Обязательные для сдачи задания

Задание 1. Наследование

В этом задании нужно будет разработать иерархию из трёх классов, которые представляют макет системы заработной платы.

Для каждого работника система хранит его имя и данные, необходимые для расчёта заработной платы. Способ расчёта заработной платы зависит от категории работника: зарплата постоянного работника фиксирована и известна заранее; зарплата работника с почасовой оплатой определяется как произведение ставки на число отработанных часов.

Для реализации макета следует реализовать иерархию из трех классов, показанную на рисунке 2.

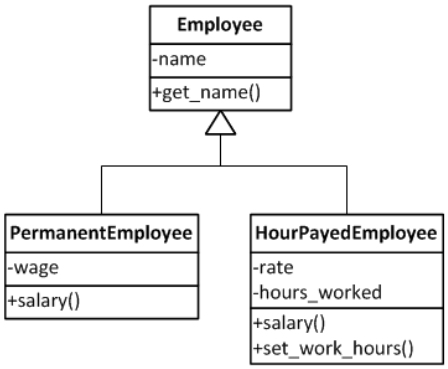

Рис. 2. Классы для модели вычисления зарплаты

Реализуйте эти классы в следующем порядке:

1. Класс Employee

Этот класс хранит общую информацию о работнике. В нашем случае это его имя.

Класс должен содержать:

- конструктор, единственным параметром которого (кроме self) является имя работника (name);
- метод get_name(), который возвращает имя работника.

2. Класс PermanentEmployee

Этот класс хранит информацию о работнике с постоянным окладом. Класс должен быть унаследован от класса Employee.

Класс должен содержать:

- конструктор, параметрами которого являются имя работника (name) и оклад (wage), в конструкторе следует вызвать конструктор базового класса и передать ему имя работника;
- метод salary(), который возвращает размер заработной платы, в данном случае – просто значение оклада.

3. Класс HourPayedEmployee

Этот класс хранит информацию о работнике с почасовой оплатой. Класс должен быть унаследован от класса Employee.

Класс должен содержать:

- конструктор, параметрами которого являются имя работника (name) и размер почасовой ставки (rate); конструктор должен:
  - вызывать конструктор базового класса и передать ему имя работника;
  - установить число отработанных часов (hours_worked) в ноль;
- метод set_work_hours(), параметром которого является число отработанных часов, и который устанавливает соответствующую переменную класса;
- метод salary(), который возвращает зарплату, равную произведению отработанного числа часов на почасовую ставку.

В случае корректной реализации, на приведённом в конце шаблона примере код должен выдать:

```
Работник, зарплата
Иванов 40000
Петров 16000
```

Задание 2. Числа Фибоначчи

Числа Фибоначчи – элементы числовой последовательности, в который каждый следующий элемент является суммой двух предыдущих. Последовательность начинают с чисел [0, 1]. Таким образом, числа Фибоначчи задаются следующей рекурсивной формулой:

$$F_n = \begin{cases} n, & n < 2; \\ F_{n-1} + F_{n-2}, & n \geq 2. \end{cases}$$

Реализуйте функцию `fibonacchi(n)`, которая должна вычислять n-ое число Фибонначи F_n.

Для вычисления числа Фибоначчи функция должна использовать рекурсию.

Задание 3. Двоичный поиск

Двоичный поиск – алгоритм поиска элемента в отсортированном массиве, использующий разделение массива на половины.

В этом задании необходимо рекурсивно реализовать алгоритм поиска элемента в списке, отсортированном по возрастанию. Результатом работы должно быть значение `True`, если элемент входит в список, и `False` – если не входит.

Алгоритм может быть сформулирован следующим образом.

1. Если список пуст, вернуть значение False.
2. Если список состоит из одного элемента, проверить, равен он искомому или нет, вернуть результат сравнения.
3. Вычислить индекс среднего элемента.
4. Если средний элемент равен искомому, вернуть `True`.
5. Если средний элемент больше искомого, выполнить поиск в половине списка, включающей все элементы с начала до середины.
6. Если средний элемент меньше искомого, выполнить поиск в половине списка начиная со следующего за средним и до конца.

Этот алгоритм выполняет манипуляции с объектом «половина списка». Если на каждом шаге создавать отдельный массив, равный половине исходного, то это приведёт к избыточному выделению памяти и копированию данных. Поэтому, в практических реализациях данные оставляют в исходном массиве, а для выделения фрагмента, с которым работает алгоритм, используют индексы начала и конца. Для упрощения работы с индексами в шаблоне задания содержится реализация класса Range (англ. диапазон), отвечающего за хранение подобных индексов и доступ к данным массива.

При создании объекта класса Range его конструктору передаётся исходный список, в котором будет производиться поиск. Исходные значения границ диапазона выставляются так, чтобы покрыть весь список, начиная от первого и заканчивая последним элементами. Затем можно перемещать эти границы с помощью методов класса. При этом объект Range обеспечит удобный доступ к элементам, входящим в соответствующий диапазон. Методы класса Range описаны в таблице 1.

Таблица 1. Методы класса Range

Метод и аргументы	**Описание**
__init__(data) **data – список или кортеж с данными.**	Создаёт диапазон, описывающий список или кортеж data. Диапазон покрывает в точности все элементы списка от первого до последнего.
__str__()	Возвращает описание данных и диапазона: полный список всех элементов в квадратных скобках, круглыми скобками показывается часть, в данный момент входящая в диапазон.
empty()	Возвращает `True`, если в диапазоне нет элементов.
__len__()	Позволяет использовать функцию len() с диапазоном. Возвращается количество элементов, входящих в диапазон.
__get_item__(index) **index – индекс элемента относительно начала диапазона.**	Позволяет использовать оператор [] для доступа к элементам исходного списка, входящим в диапазон. Элементы индексируется, начиная с 0 и относительно начала диапазона, т.е. по индексу 0 выдаётся первый элемент диапазона, и так далее до индекса len(…)-1, для последнего элемента.
set_first(index) **index – индекс элемента относительно начала диапазона.**	Передвигает начало диапазона на указанный элемент.
set_last(index) **index – индекс элемента относительно начала диапазона.**	Передвигает конец диапазона на указанный элемент.

Обратите внимание, что индекс всегда указывается относительно начала диапазона, таким образом, первый индекс «внутри» всегда 0, а средний элемент – всегда «длина делить на 2». При изменении границ диапазона индексы тоже всегда указываются относительно его начала.

Структура алгоритма двоичного поиска очень проста, однако его реализация требует большой аккуратности. Даже опытные программисты часто допускают при его реализации ошибки[1]. Список «традиционных» ошибок в реализации этого алгоритма включает:

1. алгоритм не работает на пустом списке;
2. алгоритм не работает при поиске элемента, который меньше минимального или больше максимального;
3. алгоритм не работает при поиске минимального или максимального элементов;
4. переполнение при вычислении индекса среднего элемента (в языках с ограниченной разрядностью целых чисел, в Python такого быть не может);

Использование класса Range позволяет избежать части этих проблем, но всё равно будьте осторожны.

В задании следует реализовать функцию `binary_search(element, array)`.

Аргументами этой функции являются:

1. element – значение элемента, который следует найти;
2. array – объект класса Range, описывающий диапазон для поиска.

Функция должна вернуть значение `True` если элемент присутствует в диапазоне, и `False` если его там нет. В том числе, следует вернуть `False` если диапазон пуст.

Функция должна быть реализована с помощью рекурсии.

В конце шаблона находится несколько примеров вызова этой функции. Вспомогательная функция `print_search_result()` создаёт объект класса `Range` и обеспечивает красивую печать результатов.

В начале функции binary_search() находится оператор печати, который печатает состояние диапазона поиска. Если Вы оставите эту печать, то сможете увидеть, как работает алгоритм. Это позволит лучше понять работу алгоритма и поможет его отладить, если он будет работать некорректно.

Оценка обязательной части

1	Наследование	- 45 %
2	Числа Фибоначчи	- 10 %.
3	Двоичный поиск	- 45 %.

[1] Исследование 1988 года опубликованное в статье Pattis, Richard E.Textbook Errors in Binary Search, SIGCSE '88 Proceedings of the nineteenth SIGCSE technical symposium on Computer science education, ACM: New York, NY, USA, ISBN: 0-89791-256-X показало, что три четверти учебников по программированию приводят в качестве примеров версии с ошибкой.

Дополнительные задания

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[2] 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

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

[2] 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 4

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 5

В конце шаблона для задания 3 находится цикл, который вычисляет и печатает оптимальный размен для сумм с 0 до 25. Если увеличить верхнюю границу, 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).

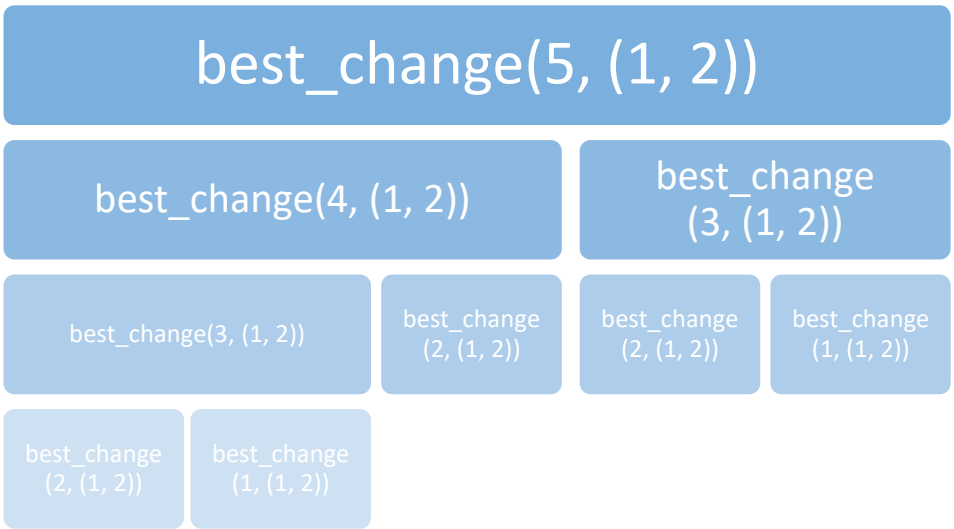

Рис. 2. Рекурсивные вызовы процедуры best_change(5, (1, 2))

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.

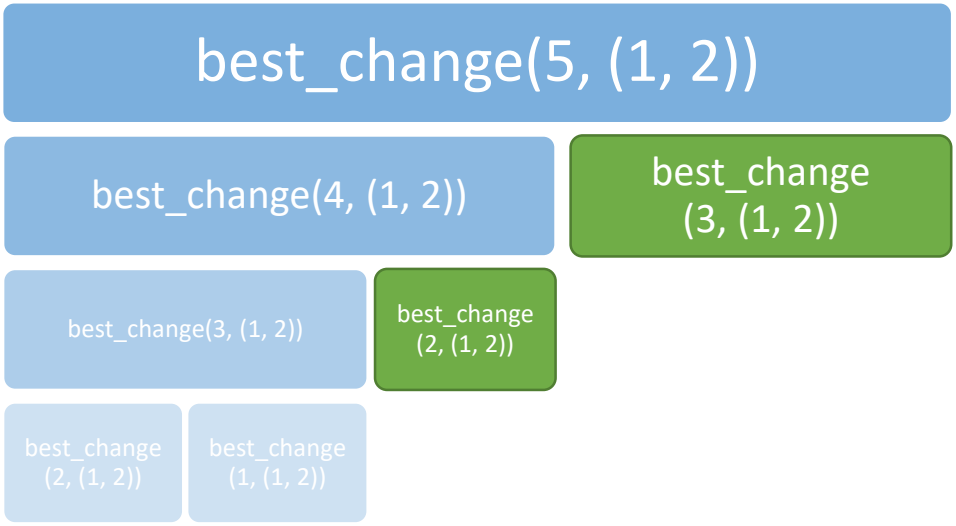

Рис. 3. Рекурсивные вызовы процедуры best_change(5, (1, 2))

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.

Рекомендуется стереть словарь в конце файла, чтобы не повлиять на работу автоматических тестов.

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.

Оценка дополнительной части

Задание 4. Размен – 60 %.

Задание 5.1. Быстрая версия размена – 30 %.

Задание 5.2. Быстрая версия размена, устойчивая к замене номиналов – 10 %.