Задача 1

Реализуйте алгоритм Евклида для нахождения наибольшего общего делителя с помощью рекурсивной функции.

Алгоритм Евклида находит наибольший общий делитель чисел *A* и *B* по следующему правилу:

$$\text{НОД}(A,B) = \begin{cases} \text{НОД}(B, A\%B), & B \neq 0; \\ A, & B = 0. \end{cases}$$

Задача 2

С помощью рекурсивной функции воспроизведите следующий рисунок.

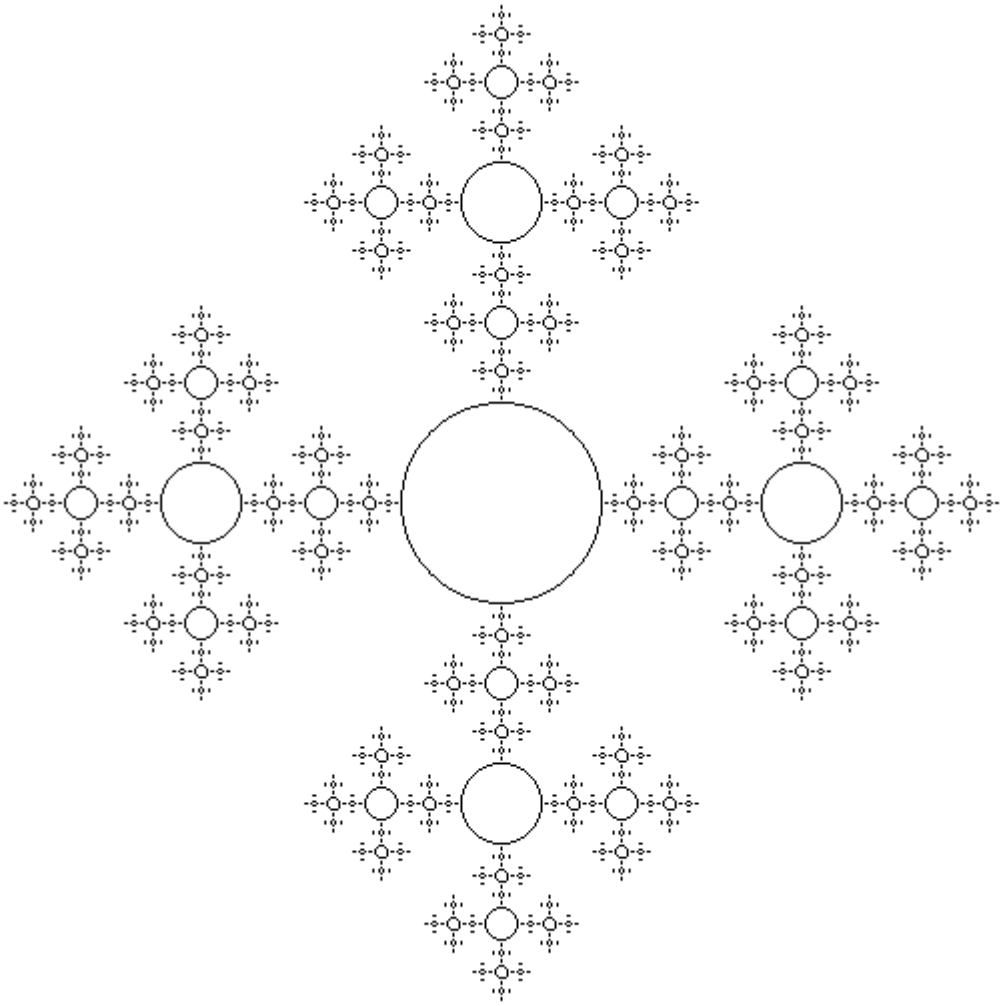

Рис. 1. Фрактальный орнамент

Указание: для рисования напишите рекурсивную функцию, параметрами которой является координаты центра области, в которой производится рисование и уровень рекурсии.На рисунке диаметр каждой следующей окружности в 2.5 раза меньше предыдущей.

Размен

Часто возникает задача, как набрать определённую сумму используя минимальное число купюр или монет.

На первый взгляд, эта задача может показаться тривиальной: казалось бы, достаточно взять как можно больше монет максимального достоинства, потом как можно больше монет следующего достоинства и так далее. Например, если требуется набрать 27 рублей можно взять 2 монеты по 10 рублей, одну монету 5 рублей и одну монету 2 рубля. Всего 4 монеты и этот вариант действительно является минимальным.

К сожалению, этот простой алгоритм[1] работает не во всех случаях. Допустим, что в некоторой стране используются монеты достоинствами 1, 5 и 8. Требуется набрать 10. Наш алгоритм говорит, что следует сначала взять монету 8 и после этого остаётся только взять три монеты по 1. Всего получается 4 монеты. Однако этот результат можно улучшить: следует взять две монеты по 5 рублей.

Одним из возможных вариантов решения задачи нахождения оптимального решения является использование следующего рекурсивного алгоритма:

1. пусть нам нужно набрать сумму N;
2. для всех доступных номиналов монет m:
   2.1. вычислить оптимальный способ набрать сумму N-m;
3. среди перечисленных на шаге 2 выбрать вариант с минимальным числом монет;
4. добавить к этому варианту соответствующую монету.

Например, если требуется набрать 10 монетами достоинствами (1, 5, 8), то нужно рассмотреть 3 варианта:

1. набрать 9 и добавить одну монету достоинством 1;
2. набрать 5 и добавить одну монету достоинством 5;
3. набрать 2 и добавить одну монету достоинством 8.

Рекурсивные вызовы дадут следующий ответ для этих случаев:

1. Набрать 9 можно 2 монетами (8 и 1);
2. Набрать 5 можно 1 монетой (5);
3. Набрать 2 можно 2 монетами (1 и 1).

Таким образом, первый и третий вариант дадут способы набрать 10 тремя монетами, а второй – двумя. Следует предпочесть второй вариант, который обеспечивает оптимальное решение.

Вычисление минимального числа монет можно описать рекурсивной формулой

$$best\_change(N, coins) = \min_{m \in coins, m \le N} best\_change(N - m, coins) + 1.$$

[1] Такие алгоритмы называют жадными. Жадный алгоритм пытается построить сложное решение делая множество шагов, на каждом из которых делается максимально эффективное простое действие. Например, в данном случае алгоритм делает решение взять самую большую из возможных монет. Но так как решения принимаются изолировано одно от другого, такой подход не всегда даёт оптимальный конечный результат.

Задание 3

Реализуйте алгоритм вычисления минимального размена в функции best_change(amount, coin). Аргументами функции являются:

1. amount – сумма, которую нужно набрать;
2. coins – кортеж со значениями доступных номиналов монет, запас монет каждого номинала считается безграничным.

Функция должна вернуть кортеж из двух элементов, включая:

1. минимальное число монет, которым можно набрать сумму amount;
2. словарь (тип dict), в котором ключами являются использованные номиналы монет, а значениями – сколько таких момент входит в минимальный набор.

Если набрать требуемую сумму не возможно, функция должна вернуть кортеж (0, None).

Примеры некоторых входов и выходов этой функции приведены в таблице 2.

Таблица 2. Ожидаемые результаты работы функции best_change

Параметры функции	Результат
best_change(0, (1, 2, 5, 10))	(0, None)
best_change(4, (5, 10))	(0, None)
best_change(10, (1, 5, 8))	(2, {5: 2})
best_change(23, (1, 3, 8))	(5, {8: 2, 1: 1, 3: 2})

Задание 4

Если попробовать использовать функцию best_change() для сумм, даже немного больших чем 20, то можно увидеть, что даже для небольших чисел функция работает долго. Причиной этого является то, что она перебирает большое количество вариантов. Рассмотрим, как вычисляется такой простой вызов как best_change(5, (1, 2)) (рис. 2).

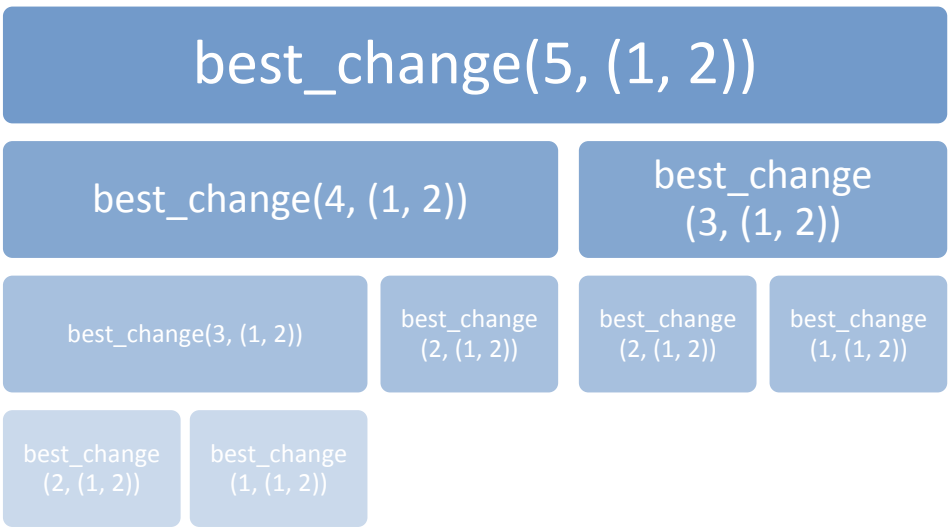

Рис. 2. Рекурсивные вызовы процедуры best_change(5, (1, 2))

Как можно видеть, в ходе обработки исходной задачи в разных ветках выполнения алгоритма по многу раз встречаются вызовы с одними и теми же параметрами, в частности:

- два раза вычисляется best_change(3,…);

- два раза вычисляется best_change(1,…);
- три раза вычисляется best_change(2,…).

Вызов best_change() с бóльшими суммами будет включать ещё больше подобных многократных вычислений одних и тех же значений.

Работу функции best_change() можно ускорить простым способом: достаточно запоминать вычисленные один раз значение для заданного входа, а затем при каждом вызове проверять, не было ли нужное нам значение уже вычислено. Если да, то следует сразу взять сохранённый результат. Это приведёт к тому, что в дереве рекурсии будет «обрезана» соответствующая ветка вычислений.

На рисунке 3 оранжевым цветом показано, какие из рекурсивных вызовов при расчёте `best_change(5, (1, 2))` смогут завершиться, не выполняя расчётов, за счёт использования запомненных значений.

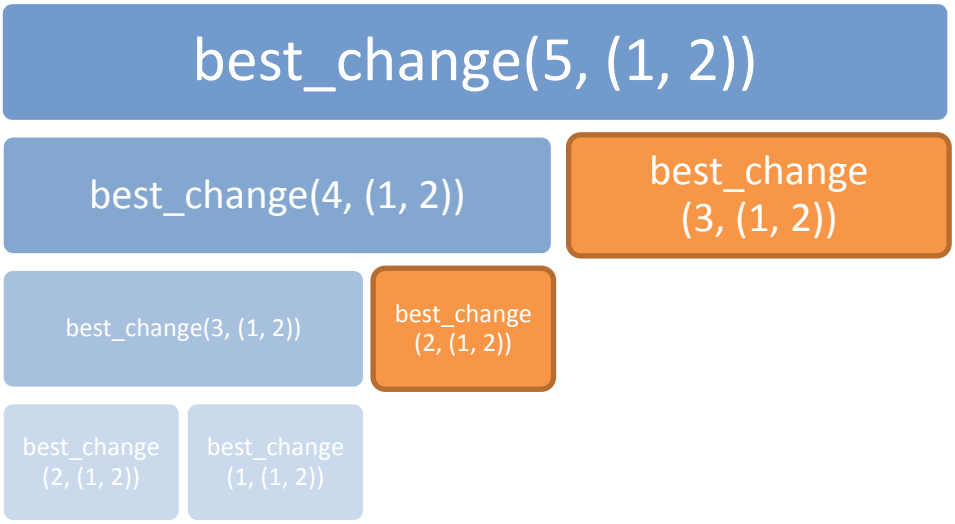

Рис. 3. Рекурсивные вызовы процедуры best_change(5, (1, 2))

Вашей задачей является реализация такой процедуры вычисления с запоминанием ранее вычисленных значений. Это можно сделать, например, следующим образом.

1. Создайте глобальную переменную со словарём, ключом в котором будет значение суммы (amount), а значением – результат работы алгоритма на этой сумме (кортеж, включающий число монет и словарь).
2. В начале функции best_change(), проверьте, нет ли в словаре ответа. Если есть, верните его.
3. В самом конце функции best_change() перед возвращением результата занесите его в словарь.
4. Не забывайте, что переменные в Python это на самом деле ссылки: может потребоваться сделать копии «ответов» из словаря, чтобы дальнейшая обработка не испортила их.

Подобная реализация сможет быстро посчитать размен для бóльших сумм, чем простой рекурсивный вариант. Обратите внимание, что, так как словарь является глобальной переменной, то он сохраняется при последующих вызовах функции. Поэтому, при вызове её в цикле для увеличивающихся значений суммы, каждый новый вызов сделает только одну итерацию рекурсивных вызовов: все они смогут сразу же вернуть готовые ответы из словаря.

При тестировании функции учтите, что если в качестве ключа словаря используется только сумма, то при последовательном вызове best_change() с разными наборами номиналов монет Вы получите неправильный ответ: второй вызов возьмёт из словаря ответ для старого набора монет. В таком случае нужно стереть словарь перед заменой набора монет.

Сделать универсальный вариант можно несколькими способами.

1. Использовать в качестве ключа в словаре пару (сумма, набор номиналов). Такой вариант подходит, если ожидается большое число вызовов функции с несколькими разными наборами номиналов в разнобой.
   Альтернативно, можно организовать глобальный словарь словарей, в котором по набору номиналов монет можно найти словарь с ответами.
2. Можно выделить отдельную функцию, которая запускает вычисления. Эта функция может создать словарь как локальную переменную, и затем запустить рекурсивную часть, отвечающую за выполнение расчётов. Словарь можно передать этой части как дополнительный параметр.
   Если желательно сохранить возможность использования результатов вычислений между вызовами функции «с наружи», то словарь можно оставить глобальной переменной и добавить ещё одну глобальную переменную, в которой будет храниться набор номиналов монет, для которого построен словарь. Нерекурсивная часть может проверять, не изменился ли набор номиналов, и в случае необходимости, стирать словарь.