|  |  |  |
| --- | --- | --- |
| 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 6 | | |
|  | | Работу выполнил  студент группы ИТ-6,И-2024 1 курса  Кочетов Иван Витальевич  «18» Июня 2024 г. |
| Работу проверил  \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фамилия И.О.  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2024 г. |
| Пермь 2024 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#_Toc153130027)

[Алгоритм решения 3](#_Toc153130028)

[Тестирование 3](#_Toc153130029)

[Код программы 3](#_Toc153130030)

# Постановка задачи

# Для заданного подмножества набора костей домино определить, можно ли их выложить в ряд, не нарушая правил. Если можно, то представить один любой вариант такого разложения. Например, для входных данных 31, 00, 13, получаем ответ: некорректные входные данные; для входных данных 02, 04, 42 ответ: можно, 04, 42, 20. Использовать двухсвязный список. Алгоритм решения

**Разработать программу, которая**:

1. Принимает на вход набор костей домино в виде строки (например, "12, 34, 56")
2. Проверяет возможность построения из них непрерывной цепи по правилам домино
3. Выводит результат - либо построенную цепь, либо сообщение о невозможности построения

**Алгоритм решения задачи.**

* **1. Ввод и валидация данных**
* **Шаг 1.1**: Пользователь вводит набор костей домино в формате 12, 34, 56.
* **Шаг 1.2**: Программа проверяет корректность ввода:
  + Каждая кость должна состоять из 2 цифр.
  + Цифры должны быть в диапазоне от 0 до 6.
  + Недопустимы пустые значения или символы.
* **Шаг 1.3**: Если данные некорректны — выводится ошибка.
* **2. Построение двусвязного списка костей**
* **Шаг 2.1**: Создается экземпляр класса DominoLinkedList.
* **Шаг 2.2**: Для каждой кости из ввода:
  + Создается узел DominoNode с полями:
    - value (значение кости, например, "12"),
    - prev и next (ссылки на соседние узлы),
    - used (флаг использования в цепи).
  + Узел добавляется в конец списка.
* **3. Построение графа смежности**
* **Шаг 3.1:**Создается граф в формате словаря, где**:**
  + Ключи — числа на костях (от 0 до 6),
  + Значения — списки смежных чисел.
* **Шаг 3.2: Для каждой кости ab в списке:**
  + Добавляются рёбра a → b и b → a.  
    *Пример:* Для кости 12 граф получит связи 1: [2] и 2: [1].
* **4. Проверка условий существования эйлерова пути**
* **Шаг 4.1:**Вычисляются степени всех вершин (количество рёбер).
* **Шаг 4.2:**Определяются вершины с нечётными степенями**:**
  + Если таких вершин 0 или 2 — путь существует.
  + Если больше 2 — цепь построить нельзя.
* **Шаг 4.3:**Выбор начальной вершины:
  + Если есть 2 нечётных — начинаем с одной из них.
  + Если все чётные — начинаем с любой.
* **5**. Поиск эйлерова пути алгоритмом на основе стека
* **Шаг 5.1:**Инициализация стека с начальной вершиной.
* **Шаг 5.2:**Пока стек не пуст:
  1. Берем вершину с вершины стека (current).
  2. Если у нее есть непосещённые рёбра:
     + Переходим по первому ребру, удаляем его из графа.
     + Добавляем следующую вершину в стек.
  3. Если рёбер нет — переносим current в путь.
* **Шаг 5.3**: Проверка длины пути:
  1. Если путь включает все рёбра + 1 (n костей + 1), он корректен.
* **6. Построение цепи домино**
* **Шаг 6.1:**Для каждой пары вершин в пути (например, 1 → 2 → 3):
  + Ищется соответствующая кость в списке (12, 23).
  + Проверяется, не использована ли она ранее.
* **Шаг 6.2:**Кость добавляется в цепь в правильной ориентации:
  + Если 12 подходит под 1 → 2, оставляем 12.
  + Если подходит только 21 — разворачиваем (12 → 21).
* **Шаг 6.3**: Если все кости использованы — цепь построена.
* **7. Вывод результата**
* Если цепь возможна:  
  Выводится последовательность костей, например:  
  Можно: 12, 23, 34.
* Если цепь невозможна:  
  Выводится сообщение: Нельзя выложить в ряд.

# Тестирование

**Корректные цепи (должны работать)**

Простая линейная цепьИзображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Цепь с дублями

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

Длинная цепь

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание

Цепь с нулями

Изображение выглядит как текст, снимок экрана, Шрифт, часы

Автоматически созданное описание

**Некорректные цепи (должны выдавать "Нельзя")**

Несвязный граф

Изображение выглядит как текст, Шрифт, снимок экрана, дизайн

Автоматически созданное описание

Нет связей между костями

Изображение выглядит как текст, Шрифт, снимок экрана, дизайн

Автоматически созданное описание

**Тесты валидации (должны выдавать ошибку)**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

# Код программы

class DominoNode:  
 #узел двусвязного списка для хранения кости домино"""  
  
 def \_\_init\_\_(self, value):  
 self.value = value # значение кости (например, "12")  
 self.prev = None # ссылка на предыдущий узел  
 self.next = None # ссылка на следующий узел  
 self.used = False # флаг использования в цепи  
  
  
class DominoLinkedList:  
 #класс двусвязного списка для хранения набора костей домино"""  
  
 def \_\_init\_\_(self):  
 self.head = None  
 self.tail = None  
 self.length = 0  
  
 def add(self, value):  
 # добавление кости в конец списка  
 new\_node = DominoNode(value)  
  
 if not self.head:  
 self.head = new\_node  
 self.tail = new\_node  
 else:  
 self.tail.next = new\_node  
 new\_node.prev = self.tail  
 self.tail = new\_node  
 self.length += 1  
  
 def find(self, a, b):  
 # поиск кости с заданными значениями (прямого или перевернутого варианта)  
 current = self.head  
 while current:  
 x, y = map(int, current.value)  
 if (x == a and y == b) or (y == a and x == b):  
 return current  
 current = current.next  
 return None  
  
  
class DominoChainSolver:  
 #класс для проверки возможности построения цепи  
  
 def \_\_init\_\_(self, linked\_list):  
 self.linked\_list = linked\_list  
 self.graph = self.\_build\_graph()  
  
 def \_build\_graph(self):  
 # построение графа смежности  
 graph = {}  
 current = self.linked\_list.head  
 while current:  
 a, b = map(int, current.value)  
 graph.setdefault(a, []).append(b)  
 graph.setdefault(b, []).append(a)  
 current = current.next  
 return graph  
  
 def \_find\_eulerian\_path(self):  
 # поиск эйлерова пути  
 degrees = {node: len(edges) for node, edges in self.graph.items()}  
 odd\_nodes = [node for node in degrees if degrees[node] % 2 != 0]  
  
 if len(odd\_nodes) not in (0, 2):  
 return None  
  
 start\_node = odd\_nodes[0] if odd\_nodes else next(iter(self.graph.keys()))  
 path = []  
 stack = [start\_node]  
 temp\_graph = {k: v.copy() for k, v in self.graph.items()}  
  
 while stack:  
 current = stack[-1]  
 if temp\_graph.get(current):  
 next\_node = temp\_graph[current].pop()  
 temp\_graph[next\_node].remove(current)  
 stack.append(next\_node)  
 else:  
 path.append(stack.pop())  
  
 return path[::-1] if len(path) == self.linked\_list.length + 1 else None  
  
 def solve(self):  
 # проверка возможности построения цепи  
 path = self.\_find\_eulerian\_path()  
 if not path:  
 return None  
  
 chain = []  
 for i in range(len(path) - 1):  
 a, b = path[i], path[i + 1]  
 node = self.linked\_list.find(a, b)  
 if node and not node.used:  
 node.used = True  
 if node.value[0] == str(a):  
 chain.append(node.value)  
 else:  
 chain.append(node.value[::-1])  
 return chain if len(chain) == self.linked\_list.length else None  
  
  
class DominoGame:  
 #класс игры  
  
 def \_\_init\_\_(self):  
 self.list = DominoLinkedList()  
  
 def run(self):  
 # основной игровой цикл  
 print("╔════════════════════════════════════╗")  
 print("║ ДОМИНО: ПРОВЕРКА ЦЕПИ ║")  
 print("╚════════════════════════════════════╝")  
 print("Введите кости домино через запятую (например: 12, 34, 56)")  
  
 while True:  
 try:  
 input\_str = input("> ").strip()  
 if not input\_str:  
 print("Ошибка: введите хотя бы одну кость!")  
 continue  
  
 self.\_process\_input(input\_str)  
 chain = self.\_solve\_chain()  
 self.\_print\_result(chain)  
  
 except ValueError as e:  
 print(f"Ошибка: {e}")  
 except KeyboardInterrupt:  
 print("\nИгра завершена.")  
 break  
 except Exception as e:  
 print(f"Неожиданная ошибка: {e}")  
  
 def \_process\_input(self, input\_str):  
 # обработка ввода пользователя  
 self.list = DominoLinkedList() # сброс предыдущего списка  
  
 for item in input\_str.split(","):  
 value = item.strip()  
 if not value:  
 continue  
 if len(value) != 2 or not value.isdigit():  
 raise ValueError(f"Некорректная кость: {value}")  
 a, b = map(int, value)  
 if not (0 <= a <= 6 and 0 <= b <= 6):  
 raise ValueError(f"Значения кости должны быть от 0 до 6")  
 self.list.add(value)  
  
 if not self.list.head:  
 raise ValueError("Не введено ни одной кости")  
  
 def \_solve\_chain(self):  
 # попытка построить цепь  
 solver = DominoChainSolver(self.list)  
 return solver.solve()  
  
 def \_print\_result(self, chain):  
 # вывод результата  
 if chain:  
 result = "Можно: " + ", ".join(chain)  
 border = "═" \* (len(result) + 2)  
 print(f"\nРезультат:\n╔{border}╗\n║ {result} ║\n╚{border}╝\n")  
 else:  
 print("\nРезультат: Нельзя выложить в ряд\n")  
  
  
# запуск программы  
if \_\_name\_\_ == "\_\_main\_\_":  
 game = DominoGame()  
 game.run()