': 'e', 'ella': 'e', 'usted': 'e',
                'nosotros': 'imos',
                'nosotras': 'imos',
                'vosotros': 'ís',
                'vosotras': 'ís',
                'ellos': 'en',
                'ellas': 'en', 'ustedes': 'en'
            }
        elif tense == 'pretérito':
            di = {'yo': 'í', 'tú': 'iste',
                'él': 'ió', 'ella': 'ió',
                'usted': 'ió',
                'nosotros': 'imos',
                'nosotras': 'imos',
                'vosotros': 'isteis',
                'vosotras' : 'isteis',
                'ellos': 'ieron' , 'ellas': 'ieron',
                'ustedes': 'ieron'
            }
        elif tense == 'imperfecto':
            di = {'yo': 'ía', 'tú': 'ías',
                'él': 'ía', 'ella': 'ía',
                'usted': 'ía',
                'nosotros': 'íamos',
                'nosotras': 'íamos',
                'vosotros': 'íais',
                'vosotras': 'íais',
                'ellos': 'ían',
```

```
                'ellas': 'ían',
                'ustedes': 'ían'
            }
    '''
    Если задано будущее время, то
    форма глагола не имеет значения
    '''
    if tense == 'futuro':
        di = {'yo': 'é', 'tú': 'ás',
              'él': 'á', 'ella': 'á',
              'usted': 'á',
              'nosotros': 'emos',
              'nosotras': 'emos',
              'vosotros': 'éis',
              'vosotras': 'éis',
              'ellos': 'án', 'ellas': 'án',
              'ustedes': 'án'
        }
        '''
        Если задано будущее время, то
        последние две буквы в глаголе удалять не надо.
        '''
        return verb + di[subject]
    '''
    Если задано не будущее время, то
    надо удалить две последние буквы в глаголе.
    '''
    return verb[:len(verb) - 2:] + di[subject]
```

4.12. Выбор слов с одним и тем же набором букв

Подстрока – это часть оригинальной строки, сохраняющая те же символы в том же порядке, а производная перестановка – это строка, в которой порядок букв не обязательно совпадает с исходной строкой. Например, «tho» – это подстрока «python», а «thypno» – это производная перестановка от «python». Ваша задача: написать функцию, которая принимает список букв *letter* и список слов *words* и возвращает все слова из *words*, содержащие только буквы, имеющиеся в *letter*.

В табл. 4.12 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.12. Некоторые ожидаемые результаты для задачи выбора слов с одним и тем же набором букв

Words, letter	Ожидаемый результат
['suits', 'refluxed', 'trip', 'refluxing', 'retroflux'], 'reflux'	['refluxed', 'refluxing', 'retroflux']
['vasomotor', 'bathythermogram', 'benzhydroxamic', 'dialer'], 'byoam'	['bathythermogram', 'benzhydroxamic']

Алгоритм

Алгоритм использует комбинацию вложенных циклов и операций со строками для поиска слов, включающих один и тот же набор букв. Он перебирает каждое слово во входном списке и проверяет длину слова, сравнивая с длиной строки с искомыми буквами. Если длина слова больше или равна длине строки с буквами, то слово преобразуется в список символов, а затем запускается другой цикл, проверяющий присутствие в слове каждой искомой буквы. Если в слове найдены все буквы, то оно добавляется в результирующий список. В завершение алгоритм возвращает этот список.

В листинге 4.12 приводится код на Python, решающий задачу выбора слов с одним и тем же набором букв.

Листинг 4.12. Выбор слов с одним и тем же набором букв

```
def subsequent_letters(words, letters):
    '''Получить длину letters - списка
    искомых букв
    '''
    len_let = len(letters)
    '''
    Создать пустой список
    для хранения результата
    '''
    ST = []

    # Цикл по словам во входном списке
    for term in words:
        '''
        Длина рассматриваемого слова должна быть
        не меньше длины строки с искомыми символами
        '''
        if len(term) >= len(letters):
            # Преобразовать слово в список символов
```

```
            indterm = list(term)
            '''
            Инициализировать счетчик и индекс для
            отслеживания найденных букв
            '''
            counter = 0
            CurrentIndex = -1

            '''
            цикл по искомым буквам
            '''
            for let in letters:
                '''
                Проверить присутствие искомой буквы среди
                оставшихся букв в слове
                '''
                if let in indterm[CurrentIndex + 1:]:
                    '''
                    Если буква найдена,
                    то обновить индекс и счетчик
                    '''
                    indexlet = \
                        indterm.index(let, CurrentIndex + 1)
                    CurrentIndex = indexlet
                    counter += 1

                    '''
                    Если найдены все буквы, то
                    добавить слово в список
                    результатов и прервать цикл
                    '''
                    if counter == len_let:
                        ST.append(term)
                        break

    # Вернуть список с найденными словами
    return ST
```

4.13. Выбор слов из текстового корпуса, соответствующих шаблону

В этом задании дается шаблон строки и нужно найти все слова, включающие этот шаблон. Например, если *p* = ****b*ls* – заданный шаблон, то слова

'jumbals' и *'verbals'* будут соответствовать этому шаблону (т.е. включать его). Ваша задача: написать функцию, которая принимает строковый шаблон и возвращает все слова, содержащие его.

В табл. 4.13 показаны ожидаемые результаты для некоторых входных данных.

Таблица 4.13. Некоторые ожидаемые результаты для задачи выбора слов из текстового корпуса, соответствующих шаблону

Words, pattern	Ожидаемый результат
['pepsi', 'fissury', 'dark', 'missary', 'missort'], '*iss*r*'	['fissury', 'missary', 'missort']
['havened', 'car', 'hoveled', 'people', 'hovered'], 'h*ve*ed'	['havened', 'hoveled', 'hovered']

Алгоритм

Пусть *words* – это список заданных слов, а *pattern* – шаблон искомой строки. Алгоритм проверяет каждое слово *word* в списке *words* на соответствие шаблону *pattern*, для этого он проверяет, все ли буквы, имеющиеся в шаблоне *pattern*, присутствуют в слове *word* в той же последовательности и в той же кодировке, и если да, то сохраняет слово *word* в массив *result*.

В листинге 4.13 приводится код на Python, решающий задачу выбора слов из текстового корпуса, соответствующих шаблону.

Листинг 4.13. Выбор слов из текстового корпуса, соответствующих шаблону

```
def encode(word, pattern):
    '''
    Символ * имеет код 42
    '''
    encoding_pattern = list(pattern.encode())
    encoding_word = list(word.encode())
    pattern_filter = \
        [x for x in encoding_pattern if x != 42]
    encoded_word = []
    '''
    Достаточно, чтобы в слове присутствовали те же буквы, что и в шаблоне.
    И не важно, какие другие буквы присутствуют.
    '''
    for int_ in encoding_word:
        if int_ in pattern_filter:
            encoded_word.append(int_)
        else:
            encoded_word.append(42)
    return encoding_pattern , encoded_word
```

```
20
21 def possible_words(words, pattern):
22     '''
23     Проверяет каждое слово на соответствие шаблону, и
24     если оно соответствует, то добавляет
25     слово в результирующий список.
26     '''
27     result = []
28     pattern_encoding = encode('', pattern)[0]
29     for word in words:
30         '''
31         Чтобы избежать проверки всех слов,
32         проверять только слова, имеющие ту же длину,
33         что и шаблон.
34         '''
35         if len(word) == len(pattern):
36             word_encoding = encode(word, pattern)[1]
37             if word_encoding == pattern_encoding:
38                 result.append(word)
39     return result
```

Глава 5

Игры

В этой главе рассматриваются 14 игровых задач, которые сопровождаются примерами решения на Python. Вот эти задачи:

1. Определение выигрышной карты.
2. Подсчет карт каждой масти в игре бридж.
3. Получение сокращенного представления карт в раздаче в игре «Контрактный бридж».
4. Наборы карт одинаковой формы в карточной игре.
5. Подсчет количества раундов обработки чисел в порядке возрастания.
6. Достижение стабильного состояния в распределении конфет.
7. Игра вари.
8. Количество безопасных полей на шахматной доске с ладьями.
9. Количество безопасных полей на шахматной доске со слонами.
10. Достижимость поля для коня в многомерных шахматах.
11. Захват максимального количества шашек на шахматной доске.
12. Количество безопасных полей для размещения дружественных фигур на шахматной доске.
13. Количество очков в игре в кости «Скала».
14. Наилучший результат из нескольких бросков в игре в кости «Скала».

5.1. Определение выигрышной карты

Среди карточных игр есть игра с четырьмя игроками, в которой каждому игроку сдается одна карта и затем они по очереди вскрывают свои карты. В игре может назначаться или не назначаться козырная масть: «трефы», «пики», «червы» или «бубны». Достоинства карт определяются в порядке убывания так: «туз» > «король» > «дама» > «валет» > 10 > 9 > 8 > 7 > 6 > 5 > 4 > 3 > 2 > 1. Победитель в игре определяется по следующим правилам:

1. Если в игре есть козыри, то выигрывает игрок, имеющий козырь с наибольшим достоинством.
2. Если в игре нет козырей, то выигрывает игрок, имеющий карту с наивысшим достоинством и с мастью, совпадающей с мастью карты, которая была вскрыта первой.

Например, если игрокам сданы карты [(тройка, трефы), (король, пики), (дама, трефы), (валет, червы)] и козырями являются трефы, то выигрывает игрок с картой (дама, трефы) как обладающий козырной картой наибольшего достоинства.

Другой пример: игрокам сданы карты [(четверка, бубны), (туз, трефы), (четверка, пики), (король, трефы)] и в игре нет козырей. В таком случае выигрывает игрок с картой (четверка, бубны), потому что у других игроков карты другой масти и они не могут побить первую карту.

Ваша задача: написать функцию, которая принимает список карт в раздаче и возвращает выигравшую карту.

В табл. 5.1 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.1. Некоторые ожидаемые результаты для задачи определения выигрышной карты

Cards, trump	Ожидаемый результат
[('three', 'clubs'), ('king', 'spades'), ('queen', 'clubs'), ('jack', hearts')], 'clubs'	('queen', 'clubs')
[('four', 'diamonds'), ('ace', 'clubs'), ('four', 'spades'), ('king', 'clubs')], None	('four', 'diamonds')
[('eight', 'hearts'), ('three', 'diamonds'), ('nine', 'spades'), ('queen', 'hearts')], None	('queen','hearts')

Алгоритм

Для каждой масти – пики (spades), червы (hearts), бубны (diamonds) и трефы (clubs) – программа сохраняет соответствующие ранги и сортирует их в соответствии с правилами, используя алгоритм пузырьковой сортировки. На следующем этапе программа выбирает карту с самым высоким рангом на основе следующих критериев: если козырная карта указана на входе, то выбирается первая козырная карта с самым высоким рангом. Если козырная карта не указана, то выбирается первая карта с самым высоким рангом.

В листинге 5.1 приводится код на Python, определяющий выигрышную карту.

Листинг 5.1. Определение выигрышной карты

```
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, (len(arr) - i ) - 1):
            if arr[j] > arr[j + 1]:
                # Поменять местами, если необходимо
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
def Winner_in_Card_Game(cards, trump=None):
    names = [x[1] for x in cards]
    '''
    Массивы для определения рангов, если
    козыри определены.
    '''
    spades = []
    hearts = []
    diamonds = []
    clubs = []
    '''
    dicnames служит для поиска ранга
    по числовому значению
    '''
    dicnames = {1: 'one' , 2: 'two' , 3: 'three', 4: 'four',
                5: 'five', 6: 'six', 7: 'seven',
                8: 'eight', 9: 'nine', 10: 'ten', 50: 'ace',
                40: 'king', 30: 'queen', 20: 'jack'}
    for item in cards :
        if item[1] == 'spades':
            if item[0] == 'ace':
                spades.append(50)
            elif item[0] == 'king':
                spades.append(40)
            elif item[0] == 'queen':
                spades.append(30)
            elif item[0] == 'jack':
                spades.append(20)
            elif item[0] == 'one':
                spades.append(1)
            elif item[0] == 'two':
                spades.append(2)
            elif item[0] == 'three':
                spades.append(3)
```

```
42              elif item[0] == 'four':
43                  spades.append(4)
44              elif item[0] == 'five':
45                  spades.append(5)
46              elif item[0] == 'six':
47                  spades.append(6)
48              elif item[0] == 'seven':
49                  spades.append(7)
50              elif item[0] == 'eight':
51                  spades.append(8)
52              elif item[0] == 'nine':
53                  spades.append(9)
54              elif item[0] == 'ten':
55                  spades.append(10)
56
57          elif item[1] == 'hearts':
58              if item[0] == 'ace':
59                  hearts.append(50)
60              elif item[0] == 'king':
61                  hearts.append(40)
62              elif item[0] == 'queen':
63                  hearts.append(30)
64              elif item[0] == 'jack':
65                  hearts.append(20)
66              elif item[0] == 'one':
67                  hearts.append(1)
68              elif item[0] == 'two':
69                  hearts.append(2)
70              elif item[0] == 'three':
71                  hearts.append(3)
72              elif item[0] == 'four':
73                  hearts.append(4)
74              elif item[0] == 'five':
75                  hearts.append(5)
76              elif item[0] == 'six':
77                  hearts.append(6)
78              elif item[0] == 'seven':
79                  hearts.append(7)
80              elif item[0] == 'eight':
81                  hearts.append(8)
82              elif item[0] == 'nine':
83                  hearts.append(9)
84              elif item[0] == 'ten':
```

```
                hearts.append(10)
        elif item[1] == 'diamonds':
            if item[0] == 'ace':
                diamonds.append(50)
            elif item[0] == 'king':
                diamonds.append(40)
            elif item[0] == 'queen':
                diamonds.append(30)
            elif item[0] == 'jack':
                diamonds.append(20)
            elif item[0] == 'one':
                diamonds.append(1)
            elif item[0] == 'two':
                diamonds.append(2)
            elif item[0] == 'three':
                diamonds.append(3)
            elif item[0] == 'four':
                diamonds.append(4)
            elif item[0] == 'five':
                diamonds.append(5)
            elif item[0] == 'six':
                diamonds.append(6)
            elif item[0] == 'seven':
                diamonds.append(7)
            elif item[0] == 'eight':
                diamonds.append(8)
            elif item[0] == 'nine':
                diamonds.append(9)
            elif item[0] == 'ten':
                diamonds.append(10)
        elif item[1] == 'clubs':
            if item[0] == 'ace':
                clubs .append(50)
            elif item[0] == 'king':
                clubs.append(40)
            elif item[0] == 'queen':
                clubs.append(30)
            elif item[0] == 'jack':
                clubs.append(20)
            elif item[0] == 'one':
                clubs.append(1)
            elif item[0] == 'two':
                clubs.append(2)
```

```
            elif item[0] == 'three':
                clubs.append(3)
            elif item[0] == 'four':
                clubs.append(4)
            elif item[0] == 'five':
                clubs.append(5)
            elif item[0] == 'six':
                clubs.append(6)
            elif item[0] == 'seven':
                clubs.append(7)
            elif item[0] == 'eight':
                clubs.append(8)
            elif item[0] == 'nine':
                clubs.append(9)
            elif item[0] == 'ten':
                clubs.append(10)
    bubble_sort(spades)
    bubble_sort(hearts)
    bubble_sort(diamonds)
    bubble_sort(clubs)
    '''
    Функция eval вычисляет выражение.
    Проще говоря,
    она принимает строку и возвращает
    результат.
    t = eval('9') даст t = 9, где t - целое число.
    t = eval('2*x+1'даст t = 13
    '''
    # Использовать правила для определения победителя
    '''
    eval(cards[0][1]) вернет
    значение 'spades', или 'hearts', или 'diamonds',
    или 'clubs', [-1] вернет
    наибольший ранг, потому что массив
    отсортирован по возрастанию.
    '''
    if trump not in names:
        return dicnames[eval(cards[0][1])[-1]], cards[0][1]
        # Если козырь задан
    elif trump in names:
        '''
        eval(trump) вернет значение
        'spades', или 'hearts', или 'diamonds', или 'clubs',
```

```
171         [-1] вернет наибольший ранг,
172         потому что массив отсортирован по возрастанию.
173         '''
174     return dicnames[eval(trump)[-1]], trump
```

5.2. Подсчет карт каждой масти в игре бридж

В карточной игре бридж каждому игроку сдается по тринадцать карт. В этом задании необходимо подсчитать и вернуть количество карт каждой масти в порядке {«пики», «червы», «бубны», «трефы»}. Напишите функцию, которая принимает тринадцать карт и возвращает количество карт каждой масти в порядке {«пики», «червы», «бубны», «трефы»}.

В табл. 5.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.2. Некоторые ожидаемые результаты для задачи подсчета карт каждой масти в игре бридж

Cards	Ожидаемый результат
[('jack', 'diamonds'), ('jack', 'hearts'), ('seven', 'clubs'), ('five', 'clubs'), ('ace', 'diamonds'), ('three', 'clubs'), ('four', 'spades'), ('three', 'spades'), ('eight', 'spades'), ('ace', 'hearts'), ('five', 'diamonds'), ('two', 'clubs'), ('queen', 'diamonds')]	[3, 2, 4, 4]

Алгоритм

Чтобы выполнить задание, нужно просто обойти в цикле все сданные карты и подсчитать количество карт каждой масти в порядке: «пики», «червы», «бубны» и «трефы».

В листинге 5.2 приводится код на Python, выполняющий подсчет карт каждой масти.

Листинг 5.2. Подсчет карт каждой масти

```
 1 def Hand_Shape_in_Bridge_Game(hand):
 2     cards = []
 3     countinglist = []
 4     for idx, shapes in hand:
 5         # shapes – масть карты
 6         cards.append(shapes)
 7     spades = cards.count('spades')
 8     hearts = cards.count('hearts')
 9     diamonds = cards.count('diamonds')
10     clubs = cards.count('clubs')
11     countinglist = [spades, hearts, diamonds, clubs]
```

```
12      return countinglist
```

5.3. Получение сокращенного представления карт в раздаче в игре «Контрактный бридж»

В игре «Контрактный бридж» (или просто бридж) каждому игроку раздается по тринадцать карт. Достоинства этих карт можно сократить для создания более понятного визуального представления. Цель состоит в том, чтобы найти соответствующие сокращения для каждого достоинства, причем порядок карт всегда следующий: пики (spades), червы (hearts), бубны (diamonds) и трефы (clubs). Для обозначения достоинств используются следующие символы: «A» обозначает туза (от англ. «ace» – «туз»), «K» – короля (от англ. «king» – «король»), «Q» – даму (от англ. «queen» – «королева», или «дама»), «J» – валета (от англ. «jack» – «валет»), и числа обозначают все остальные достоинства. Ваша задача: написать функцию, которая принимает список из тринадцати карт и возвращает его сокращенную форму в порядке «AQKJx», где «x» представляет числа, причем числа не требуется упорядочивать по их величине, а если карты какой-то масти отсутствуют в раздаче, то в соответствующую позицию должен быть помещен знак «–».

В табл. 5.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.3. Некоторые ожидаемые результаты для задачи получения сокращенной формы представления карт в игре «Контрактный бридж»

Cards	Ожидаемый результат
[('three', 'clubs'), ('ten', 'spades'), ('jack', 'hearts'), ('five', 'hearts'), ('jack', 'clubs'), ('two', 'diamonds'), ('eight', 'hearts'), ('eight', 'clubs'), ('three', 'spades'), ('ace', 'hearts'), ('jack', 'spades'), ('king', 'diamonds'), ('six', 'hearts')]	'Jxx AJxxx Kx Jxx'

Алгоритм

Алгоритм сначала кодирует достоинства карт каждой масти в виде числовых значений, где тузу присваивается наименьшее значение (0), а числовому достоинству «x» (независимо от его фактической величины) – самое высокое значение (4). Затем вызывается алгоритм пузырьковой сортировки, чтобы отсортировать карты каждой масти в порядке возрастания их закодированного значения. И в заключение закодированные значения заменяются соответствующими сокращенными именами карт: «A» для туза, «K» для «короля» и т.д.

В листинге 5.3 приводится код на Python, получающий сокращенное представление карт в раздаче.

Листинг 5.3. Получение сокращенного представления карт в раздаче

```
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, (len(arr) - i ) - 1):
            if arr[j] > arr[j + 1]:
                # Поменять местами, если необходимо
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
def contract_bridge_game(hand):

    spades = []
    hearts = []
    diamonds = []
    clubs = []
    '''
    Закодировать достоинства карт
    для сортировки на следующем шаге.
    '''
    for item in hand:
        if item[1] == 'spades':
            if item[0] == 'ace':
                spades.append('0')
            elif item[0] == 'king':
                spades.append('1')
            elif item[0] == 'queen':
                spades.append('2')
            elif item[0] == 'jack':
                spades.append('3')
            else:
                spades.append('4')
        elif item[1] == 'hearts':
            if item[0] == 'ace':
                hearts.append('0')
            elif item[0] == 'king':
                hearts.append('1')
            elif item[0] == 'queen':
                hearts.append('2')
            elif item[0] == 'jack':
                hearts.append('3')
            else:
                hearts.append('4')
        elif item[1] == 'diamonds':
            if item[0] == 'ace':
```

```
                diamonds.append('0')
            elif item[0] == 'king':
                diamonds.append('1')
            elif item[0] == 'queen':
                diamonds.append('2')
            elif item[0] == 'jack':
                diamonds.append('3')
            else:
                diamonds.append('4')
        elif item[1] == 'clubs':
            if item[0] == 'ace':
                clubs.append('0')
            elif item[0] == 'king':
                clubs.append('1')
            elif item[0] == 'queen':
                clubs.append('2')
            elif item[0] == 'jack':
                clubs.append('3')
            else:
                clubs.append('4')
    # Обозначить отсутствующие карты
    if len(spades) == 0:
        spades.append('-')
    if len(hearts) == 0:
        hearts.append('-')
    if len(diamonds) == 0:
        diamonds.append('-')
    if len(clubs) == 0:
        clubs.append('-')
    bubble_sort(spades)
    bubble_sort( hearts )
    bubble_sort(diamonds)
    bubble_sort(clubs)
    # Заменить коды аббревиатурами
    output = ''.join(spades) + ''' '''+ \
        ''''''.join(hearts) + ''' ''' + \
        ''''''.join(diamonds) + ''' ''' + \
        ''''''.join(clubs)
    output = output.replace('0', 'A')\
        .replace('1', 'K').replace(
        '2', 'Q' ).replace('3', 'J')\
        .replace('4', 'x')
    return output
```

5.4. Наборы карт одинаковой формы в карточной игре

В этой задаче предлагается считать два набора карт как имеющие одинаковую форму, если они содержат одинаковое количество карт одинаковой масти, пусть и в разном порядке. Например, наборы [6, 3, 2, 2] и [2, 3, 6, 2] имеют одинаковую форму. Ваша задача: написать функцию, которая принимает список наборов карт и возвращает список кортежей одинаковой формы, при этом содержимое кортежей должно быть отсортировано по убыванию, а список кортежей – по возрастанию.

В табл. 5.4 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.4. Некоторые ожидаемые результаты для задачи определения наборов карт одинаковой формы

Cards	Ожидаемый результат
[('two', 'hearts'), ('nine', 'spades'), ('two', 'clubs'), ('eight', 'diamonds'), ('queen', 'diamonds'), ('ace', 'clubs'), ('six', 'diamonds'), ('queen', 'hearts'), ('three', 'hearts'), ('queen', 'clubs'), ('ten', 'spades'), ('nine', 'clubs'), ('six', 'spades')], [('five', 'diamonds'), ('jack', 'hearts'), ('three', 'spades'), ('king', 'clubs'), ('two', 'spades'), ('king', 'spades'), ('jack', 'spades'), ('jack', 'diamonds'), ('seven', 'clubs'), ('nine', 'hearts'), ('two', 'hearts'), ('king', 'hearts'), ('eight', 'hearts')], [('queen', 'clubs'), ('jack', 'clubs'), ('two', 'diamonds'), ('nine', 'spades'), ('six', 'clubs'), ('ace', 'clubs'), ('ten', 'diamonds'), ('nine', 'diamonds'), ('three', 'clubs'), ('five', 'hearts'), ('eight', 'spades'), ('six', 'spades'), ('eight', 'hearts')], [('seven', 'spades'), ('ten', 'spades'), ('two', 'hearts'), ('two', 'clubs'), ('jack', 'spades'), ('five', 'spades'), ('queen', 'clubs'), ('king', 'hearts'), ('king', 'clubs'), ('seven', 'diamonds'), ('eight', 'clubs'), ('queen', 'spades'), ('four', 'spades')]]	[((4, 3, 3, 3), 1), ((5, 3, 3, 2), 1), ((5, 4, 2, 2), 1), ((6, 4, 2, 1), 1)]

Алгоритм

Методом линейного поиска определяются масти и вставляются в массив. Затем к массиву применяются сортировки в нужном порядке.

В листинге 5.4 приводится код на Python, определяющий наборы карт одинаковой формы.

Листинг 5.4. Определение наборов карт одинаковой формы

```
1 def insertion_sort_descending(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j= i - 1
```

```
        while j >= 0 and key > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, (len(arr) - i) - 1):
            if arr[j] > arr[j + 1]:
                # Поменять местами,если необходимо
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
def hand_shape_distribution(hands):
    arr = []
    arr1 = []
    for i in range(len(hands)):
        hand_list = [0, 0, 0, 0]
        # Подсчет числа вхождений
        for j in hands[i]:
            if j[1] == 'spades':
                hand_list[0] += 1
            elif j[1] == 'hearts':
                hand_list[1] += 1
            elif j[1] == 'diamonds':
                hand_list[2] += 1
            elif j[1] == 'clubs':
                hand_list[3] += 1
        hand_list = insertion_sort_descending(hand_list)
        arr.append(hand_list)
    for i in arr:
        arr1.append([i, arr.count(i)])
    # Удалить наборы с отличающейся формой
    temp_list = []
    for i in arr1:
        if i not in temp_list:
            temp_list.append(i)
    arr1 = temp_list
    # Преобразовать в кортеж
    for j in range(len(arr1)):
        arr1[j][0] = tuple(arr1[j][0])
        arr1[j] = tuple(arr1[j])
    # Отсортировать по возрастанию
    arr1 = bubble_sort(arr1)
    return arr1
```

5.5. Подсчет количества раундов обработки чисел в порядке возрастания

В этом задании дается массив чисел, возможно несортированный. Массив заполнен числами от 0 до n – 1. Цель этого задания – подсчитать количество раундов перестановки чисел, необходимых для упорядочения их в порядке возрастания. В первом раунде обрабатываются числа в порядке возрастания. В последующих раундах обрабатываются числа, которые не были обработаны в предыдущих раундах, и снова в порядке возрастания. Этот процесс повторяется до тех пор, пока не будут обработаны все числа в порядке возрастания. Для примера рассмотрим список l = [0, 4, 3, 5, 2, 1]. В первом раунде обрабатываются числа 0 и 1 в порядке возрастания, во втором раунде – только 2, в третьем раунде – 3 и в четвертом раунде – 4 и 5. Следовательно, для обработки данного списка l в порядке возрастания требуется четыре раунда. Ваша задача: написать функцию, которая принимает перестановку чисел от 0 до n – 1 и возвращает количество раундов.

В табл. 5.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.5. Некоторые ожидаемые результаты для задачи подсчета раундов обработки чисел в порядке возрастания

Perm	Ожидаемый результат
[0, 1, 5, 4, 2, 3]	3
[0, 1, 2, 4, 3, 5]	2
[4, 11, 2, 8, 6, 9, 5, 3, 0, 10, 1, 12, 7]	6
[0, 1, 4, 3, 2]	3

Алгоритм

Для решения этой задачи используется метод обратной перестановки. Пусть *perm* – исходный массив, а *inverseperm* – массив того же размера, что и *perm*, заполненный нулями. Массив *inverseperm* используется для инверсии заданной перестановки. Для каждого *i* в *perm* инструкция *inverseperm* [*perm* [*i*]] = *i* инвертирует перестановки. Для подсчета числа раундов определяется счетчик *rounds*. Первоначально переменной *rounds* присваивается значение 1, так как всегда существует последовательность чисел в порядке возрастания, мощность которой равна хотя бы единице. Пусть *n* – мощность *inverseperm*, для *k* = от 1 до *n*, если *inverseperm* [*k*] < *inverseperm* [*k* – 1], это означает, что *k* находится слева от *k* – 1 и количество раундов должно быть увеличено на единицу.

В листинге 5.5 приводится код на Python, подсчитывающий количество раундов для обхода заданной перестановки в порядке возрастания.

Листинг 5.5. Подсчет раундов обработки чисел в порядке возрастания

```
def Number_round_counter(perm):
    '''
    perm – это массив с числами
    от 0 до n - 1
    '''
    # Для хранения количества раундов
    rounds = 1
    '''
    Создать массив того же
    размера, что и массив perm
    '''
    inverse_perm = [0] * len(perm)
    for i in range(len(perm)):
        '''
        Инверсия массива perm
        '''
        inverse_perm[perm[i]] = i
    for k in range(1, len(inverse_perm)):
        '''
        В массиве inverse_perm,
        если очередной элемент меньше
        предыдущего,
        это означает, что он должен
        находиться левее предыдущего
        числа в массиве perm, соответственно,
        переменную round нужно увеличить на единицу.
        '''
        if inverse_perm[k] < inverse_perm[k - 1]:
            rounds += 1
    return rounds
```

5.6. Достижение стабильного состояния в распределении конфет

Группа детей сидит за круглым столом, и перед ними лежат конфеты. Все конфеты совершенно одинаковые. Игра начинается с того, что воспитатель звонит в колокольчик. После звонка каждый ребенок, у которого есть хотя бы две конфеты, должен отдать одну конфету ребенку слева и одну

конфету ребенку справа. Дети, у которых одна конфета или нет конфет, ничего не должны делать. Согласно принципу Дирихле, если конфет больше, чем детей, то такое деление конфет никогда не завершится. Ваша задача: написать функцию, которая принимает исходное распределение конфет и возвращает количество шагов, необходимых для достижения стабильного состояния.

В табл. 5.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.6. Некоторые ожидаемые результаты для задачи подсчета шагов для достижения стабильного состояния в распределении конфет

Candies	Ожидаемый результат
[0, 0, 0, 3]	1
[2, 0, 0, 1]	2
[2, 0, 0, 0, 0, 0, 4, 0, 1]	8
[0, 4, 0, 0, 0, 3, 0, 1, 0, 0]	4

Алгоритм

Алгоритм рекурсивно применяет набор правил ко входному списку конфет `candies`, пока не будет достигнуто стабильное состояние, когда у каждого ребенка окажется не более одной конфеты. В каждой итерации алгоритм идентифицирует элемент массива, в котором имеются две или более конфет, и перекладывает по одной конфете в соседние элементы. Этот процесс повторяется до тех пор, пока во всех элементах будет не более одной конфеты, после чего алгоритм завершает работу и возвращает количество шагов, потребовавшихся для достижения этого состояния. Вот подробное описание шагов алгоритма.

1. Принимается два параметра: `candies` – список целых чисел, представляющий количество конфет, которые есть у каждого ребенка, и `states` (необязательный параметр) – целое число, представляющее количество шагов, потребовавшихся программе для достижения текущего состояния (по умолчанию имеет значение 0).
2. Проверяется, достигнут ли базовый случай. Если количество детей, имеющих 0 или 1 конфету, равно общему количеству детей, то возвращается количество шагов.
3. Создается копия списка `candies` для текущего шага.
4. Выполняется обход элементов в списке `candies`.
5. Проверяется, есть ли у текущего ребенка хотя бы две конфеты. Если да, то из соответствующего элемента удаляются две конфеты и в со-

седние элементы добавляется по одной конфете (т. е. передаются детям слева и справа).

6. Вышеуказанные шаги продолжают выполняться рекурсивно с обновленным списком `candies` и увеличенным значением `states`.
7. По достижении базового случая возвращается конечное значение `states`.

В листинге 5.6 приводится код на Python, подсчитывающий количество шагов для достижения стабильного состояния в распределении конфет.

Листинг 5.6. Подсчет количества шагов для достижения стабильного состояния в распределении конфет

```
def candy_share(candies, states = 0):
    '''
    Возвращает количество
    шагов, потребовавшихся программе
    для достижения стабильного состояния.
    '''
    # Базовый случай: все дети имеют не более одной конфеты
    if candies.count(0) + candies.count(1) == len(candies):
        return states

    # Создать копию списка candies для текущего шага
    previous_state = candies.copy()

    '''
    Применить правила
    игры к каждому ребенку
    '''
    for i in range(len(candies)):
        '''
        Передавать конфеты соседям
        должны только дети, имеющие
        две или более
        конфет
        '''
        if previous_state[i] >= 2:
            # Уменьшить число конфет у текущего ребенка
            candies[i] -= 2

            # И передать их соседям слева и справа
            candies[(i - 1) % len(candies)] += 1
```

```
            candies[(i + 1) % len(candies)] += 1

    # Выполнить рекурсивный вызов с обновленными параметрами candies и states
    return candy_share(candies, states + 1)
```

5.7. Игра вари

Вари (оваре)[1] – африканская настольная игра со множеством вариаций, самая популярная, пожалуй, из семейства игр манкала. В игре используется игровая доска с двумя рядами лунок. Игроки могут «засевать» и «захватывать» семена в лунках. В этом задании дается доска с 2*n* лунками (домами). Первый ряд с *n* лунками принадлежит первому игроку, а второй – второму. Первоначально в каждой лунке находится определенное количество семян и их нужно собрать. Если в лунке находятся 2 или 3 семени, то их можно захватить. Рассмотрим раунд игры на примере. Пусть дана доска *board* = [0, 2, 1, 2] и дом *house* = 1. Нужно вывести массив, отражающий изменения после сбора семян. Первая фаза – сбор урожая и посев – начинается с позиции *house* + 1 и продолжается до индекса *board* [*house*] (т.е. выполняется столько шагов, сколько семян в лунке *board* [*house* + 1]). Значение *board* [*house*] обнуляется (из лунки извлекаются два зерна), поэтому *board* = [0, 0, 1, 2]. Далее одно зерно засевается во второй элемент (напомню, что счет начинается с 0), поэтому *board* = [0, 0, 2, 2], и еще одно зерно засевается в третий элемент, поэтому *board* = [0, 0, 2, 3]. Вторая и третья лунки содержат 2 и 3 зерна соответственно, поэтому они захватываются. В результате на выходе получается массив *board* = [0, 0, 0, 0] и игра завершается.

Ваша задача: написать функцию, которая принимает доску и возвращает массив, показывающий изменения после сбора урожая и посева.

В табл. 5.7 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.7. Некоторые ожидаемые результаты для задачи определения состояния игровой доски в африканской игре вари

Board, house	Ожидаемый результат
[2, 1, 2, 0], 1	[2, 0, 0, 0]
[1, 4, 5, 6], 1	[2, 0, 7, 7]
[7, 7, 7, 68, 0, 1, 0], 3	[18, 18, 18, 0, 0, 1, 1]
[2, 0, 7, 7], 1	[2, 0, 7, 7]

[1] Подробное описание правил игры вы найдете в Википедии: **https://ru.wikipedia.org/wiki/Вари_ (игра)**. – *Прим. перев.*

Алгоритм

Алгоритм использует циклы `while` и `for` для обхода игровой доски, посева семян в дома и захвата семян противника, если это возможно.

1. Принимается список `board`, представляющий текущее состояние игровой доски, и целое число `house`, представляющее индекс лунки – дом, – откуда начинаются сбор урожая и посев.
2. Определяется индекс `last_index` последнего дома на доске вычитанием 1 из длины списка `board`.
3. Количество семян в выбранном доме сохраняется в переменной `seeds_to_sow`.
4. В переменную `current_index` (текущий индекс) записывается индекс дома, следующего за выбранным.
5. Количество семян в выбранном доме устанавливается равным `0`.
6. Пока значение `seed_to_sow` больше `0`, выполняются следующие действия:
   а) если текущий индекс (`current_index`) выходит за пределы последнего индекса, то выполняется переход к первому индексу;
   б) если текущий индекс равен индексу дома, откуда были извлечены семена для посева, то этот индекс пропускается;
   в) в текущий дом добавляется одно семя;
   г) количество семян для посева (`seed_to_sow`) уменьшается на единицу;
   д) выполняется переход к следующему дому увеличением текущего индекса (`current_index`) на единицу;
   е) по достижении последнего дома в своем ряду происходит переход к последнему дому в ряду противника и движение продолжается от конца к началу ряда;
   ж) для каждого дома в ряду противника выполняются следующие действия:
      i) если в доме 2 или 3 семени, то они захватываются и их количество в доме устанавливается равным нулю;
      ii) если семян в доме противника больше, то они не захватываются.
7. Возвращается обновленный список *board*.

В листинге 5.7 приводится код на Python, определяющий состояние игровой доски в африканской игре вари.

Листинг 5.7. Определение состояния игровой доски в африканской игре вари

```
def oware_game(board, house):
    # Определить последний индекс массива board
    last_index = len(board) - 1
    # Сохранить количество семян в выбранном доме
    seeds_to_sow = board[house]
    # Запомнить индекс следующего дома как текущий
    current_index = house + 1
    # Обнулить количество семян в выбранном доме
    board[house] = 0
    # Выполнить посев семян в последующие дома
    while seeds_to_sow > 0:
        '''
        Если текущий индекс
        превысил последний индекс,
        то вернуться к первому индексу
        '''
        if current_index > last_index:
            current_index = 0
        # Пропустить начальный дом, откуда были взяты семена для посева
        if current_index == house:
            current_index += 1
        # Добавить одно семя в текущий дом
        board[current_index] += 1
        # Уменьшить число семян для посева на единицу
        seeds_to_sow -= 1
        # Перейти к следующему дому
        current_index += 1

        # Захватить семена противника, если возможно
        end_index = current_index - 1
        if end_index >= len(board) / 2:
            for i in range(end_index, len(board) // 2 - 1, -1):
                '''
                Если в доме противника 2 или 3 семени,
                то захватить их и обнулить
                количество семян в доме
                '''
                if board[i] in (2, 3):
                    board[i] = 0
                    '''
                    Если семян больше, то
```

```
                    прекратить захват
                    '''
                else:
                    break

    # Вернуть обновленный массив board
    return board
```

5.8. Количество безопасных полей на шахматной доске с ладьями

Ладья – это шахматная фигура, и позиция каждой ладьи представлена парой (горизонталь, вертикаль), где горизонтали и вертикали пронумерованы от 0 до *n* – 1. Ладьи могут атаковать фигуры, стоящие на одной с ними горизонтали или вертикали. Цель этой задачи – определить набор безопасных полей на шахматной доске, где ни одна ладья не сможет атаковать их. Напишите функцию, которая принимает координаты ладей в виде списка кортежей и размер шахматной доски и возвращает количество безопасных полей.

В табл. 5.8 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.8. Некоторые ожидаемые результаты для задачи определения безопасных полей на шахматной доске с ладьями

n, rooks	Ожидаемый результат
10, [(2, 3), (4, 4)]	64
91, [(1,1), (4,4), (3, 5), (0, 7)]	7569
20, [(1,0), (3,6), (11, 5), (1, 2)]	272
7, [(0,2), (3,9), (3, 4)]	20

Алгоритм

Пусть *rooks* – список ладей, а *n* – размер шахматной доски. Для каждого *i* в *rooks* алгоритм сохраняет *i*[0] в массиве *Unsaferow* и *i*[1] в массиве *Unsafecol*. На следующем шаге он получает количество безопасных горизонталей и вертикалей как *safedrow* = *n* – *len*(*set*(*Unsaferow*)) и *safecol* = *n* – *len*(*set* (*Unsafecol*)) соответственно. На последнем шаге подсчитывается число безопасных полей как *safedrow* * *safecol*.

В листинге 5.8 приводится код на Python, вычисляющий количество безопасных полей на шахматной доске.

Листинг 5.8. Определение количества безопасных полей на шахматной доске с ладьями

```
def Safe_Squares_Rooks_of_Chessboard(n, rooks):
    Unsafe_squares_in_rows = []
    Unsafe_squares_in_columns = []
    for i in rooks:
        Unsafe_squares_in_rows.append(i[0])
        Unsafe_squares_in_columns.append(i[1])
    # Чтобы исключить повторяющиеся числа, используется множество
    safed_rows = n - len(set(Unsafe_squares_in_rows))
    # Чтобы исключить повторяющиеся числа, используется множество
    safed_columns = n - len(set(Unsafe_squares_in_columns))
    # Получить количество безопасных полей
    safe = safed_rows * safed_columns
    return safe
```

5.9. Количество безопасных полей на шахматной доске со слонами

Дано: шахматная доска $n \times n$ с несколькими слонами на ней, где горизонтали и вертикали пронумерованы от 0 до $n - 1$. Слоны могут двигаться только по диагонали. Слоны могут атаковать фигуры, стоящие на одной с ними диагонали. Напишите функцию, которая в качестве входных данных принимает координаты слонов в виде списка кортежей и размер шахматной доски и возвращает количество безопасных полей.

В табл. 5.9 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.9. Некоторые ожидаемые результаты для задачи определения безопасных полей на шахматной доске со слонами

n, bishops	Ожидаемый результат
10, [(2, 3), (4,4)]	68
91, [(1,1), (4,4), (3, 5), (0, 7)]	8002
20, [(1,0), (3,6), (11, 5), (1, 2)]	307
7, [(0,2), (3,9), (3, 4)]	29

Алгоритм

Алгоритм принимает размер шахматной доски *n* и список *bishops* координат слонов на доске. Он вычисляет количество безопасных полей на дос-

ке, то есть количество полей, которые не заняты слонами или не находятся под угрозой атаки со стороны слона. Алгоритм сначала инициализирует переменную *safe_squares* значением 0, которая затем будет использоваться для подсчета количества безопасных полей. Далее он создает два множества: *diagonal1* и *diagonal2* – для хранения полей на каждой диагонали, находящейся под угрозой атаки.

Для каждого слона на доске алгоритм вычисляет индексы полей на двух диагоналях, где расставлены слоны, и добавляет их в соответствующие множества *diagonal1* и *diagonal2*. Наконец, алгоритм перебирает все поля на доске и проверяет, занято ли каждое поле слоном на той же диагонали или находится под угрозой. Если поле не занято и не находится под угрозой атаки слоном, то количество безопасных полей увеличивается на 1. На выходе алгоритм возвращает количество безопасных полей.

В листинге 5.9 приводится код на Python, вычисляющий количество безопасных полей на шахматной доске со слонами.

Листинг 5.9. Определение количества безопасных полей на шахматной доске со слонами

```
def Safe_Squares_not_Threaten_with_Bishops(n, bishops):
    safe_squares = 0

    # Для сохранения полей на каждой диагонали
    # множество полей на первой диагонали, находящихся под угрозой
    diagonal1 = set()
    # множество полей на второй диагонали, находящихся под угрозой
    diagonal2 = set()

    # Обойти список слонов и обновить множества, представляющие диагонали
    for bishop in bishops:
        # добавить поле в diagonal1
        diagonal1.add(bishop[0] + bishop[1])
        # добавить поле в diagonal2
        diagonal2.add(bishop[0] - bishop[1])

    # Подсчитать количество безопасных полей
    for i in range(n):
        for j in range(n):
            if i + j not in diagonal1 \
               and i - j not in diagonal2:
                # увеличить счетчик безопасных полей
                safe_squares += 1
    return safe_squares
```

5.10. Достижимость поля для коня в многомерных шахматах

Обычная шахматная доска имеет два измерения, но ее можно расширить до *k* измерений. На двумерной шахматной доске конь может перемещаться на одно из восьми полей на плоской доске, а на многомерной шахматной доске он может перемещаться в поля в других измерениях. Цель этой задачи – определить, сможет ли конь, для которого определено количество полей, пересекаемых в каждом направлении, достичь заданного поля на шахматной доске, исходя из начальных и конечных координат. Координаты представлены кортежами отрицательных или неотрицательных целых чисел. Например, для коня, который выполняет шаги (2, 1, 7) в трех измерениях, нужно определить, сможет ли конь прыгнуть из поля с координатами (3, 5, 9) в поле с координатами (8, 11, 13).

В табл. 5.10 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.10. Некоторые ожидаемые результаты для задачи определения достижимости поля для коня в многомерных шахматах

Knight, start, end	Ожидаемый результат
(2, 1, 7), (3, 5, 9), (8, 11, 13)	False
(3, 4, 2), (3, 5, 9), (1, 7, 9)	False
(2, 1), (12, 10), (11, 12)	True
(10, 5, 1), (20, 11, 16), (10, 12, 11)	True

Алгоритм

Алгоритм вычитает каждое значение в кортеже *start* из каждого значения в кортеже *end* и сохраняет абсолютные значения. Для нашего примера абсолютные разности |3–8 |, |5–11 | и |9–13 | равны (5, 6, 4) соответственно. Структура хода коня должна в точности соответствовать результатам вычитания. Следовательно, поскольку (2, 1, 7) не совпадает с полученным результатом (5, 6, 4), можно сделать вывод, что поле (8, 11, 13) недостижимо для коня, находящегося в поле (3, 5, 9) и выполняющего перемещение (2, 1, 7) по каждому из измерений. Следует отметить, что если *c* является результатом вычитания, то *c* необходимо отсортировать в порядке убывания. Напишите функцию, которая принимает структуру хода коня, начальную и конечную координаты и возвращает `True`, если конь сможет прыгнуть из поля *start* в поле *end*, в противном случае возвращает `False`. Вот подробное описание шагов алгоритма.

1. Принимается три аргумента: структура хода `knight`, начало `start` и конец `end`. Возвращает `True`, если конь может прыгнуть из поля `start` в поле `end`, и `False` в противном случае.
2. Создается пустой список с именем `list_numbers`.
3. Выполняется обход элементов в `start` и `end` позиции и вычисляется абсолютная разность соответствующих элементов.
4. Эти разности добавляются в `list_numbers`.
5. Список `list_numbers` сортируется в порядке убывания с использованием `bubble_sort`.
6. Выполняется обход элементов в списке `knight`. Если какой-то элемент в `knight` отсутствует в `list_numbers`, то это означает, что конь не может достичь конечного поля. В этом случае функция возвращает `False`.
7. Если элементы в `knight` присутствуют в `list_numbers`, то это означает, что конь может достичь целевого поля. В этом случае функция возвращает `True`.

В листинге 5.10 приводится код на Python, определяющий достижимость поля для коня в многомерных шахматах.

Листинг 5.10. Определение достижимости поля для коня в многомерных шахматах

```
'''
Выполняет пузырьковую сортировку входного массива
и возвращает массив, отсортированный в порядке убывания
'''
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if arr[j] < arr[j + 1]:
                '''
                Поменять два элемента местами,
                если они стоят в неправильном порядке
                '''
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
'''
Определяет, сможет ли конь прыгнуть
в заданную позицию на k-мерной шахматной доске
'''
def reaching_knight_jump(knight, start, end):
    '''
```

```
    # Создать пустой список для хранения разностей
    координат между начальной и конечной позициями
    '''
    list_numbers = []
    for i in range(len(start)):
        '''
        # Вычислить абсолютные разности между
        элементами start и end
        и добавить их в list_numbers
        '''
        list_numbers.append(abs(start[i] - end[i]))
    # Отсортировать список разностей в порядке убывания
    list_numbers = bubble_sort(list_numbers)
    for num in knight:
        '''
        Если какой-то элемент в knight
        отсутствует в list_numbers,
        значит, конь не сможет
        прыгнуть в указанную позицию
        '''
        if num not in list_numbers:
            return False
    '''
    Иначе вернуть True и тем самым показать,
    что конь сможет прыгнуть в указанное поле
    '''
    return True
```

5.11. Захват максимального количества шашек на шахматной доске

Среди настольных игр на шахматной доске $n \times n$ с черными и белыми полями широко известна игра в шашки. Она предполагает взятие фигур противника. Игроки поочередно делают ходы, перемещая шашки на соседнее поле по диагонали, причем фигуры всегда должны оставаться на белых полях и не могут двигаться назад. Если две фигуры разного цвета соседствуют по диагонали, то игрок, кому выпал черед делать следующий ход, может перепрыгнуть через фигуру противника и взять ее, при условии что позади фигуры противника есть свободное белое поле. Если у игрока есть возможность взять фигуру противника, он должен это сделать, а если есть возможность последовательно взять несколько фигур, то игрок должен взять их все. Игрок, потерявший все фигуры, считается проигравшим. Цель

этого задания состоит в том, чтобы взять максимальное количество фигур противника за один ход. Ход со взятием фигуры противника, находящейся по соседству на диагонали, может выполняться только в направлениях (1, 1), (1,–1), (–1, 1) и (–1,–1). Ваша задача: написать функцию, которая принимает размер шахматной доски, координату фигуры, выполняющей ход, и координаты фигур противника и возвращает количество фигур, взятых за один ход.

В табл. 5.11 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.11. Некоторые ожидаемые результаты для задачи определения максимального количества шашек на шахматной доске, которые можно взять из текущей позиции

n, x, y, pieces	Ожидаемый результат
5, 1, 3, set ([(2, 4), (2, 3), (2, 2), (1, 2), (3, 3), (1, 3)])	1
8, 0, 4, set ([(2, 1), (3, 1), (1, 3), (4, 1), (1, 4)])	2
10, 4, 7, set ([(2, 1), (9, 1), (2, 1), (6, 5)])	0

Алгоритм

Алгоритм принимает размер шахматной доски *n*, координаты *x* и *y* фигуры, выполняющей ход, множество *pieces* с координатами фигур противника и возвращает количество взятых фигур. Для исследования всех возможных ходов из текущего состояния алгоритм использует поиск в ширину. Он инициализирует множество *visited* посещенных полей, чтобы отслеживать поля, посещавшиеся во время поиска в ширину, и очередь *queue*, содержащую начальное состояние. Он также инициализирует текущее максимальное число взятых фигур. Затем алгоритм входит в цикл `while` и удаляет следующее возможное состояние из очереди. Он обновляет текущее максимальное количество фигур, взятых в этом состоянии. Для каждого возможного направления движения по диагонали алгоритм вычисляет новую позицию и позицию взятой фигуры. Если алгоритм находит новое состояние со взятой фигурой, он создает новое множество фигур без взятой фигуры и проверяет, посещалось ли уже это состояние. Если нет, то алгоритм добавляет новое состояние в множество посещавшихся состояний в очередь *queue*. Поиск производится до тех пор, пока очередь не опустеет, после чего возвращается текущее максимальное количество взятых фигур.

В листинге 5.11 приводится код на Python, определяющий максимальное количество шашек на шахматной доске, которые можно взять из текущей позиции.

Листинг 5.11. Определение максимального количества шашек на шахматной доске, которые можно взять из текущей позиции

```
'''
Эта функция добавляет элемент
в конец очереди
'''
def enqueue(queue, item):
    queue.append(item)

'''
Эта функция удаляет и возвращает
первый элемент из очереди
'''
def dequeue(queue) :
    return queue.pop(0)

'''
Эта функция проверяет наличие элементов в очереди
'''
def is_empty(queue) :
    return len(queue) == 0

'''
Эта функция отыскивает максимальное число
фигур, которые можно взять
'''
'''
одной шашкой, находящейся в позиции (x, y)
на шахматной доске n x n,
'''
'''
принимает позиции фигур противника
в множестве pieces
'''
'''
и количество уже взятых
фигур в множестве removed.
'''
def Capturing_Max_Checkers(n, x, y, pieces, removed = 0):
    '''
    Для отслеживания возможных состояний
    используется множество visited
```

```
    '''
    visited = set ()
    '''
    Поиск начинается с начальной позиции,
    находящейся в очереди queue
    '''
    queue = [(x, y, pieces, removed)]
    '''
    Инициализировать текущий максимум
    количеством уже взятых
    фигур
    '''
    current_max = removed

    '''
    Продолжать поиск в ширину,
    пока очередь не опустеет
    '''
    while not is_empty(queue):
        '''
        Извлечь из очереди очередное состояние
        '''
        x, y, pieces, removed = dequeue(queue)
        '''
        Обновить текущий максимум
        максимальным количеством
        фигур, взятых
        к настоящему моменту
        '''
        current_max = max(current_max, removed)

        # Для каждой возможной диагонали
        for dx, dy in [(1, 1), (-1, 1), (1, -1), (-1, -1)]:
            '''
            Вычислить новую позицию фигуры
            после выполнения хода
            '''
            new_x, new_y = x + 2 * dx, y + 2 * dy
            jump_x, jump_y = x + dx, y + dy

            '''
            Если новая позиция оказалась за пределами доски,
            то пропустить это направление
```

```
            '''
            if not (0 <= new_x < n and 0 <= new_y < n):
                continue

            '''
            Если в новой позиции имеется фигура,
            то пропустить это направление
            '''
            if (new_x, new_y) in pieces:
                continue

            '''
            Если в новой позиции
            нет фигуры противника,
            то пропустить это направление
            '''
            if (jump_x, jump_y) not in pieces:
                continue

            '''
            Создать новое множество фигур pieces
            без только что взятой фигуры
            '''
            new_pieces = pieces.copy()
            new_pieces.remove((jump_x, jump_y))

            '''
            Если это поле уже посещалось,
            то пропустить его
            '''
            if (new_x, new_y, tuple(new_pieces))\
               in visited :
                continue

            '''
            Добавить новое состояние
            в множество visited и в очередь queue
            '''
            visited.add((new_x, new_y, \
                         tuple(new_pieces)))
            enqueue(queue, (new_x,new_y, \
                    new_pieces,removed+1))

```

```
    '''
    Вернуть текущее максимальное
    количество взятых фигур
    '''
    return current_max
```

5.12. Количество безопасных полей для размещения дружественных фигур на шахматной доске

Едва ли можно найти человека, незнакомого с игрой в шахматы или хотя бы не слышавшего о ней. Однако задумывались ли вы когда-нибудь над вопросом, как бы развивалась игра, если бы на доске присутствовали только ладьи? Рассмотрим шахматную доску $n \times n$ с множеством ладей, принадлежащих двум игрокам. Ладьи могут перемещаться по горизонтали или вертикали на любое количество полей в пределах шахматной доски. Ваша задача: написать функцию, которая принимает размер шахматной доски n и списки координат белых и черных ладей на доске и возвращает количество полей, в которых безопасно можно разместить другие белые фигуры.

В табл. 5.12 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.12. Некоторые ожидаемые результаты для задачи определения количества безопасных полей для размещения дружественных фигур на шахматной доске

n, friends, enemies	Ожидаемый результат
22, [(11, 7), (2, 4), (15, 7)], [(10, 20), (18, 12)]	397
9, [(5, 5)], [(2, 5), (1, 3)]	52
4, [(2, 2)], [(2, 1)]	10
7, [(2, 2)], [(2, 1), [6, 4], [6, 3]]	22

Алгоритм

Принимает три входных параметра: целое число n, представляющее размер шахматной доски, список кортежей с координатами белых ладей на доске *friends_rooks* и список кортежей с координатами черных ладей на доске *enemy_rooks* – и возвращает целое число, представляющее количество полей на доске, на которых можно безопасно разместить белые фигуры. На первом этапе создается доска с размерами $n \times n$, все поля которой инициализируются нулями, обозначающими пустые поля. Затем

на доске размещаются белые и черные фигуры, обозначаемые значениями 1 и –1 соответственно. На следующем этапе проверяется безопасность каждого поля для белых фигур путем проверки окружающих его полей в горизонтальном и вертикальном направлениях. Если поле безопасно со всех этих направлений, т. е. если в этих направлениях нет черных фигур, или имеются белые фигуры, или они пустые, то оно считается безопасным. Для каждого встреченного безопасного поля переменная-счетчик *safe_squares* увеличивается на единицу. Наконец, на последнем этапе функция возвращает *safe_squares* – количество полей, безопасных для размещения белых фигур.

В листинге 5.12 приводится код на Python, определяющий количество безопасных полей для размещения дружественных фигур на шахматной доске.

Листинг 5.12. Определение количества безопасных полей для размещения дружественных фигур на шахматной доске

```
def Safe_rooks_with_friends(n, friend_rooks, enemy_rooks):
    # Инициализировать шахматную доску
    chess = [[0] * n for _ in range(n)]
    # Разместить ладьи на доске
    for x, y in friend_rooks + enemy_rooks:
        value = 1 if (x, y) in friend_rooks else -1
        chess[x][y] = value

    # Подсчитать безопасные поля
    safe_squares = 0
    for i in range(n):
        for j in range(n):
            if chess[i][j] != 0:
                continue # Пропустить занятые поля

            # Проверить безопасность поля
            is_safe = True
            for k in range(j - 1, -1, -1): # Проверка полей слева
                if chess[i][k] == -1:
                    is_safe = False
                    break
                elif chess[i][k] == 1:
                    break
            for k in range(j + 1, n): # Проверка полей справа
                if chess[i][k] == -1:
                    is_safe = False
```

```
                    break
                elif chess[i][k] == 1:
                    break
            for k in range(i + 1, n): # Проверка полей ниже
                if chess[k][j] == -1:
                    is_safe = False
                    break
                elif chess[k][j] == 1:
                    break
            for k in range(i - 1, -1, -1): # Проверка полей выше
                if chess[k][j] == -1:
                    is_safe = False
                    break
                elif chess[k][j] == 1:
                    break

            if is_safe:
                # Поле безопасно
                safe_squares += 1
    return safe_squares
```

5.13. Количество очков в игре в кости «Скала»

«Скала» – это игра в кости, в которой каждый игрок бросает три кубика. Цель этого задания – сымитировать бросок трех кубиков и вернуть полученное количество очков. Правила для этого задания проиллюстрированы в таблице ниже. Если комбинация выброшенных кубиков не соответствует ни одной из показанных в таблице ниже, то в качестве результата принимается наибольшее количество очков, выпавших на одном из кубиков. Ваша задача: написать функцию, которая принимает результат броска трех кубиков и возвращает количество очков, подсчитанных в соответствии с правилами.

Категория	Описание	Очки	Комбинация
Скала	Любая комбинация с парой одинакового количества очков, дающая в сумме 13	50	
Тринадцать	Любая комбинация, дающая в сумме 13 очков	26	
Тройка	Любая тройка одинаковых очков	25	
Нижний ряд	1–2–3	20	

Категория	Описание	Очки	Комбинация
Верхний ряд	4–5–6	20	
Нечетный ряд	1–3–5	20	
Четный ряд	1–3–5	20	

В табл. 5.13 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.13. Некоторые ожидаемые результаты для задачи определения количества очков в игре в кости «Скала»

Dice	Ожидаемый результат
[3,2,1]	20
[4,4,4]	25
[6,6,6]	25
[1,3,6]	6

Алгоритм

Сначала создаются два массива: *index* и *Sum_of_each_Number*, оба размером 6. Массив *index* используется для отслеживания того, сколько раз каждое число появляется в списке *dice*, а массив *Sum_of_each_Number* – для хранения сумм каждого числа в списке *dice*. Затем циклически перебираются первые три элемента в списке *dice* и обновляются соответствующие элементы массивов *index* и *Sum_of_each_Number* с применением правил подсчета очков. Если обнаруживается совпадение с одной из комбинаций в таблице выше, то счет соответствующим образом обновляется. Если совпадения не найдено, возвращается наибольший элемент в списке *dice*.

В листинге 5.13 приводится код на Python, определяющий количество очков в игре в кости «Скала».

Листинг 5.13. Определение количества очков в игре в кости «Скала»

```
def Highest_Score_in_Crag_Score(dice):
    '''
    Определить массив с 6 элементами,
    по одному для каждой из 6 граней кубика.
    '''
    index = [0] * 6
    # Суммы очков на каждой грани
```

```
    Sum_of_each_Number = [0] * 6
    Score = 0
    for item in range(3):
        index[dice[item] - 1] = index[dice[item] - 1] + 1

        Sum_of_each_Number[dice[item] - 1] = \
            Sum_of_each_Number[dice[item] - 1] + dice[item]
        '''
        Выполнить подсчет с учетом
        правил.
        '''
        # Скала
        if (index[0] == 1 and index[5] == 2)\
           or (index[2] == 1
           and index[4] == 2) or \
           (index[4] == 1 and index[3] == 2):
                Score = 50
        # Тройка
        elif (index[0] == 3 \
             or index[1] == 3 or index[2] == 3 or
             index[3] == 3 or index[4] == 3\
             or index[5] == 3):
            Score = 25
        # Тринадцать
        elif ((index[2] == 1 and index[3] == 1 \
             and index[5] == 1) or
             (index[1] == 1 and index[4] == 1\
             and index[5] == 1)):
            Score = 26
        # 246
        elif index[1] == 1 and\
             index[3] == 1 and index[5] == 1:
            Score = 20

        # 456
        elif index[3] == 1 and \
             index[4] == 1 and index[5] == 1:
            Score = 20
        # 123
        elif index[0] == 1 and \
             index[1] == 1 and index[2] == 1:
            Score = 20
        # 135
```

```
51          elif index[0] == 1 and \
52               index[2] == 1 and index[4] == 1:
53              Score = 20
54          else:
55              for i in range(6):
56                  '''
57                  Если Score меньше суммы
58                  в позиции, то присвоить переменной Score
59                  это число, потому что
60                  оно максимальное
61                  '''
62                  if Score < Sum_of_each_Number[i]:
63                      Score = Sum_of_each_Number[i]
64      return Score
```

5.14. Наилучший результат из нескольких бросков в игре в кости «Скала»

Эта задача похожа на предыдущую, только вместо одного броска выполняется несколько бросков, для каждого из которых подсчитывается количество набранных очков, причем одна и та же комбинация не может повторяться два и более раз. Ваша задача: написать функцию, которая принимает результаты нескольких бросков и возвращает максимальное набранное количество очков. Если количество бросков равно нулю, то функция должна вернуть ноль.

В табл. 5.14 показаны ожидаемые результаты для некоторых входных данных.

Таблица 5.14. Некоторые ожидаемые результаты для задачи определения наилучшего результата из нескольких бросков в игре в кости «Скала»

Rolls	Ожидаемый результат
[(3, 3, 3), (2, 5, 5), (1, 5, 6), (2, 3, 3)]	47
[(1, 4, 6), (2, 3, 5), (1, 5, 1)]	14
[(1, 1, 1), (4, 3, 6)]	51

Алгоритм

Алгоритм тот же, что и в предыдущем задании, но есть некоторые важные отличия: на входе принимается несколько бросков. На первом шаге этот алгоритм генерирует комбинации для заданного числа бросков. Затем он перебирает комбинации и оценивает каждую из них с учетом правил.

Правила оценки приведены в предыдущем разделе. Список *category_is_used* используется для отслеживания того, какие правила уже применялись к текущей комбинации. Если правило уже было применено, его нельзя использовать снова для текущей комбинации. В завершение алгоритм возвращает наибольшую из оценок в заданной серии бросков.

В листинге 5.14 приводится код на Python, определяющий наилучший результат из нескольких бросков в игре в кости «Скала».

Листинг 5.14. Определение наилучшего результата из нескольких бросков в игре в кости «Скала»

```
'''
Эта функция генерирует
заданное количество комбинаций (бросков)
'''
def generate_combinations(rolls, n):
    if n == 0:
        return [[]]
    else:
        combinations = []
        for i in range(len(rolls)):
            sub_combinations =\
                generate_combinations(rolls, n - 1)
            for combination in sub_combinations:
                if i not in combination:
                    combinations.append([i] + combination)
        return combinations

'''
Эта функция определяет наилучший
результат по заданному списку бросков
'''
def Optimal_Crag_Score_with_Multiple_Rolls(rolls):
    posibilities = generate_combinations(rolls, len(rolls))
    unique_posibilities = []
    for roll in posibilities:
        if set(roll) == {i for i in range(len(rolls))}:
            unique_posibilities.append(roll)
    result = []

    category_is_used = [False] * 13
    for posibility in unique_posibilities:
        category_is_used = [False] * 13
```

```
        temp_roll = [rolls[i] for i in posibility]
        points = []
        for dice in temp_roll:
            # как перестановка из шести чисел
            index = [0] * 6
            Sum_of_each_Number = [0] * 6
            Desired_Score = 0
            for item in range(3):
                index[dice[item] - 1]=\
                dice.count(dice[item])
                Sum_of_each_Number[dice[item] - 1] = \
                    dice.count(dice[item]) * dice[item]
            '''
            # Выполнить подсчет очков для комбинации
            с учетом правил
            '''
            # Скала
            if ((index[0] == 1 and index[5] == 2) or
               (index[2] == 1 and index[4] == 2) or (
               index[4] == 1 and index[3] == 2)) \
               and not category_is_used[0]:
                Desired_Score = 50
                category_is_used[0] = True
            # Тринадцать
            elif ((index[2] == 1 and index[3] == 1
                 and index[5] == 1) or (
                 index[1] == 1 and index[4] == 1
                 and index[5] == 1))\
                 and not category_is_used[1]:
                Desired_Score = 26
                category_is_used[1] = True
            # Тройка
            elif (index[0] == 3 or index[1] == 3 or
                 index[2] == 3 or index[3] == 3 or
                 index[4] == 3 or index[5] == 3) \
                 and not category_is_used[2]:
                Desired_Score = 25
                category_is_used[2] = True
            # 246
            elif (index[1] == 1 and index[3] == 1
                 and index[5] == 1) \
                 and not category_is_used[3]:
                Desired_Score = 20
```

```
                category_is_used[3] = True
            # 456
            elif (index[3] == 1 and index[4] == 1
                 and index[5] == 1) and not \
                 category_is_used[4]:
                Desired_Score = 20
                category_is_used[4] = True
            # 123
            elif (index[0] == 1 and index[1] == 1
                 and index[2] == 1) and not \
                 category_is_used[5]:
                Desired_Score = 20
                category_is_used[5] = True
            # 135
            elif (index[0] == 1 and index[2] == 1
                 and index[4] == 1) and not \
                 category_is_used[6]:
                Desired_Score = 20
                category_is_used[6] = True

            else:
                temp_index = 0
                for i in range(6):
                    if Desired_Score \
                       < Sum_of_each_Number[i] \
                       and not category_is_used[6 + i]:
                        temp_index = i
                        Desired_Score =\
                            Sum_of_each_Number[i]
                        category_is_used[6 + temp_index] = True
            points.append(Desired_Score)
        result.append(sum(points))
    try:
        return max(result)
    except:
        return 0
```

Глава 6

Счет

В этой главе рассматриваются 8 счетных задач, которые сопровождаются примерами решения на Python. Вот эти задачи:

1. Подсчет количества переносов при сложении двух заданных чисел.
2. Подсчет количества рычащих животных.
3. Подсчет количества способов выражения вежливого числа.
4. Подсчет вхождений каждой цифры.
5. Подсчет количества максимальных слоев на двумерной плоскости.
6. Подсчет количества доминирующих чисел.
7. Подсчет количества троек в списке.
8. Подсчет пар пересекающихся кругов.

6.1. Подсчет количества переносов при сложении двух заданных чисел

При сложении двух или более цифр происходит перенос, если результат сложения превышает наибольшее число, которое может быть представлено одной цифрой. В этом задании вы должны подсчитать количество переносов, возникающих при сложении двух чисел.

В табл. 6.1 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.1. Некоторые ожидаемые результаты для задачи подсчета количества переносов при сложении двух заданных чисел

a, b	Ожидаемый результат
434, 289	2
2, 8	1
3, 3	0
8856, 5936	3

Алгоритм

В решении используются два алгоритма. Первый рекурсивно вызывает второй (*get_digit_sum_recursive*). Рекурсивный алгоритм вычисляет сумму заданных цифр *a*, *b*: *a* + *b*. Фактически он неоднократно удаляет последнюю цифру числа, используя целочисленное деление (*n*//10), и добавляет ее к сумме до тех пор, пока не останется больше цифр. Рекурсивный алгоритм выполняет следующие шаги.

1. Изначально есть положительное целое число *number*.
2. Если число *number* меньше 10, то возвращается *number*.
3. В противном случае вычисляется сумма последней цифры в *number* и суммы остальных цифр в *number*, для чего берется остаток и частное от целочисленного деления *number* на 10 соответственно.
4. Затем рекурсивно выполняются шаги, перечисленные выше.
5. И возвращается результат рекурсивного вызова.
6. Первый алгоритм выполняет следующие шаги.
7. Принимаются два положительных целых числа *num1* и *num2*.
8. Вычисляется сумма цифр в *num1* с использованием *get_digit_sum_recursive* и сохраняется в переменной *digit_sum_num1*.
9. Вычисляется сумма цифр в *num2* с использованием *get_digit_sum_recursive* и сохраняется в переменной *digit_sum_num2*.
10. Вычисляется сумма цифр в *num1* + *num2* с использованием функции *get_digit_sum_recursive* и сохраняется в переменной *digit_sum_num1_num2*.
11. Вычисляется общее количество переносов как (*digit_sum_num1* + *digit_sum_num2* – *digit_sum_num1_num2*)//9.
12. И полученное общее количество переносов возвращается.

В листинге 6.1 приводится код на Python, решающий задачу подсчета количества переносов при сложении двух заданных чисел.

Листинг 6.1. Подсчет количества переносов при сложении двух заданных чисел

```
'''
рекурсивно вычисляет сумму цифр
заданного числа
'''
def get_digit_sum_recursive(number):
    # Если число состоит из единственной цифры
    if number < 10:
        # Вернуть эту цифру
```

```
        return number
    else:
        '''
        Иначе сложить последнюю цифру
        с суммой остальных цифр
        '''
        return number % 10 + \
            get_digit_sum_recursive(number // 10)

'''
Подсчитывает количество переносов,
возникающих при сложении двух чисел
'''
def count_carries (num1, num2):
    # Вычислить сумму цифр в num1
    digit_sum_num1 = get_digit_sum_recursive(num1)
    # Вычислить сумму цифр в num2
    digit_sum_num2 = get_digit_sum_recursive(num2)
    # Вычислить сумму цифр в num1 + num2
    digit_sum_num1_num2 = get_digit_sum_recursive\
                          (num1 + num2)
    '''
    Вычислить общее количество переносов
    '''
    return (digit_sum_num1 + digit_sum_num2\
        - digit_sum_num1_num2) // 9
```

6.2. Подсчет количества рычащих животных

Дана группа животных, названия которых написаны как обычно – '*cat*' и '*dog*' – и в обратном порядке – '*tac*' и '*god*'. Животные начнут рычать, если увидят, что количество собак ('*dog*' и '*god*') в списке слева или справа (для '*cat*' и '*dog*' или '*tac*' и '*god*' соответственно) строго больше количества кошек ('*cat*' и '*tac*'). Например, если *animals* = ['*god*', ***'cat'***, '*cat*', '*tac*', ***'tac'***, '*dog*', '*cat*', '*god*'], то количество рычащих животных будет равно 2. В этом списке *animals* выделены жирным элементы ***'cat'*** и ***'tac'***, для которых количество собак ('*dog*' и '*god*') больше количества кошек ('*cat*' и '*tac*'). Ваша задача: написать функцию, которая принимает список животных и возвращает количество рычащих животных.

В табл. 6.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.2. Некоторые ожидаемые результаты для задачи подсчета количества рычащих животных

Animals	Ожидаемый результат
['tac', 'tac', 'tac', 'god', 'tac', 'dog', 'dog']	4
['tac', 'dog', 'dog', 'god', 'tac', 'dog', 'tac']	2
['tac', 'dog', 'dog', 'god', 'tac']	1
['tac', 'tac', 'dog', 'cat', 'tac']	0

Алгоритм

Алгоритм перебирает животных в заданном списке *animals* и проверяет их названия '*cat*', '*dog*', '*tac*' или '*god*', выполняя следующие шаги.

1. Принимается список животных *animals*.
2. Инициализируется переменная *num_growlers*, в которой накапливается количество рычащих животных.
3. Инициализируется переменная *current_animal_index*, отслеживающая текущее проверяемое животное.
4. Пока остаются животные для проверки (т.е. пока *current_animal_index* меньше длины *animals*):
   а) если текущее животное *'cat'* или *'dog'*, то подсчитывается количество кошек (*'cat'* и *'tac'*) и собак (*'dog'* и *'god'*), предшествующих текущему животному в списке. Если собак больше, то число *num_growlers* увеличивается на единицу;
   б) если текущее животное '*tac*' или '*god*', то подсчитывается количество кошек ('*cat*' и '*tac*') и собак ('*dog*' и '*god*'), следующих за ним в списке. Если собак больше, то число *num_growlers* увеличивается на единицу;
   в) переход к следующему животному в списке.
5. Возвращается *num_growlers* – подсчитанное количество рычащих животных.

В листинге 6.2 приводится код на Python, решающий задачу подсчета количества рычащих животных.

Листинг 6.2. Подсчет количества рычащих животных

```
def Count_the_Growl_of_Animals(animals):
    # Инициализировать нулем счетчик рычащих животных
    num_growlers = 0
```

```
    # Инициализировать нулем индекс текущего животного
    current_animal_index = 0
    # пока остаются животные для проверки
    while current_animal_index < len(animals):
        # Если текущее животное cat или dog
        if animals[current_animal_index] in ["cat","dog"]:
            '''
            инициализировать нулем количество
            кошек слева в списке
            '''
            num_cats = 0
            '''
            инициализировать нулем количество
            собак слева в списке
            '''
            num_dogs = 0
            '''
            начать проверку животных
            в списке, находящихся левее текущего
            '''
            previous_animal_index=\
                current_animal_index - 1

            '''
            подсчитать количество кошек и собак,
            находящихся левее в списке
            '''
            while previous_animal_index >= 0:
                if animals[previous_animal_index]=="cat" \
                   or animals[previous_animal_index]=="tac" :
                    num_cats += 1
                elif animals[previous_animal_index]=="dog" \
                     or animals[previous_animal_index]=="god" :
                    num_dogs += 1
                previous_animal_index -= 1
            '''
            Если перед текущим животным
            собак больше, чем кошек,
            то увеличить счетчик рычащих животных
            '''
            if num_dogs > num_cats:
                num_growlers += 1
        # Если текущее животное tac или god
```

```
47          elif animals[current_animal_index] in ["tac", "god"]:
48              '''
49              инициализировать нулем количество
50              кошек правее в списке
51              '''
52              num_cats = 0
53              '''
54              инициализировать нулем количество
55              собак правее в списке
56              '''
57              num_dogs = 0
58              '''
59              начать проверку животных
60              в списке, находящихся правее текущего
61              '''
62              next_animal_index=\
63                  current_animal_index + 1
64
65              '''
66              подсчитать количество кошек и собак,
67              находящихся правее в списке
68              '''
69              while next_animal_index < len(animals):
70                  if animals[next_animal_index]=="cat" \
71                     or animals[next_animal_index]=="tac":
72                      num_cats += 1
73                  elif animals[next_animal_index]=="dog" \
74                       or animals[next_animal_index]=="god" :
75                      num_dogs += 1
76                  next_animal_index += 1
77
78              '''
79              Если после текущего животного
80              собак больше, чем кошек,
81              то увеличить счетчик рычащих животных
82              '''
83              if num_dogs > num_cats:
84                  num_growlers += 1
85          # Перейти к следующему животному
86          current_animal_index += 1
87      # Вернуть общее количество рычащих животных
88      return num_growlers
```

6.3. Подсчет количества способов выражения вежливого числа

В теории чисел положительные целые числа, которые можно выразить в виде суммы двух или более последовательных чисел, называются вежливыми числами. С другой стороны, положительные целые числа, которые нельзя выразить подобным образом, называются невежливыми. Важно отметить, что вежливые и невежливые числа относятся к категории натуральных чисел. Целью этой задачи является подсчитать количество способов, которыми вежливое число можно выразить в виде суммы последовательных положительных целых чисел. Например, число 42 можно выразить как

а) 3+ 4 + 5 + 6 + 7 + 8 + 9;

б) 9 + 10 + 11 + 12;

в) 13 + 14 + 15;

г) 42.

А число 3 – как 2 + 1 и 3. Ваша задача: написать функцию, которая принимает положительное целое число *n* и возвращает количество способов, которыми это число можно выразить в виде суммы последовательных положительных целых чисел.

В табл. 6.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.3. Некоторые ожидаемые результаты для задачи подсчета количества способов выражения вежливого числа

n	Ожидаемый результат
3	2
96000	8
2	1
42	4

Алгоритм

Алгоритм подсчитывает количество делителей числа *n*, что эквивалентно количеству способов, которыми *n* можно выразить в виде суммы последовательных положительных целых чисел.

1. Принимается положительное целое число.
2. Инициализируется переменная *NumberofPolites* значением 0.

3. В цикле выполняется перебор чисел в диапазоне от 1 до $n + 1$.
4. Для каждого целого числа i в цикле:
   а) проверяется нечетность i и делится ли n на i;
   б) если оба условия выполняются, то переменная *NumberofPolites*, представляющая количество нечетных делителей, увеличивается на 1.
5. *NumberofPolites* возвращается как результат.

В листинге 6.3 приводится код на Python, решающий задачу подсчета количества способов выражения вежливого числа.

Листинг 6.3. Подсчет количества способов выражения вежливого числа

```
1 def count_consecutive_summers(n):
2     NumberofPolites = 0
3     for i in range(1, n + 1):
4         if n % i == 0 and i % 2 == 1:
5             NumberofPolites += 1
6     return NumberofPolites
```

6.4. Подсчет вхождений каждой цифры

Дана строка, содержащая только цифры «0123456789». Требуется подсчитать количество вхождений подряд каждой цифры и представить эту информацию в виде «количество–цифра». Например, для входной строки «222274444499966» результатом будет строка «4217543926». Она означает: «четыре двойки, одна семерка, пять четверок, три девятки и две шестерки».

Ваша задача: написать функцию, которая принимает строку цифр и возвращает строку с цифрами в исходной строке и количеством вхождений подряд каждой цифры.

В табл. 6.4 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.4. Некоторые ожидаемые результаты для задачи подсчета вхождений каждой цифры

Digits	Ожидаемый результат
'7779981'	'37291811'
'1333334'	'115314'
'2115131114'	'12211511133114'
'37291811'	'13171219111821'

Алгоритм

Алгоритм принимает строку с цифрами и выполняет итерации, подсчитывая количество одинаковых цифр, следующих подряд, в переменной *count*. Когда встречается другая цифра, он добавляет количество вхождений и саму цифру в список с именем *orders* и сбрасывает счетчик в 1. В завершение он возвращает представление исходной строки, полученное в *orders*.

В листинге 6.4 приводится код на Python, решающий задачу подсчета количества вхождений каждой цифры.

Листинг 6.4. Подсчет количества вхождений каждой цифры

```
def Count_occurrence_of_each_digit(digits):
    orders = []
    count = 1
    previous_digit = digits [0]

    '''
    Цикл по цифрам в исходной строке
    с подсчетом вхождений каждой цифры
    '''
    for current_digit in digits[1:]:
        if current_digit == previous_digit:
            count += 1
        else:
            '''
            Добавить количество вхождений цифры
            и ее саму в результат
            '''
            orders.append(str(count))
            orders.append(previous_digit)
            count = 1
            previous_digit = current_digit

    '''
    Добавить счетчик и последнюю цифру
    в результат
    '''
    orders.append(str(count))
    orders.append(previous_digit)
    return ''.join(orders)
```

6.5. Подсчет количества максимальных слоев на двумерной плоскости

Точка на двумерной плоскости с координатами (x_1, y_1) доминирует над точкой (x_2, y_2), если $x_1 > x_2$ и $y_1 > y_2$. Если над точкой не доминирует никакая другая точка, то она считается максимальной, причем на двумерной плоскости может иметься произвольное количество точек, образующих максимальный слой. Цель этого задания: подсчитать количество максимальных слоев. Например, пусть дан список точек *points* = [(1, 5), (3, 10), (2, 1), (9, 2)] и для него нужно подсчитать количество максимальных слоев. Входные данные просматриваются слева направо, точка (1, 5) не является максимальной, потому что над ней доминирует точка (3, 10), т.е. 3 > 1 и 10 > 5. Следующая рассматриваемая точка – точка (3, 10), это максимальная точка, потому что над ней не доминирует никакая другая точка: для пары (1, 5) и (3, 10) 1 не больше 3 и 5 не больше 10, для пары (2, 1) и (3, 10) 2 не больше 3 и 1 не больше 10, и для пары (9, 2) и (3, 10) 9 больше 3, но 2 не больше 10. Следующей рассматривается точка (2, 1), и она не является максимальной, потому что над ней доминирует точка (3, 10). Следующая точка – (9, 2) – является максимальной, потому что над ней, как и над (3, 10), не доминирует никакая другая точка. Следовательно, в списке *points* = [(1, 5), (3, 10), (2, 1), (9, 2)] есть два максимальных слоя, один из них образован точкой (10, 3), а другой – точкой (9, 2). Ваша задача: написать функцию, которая принимает список точек и возвращает количество максимальных слоев.

В табл. 6.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.5. Некоторые ожидаемые результаты для задачи подсчета количества максимальных слоев на двумерной плоскости

Points	Ожидаемый результат
[(7, 1), (6, 9)]	1
[(11, 4), (1, 1), (11, 1)]	2
[(1, 5), (3, 10), (2, 1), (9, 2)]	2
[(796, 1024), (11, 13), (19, 221), (700, 3)]	3

Алгоритм

Алгоритм принимает список точек *points* и подсчитывает количество максимальных слоев. Он циклически перебирает точки в списке и проверяет, является ли каждая точка максимальной (т.е. нет ли других точек с большими координатами *x* и *y*). Если точка максимальна, она добавляется в список максимальных точек. После выявления всех максимальных

точек функция увеличивает счетчик *layerCounter* и удаляет максимальные точки из исходного списка. Цикл продолжается до тех пор, пока в списке не останется точек и не будет получено окончательное количество слоев.

В листинге 6.5 приводится код на Python, подсчитывающий количество максимальных слоев.

Листинг 6.5. Подсчет количества максимальных слоев

```
def count_maximal_layers_in_points(points_list):
    # Инициализировать счетчик слоев значением 0
    layer_count = 0

    # Цикл, пока points_list не опустеет
    while points_list:
        # Отыскать все максимальные точки в списке
        maximal_points = []
        for point in points_list:
            is_maximal = True
            for other_point in points_list:
                if other_point[0] > point[0] \
                   and other_point[1] > point[1]:
                    is_maximal = False
                    break
            if is_maximal:
                maximal_points.append(point)

        '''
        Увеличить layer_count на 1,
        если есть хотя бы одна максимальная точка
        '''
        if maximal_points:
            layer_count += 1

        '''
        Удалить все максимальные точки
        из исходного списка
        '''
        for point in maximal_points:
            points_list.remove(point )

    # Вернуть количество максимальных слоев
    return layer_count
```

6.6. Подсчет количества доминирующих чисел

Число в списке считается доминирующим, если числа справа от него меньше. Цель этого задания – найти количество доминирующих чисел в заданном списке. Согласно определению, последний элемент списка всегда является доминирующим. Например, рассмотрим список *items* = [–492, 124, 113,–38,–28,–15]. Мы видим, что число 124 является доминирующим, потому что оно больше, чем 113,–38,–28 и –15. Элемент 113 тоже является доминирующим, потому что он больше, чем –38,–28 и –15. Наконец, последний элемент –15 тоже является доминирующим. Следовательно, количество доминирующих чисел в этом списке равно трем. Если на вход подается пустой список, то должно быть возвращено число 0.

В табл. 6.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.6. Некоторые ожидаемые результаты для задачи подсчета количества доминирующих чисел

items	Ожидаемый результат
[-492, 124, 113,–38,–28,–15]	3
[77, 1, 2,–4, 13, 7, 9, 1]	4
[13, 7, 9, 1, 77, 1, 2,–4]	3
[77, 1, 2,–4, 13, 77, 7, 9, 1]	3

Алгоритм

1. Принимается список целых чисел.
2. Если список пустой, то возвращается 0.
3. Переменная *max_num* инициализируется значением «отрицательной бесконечности», а переменная *dominator_count* – значением 0.
4. Цикл по элементам списка в обратном порядке.
5. Для каждого числа в списке проверяется, больше ли оно *max_num*.
6. Если больше, то в *max_num* записывается значение текущего элемента и *dominator_count* увеличивается на 1.
7. После перебора всех элементов списка возвращается *dominator_count*.

В листинге 6.6 приводится код на Python, подсчитывающий количество доминирующих чисел.

Листинг 6.6. Подсчет количества доминирующих чисел

```
# Перевернуть список
def reverse_list(lst):
    '''
    создать пустой список для
    сохранения элементов в обратном порядке
    '''
    reversed_lst = []
    '''Цикл по элементам
    в обратном порядке
    '''
    for i in range(len(lst)  - 1, -1, -1):
        reversed_lst.append(lst[i])
    # Вернуть перевернутый список
    return reversed_lst
"""
Подсчитывает количество доминирующих чисел
в заданном списке целых чисел.
"""
def Counting_Dominator_Numbers(items):

    # Если список пуст, то вернуть 0
    if not items:
        return 0

    '''
    В языке -inf представляет отрицательную бесконечность.
    '''
    max_num = float('-inf')
    # Инициализировать счетчик доминирующих чисел
    dominator_count = 0
    # Цикл по перевернутому списку элементов
    for num in reverse_list(items):
        '''
        если текущее число больше максимального,
        встреченного до сих пор
        '''
        if num > max_num:
            # Сохранить его как текущее максимальное
            max_num = num
            # Увеличить счетчик доминирующих чисел
            dominator_count += 1
```

```
42      # Вернуть количество доминирующих чисел
43      return dominator_count
```

6.7. Подсчет количества троек чисел

Пусть *items* – список чисел такой, что элементы этого списка в позициях $i < j < k$ составляют тройку, если *items* [*i*] == *items* [*j*] == *items* [*k*] и $j - i == k - j$. Например, подсчитаем количество троек в списке *items* = [42, 9, 42, 42, 42, 103]. В 0-й, 2-й и 4-й позициях находятся числа 42, 42 и 42 соответственно, при этом 2 – 0 = 4 – 2, т.е. 2 = 2. Числа в этих позициях образуют тройку. Другая тройка образуется числами в 2-й, 3-й и 4-й позициях, потому что 3 – 2 = 4 – 3, т.е. 1 = 1. Таким образом, в рассматриваемом списке имеются две тройки. Ваша задача: написать функцию, которая принимает список положительных и отрицательных целых чисел и возвращает количество троек в нем.

В табл. 6.7 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.7. Некоторые ожидаемые результаты для задачи подсчета троек

items	Ожидаемый результат
[42, 9, 42, 42, 42, 103]	2
[–8,–8, 109,–8,–8,–36]	2
[21,–41, 21, 76, 21,–71, 21, 17, 17, 17, 17, 17, 47]	6
[77, 1, 2,–4, 13, 77, 7, 9, 1]	0

Алгоритм

Пусть функция *TroikaCounter* – это функция, подсчитывающая количество троек в списке *items*. Для проверки условия $j - i == k - j$ она проверяет состояние *items* [*i*] == *items* [*j*] == *items* [2 * *j* – *i*] для каждого *i* в диапазоне от 0 до |*items*| и для каждого *j* в диапазоне от *i* + 1 до |*items*|. Если условие *items* [*i*] == *items* [*j*] == *items* [2 * *j* – *i*] выполняется и числа во всех трех позициях одинаковые (третья позиция определяется выражением (2 * *j* – *i*)), то счетчик *TroikaCounter* увеличивается на единицу. В завершение функция возвращает *TroikaCounter* как количество троек.

В листинге 6.7 приводится код на Python, подсчитывающий количество троек в списке.

Листинг 6.7. Подсчет количества троек в списке

```
1 def TroikaCounter(items):
2     TroikaCounter = 0
```

```
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            k = 2 * j - i
            if k > (len(items) - 1):
                pass
            else:
                if items[i] == items[j] == items[k]:
                    TroikaCounter += 1
    return TroikaCounter
```

6.8. Подсчет пар пересекающихся кругов

На двумерной плоскости существуют круги (x_1, y_1, r_1) и (x_2, y_2, r_2), где (x, y) – это координаты центра круга, а r – радиус. Если для них выполняется неравенство Пифагора $(x_2 - x_1)$ ** 2 + $(y_2 - y_1)$ ** 2 <= $(r_1 + r_2)$ ** 2, то эти два круга (x_1, y_1, r_1) и (x_2, y_2, r_2) имеют область пересечения. В языке Python оператор ** вычисляет степень. Следует отметить, что формула работает только для целых чисел. Ваша задача: написать функцию, которая принимает список координат центров и радиусов кругов и возвращает количество пар кругов, пересекающихся хотя бы в одной точке.

В табл. 6.8 показаны ожидаемые результаты для некоторых входных данных.

Таблица 6.8. Некоторые ожидаемые результаты для задачи подсчета пар пересекающихся кругов

Disks	Ожидаемый результат
[(1, 1, 9), (1, 0, 9)]	1
[(4, 7, 11), (11, 2, 8), (1, 1, 1)]	2
[(1, 1, 3), (6, 0, 3), (6, 5, 3), (0, 6, 3)]	3
[(34, 9, 67)]	0

Алгоритм

Для решения этой задачи обычно используется алгоритм заметающей прямой, который широко применяется в вычислительной геометрии для решения различных задач, связанных с расположением и пересечением геометрических объектов. Он позволяет эффективно обнаруживать пересечения, перекрытия и другие геометрические свойства. Количество сравнений в случае кругов равно. Чтобы уменьшить количество сравнений, особенно когда круги могут находиться на большом расстоянии друг от друга, используется алгоритм заметающей прямой, вычисляющий активное пространство. В активном пространстве пересечение кругов более

вероятно. Считается, что круг попадает в активное пространство, если вертикальная линия, отличная от оси *x*, входит в пространство (*x* – *r*), и не попадает, если вертикальная линия покидает (*r* + *x*).

В листинге 6.8 приводится код на Python, подсчитывающий количество пар пересекающихся кругов.

Листинг 6.8. Подсчет количества пар пересекающихся кругов

```
def Counting_Intersected_Disks(disks):
    # Отсортировать круги
    active_disks = sorted([(x - r, x + r, x, y, r)
                           for x, y, r in disks ])

    # Инициализировать счетчик пересекающихся пар кругов
    count = 0

    '''
    Цикл по парам кругов
    и подсчет пересекающихся пар
    '''
    for i in range(len(active_disks) - 1):
        # Начать с круга, следующего за i-м кругом
        j = i + 1

        # Проверить пересечение с i-м кругом
        while j < len(active_disks) \
              and active_disks[j][0] <= active_disks[i][1]:
            '''Вычислить расстояние между
            центрами i-го и j-го кругов
            '''
            dx = active_disks[i][2] - active_disks[j][2]
            dy = active_disks[i][3] - active_disks[j][3]
            r_sum = active_disks[i][4] + active_disks[j][4]

            # Проверить пересечение i-го и j-го кругов
            if dx ** 2 + dy ** 2 <= r_sum ** 2:
                count += 1

            # Перейти к следующему кругу в списке
            j += 1

    # Вернуть счетчик пар пересекающихся кругов
    return count
```

Глава 7

Разные задачи

В этой главе рассматриваются шесть разных задач, не подходящих под темы, разобранные в предыдущих главах. Они сопровождаются примерами решения на Python. Вот эти задачи:

1. Идеальное перемешивание элементов списка.
2. Точный размен монет с учетом имеющихся номиналов.
3. Удаление избыточных элементов из списка.
4. Когда две лягушки встретятся в одном квадрате.
5. Определение позиции числа в массиве Витхофа.
6. Интерпретация программы на Fractran.

7.1. Идеальное перемешивание элементов списка

В этом задании дается список с четным числом элементов и требуется перемешать их. Для перемешивания список делится на две равные половины, и из каждой половины поочередно выбирается по одному элементу. Существует два типа идеального перемешивания: out-shuffle, когда первым берется первый элемент из первой половины, за ним следует первый элемент из второй половины – и так до тех пор, пока все элементы не будут перемешаны. Второй: in-shuffle, когда первым берется первый элемент из второй половины, а за ним – первый элемент из первой половины. Например, пусть дан список *items* = [19, 456, 3, 4, 5, 6, 64, 11] и требуется перемешать элементы, используя тип перемешивания out-shuffle. Сначала разбиваем список *items* на две равные половины и получаем p_1 = [19, 456, 3, 4] и p_2 = [5, 6, 64, 11]. Пусть есть список *st*, куда будут складываться элементы в процессе перемешивания. Вот как выглядит порядок вставки элементов в выходной список: *st* = [19] → *st* = [19, 5] → *st* = [19, 5, 456] → *st* = [19, 5, 456, 6] → *st* = [19, 5, 456, 6, 3] → *st* = [19, 5, 456, 6, 3, 64] → *st* = [19,

5, 456, 6, 3, 64, 4] → *st* = [19, 5, 456, 6, 3, 64, 4, 11]. Итак, [19, 5, 456, 6, 3, 64, 4, 11] – это перемешанный список. Результат перемешивания элементов при использовании in-shuffle выглядит так: *st* = [5, 19, 6, 456, 64, 3, 11, 4]. Он получен следующим образом: *st* = [5] → *st* = [5, 19] → *st* = [5, 19, 6] → *st* = [5, 19, 6, 456] → *st* = [5, 19, 6, 456, 64] → *st* = [5, 19, 6, 456, 64, 3] → *st* = [5, 19, 6, 456, 64, 3, 11] → *st* = [5, 19, 6, 456, 64, 3, 11, 4].

Ваша задача: написать функцию, которая принимает список элементов и логическую переменную, определяющую тип перемешивания, и возвращает перемешанный список элементов.

В табл. 7.1 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.1. Некоторые ожидаемые результаты для задачи идеального перемешивания элементов списка

Items, outin	Ожидаемый результат
[19, 456, 3, 4, 5, 6, 64, 11], True	[19, 5, 456, 6, 3, 64, 4, 11]
[19, 456, 3, 4, 5, 6, 64, 11], False	[5, 19, 6, 456, 64, 3, 11, 4]
[0, 7, 89, 654, 5, 0, 1, 7]	[0, 5, 7, 0, 89, 1, 654, 7]

Алгоритм

Пусть *items* – заданный список элементов, *storage* – выходной список, куда будут сохраняться элементы в процессе перемешивания, а *outin* – логическое значение, значение *True* которого определяет тип перемешивания out-shuffle, а значение *False* – тип in-shuffle. Первая половина исходного списка сохраняется в *part1*, а вторая – в *part2*. |*items*|[2] делится пополам, и результат сохраняется в *halves*. Тогда если *outin* = *True*, сначала в *storage* сохраняется *w*-й элемент *part1* [*w*], а вслед за ним – элемент *part2* [*w*]. Иначе если *outin* = *False*, сначала в *storage* сохраняется *w*-й элемент *part2* [*w*], а вслед за ним – элемент *part1* [*w*]. По завершении цикла в *storage* будет храниться перемешанный список.

В листинге 7.1 приводится код на Python, решающий задачу идеального перемешивания элементов списка.

Листинг 7.1. Идеальное перемешивание элементов списка

```
1 def perfect_riffle(items, outin = True):
2     '''
3     out = True: out-shuffle
4     out = False: in-shuffle
5     '''
```

[2] Длина списка.

```
    # Для хранения результата
    storage=[]
    # Разбить список на две равные половины
    halves = int(len(items) / 2)
    totallength = int(len(items))
    # part1 – первая половина
    part1 = items[0:halves]
    # part2 – вторая половина
    part2 = items[halves:totallength]
    for w in range(halves):
        # out-shuffle
        if outin == True:
            storage.append(part1[w])
            storage.append(part2[w])
        # in-shuffle
        else:
            storage.append(part2[w])
            storage.append(part1[w])
    return storage
```

Существует более эффективное решение, использующее меньше памяти. Его реализация приводится в листинге 7.2.

Листинг 7.2. Эффективное идеальное перемешивание элементов списка

```
def perfect_riffle(items, outin = True):
    '''
    out = True: out-shuffle
    out = False: in-shuffle
    '''
    # Вычислить количество элементов в каждой половине
    n = len(items) // 2
    # Разбить список на две половины
    half1 = items[:n]
    half2 = items[n:]
    # Цикл по одной из половин
    for i in range(n):
        # out-shuffle
        if outin:
            items[2 * i] = half1[i]
            items[2 * i + 1] = half2[i]
        # in-shuffle
        else:
```

```
19              items[2 * i] = half2[i]
20              items[2 * i + 1] = half1[i]
21      return items
```

7.2. Точный размен монет с учетом имеющихся номиналов

Пусть дано положительное целое число – некоторая сумма денег – и массив номиналов монет в порядке убывания и нужно выразить эту сумму в номиналах монет. Гарантируется, что наименьший номинал монеты в массиве равен 1. Например, если сумма равна 583 и в нашем распоряжении имеются монеты с номиналами [142, 73, 45, 17, 13, 7, 1], то эту сумму можно выразить в виде набора монет как [142, 142, 142, 142, 13, 1, 1]. Ваша задача: написать функцию, которая принимает положительное целое число и массив номиналов монет и возвращает массив номиналов монет, общая стоимость которых в точности равна введенной сумме.

В табл. 7.2 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.2. Некоторые ожидаемые результаты для задачи размена монет с учетом имеющихся номиналов

Amount, coins	Ожидаемый результат
583, [142, 73, 45, 17, 13, 7, 1]	[142, 142, 142, 142, 13, 1, 1]
26, [142, 45, 7, 1]	[7, 7, 7, 1, 1, 1, 1, 1]
174, [94, 73, 17, 13, 7, 4, 3, 1]	[94, 73, 7]
1, [142, 3, 1]	[1]

Алгоритм

1. Принимаются *amount* – заданная сумма и *coins* – список номиналов имеющихся монет.
2. Создается пустой список *smallercoins* для хранения наименьшего числа монет, составляющих сумму.
3. Для каждого номинала в списке *coins* повторяются шаги 4–6.
4. Проверяется, превышает ли оставшаяся сумма *amount* значение *coin* – номинал текущей монеты.
5. Если оставшаяся сумма больше или равна номиналу текущей монеты, то значение номинала текущей монеты *coin* вычитается из суммы *amount* и добавляется в список *smallercoins*.

6. Если оставшаяся сумма *amount* меньше номинала текущей монеты *coin*, то выбирается следующая монета из списка *coins*.
7. После проверки всех монет возвращается список *smallercoins*, содержащий наименьшее число монет, составляющих заданную сумму.

В листинге 7.3 приводится код на Python, решающий задачу размена монет.

Листинг 7.3. Размен монет

```
def calculate_smaller_coins(amount, coins):
    # Создать пустой список для хранения наименьшего числа монет
    smallercoins = []
    # Цикл по номиналам монет в списке
    for coin in coins:
        '''Продолжать вычитать номинал монеты из
        суммы, пока она не станет меньше текущего номинала
        '''
        while amount >= coin:
            '''
            Добавить номинал монеты
            в список
            '''
            smallercoins.append(coin)
            # Вычесть номинал монеты из суммы
            amount -= coin
    # Вернуть список монет, соответствующих заданной сумме
    return smallercoins
```

7.3. Удаление избыточных элементов из списка

По условию задачи дается список произвольных элементов. Его нужно проанализировать и удалить избыточные элементы, чтобы любой элемент исходного списка встречался в списке результата не более *n* раз. Например, пусть дан список *i* = [«Python», 4, 99, «R», 4, «43», 123, 80, 99, «Python», 4] и *n* = 2. В соответствии с этими условиями в результате должен получиться список *j* = [«Python», 4, 99, «R», 4, «43», 123, 80, 99, «Python», 4], соответствующий исходному, из которого удалены избыточные элементы так, чтобы их количество не превышало *n*. В списке *i* все элементы повторяются два раза и только элемент 4 повторяется три раза, поэтому, поскольку *n* = 2, последний элемент 4 исключается из результата. Ваша задача: написать функцию, которая принимает положительное целое число *n* и список элементов и возвращает новый список, не содержащий избыточных элемен-

тов, но сохраняющий порядок элементов в исходном списке. Если $n = 0$, функция должна вернуть пустой список.

В табл. 7.3 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.3. Некоторые ожидаемые результаты для задачи удаления избыточных элементов из списка

Items, n	Ожидаемый результат
[4, 4, 4, 4, 4, 4], 4	[4, 4, 4, 4]
[4, 5, 1000, 1, 2, 3], 5	[4, 5, 1000, 1, 2, 3]
[0, 0, 0, 0, 2, 0, 0, 0, 0], 5	[0, 0, 0, 0, 2, 0]
[1001, 'python', 1003, 'world', 1001, 1001]	[1001, 'python', 1003, 'world']

Алгоритм

Алгоритм принимает исходный список *items_list* и пороговое значение *max_frequency* и инициализирует пустой список *top_frequency_items* для хранения результата. Затем создается пустой словарь, ключами в котором служат элементы исходного списка. Значения ключей инициализируются нулями. В этом словаре подсчитывается количество вхождений каждого элемента в исходном списке. Если в процессе обхода исходного списка обнаруживается, что какой-то элемент встретился *max_frequency* или более раз, то соответствующему ключу присваивается значение *max_frequency*. Далее, после создания словаря *top_frequency_items*, выполняется цикл по списку *items_list*. Если соответствующее значение в *top_frequency_dict* больше 0, то элемент добавляется в список *top_frequency_items*, а соответствующее значение в словаре *top_frequency_dict* уменьшается на 1. По завершении обхода элементов в *items_list* возвращается список *top_frequency_items*. Алгоритм генерирует новый список, в котором каждый элемент повторяется не более *max_frequency* раз.

В листинге 7.4 приводится код на Python, решающий задачу удаления избыточных элементов из списка.

Листинг 7.4. Удаление избыточных элементов из списка

```
def calculate_item_frequency(items_list, max_frequency):
    '''
    Эта функция подсчитывает количество вхождений каждого
    элемента в заданном списке и возвращает словарь,
    ключами в котором являются элементы, а значениями –
    количество их вхождений. Она также гарантирует, что
    количество вхождений каждого элемента
```

```
    не превысит заданного значения max_frequency.
    '''
    frequency_dict = dict.fromkeys(items_list, 0)
    for item in items_list:
        frequency_dict[item] += 1
        for key in frequency_dict:
            if frequency_dict[key] > max_frequency:
                frequency_dict[key] = max_frequency
    return frequency_dict

def items_below_max_frequency(items_list, max_frequency = 1):
    '''
    Эта функция удаляет из списка избыточные элементы, количество
    вхождений которых превысило заданное значение max_frequency.
    '''
    if max_frequency == 0:
        return []
    top_frequency_dict = \
        calculate_item_frequency(items_list, max_frequency)
    top_frequency_items = []
    for item in items_list:
        if top_frequency_dict[item] > 0:
            top_frequency_items.append(item)
            top_frequency_dict[item] -= 1
    return top_frequency_items
```

7.4. Когда две лягушки встретятся в одном квадрате

Две лягушки, находясь в разных местах, начинают прыгать каждая в своем направлении. Информация о каждой лягушке представлена кортежем (*sx*, *sy*, *dx*, *dy*), где (*sx*, *sy*) обозначает начальные координаты, а (*dx*, *dy*) – постоянный вектор, определяющий направление и расстояние для каждого последующего прыжка. Проще говоря, обе лягушки представлены как *Frog1* (*sx1, sy1, dx1, dy1*) и *Frog2* (*sx1, sy1, dx1, dy1*). В этом задании нужно найти время, когда обе лягушки окажутся в одном и том же квадрате. Напишите функцию, которая принимает два кортежа, описывающих лягушек, и возвращает время, когда обе лягушки окажутся в одном квадрате. Если лягушки никогда не окажутся в одном квадрате одновременно, то функция должна вернуть *None*.

В табл. 7.4 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.4. Некоторые ожидаемые результаты для задачи определения времени встречи двух лягушек

Frog1, frog2	Ожидаемый результат
(562,–276,–10, 5), (49,–333,–1, 6)	57
(-3525,–877, 4, 1), (-4405, 2643, 5,–3)	880
(2726,–3200,–6, 7), (-2290, 2272, 5,–5)	456
(1591,–1442,–10, 9), (-329,–962, 2, 6)	160

Алгоритм

Алгоритм принимает два аргумента, *frog1* и *frog2*, представляющих начальные координаты и векторы движения двух лягушек. Алгоритм использует эту информацию, чтобы определить время встречи лягушек.

В листинге 7.5 приводится код на Python, определяющий время встречи двух лягушек.

Листинг 7.5. Определение времени встречи двух лягушек

```
def Collision_time_of_frogs(frog1, frog2):
    '''
    На основе frog1(sx1,sy1, dx1, dy1) и
    frog2(sx1, sy1, dx1, dy1) заполнить
    переменные с исходными данными.
    '''
    sx1 = frog1[0]
    sx2 = frog2[0]
    sy1 = frog1[1]
    sy2 = frog2[1]
    dx1 = frog1[2]
    dx2 = frog2[2]
    dy1 = frog1[3]
    dy2 = frog2[3]

    tx=0
    ty=0
    '''
    1:
    Если компоненты векторов движения равны нулю
    (dx1==dx2==0 и dy1==dy2==0), то это означает,
    что лягушки вообще не движутся:

```

```
    Если при этом лягушки первоначально находятся
    в одной точке, то нужно вернуть
    ноль, т. е. время до встречи равно нулю,
    так как лягушки уже встретились, не начав движения.

    Иначе, если лягушки находятся в разных точках,
    они никогда не встретятся, и нужно вернуть
    None.
    '''
    if dx1 == dx2 == 0 and dy1 == dy2 == 0:
        if sx1 == sx2 and sy1 == sy2:
            t = 0
            return t
        else:
            t = None
            return t

    '''
    2:
    Если векторы движения определяют смещение вдоль
    оси y и (dx1 == dx2 == 0), то вычислить
    позицию y точки встречи:

    Если ty недробное число
    и положительное, то вернуть ty.

    Иначе, если предыдущие условия не выполняются,
    вернуть None.
    '''
    if dx1 == dx2 == 0:
        ty = (sy2 - sy1) / (dy1 - dy2)
        if int(ty) == ty and ty > 0:
            t = int(ty)
        else:
            t = None
            return t

    '''
    3:
    Если векторы движения определяют смещение вдоль
    оси x и (dy1 == dy2 == 0),
    позицию x точки встречи:

```

```
    Если tx недробное число
    и положительное,
    то вернуть tx.

    Иначе, если предыдущие условия не выполняются,
    вернуть None.
    '''
    if dy1 == dy2 == 0:
        tx = (sx2 - sx1) / (dx1 - dx2)
        if int(tx) == tx and tx > 0:
            t = int(tx)
        else:
            t = None
            return t

    '''
    4:
    Если dx1 != dx2 (не равны):
    1: Если dy1 != dy2, то вычисляются tx и ty.
    1.2: Если tx и ty равны и имеют
    недробные положительные значения,
    то вернуть tx, иначе вернуть None.
    2: Если dy1 = dy2, то вычисляется tx.
    2.1: Если tx недробное и положительное число,
    то вернуть tx,
    иначе вернуть None.
    '''
    if dx1 != dx2:
        if dy1 != dy2:
            tx = (sx2 - sx1) / (dx1 - dx2)
            ty = (sy2 - sy1) / (dy1 - dy2)
            if tx == ty and int(tx) == tx and tx > 0:
                t=int(tx)
            else:
                t = None
                return t
        elif dy1 == dy2:
            tx = (sx2 - sx1) / (dx1 - dx2)
            if int(tx) == tx and tx > 0 and sy1 == sy2:
                t = int(tx)
            else:
                t = None
                return t
```

```
110
111     '''
112     5:
113     Если dx1 == dx2 (равны)
114     1: Если dy1 != dy2, то вычислить ty.
115     1.1: Если ty недробное и
116     положительное число, то
117     вернуть tx, иначе
118     вернуть None.
119     2: Если dy1 == dy2, то вернуть None.
120     '''
121     if dx1 == dx2:
122         if dy1 != dy2:
123             ty = (sy2 - sy1) / (dy1 - dy2)
124             if int(ty) == ty and ty > 0 and sx1 == sx2:
125                 t = int(ty)
126             else:
127                 t = None
128                 return t
129         elif dy1 == dy2:
130
131             t=None
132             return t
133     return t
```

7.5. Определение позиции числа в массиве Витхофа

Массив Витхофа – это двумерная таблица, которая характеризуется первой строкой, содержащей последовательность Фибоначчи (1, 2, 3, 5, …). Массив можно определить следующим образом: пусть *pr* обозначает предыдущую строку, *cr* обозначает текущую строку, а *a* и *b* – первый и второй элементы *pr* соответственно. Первый элемент следующей строки (исключая первую строку) определяется как наименьшее число, которое не встречается ни в одной из предыдущих строк, и обозначается как *c*. Второй элемент следующей строки определяется на основе следующих двух условий:

1) если $c - a = 2$, то значение второго элемента равно $b + 3$;
2) если $c - a \neq 2$, то значение второго элемента равно $b + 5$.

$$x = \begin{cases} b+3 & \text{если } c - a == 2 \text{ (равно 2)} \\ b+5 & \text{если } c - a \text{ != 2 (не равно 2)} \end{cases}. \quad (7.1)$$

Значение каждого следующего элемента определяется подобно числам в последовательности Фибоначчи – как сумма двух предыдущих элементов. Например, пусть *x1*, *x2* и *x3* обозначают первую, вторую и третью строки массива Витхоффа.

x1 = 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

x2 = 4, 7, 11, 18, 29, 47, 76, 123, 199,

x3 = 6, 10, 16, 26, 42, 68, 110, 178, 288.

Согласно правилам, первый элемент *x2* равен 4 – наименьшему числу, которого нет в *x1*. Чтобы определить второй элемент в *x2*, используем $a = 1$ и $b = 2$ из *x1* и $c = 4$ из *x2*. Вычисляем $c - a$: 4–1 = 3. Поскольку $3 \neq 2$, используем формулу $b + 5$ и получаем 7 – значение для второго элемента в *x2*. После получения первых двух элементов в *x2* мы можем продолжить эту последовательность, используя алгоритм Фибоначчи. Например, третий элемент получается сложением 7 и 4 – двух предыдущих элементов, в результате получается 11. Используя аналогичный подход, можно получить *x3* из *x2*. Цель этой задачи – найти позицию (индекс) заданного числа *n* в виде кортежа (*строка*, *столбец*). Ваша задача: написать функцию, которая принимает положительное целое число *n* и возвращает его позицию в виде кортежа (*строка*, *столбец*) в массиве Витхофа.

В табл. 7.5 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.5. Некоторые ожидаемые результаты для задачи поиска позиции числа в массиве Витхофа

n	Ожидаемый результат
1042	(8, 8)
13	(0, 5)
127	(48, 0)
20022	(4726, 1)

Алгоритм

Пусть *n* – заданное искомое число. Алгоритм начинается с инициализации переменных и множеств. Затем он перебирает максимально допустимое количество столбцов ($n + 1$) и генерирует два числа в текущей строке на основе меньшего и большего чисел из предыдущей строки. Меньшее число выбирается в качестве отправной точки, и алгоритм выполняет итерации до тех пор, пока не будет найдено непосещенное число. Затем на основе формулы 7.1 вычисляется следующее число. Далее алгоритм проверяет, присутствует ли искомое число в текущей строке и, если присутствует,

возвращает найденную позицию. Если число отсутствует, то алгоритм добавляет два новых числа в множество посещенных чисел, проверяет остальную часть строки на наличие целевого числа и переходит к следующей строке. Этот процесс повторяется до тех пор, пока не будет достигнуто максимальное количество столбцов.

В листинге 7.6 приводится код на Python, определяющий позицию искомого числа в массиве Витхофа.

Листинг 7.6. Определение позиции числа в массиве Витхофа

```
def Positioning_in_Wythoff_Array(n):
    # Инициализировать переменные и множества
    visited_numbers = set([])
    current_row = 0
    smaller_number, larger_number = -1, -1

    '''
    Цикл по максимально возможному
    количеству столбцов
    '''
    for _ in range(n + 1):

        if len(visited_numbers) == 0:
            previous_number, last_number = 1, 2
        else:
            '''
            Выбрать меньшее число
            в качестве отправной точки
            '''
            current_number = smaller_number

            # Выполнять цикл, пока не будет встречено непосещавшееся число
            while True:
                if current_number not in visited_numbers:

                    # Вычислить следующее число в строке
                    if current_number - smaller_number == 2:
                        next_number = larger_number + 3
                    else:
                        next_number = larger_number + 5

                    '''
                    Вернуть два числа
```

```
                    для следующей строки
                    '''

                    previous_number, last_number =\
                        current_number, next_number
                    break

                '''
                Увеличить current_number
                и повторить попытку
                '''
                current_number += 1

        '''
        Обновить переменные
        для следующей итерации
        '''
        smaller_number, larger_number =\
            previous_number, last_number

        '''
        Проверить присутствие
        целевого числа в текущей строке
        '''
        if n == previous_number:
            return (current_row, 0)
        elif n == last_number:
            return (current_row, 1)

        '''
        Добавить два числа
        в множество посещенных чисел
        '''
        visited_numbers.add(previous_number)
        visited_numbers.add(last_number)

        '''
        Проверить наличие искомого
        числа в оставшейся части строки
        '''
        column_number = 2
        while last_number <= n:
            new_number = previous_number + last_number
```

```
77
78              '''
79              Проверить, равно ли число в текущем столбце
80              искомому числу
81              '''
82              if new_number == n:
83                  return (current_row, column_number)
84
85              # Обновить переменные для следующей итерации
86              previous_number = last_number
87              last_number = new_number
88              visited_numbers.add(new_number)
89              column_number += 1
90
91          # Перейти к следующей строке
92          current_row += 1
```

7.6. Интерпретация программы на Fractran

Fractran – это язык программирования, который использует дробные числа для описания своих необычных программ. Цель этого задания – реализовать интерпретатор языка Fractran. Чтобы интерпретировать программу на Fractran, целое число *n* умножается на дроби в данном состоянии, пока не будет получено целое число, что сигнализирует об окончании программы. Первоначально состояние равно *n* и обновляется на последующих шагах. Например, предположим, что дано целое число *n*, равное 3, программа, состоящая из дробей *program* = [(7, 3), (12, 7)], и предельное количество итераций *itera*, равное 4. Попробуем интерпретировать эту программу. Инициализируем нулем указатель *i*, который может принимать значения от 0 до *itera* – 1. Затем умножаем $\frac{7}{3}$ на 3, получаем $\frac{21}{3} = 7$ и сохраняем результат в *temp*. Поскольку *temp* – целое число, сохраняем его в массив *R* и обновляем состояние *state* = *temp* = 7, начинаем выполнять программу *program* с самого начала, так как на предыдущем шаге было найдено целое число. Увеличиваем *i* на единицу: *i* = 0 + 1 = 1. Начинаем с $\frac{7}{3}$ и получаем *temp* = $7 \times \frac{7}{3} = 16.33$ – дробное число, поэтому переходим к (12,7) – следующей дроби в программе *program*. Для $\frac{12}{7}$ получаем *temp* = $7 \times \frac{12}{7} = 12$. Теперь *temp* является целым числом, поэтому сохраняем его в *R* и *state* и снова переходим к началу программы, но уже с состоянием *state* = 12, увеличиваем *i* на единицу. Теперь *i* = 2. Начинаем с $\frac{7}{3}$ и получаем *temp* = $12 \times \frac{7}{3} = 28$. *temp* является целым числом, поэтому сохраняем его в *R* и *state* и увеличиваем *i* = 3. Опять переходим к началу программы и начинаем с $\frac{7}{3}$. Получаем

temp = $28 \times \frac{7}{3} = 65.33$ – дробное число, поэтому переходим к (12, 7) – следующей дроби в программе *program*. Для $\frac{12}{7}$ получаем *temp* = $28 \times \frac{12}{7} = 48$. *temp* является целым числом, поэтому сохраняем его в *R* и *state*. Теперь *i* = *itera* – 1, и это означает, что все итерации выполнены и в *R* находится результат интерпретации. Ваша задача: написать функцию, которая принимает положительное целое число *n*, массив дробей и число итераций и возвращает массив целых чисел, созданных программой на Fractran.

В табл. 7.6 показаны ожидаемые результаты для некоторых входных данных.

Таблица 7.6. Некоторые ожидаемые результаты для задачи интерпретации программы на Fractran

n, program, itera	Ожидаемый результат
6, [(228, 131), (327, 158), (161, 394), (77, 425)], 4	[6]
2, [(345, 162), (108, 125), (90, 45), (293, 277)], 5	[2, 4, 8, 16, 32, 64]
10, [(1, 145), (7, 349), (156, 24)], 65	[10, 65]

Алгоритм

Алгоритм интерпретации программы на Fractran использует дроби для генерации последовательности чисел. Он принимает начальное число *n* и множество дробей *program*, умножает начальное число на каждую дробь (по очереди), а затем проверяет, можно ли упростить результат до целого числа. Если это возможно, то целое число добавляется в список и становится новым начальным числом для следующей итерации. Процесс продолжается до тех пор, пока не будет выполнено определенное количество итераций или пока не перестанут генерироваться целые числа. Вот подробное описание шагов, выполняемых алгоритмом.

1. Принимаются три параметра: *n*, *program* и *itera*, где *n* – начальное значение, *program* – список дробей, представленных в виде кортежей (*числитель*, *знаменатель*), и *itera* – количество итераций.
2. Затем создается список *result*, в который помещается начальное значение *n*, и устанавливаются числитель *numerator* и знаменатель *denominator* равными *n* и 1 соответственно.
3. Далее запускается цикл, выполняющий *itera* итераций, а внутри этого цикла запускается еще один цикл, перебирающий дроби в списке *program*.
4. Для каждой дроби алгоритм умножает числитель *numerator* на первое число в дроби и знаменатель *denominator* на второе число в дроби.

5. После этого вычисляется наибольший общий делитель (НОД) для получившихся числителя и знаменателя вызовом функции `gcd()` и числитель со знаменателем делятся на НОД (чтобы упростить дробь).
6. Если получившийся знаменатель равен 1, то алгоритм добавляет числитель в список результатов *result* (потому что это целое число), обновляет числитель и знаменатель до уменьшенных значений и выходит из внутреннего цикла.
7. После того как внешний цикл выполнит заданное количество итераций, алгоритм возвращает список сгенерированных значений.

В листинге 7.7 приводится код на Python, выполняющий интерпретацию программы на Fractran.

Листинг 7.7. Интерпретация программы на Fractran

```
'''
Эта функция вычисляет наибольший общий
делитель (НОД) двух чисел
'''
'''Она принимает два аргумента: a и b -
и возвращает их НОД.
'''
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)


'''Эта функция принимает три аргумента:
начальное значение n, список дробей program
и количество итераций itera.
'''
def Fractran_Interpreter(n, program, itera):
    # Инициализировать список начальным значением
    result = [n]
    numerator = n
    denominator = 1
    for i in range(itera):
        for j in range(len(program)):
            '''
            # Умножить numerator на первое
            число в дроби
```

```
            '''
            temp_num = program[j][0] * numerator
            '''
            # Умножить знаменатель на
            второе число в дроби
            '''
            temp_denom = program[j][1] * denominator
            '''
            # Вычислить НОД числителя
            и знаменателя
            '''
            div = gcd(temp_num, temp_denom)
            '''
            Уменьшить числитель, разделив
            его на НОД
            '''
            temp_num //= div
            '''
            # Уменьшить знаменатель,
            разделив его на НОД
            '''
            temp_denom //= div
            '''
            # Если знаменатель равен 1,
            то добавить числитель в список result
            и обновить значения numerator и denominator
            '''
            if temp_denom == 1:
                result.append(temp_num)
                numerator = temp_num
                denominator = temp_denom
                '''Прервать интерпретацию дробей,
                так как получено целое число
                '''
                break
    '''
    # Вернуть список значений,
    сгенерированных программой
    '''
    return result
```

Предметный указатель

И

Т

У

Ф

Ц

Ч

Ш

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать
адрес (полностью), по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Хабиб Изадха, Рашид Бехзадидуст

Решение трудных и увлекательных задач на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 19.5. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Цель данной книги – укрепить навыки логического рассуждения и развить творческое мышление, представив и решив 90 не самых простых задач на языке Python.

Задачи изложены доходчиво и сжато, снабжены с алгоритмами и комментариями, что помогает читателям следить за процессом их решения и понимать его суть.

Рассмотрены математические задачи, задачи, связанные с обработкой строк, игровые задачи и др.

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

Хабиб Изадха – доцент кафедры информатики Тебризского университета, Иран. Занимается алгоритмами, графами, применением глубокого обучения в иедиине и биоинформатике. Автор ряда научных статей, а также пяти книг, в том числе вышедших в издательствах Springer и Elsevier.

Рашид Бехзадидуст – профессор кафедры информатики Тебризского университета, Иран. Специализируется на искусственном интеллекте и обработке естественного языка.

Springer

ДМК издательство
www.дмк.рф

Страница книги на dmkpress.com

ISBN 978-5-93700-280-8