| МИНОБРНАУКИ РОССИИ | | |
| --- | --- | --- |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Язык программирования Python»  Вариант 2 | | |
|  | | Работу выполнил  студент группы ИТ-5-2024 1 курса  Шутов С.А  «12» июня 2025 г. |
| Работу проверил  \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фамилия И.О.  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2025 г. |
| Пермь 2024 | | |

**СОДЕРЖАНИЕ**

[Постановка задачи 3](#_m1b30i7a8e73)

[Алгоритм решения 3](#_afeyqu8cz5ke)

[Тестирование 3](#_j59pgw98e7eq)

[Код программы 3](#_af8hh428f120)

# Постановка задачи

# Игра в слова. Дан набор слов. Известно, что эти слова использовались для игры, в которой очередное слово должно заканчиваться той буквой, на которую закончилось предыдущее слово. С какого слова начинается игра, не имеет значения. Если слово оканчивается на «ь», то используется предыдущая буква. Последнее слово цепочки должно заканчиваться буквой первого слова цепочки. Входной файл содержит все слова в одной строке, слова записаны строчными буквами русского алфавита, между словами может быть один или более пробелов. Требуется выстроить данные слова в цепочку согласно правилам игры. Достаточно найти одно подходящее решение. Возможен вариант, что решений не существует.

# Алгоритм решения

**1. Постановка задачи** Дан набор слов, изначально записанных в одну строку через пробелы. Требуется упорядочить их в «цепочку» так, чтобы каждое последующее слово начиналось на ту букву, на которую окончится предыдущее. Если слово оканчивается на мягкий знак «ь», считается пред­последняя буква. Начать можно с любого слова, но при этом последнее слово должно заканчиваться на первую букву первого. Если такого порядка не существует, сообщить об этом.

**2. Ключевая идея** – Каждое слово представляем как ориентированное ребро: из вершины-буквы, равной первой букве, в вершину-букву, равную последнему значимому символу.  
 – Задача превращается в поиск цикла, проходящего по всем рёбрам ровно один раз (эйлеров цикл).  
 – На практике, поскольку граф небольшой, достаточно выполнить глубинный поиск (DFS) с учётом использованных слов и вернуть первую найденную замкнутую последовательность.

**3. Подробный алгоритм**

1. **Разбор входных данных**
   * Считываем строку, разбиваем по пробелам, фильтруем пустые токены.
   * Проверяем: есть ли хотя бы одно слово.
2. **Определение значимых букв**
   * Для каждого слова w:  
     1. start = w[0]
     2. end = w[-2], если w.endswith('ь') и len(w)>1, иначе w[-1].
3. **Построение графа**
   * Создать словарь смежности graph: Dict[буква] → List[(буква, слово)].
   * Для каждого слова добавить в graph[start].append((end, слово)).
4. **Поиск цепочки (DFS + бэктрекинг)**
   * Для каждой вершины-начальной буквы или прямо для каждого слова:  
     1. Завести пустые структуры:  
        + used\_words — множество уже помещённых в цепочку слов.
        + current\_path — список слов в порядке обхода.
     2. Рекурсивная функция search(word):  
        + Добавить word в current\_path и used\_words.
        + Если len(current\_path) == total\_words:  
          1. Проверить, совпадает ли конец последнего слова с началом первого (end(current\_path[-1]) == start(current\_path[0])).
          2. Если да → решение найдено. Иначе — откатиться.
        + Иначе:  
          1. Вычислить last\_letter текущего word.
          2. Перебрать все (next\_letter, next\_word) в graph[last\_letter].
          3. Для каждого next\_word, не входящего в used\_words, вызвать search(next\_word).

Если вложенный вызов вернул успех → сразу вернуть успех.

* + - * Если ни один путь не привёл к успеху → удалить word из used\_words и current\_path (откат) и вернуть неуспех.
  + Запустить search(initial\_word) для каждого initial\_word из списка.

**4. Сложность**

* **Временная:** в худшем случае при полном переборе путей DFS получается экспоненциальная сложность, примерно O(k·n!), где n — количество слов, k — среднее число вариантов продолжения на каждом шаге. На практике для n≲15 алгоритм работает быстро, поскольку большая часть ветвей отсеивается при проверке использованных слов и замкнутости цикла.
* **Память:** O(n) для хранения графа и O(d) дополнительной памяти на рекурсивный стек (где d ≤ n — глубина поиска, равная длине пути).

**5. Итог** Алгоритм для «Игры в слова» использует только примитивные операции языка (строки, списки, множества, простая арифметика индексов) и собственные структуры данных (словарь для графа и рекурсивный вызов вместо сторонних библиотек). Решение универсально, его легко обосновать через инвариант DFS: на каждом шаге текущий путь корректен и не содержит повторений, а проверка замкнутости гарантирует выполнение условий задачи.

# Тестирование

| № | Входная строка | Ожидаемый результат | Что проверяем |
| --- | --- | --- | --- |
| 1 | хмель мороз налог лимон запах дым лошадь холм гол молох налог гол лошадь дым мороз запах холм молох хмель лимон | налог гол лошадь дым мороз запах холм молох хмель лимон | Большой набор, цепочка существует |
| 2 | а б в   |  | | --- |  |  | | --- | | Цепочку составить невозможно.   |  | | --- |  |  | | --- | | Нет пути   |  | | --- |  |  | | --- | |
| 3 | лошадь даль   |  | | --- |  |  | | --- | | лошадь даль   |  | | --- |  |  | | --- | | Учёт «ь»   |  | | --- |  |  | | --- | |
| 4 | дом | Цепочку составить невозможно.   |  | | --- |  |  | | --- | | Одиночное слово, конец ≠ начало   |  | | --- |  |  | | --- | |
| 5 | токот   |  | | --- |  |  | | --- | | токот | Одиночное слово, конец = начало   |  | | --- |  |  | | --- | |
| 6 | абв гва авг   |  | | --- |  |  | | --- | | абв гва авг   |  | | --- |  |  | | --- | | Убираются лишние пробелы |
| 7 | а а а   |  | | --- |  |  | | --- | | а а а   |  | | --- |  |  | | --- | | Повторы слов   |  | | --- |  |  | | --- | |
| 8 |  | Ошибка: не введено ни одного слова.   |  | | --- |  |  | | --- | | Пустая строка   |  | | --- |  |  | | --- | |
| 9 | рак кар   |  | | --- |  |  | | --- | | рак кар   |  | | --- |  |  | | --- | | Минимальная цепочка из двух слов   |  | | --- |  |  | | --- | |
| 10 | мать тьма ам   |  | | --- |  |  | | --- | | мать тьма ам   |  | | --- |  |  | | --- | | «ь» в двух словах подряд   |  | | --- |  |  | | --- | |
| 11 | а б а   |  | | --- |  |  | | --- | | а б а   |  | | --- |  |  | | --- | | Однобуквенные слова   |  | | --- |  |  | | --- | |
| 12 | ананас скала лад   |  | | --- |  |  | | --- | | ананас скала лад   |  | | --- |  |  | | --- | | Простая трёхсловная цепочка |

# Код программы

[ссылка на гитхаб](https://github.com/SaveliyShutov/IKM_ShutovSaveliy)

class WordChainGame:

"""

Строит цепочку слов, где каждое следующее слово начинается

на букву, на которую заканчивается предыдущее (без учёта 'ь'),

и последний элемент замыкается на первую букву первого.

Различает даже одинаковые слова по их позициям.

"""

def \_\_init\_\_(self, rawInput: str):

# Разбираем строку: убираем лишние пробелы, оставляем только слова

self.words = self.splitWords(rawInput)

if not self.words:

raise ValueError("Ошибка: не введено ни одного слова.")

# Граф: ключ — буква, значение — список (целёвая буква, индекс слова)

self.graph = self.buildGraph()

def splitWords(self, line: str) -> list[str]:

"""Разбивает входную строку на слова, игнорируя пустые."""

return [tok for tok in line.strip().split() if tok.isalpha()]

def getLastLetter(self, word: str) -> str:

"""Возвращает последнюю значимую букву (не считая 'ь')."""

if word.endswith('ь') and len(word) > 1:

return word[-2]

return word[-1]

def buildGraph(self) -> dict[str, list[tuple[str, int]]]:

"""

Для каждого слова с индексом i:

определяет start = word[0], end = getLastLetter(word)

и кладёт в graph[start].append((end, i))

"""

graph: dict[str, list[tuple[str, int]]] = {}

for i, w in enumerate(self.words):

start = w[0]

end = self.getLastLetter(w)

graph.setdefault(start, []).append((end, i))

return graph

def searchChain(self,

currentIndex: int,

usedIndices: set[int],

pathIndices: list[int]) -> bool:

"""

Рекурсивный DFS + бэктрекинг по индексам слов:

• currentIndex — индекс текущего слова;

• usedIndices — уже взятые индексы;

• pathIndices — текущий путь индексов.

Возвращает True, если найден замкнутый путь длины total\_words.

"""

pathIndices.append(currentIndex)

usedIndices.add(currentIndex)

# Если включили все слова — проверяем замыкание

if len(pathIndices) == len(self.words):

firstLetter = self.words[pathIndices[0]][0]

lastLetter = self.getLastLetter(self.words[currentIndex])

return lastLetter == firstLetter

# Иначе продолжаем DFS по ребрам из буквы конца currentWord

nextLetter = self.getLastLetter(self.words[currentIndex])

for destLetter, nxtIdx in self.graph.get(nextLetter, []):

if nxtIdx not in usedIndices:

if self.searchChain(nxtIdx, usedIndices, pathIndices):

return True

# Откат: убираем текущий индекс

usedIndices.remove(currentIndex)

pathIndices.pop()

return False

def findChain(self) -> list[str] | None:

"""

Перебираем все слова по индексам в качестве старта.

Если из какого-то индекса DFS возвращает True — собираем цепочку строк.

"""

for startIdx in range(len(self.words)):

used: set[int] = set()

path: list[int] = []

if self.searchChain(startIdx, used, path):

# Возвращаем слова в порядке найденного пути

return [self.words[i] for i in path]

return None

def main():

"""

Читает одну строку со словами через пробел,

запускает поиск цепочки и выводит результат.

"""

print("Введите слова через пробелы:")

raw = input().strip().lower()

try:

game = WordChainGame(raw)

chain = game.findChain()

if chain:

print("Найденная цепочка:")

print(" ".join(chain))

else:

print("Цепочку составить невозможно.")

except ValueError as err:

print(err)

if \_\_name\_\_ == "\_\_main\_\_":

main()