Министерство науки и высшего образования РФ
Федеральное государственное бюджетное учреждение высшего образования
Тверской государственный технический университет
(ТвГТУ)
Факультет информационных технологий
Кафедра Программное обеспечение

Теория алгоритмов

Курсовая работа
Задача о Ханойской башне

Работу выполнил:
А. В. Малов
Группа:
Б.ПИН.РИС-20.05
Проверила:
старший
преподователь
кафедры ПО
Е. И. Корнеева

Тверь
2022

Содержание

Введение

Алгоритм в задаче о Ханойской башне находит последовательность действий, чтобы переместить диски с одной башни на другую, ограничиваясь правилами игры.

Актуальность данной темы средняя, так как очень мало применений, кроме как образовательных. Программистам-новичкам с помощью данной задачи легко показать как работает рекурсия. Рекурсия очень важна в компьютерных науках, ведь только с помощью рекурсии можно строго математически перейти к процедурному-императивному циклу. Зачастую рекурсивные задачи выглядят элегантно и красиво, что даёт особой шарм, при написании такого кода.

Цель: реализовать графическое приложение с алгоритмами решения задачи о Ханойское башне рекурсивным и итеративным алгоритмами на динамически типизированном языке программирования Python[3].

Задачи:

- Научиться решать задачу о Ханойской башне
- Сделать графический интерфейс для игры в этот пазл
- Вся реализация должна быть на языке программирования Python
- Написать юнит-тесты для библиотеки

1 Ханойская башня

Ханойская башня (также называемая проблемой храма Бенареса или Башней Брахмы или башней Лукаса, а иногда во множественном числе как Башни, или просто пирамидальная головоломка) — математическая игра или головоломка, состоящая из трех стержней и нескольких дисков различного диаметра, которые могут перемещаться по любому стержню. Головоломка начинается с того, что диски укладываются на один стержень в порядке уменьшения размера, наименьший вверху, таким образом, приближаясь к конической форме. Цель головоломки — переместить всю стопку до последнего стержня, соблюдая следующие правила:

1. Одновременно можно перемещать только один диск.
2. Каждый ход состоит в том, чтобы взять верхний диск из одной из стопок и поместить его поверх другой стопки или на пустой стержень.
3. Ни один диск не может быть помещен поверх диска, который меньше его.

Имея 3 диска, головоломку можно решить за 7 ходов. Минимальное количество ходов, необходимых для решения головоломки Ханойской башни, равно $2n-1$, где n — количество дисков.

Загадка была представлена на Западе французским математиком Эдуардом Лукасом в 1883 году. Почти сразу же всплыли[1] многочисленные мифы о древней и мистической природе головоломки, в том числе об индийском храме в Каши Вишванатхе, в котором находится большая комната с тремя потертыми от времени столбами, окруженными 64 золотыми дисками. Исполняя повеление древнего пророчества, священники-брахманы с тех пор передвигают эти диски в соответствии с непреложными правилами Брахмы. Поэтому головоломка также известна как Башня Брахмы. Согласно легенде, когда будет выполнен последний ход головоломки, наступит конец света.[6]

Если бы легенда была правдой, и если бы священники могли перемещать диски со скоростью один в секунду, используя наименьшее количество ходов, им потребовалось бы $2^{64} - 1$ секунды или примерно 585 миллиардов лет, чтобы закончить, что примерно в 42 раза превышает нынешний возраст вселенной.

Существует множество вариаций этой легенды. Например, в некоторых рассказах храм — это монастырь, а священники — монахи. Храм или монастырь могут находиться в разных местах, включая Ханой, и могут быть связаны с любой религией. В некоторых версиях вводятся другие элементы, например, тот факт, что башня была создана в начале мира, или что священники или монахи могут совершать только один ход в день.

На текущий момент Ханойские башни используются в играх-головоломках. В образовательных целях данная задача даётся как пример элегантного рекурсивного решения. Можно найти веб-сайты, которые представляют собой игру в Ханойскую башню.

1.1 Решение

В головоломку можно играть с любым количеством дисков, хотя во многих игрушечных версиях их от 7 до 9. Минимальное количество ходов, необходимых для решения головоломки Ханойской башни, равно $2n - 1$, где n — количество дисков.[5]

1.1.1 Итеративное решение

Простое решение игрушечной головоломки состоит в том, чтобы чередовать ходы между самым маленьким и не самым маленьким фрагментами. Перемещая самую маленькую фигуру, всегда перемещайте ее на следующую позицию в том же направлении (вправо, если начальное количество фигур четное, влево, если начальное количество фигур нечетное). Если в выбранном направлении нет позиции башни, переместите фигуру в противоположный конец, но затем продолжайте двигаться в правильном направлении. Например, если вы начали с трех фигур, вы должны переместить самую маленькую фигуру в противоположный конец, а затем продолжить движение в левом направлении после этого. Когда наступает очередь переместить немаленькую фигуру, остается только один допустимый ход. Выполнение этого алгоритма позволит завершить головоломку за наименьшее количество ходов.

Более простая формулировка итеративного решения

Для четного числа дисков:

- сделайте законный ход между колышками A и B (в любом направлении),
- сделайте законный ход между колышками A и C (в любом направлении),
- сделайте законный ход между колышками B и C (в любом направлении),
- повторяйте до завершения.

Для нечетного числа дисков:

- сделайте законный ход между колышками A и C (в любом направлении),
- сделайте законный ход между колышками A и B (в любом направлении),
- сделайте законный ход между колышками B и C (в любом направлении),
- повторяйте до завершения.

В каждом случае делается в общей сложности $2n - 1$ хода.

1.1.2 Рекурсивное решение

Ключом к рекурсивному решению проблемы является признание того, что ее можно разбить на набор более мелких подзадач, к каждой из которых применяется та же общая процедура решения, которую мы ищем, и затем общее решение находится каким-то простым способом из решений этих подзадач. Каждая из этих созданных подзадач, будучи "меньшей", гарантирует, что базовый вариант (варианты) в конечном итоге будут достигнуты:

- обозначьте колышки A, B, C,
- пусть n — общее количество дисков,
- пронумеруйте диски от 1 (самый маленький, самый верхний) до n (самый большой, самый нижний).

Предполагая, что все n дисков распределены между колышками в правильном порядке; предполагая, что на исходном колышке имеется m верхних дисков, а все остальные диски больше m, поэтому их можно безопасно игнорировать; чтобы переместить m дисков с исходного колышка на целевой колышек с помощью запасного колышка, без нарушения правил:

1. Переместите диски $m-1$ от источника к запасному стержню, используя ту же общую процедуру решения. Правила не нарушаются, по предположению. Это оставляет диск m в качестве верхнего диска на исходном колышке.
2. Переместите диск m от источника к целевому стержню, что гарантированно является допустимым ходом, по предположениям — простой шаг.
3. Переместите диски $m-1$, которые мы только что поместили на запасной, с запасного на целевой колышек с помощью той же общей процедуры решения, чтобы они были размещены поверх диска m без нарушения правил.
4. Базовый вариант – переместить 0 дисков (на этапах 1 и 3), то есть ничего не делать, что, очевидно, не нарушает правила.

Затем полное решение Ханойской башни состоит в перемещении n дисков от исходного колышка A к целевому колышку C, используя B в качестве запасного колышка.

Этот подход может быть строго математически доказан с помощью математической индукции и часто используется в качестве примера рекурсии при обучении программированию.

1.1.3 Анализ сложности и сравнение алгоритмов

Проведем анализ временных затрат для ханойских башен (и всех задач, сводящихся к решению двух подзадач размерности $n-1$). Подсчитаем требуемое число ходов $T(n)$. С учетом структуры решения:

$$T(n) = 2T(n-1) + 1$$

Простое доказательство по индукции дает:

$$T(n) = -1 + -2 + \cdots + 2 + 1 = 2n - 1$$

Алгоритм решения задачи о Ханойских башнях является конечным, так как все используемые циклы выполняются конечное число раз.

Сложность — количественная характеристика алгоритма, которая говорит о том, сколько времени он работает (временная сложность), либо о том, какой объем памяти он занимает (емкостная сложность). На практике сложность рассматривают как временную сложность.

Из определения сложности следует, что она зависит от размерности входных данных или, как говорят, от длины входа. В задаче о Ханойских башнях входными данными является число дисков. Рассчитаем порядок временной сложности в соответствии с пошаговым алгоритмом. Временная сложность процедуры будет зависеть от количества переносов, которое равно $2n - 1$, значит $(2n - 1)$.

Рекурсивное и итеративное решение в приведённом виде может оптимально решать задачу. Если увеличить количество башен, то можно уменьшить общее количество ходов, путём раскидывания дисков по всем башням. Такой алгоритм можно сделать, но он требует гораздо больше усилий в реализации, чем алгоритм предназначенный для трёх башен.

2 Реализация

Был создан проект с помощью инструмента Poetry[2], который поможет регулировать зависимости. Так как проект у нас будет использовать графику, то подключим библиотеку Pygame[4], чтобы удобно рисовать объекты и текст на экране.

Проект будет в виде игры, где игрок сможет самостоятельно попробовать решить задачу о Ханойской башне. Будет возможность дать игроку наблюдать за решением, который будет реализован с помощью двух режимов: рекурсивный и итеративный. Будет возможность динамически изменять количество дисков и количество стержней.

Так как в игре будут чёткие состояние, то можно воспользоваться одним шаблоном программирования, который облегчит реализацию данной концепции. Данный шаблон проектирования называется “паттерн состояние”, про который можно прочитать в книге банды четырёх.[7]

В проекте были выделены состояния:

- GameState — главное состояние игры
- HelpState — состояние меню помощи
- ChangeDiskCountState — состояние изменения количества дисков
- ChangeTowerCountState — состояние изменения количества башен
- RestartState — состояние перезапуска игры (диски перестраиваются на начальную башню)
- SolveState — Состояние решения игры
- RecursiveSolveState — Состояние решение игры рекурсивным способом
- IterativeSolveState — Состояние решения игры итеративным способом
- MoveState — Состояние игры, в котором игрок может выбрать диск, который необходимо переместить
- SelectState — Состояние игры, в котором игрок выбрал диск для перетаскивания на другую башню

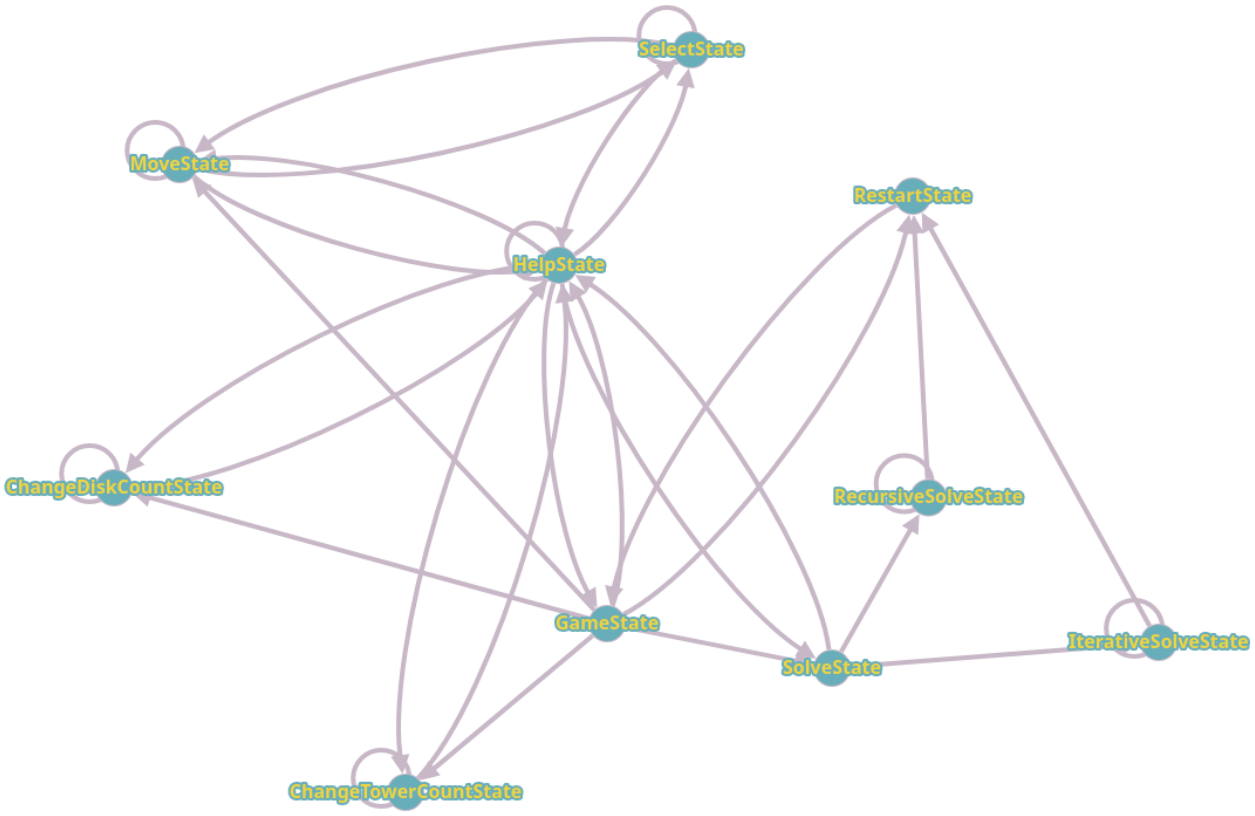

Рисунок 1: Граф состояний

С помощью такой системы состояний очень легко расширять функционал программы. Единственный минус такого способа борьбы со сложностью является написание дополнительных классов. Такой минус не критичен, если система достаточно большая.

Вся игра управляется с помощью главного состояния GameState. Из него можно вызвать меню помощи (переход в состояние GameState → HelpState). Из GameState можно перейти в состояние выбора автоматического решения (GameState → SolveState), из него можно начать рекурсивное или итеративное решение. После окончания автоматического решения происходит переход в состояние RestartState. В любом режиме, кроме RecursiveSolveState и IterativeSolveState можно попасть в режим HelpState, который может вернуть обратно в предыдущее состояние, после прочтения меню помощи. Из GameState можно перейти в режим ручного решения (GameState → MoveState). В этом режиме необходимо выбрать диск, который игрок хочет перетащить. Если диск на башне отсутствует, то мы остаёмся в режиме MoveState, иначе мы выбираем диск и переходим в состоние выбора: куда переложить диск (MoveState → SelectState). Если невозможно переложить диск (например диск большего размера не может быть положен на диск меньшего размера), то мы остаёмся в режиме SelectState, иначе мы перекладываем диск и возвращаемся в режим выбора диска для перекладывания (SelectState → MoveState).

Важно понимать, при переходе из состояния NameState→NameState название состояния остаётся тем же, но внутренние данные меняются, хотя в реализации всё иммутабельно (каждый раз создаётся неизменяемый объект с новыми данными).

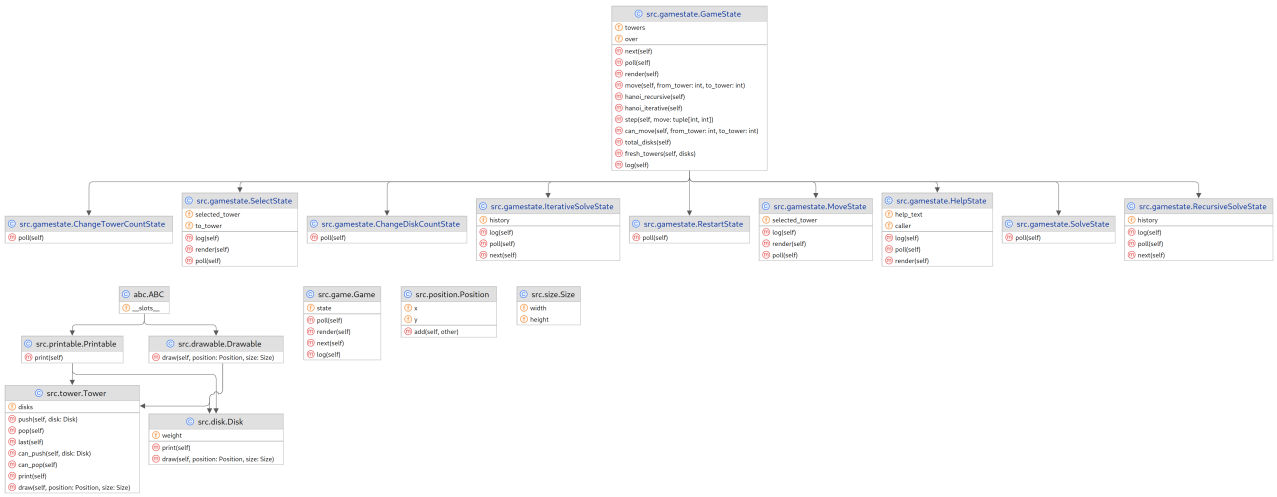

Рисунок 2: Диаграмма классов проекта

Как и любая игра нашему проекту нужна инициализация данных (создание первоначальных башен и дисков в правильном порядке). Инициализация игрового “движка”. И главный цикл игры, в котором будет происходить в чётком порядке:

1. Логгирования (log)
2. Отрисовка (render)
3. Отслеживание за нажатиями внутри игрового окна (poll)
4. Изменение состояния игры (next)

Важно отметить, что состоянии игры изменяется не только в методе next, но и, конечно, вся магия мутации будет происходить внутри poll. Так как игрок будет нажимать на клавиши клавиатуры, игра будет это воспринимать как собственное изменение, например, переключение в другой режим игры.

Почти весь интерфейс состоит из прямоугольников разного цвета. Башни располагаются симметрично и равноудалено относительно оси абсцисс. Дискам задаётся ширина логарифмически. Высота всех дисков одинаковая. В интерфейсе есть элементы текста, которые выводятся на экран, если программа находится в состоянии HelpState.

2.1 Рекурсивный алгоритм на Python

```
119     def hanoi_recursive(self) -> list:
120         def strategy(
121             disks: int, source: int, target: int, temp: int
122         ) -> list[tuple[int, int]]:
123             if disks == 0:
124                 return []
125             return (
126                 strategy(disks - 1, source, temp, target)
127                 + [(source, target)]
128                 + strategy(disks - 1, temp, target, source)
129             )
130
131         steps: list[tuple[int, int]] = strategy(
132             self.total_disks(), 0, len(self.towers) - 1, 1
```

```
        )
        history: list = [self]
        for step in steps:
            history.append(history[-1].step(step))
        return history
```

Листинг 1: Одна из рекурсивных реализаций алгоритма решения задачи о Ханойской башне

2.2 Итеративный алгоритм на Python

```
    def hanoi_iterative(self) -> list:
        def two_way_move(source, target, last_state):
            if last_state.can_move(source, target):
                return last_state.step((source, target))
            else:
                return last_state.step((target, source))

        disks: int = self.total_disks()
        source: int = 0
        target: int = len(self.towers) - 1
        temp: int = 1
        history: list = [self]
        total: int = 2**disks - 1
        if disks % 2 == 0:
            for i in range(total):
                if i % 3 == 0:
                    history.append(two_way_move(source, temp, history[-1]))
                elif i % 3 == 1:
                    history.append(two_way_move(source, target, history[-1]))
                else:
                    history.append(two_way_move(temp, target, history[-1]))
        else:
            for i in range(total):
                if i % 3 == 0:
                    history.append(two_way_move(source, target, history[-1]))
                elif i % 3 == 1:
                    history.append(two_way_move(source, temp, history[-1]))
                else:
                    history.append(two_way_move(temp, target, history[-1]))
        return history
```

Листинг 2: Одна из итеративных реализаций алгоритма решения задачи о Ханойской башне

2.3 Тестирование библиотеки

Программа-игра может автоматически решать задачу о Ханойской башне. Существует режим ручной игры, где игрок перетаскивает диски по правилам игры.

В ходе тестирования были проверены все режимы:

- Режим ручной игры
- Автоматический режим решения: рекурсивный и итеративный

- Режим изменения количества дисков
- Режим изменения количества башен

Во всех режимах не было обнаружено каких-либо ошибок в работе программы.

В ходе выполнения работы были написаны юнит-тесты для библиотеки классов. Было протестировано перемещение дисков между башнями: нельзя допустить, чтобы диск был помещён не по-правилам пазла.

Юнит-тесты дают точно понять, не нарушили мы главные библиотечные вызовы: перемещение дисков между башнями и помещение диска внутрь башни.

Игра разрешает играть с как минимум 1 диском.

Всё тестирование проходило на версии `Python 3.10.8`.

Во время ручного тестирования на ОС Linux ошибок выявлено не было. На машине с ОС MS Windows была выявлена ошибка разрешения экрана. Чтобы решить данную проблему пришлось изменять настройки масштабирования экрана. Масштабирование экрана должно быть 100%.

Если количество дисков равно 0, то задача решена, но в интерфейсе нет такой возможности. При количестве дисков ≤ 1 задача успешно решается с помощью алгоритма.

У алгоритмов не ограничений по количеству дисков. При достаточно большом количестве (> 25) программа может достаточно долго выполняться и занять всю оперативную память компьютера.

Игра ведёт журнал (логгирование) в стандартный выход (терминал), в котором есть следующая информация: текущее состояние игры (NameState), состояние башен и дисков. В режиме HelpState дублируется текст, выведенный на экран, в лог журнала. В режиме MoveState логгируется текущее положение курсора (текущая башня под курсором). В режиме SelectState логгируется положение курсора из MoveState и положение курсора куда мы хотим сделать ход. В режиме решения выводится история ходов, которые надо совершить.

2.4 Пример работы программы

Рисунок 3: Главное меню игры

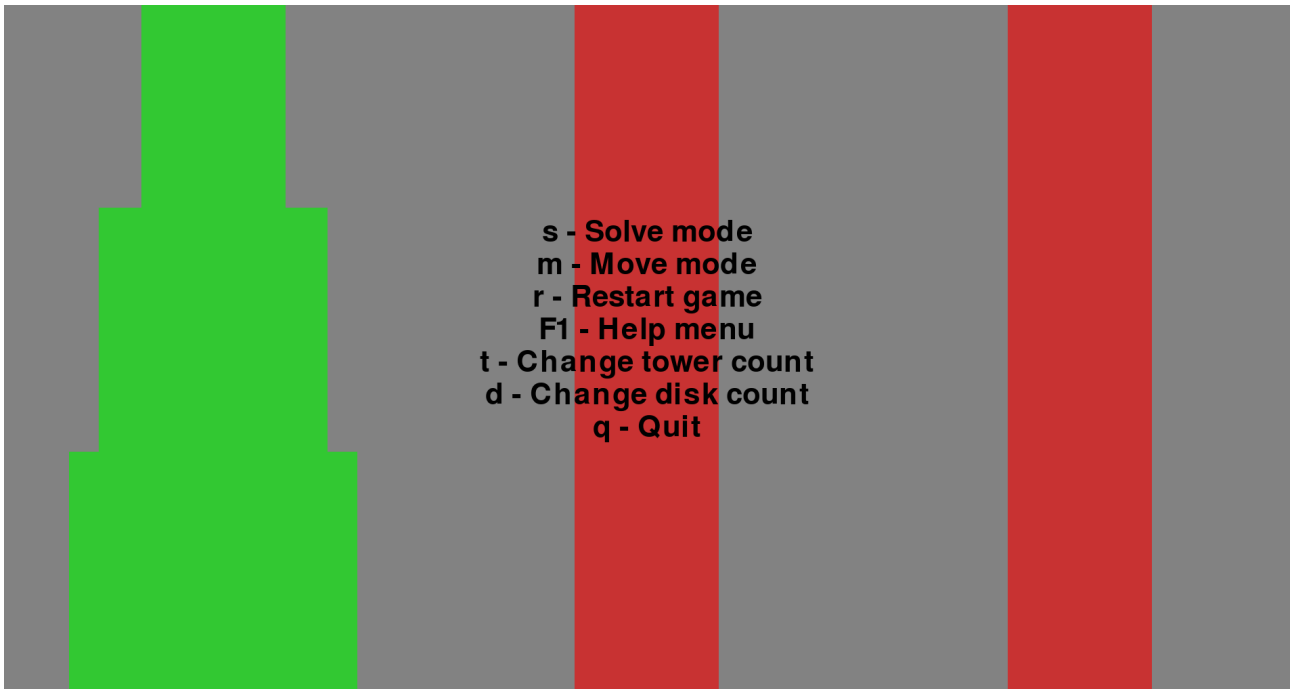

Рисунок 4: Главное меню помощи. управление

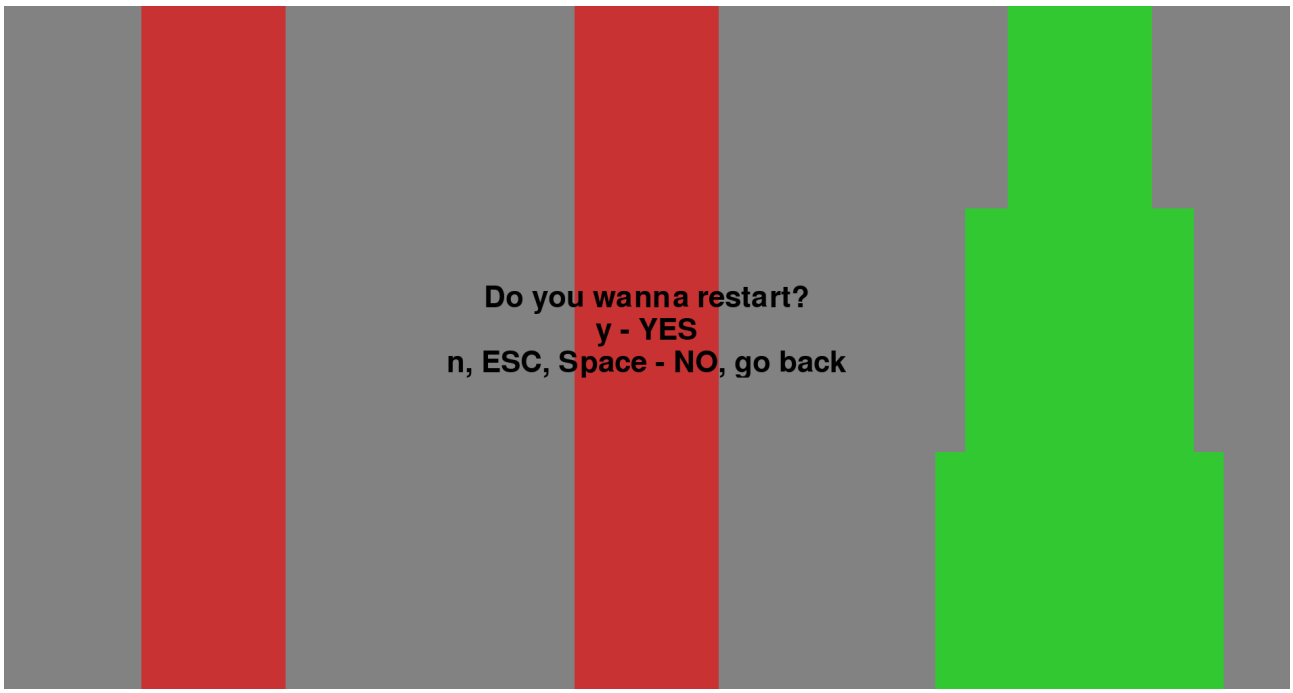

Рисунок 5: В конце авто-решения головоломки можно перезапустить игру

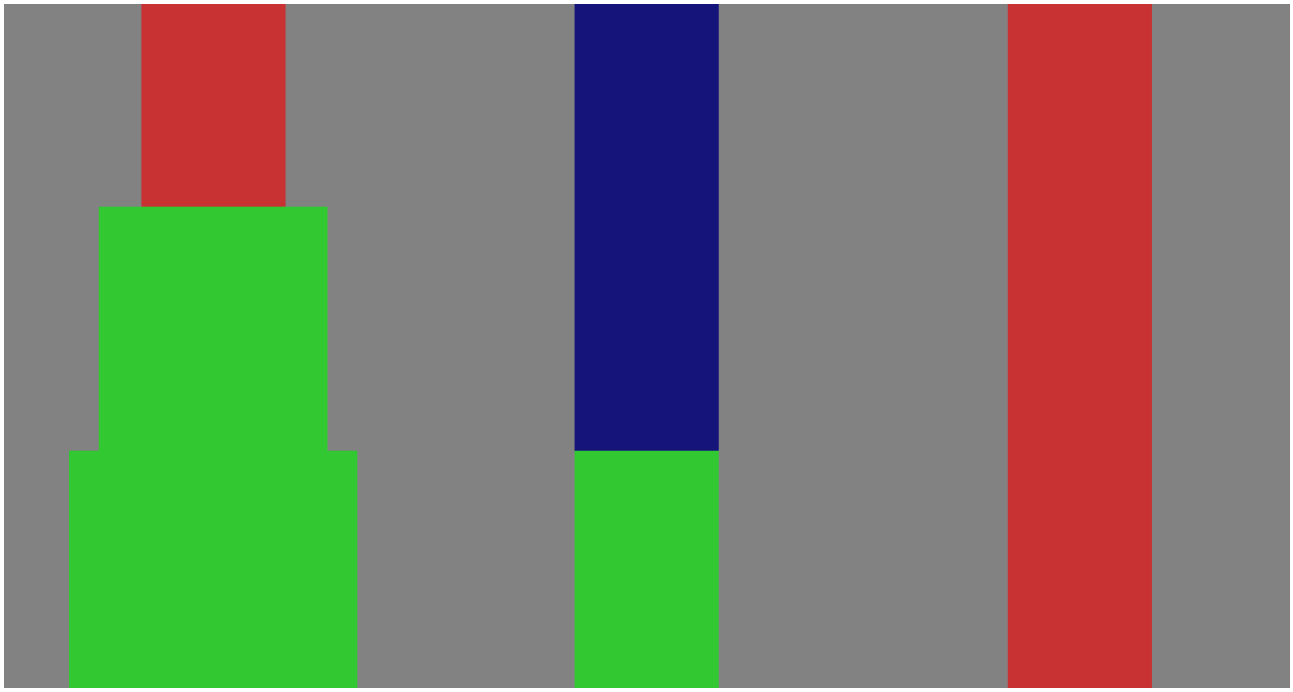

Рисунок 6: Меню передвижения курсора для взятия диска

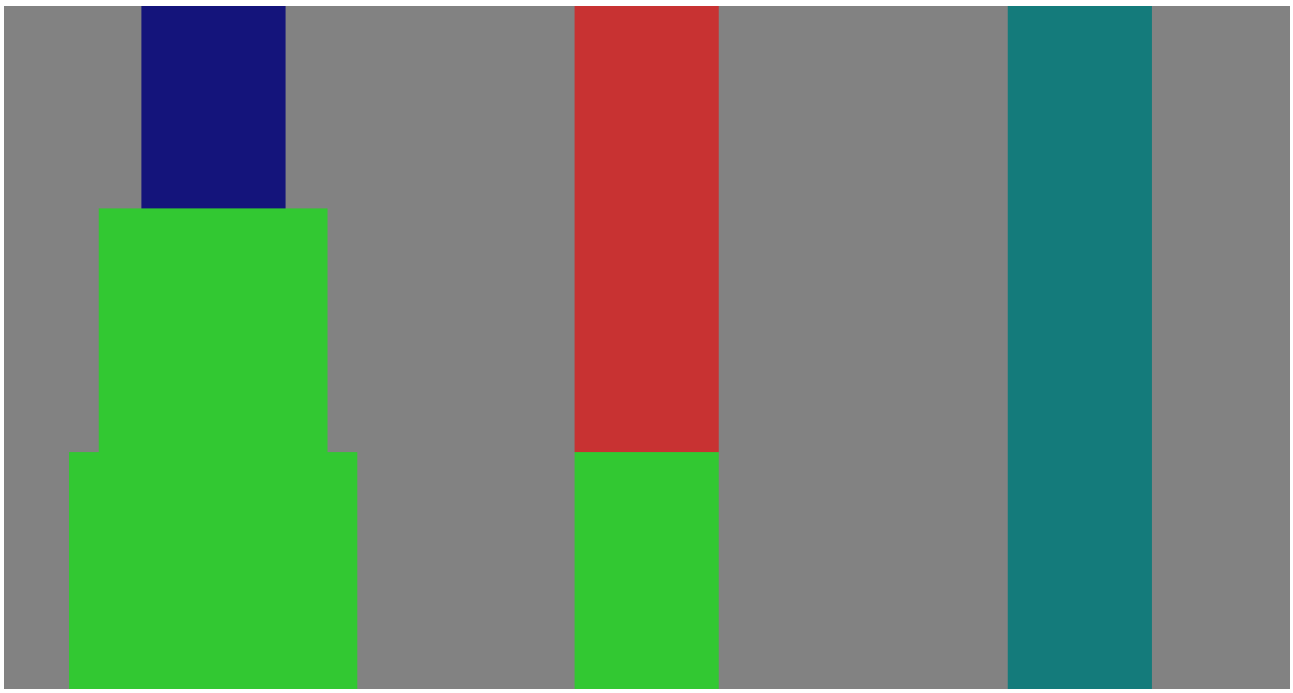

Рисунок 7: Меню выбора нового положения для выбранного диска

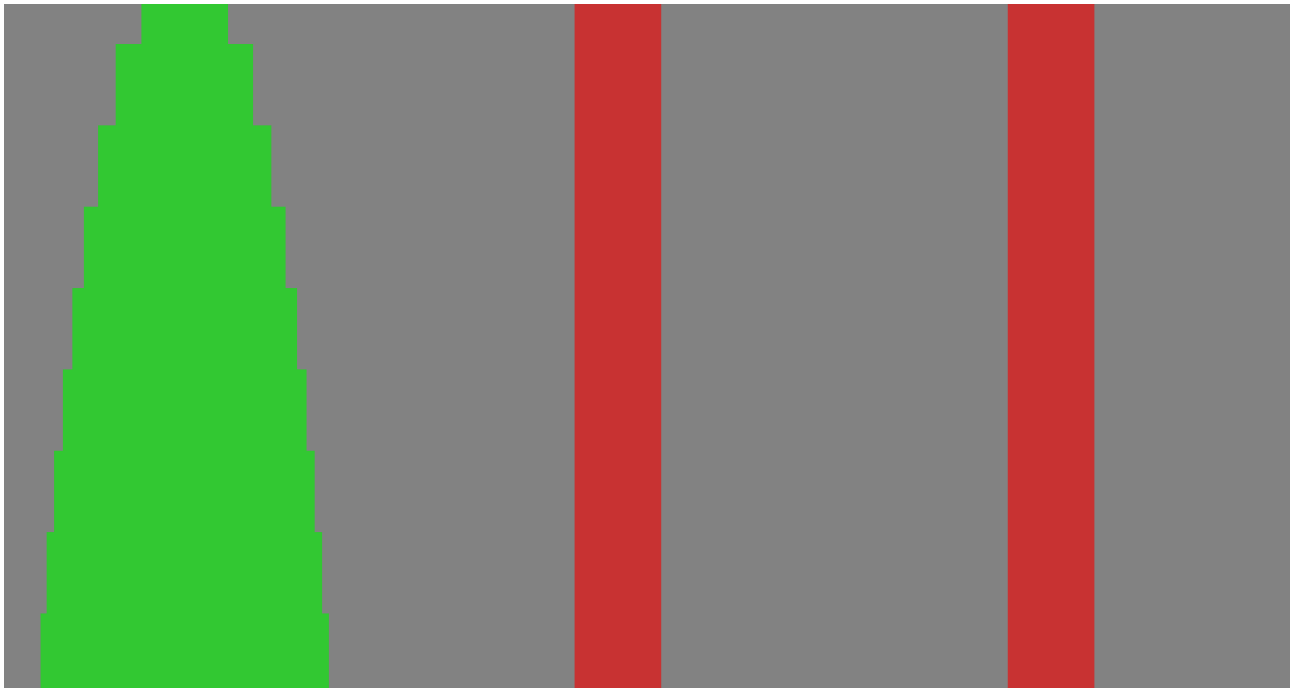

Рисунок 8: Возможно увеличить или уменьшить количество дисков

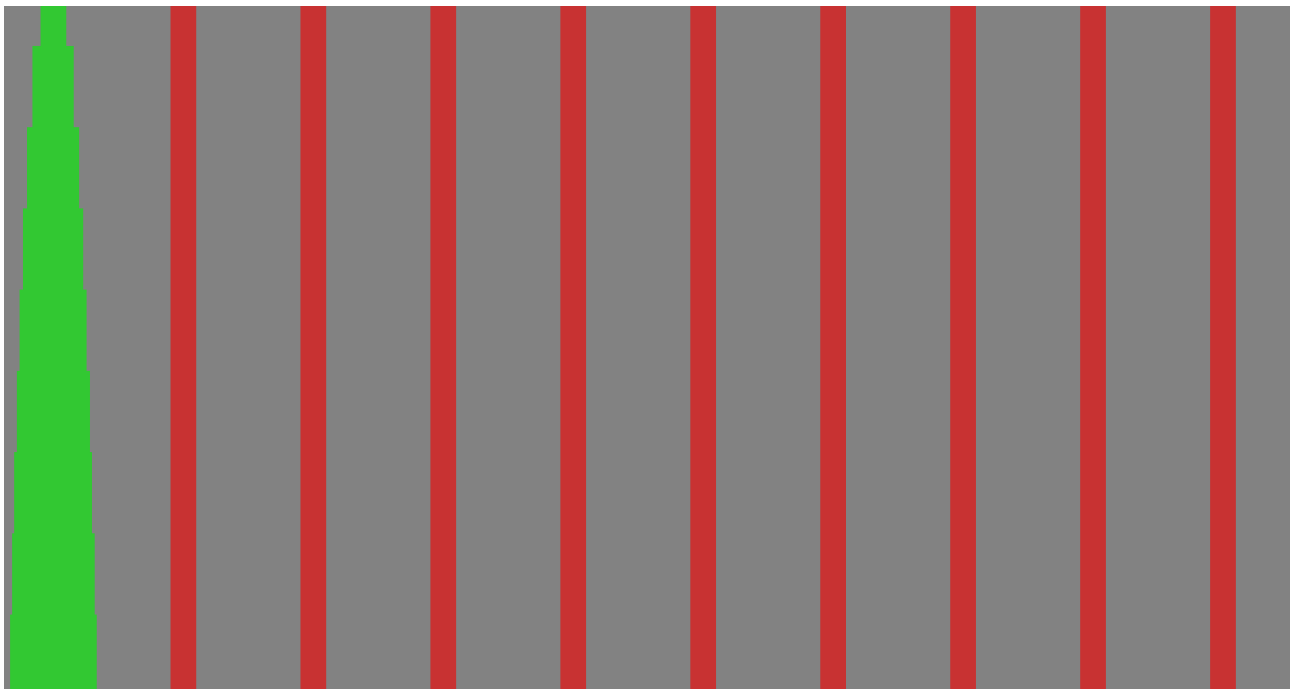

Рисунок 9: Возможно увеличить или уменьшить количество башен

Заключение

Рассмотрели теоретическую информацию о рекурсивном и итеративном алгоритме. В данной реализации итеративного алгоритма видно, что он абсолютно идентичен по ходам с рекурсивным алгоритмом. Итеративный алгоритм сложнее в реализации, для него требуется знание о том, можно ли сделать легальный ход в обе стороны.

Реализовали полноценную игру, рекурсивный и итеративные алгоритмы решения задачи о Ханойской башне. Так же пользователю дана возможность решить задачу самостоятельно. Весь код можно найти по данному адресу: `https://github.com/andreymlv/tstu-computation-theory`

Список литературы

[1] Ciril Petr Andreas M. Hinz, Sandi Klavžar. *The Tower of Hanoi - Myths and Maths.* Birkhäuser, 2018.

[2] Sébastien Eustace and Open Source. Poetry documentation. `https://python-poetry.org/docs/`, 2018-2022.

[3] Python Software Foundation. Python 3.11.1 documentation. `https://docs.python.org/3/`, 2001-2022.

[4] Raphael Holzer. Pygame documentation. `https://pygame.readthedocs.io/en/latest/index.html`, 2019.

[5] Miodrag S. Petkovic. *Famous Puzzles of Great Mathematicians.* American Mathematical Society, 2009.

[6] Edward L Spitznagel. *Selected topics in mathematics.* Holt, Rinehart and Winston, 1971.

[7] Ральф Джонсон Джон Влиссидес Эрих Гамма, Ричард Хелм. *Паттерны объектно-ориентированного проектирования.* Библиотека программиста. Питер, 2020.

Приложение А. Программа на языке Python

```python
#!/usr/bin/env python3
import pygame

from src.disk import Disk
from src.game import Game
from src.gamestate import HelpState, GameState, init
from src.tower import Tower


def main() -> None:
    pygame.init()
    first_tower_disks: list[Disk] = list(map(lambda x: Disk(x), range(3, 0, -1)))
    game: Game = Game(
        state=HelpState(
            towers=[
                Tower(first_tower_disks),
                Tower([]),
                Tower([]),
            ],
            over=False,

            help_text="Welcome to the game!\nIf you are confused with controls always use F1 bu
            caller=GameState(
                [
                    Tower(first_tower_disks),
                    Tower([]),
                    Tower([]),
                ],
                False,
            ),
        )
    )
    init()
    while not game.state.over:
        print(game.log())
        game.render()
        pygame.display.update()
        pygame.time.Clock().tick(60)
        game = game.poll()
        game = game.next()
    pygame.quit()


if __name__ == "__main__":
    main()
```

Листинг 3: main.py — главный скрипт приложения

```python
from typing import NamedTuple

from src.gamestate import GameState, HelpState


class Game(NamedTuple):
    state: GameState | HelpState
```

```
8
9       def poll(self):
10          return Game(self.state.poll())
11
12      def render(self) -> None:
13          return self.state.render()
14
15      def next(self):
16          return Game(self.state.next())
17
18      def log(self) -> str:
19          return (
20              f"{self.state.__class__.__name__} {self.state.towers} " +
                ↪  self.state.log()
21          )
```

Листинг 4: game.py — класс, который хранит текущее состояние

```
1   from collections.abc import Callable
2   from dataclasses import dataclass
3   from random import randint
4   from math import log
5
6   import pygame
7
8   from . import ptext
9
10  from src.disk import Disk
11  from src.position import Position
12  from src.size import Size
13  from src.tower import Tower
14
15
16  def init() -> None:
17      pygame.display.set_caption("Hanoi Tower")
18
19
20  def flat_map(f: Callable, xs) -> list:
21      ys = []
22      for x in xs:
23          ys.extend(f(x))
24      return ys
25
26
27  def random_color() -> pygame.Color:
28      return pygame.Color((randint(0, 255), randint(0, 255), randint(0, 255)))
29
30
31  @dataclass()
32  class GameState:
33      towers: list[Tower]
34      over: bool
35
36      def next(self):
37          return self
38
39      def poll(self):
40          for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_m:
                        return MoveState(self.towers, False, 0)
                    case pygame.K_s:
                        return SolveState(self.towers, False)
                    case pygame.K_r:
                        return RestartState(self.towers, False)
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,
                            "s - Solve mode\nm - Move mode\nr - Restart game\nF1 - Help menu\nt
                            self,
                        )
                    case pygame.K_t:
                        return ChangeTowerCountState(self.towers, False)
                    case pygame.K_d:
                        return ChangeDiskCountState(self.towers, False)
                    case pygame.K_q:
                        return GameState(self.towers, True)
        return self

    def render(self) -> None:
        backgroud_color = pygame.Color((130, 130, 130))
        screen_size = Size(1920, 1080)
        window = pygame.display.set_mode(screen_size)
        window.fill(backgroud_color)
        tower_color = pygame.Color((200, 50, 50))
        disk_color = pygame.Color((50, 200, 50))
        max_disks = self.total_disks()
        first_tower_x = screen_size.width / len(self.towers) / 3
        max_width_disk = 2 * first_tower_x
        towers_rects: list[pygame.Rect] = []
        disks_rects: list[pygame.Rect] = []
        for i, tower in enumerate(self.towers):
            tower_size = Size(
                max_width_disk / log(max_disks + 1, 2), screen_size.height
            )
            tower_position = Position(
                first_tower_x + screen_size.width / len(self.towers) * i,
                0,
            )
            center_of_tower = Position(
                tower_position.x + tower_size.width / 2, tower_position.y
            )
            towers_rects.append(
                tower.draw(
                    tower_position,
                    tower_size,
                )
            )
            for j, disk in enumerate(tower.disks):
                disk_size = Size(
                    max_width_disk / log(max_disks + 1, disk.weight + 1),
                    screen_size.height / max_disks,
                )
```

```
                disk_position = Position(
                    center_of_tower.x - disk_size.width / 2,
                    screen_size.height - disk_size.height * (j + 1),
                )
                disks_rects.append(disk.draw(disk_position, disk_size))
        for tower in towers_rects:
            pygame.draw.rect(window, tower_color, tower)
        for disk in disks_rects:
            pygame.draw.rect(window, disk_color, disk)

    def move(self, from_tower: int, to_tower: int):
        if self.can_move(from_tower, to_tower):
            towers: list[Tower] = self.towers.copy()
            temp_disk: Disk = towers[from_tower].last()
            towers[from_tower] = towers[from_tower].pop()
            towers[to_tower] = towers[to_tower].push(temp_disk)
            return GameState(towers, self.over)
        return self

    def hanoi_recursive(self) -> list:
        def strategy(
            disks: int, source: int, target: int, temp: int
        ) -> list[tuple[int, int]]:
            if disks == 0:
                return []
            return (
                strategy(disks - 1, source, temp, target)
                + [(source, target)]
                + strategy(disks - 1, temp, target, source)
            )

        steps: list[tuple[int, int]] = strategy(
            self.total_disks(), 0, len(self.towers) - 1, 1
        )
        history: list = [self]
        for step in steps:
            history.append(history[-1].step(step))
        return history

    def hanoi_iterative(self) -> list:
        def two_way_move(source, target, last_state):
            if last_state.can_move(source, target):
                return last_state.step((source, target))
            else:
                return last_state.step((target, source))

        disks: int = self.total_disks()
        source: int = 0
        target: int = len(self.towers) - 1
        temp: int = 1
        history: list = [self]
        total: int = 2**disks - 1
        if disks % 2 == 0:
            for i in range(total):
                if i % 3 == 0:
                    history.append(two_way_move(source, temp, history[-1]))
                elif i % 3 == 1:
                    history.append(two_way_move(source, target, history[-1]))
                else:
                    history.append(two_way_move(temp, target, history[-1]))
```

```
        else:
            for i in range(total):
                if i % 3 == 0:
                    history.append(two_way_move(source, target, history[-1]))
                elif i % 3 == 1:
                    history.append(two_way_move(source, temp, history[-1]))
                else:
                    history.append(two_way_move(temp, target, history[-1]))
        return history

    def step(self, move: tuple[int, int]):
        return self.move(move[0], move[1])

    def can_move(self, from_tower: int, to_tower: int) -> bool:
        return (
            len(self.towers) > 1
            and self.towers[from_tower].can_pop()
            and self.towers[to_tower].can_push(self.towers[from_tower].last())
        )

    def total_disks(self) -> int:
        return len(flat_map(lambda id: id, map(lambda tower: tower.disks,
        ↪  self.towers)))

    def fresh_towers(self, disks) -> list[Tower]:
        towers: list[Tower] = list(map(lambda _: Tower([]), range(len(self.towers))))
        towers[0].disks = list(map(lambda x: Disk(x), range(disks, 0, -1)))
        return towers

    def log(self) -> str:
        return ""


@dataclass()
class ChangeDiskCountState(GameState):
    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_ESCAPE:
                        return GameState(self.towers, False)
                    case pygame.K_p:
                        return ChangeDiskCountState(
                            self.fresh_towers(self.total_disks() + 1), False
                        )
                    case pygame.K_m:
                        if self.total_disks() == 1:
                            return self
                        return ChangeDiskCountState(
                            self.fresh_towers(self.total_disks() - 1), False
                        )
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,

                            ↪  "Any change restarts the game!\np - Increase number of disks\nm - D
                            self,
```

```
                        )
        return self


@dataclass()
class ChangeTowerCountState(GameState):
    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_ESCAPE:
                        return GameState(self.towers, False)
                    case pygame.K_p:
                        return ChangeTowerCountState(
                            self.fresh_towers(self.total_disks()) + [Tower([])],
                            ↪ False
                        )
                    case pygame.K_m:
                        if len(self.towers) == 1:
                            return self
                        return ChangeTowerCountState(
                            self.fresh_towers(self.total_disks())[:-1], False
                        )
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,

                            ↪ "Any change restarts the game!\np - Increase number of towers\nm -
                            self,
                        )
        return self


@dataclass()
class RestartState(GameState):
    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_ESCAPE | pygame.K_SPACE | pygame.K_n:
                        return GameState(self.towers, False)
                    case pygame.K_y:
                        return GameState(self.fresh_towers(self.total_disks()),
                        ↪ False)
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,

                            ↪ "Do you wanna restart?\ny - YES\nn, ESC, Space - NO, go back",
                            self,
                        )
        return self


```

```
@dataclass()
class HelpState(GameState):
    help_text: str
    caller: GameState

    def log(self) -> str:
        return f"{self.help_text} {self.caller.__class__.__name__}"

    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_ESCAPE | pygame.K_SPACE:
                        return self.caller
        return self

    def render(self):
        screen_size = Size(1920, 1080)
        super().render()
        ptext.draw(
            self.help_text,
            centerx=screen_size.width / 2,
            centery=screen_size.height / 2,
            fontsize=64,
            color=pygame.color.Color(0, 0, 0),
        )


@dataclass()
class SelectState(GameState):
    selected_tower: int
    to_tower: int

    def log(self) -> str:
        return f"{self.selected_tower} to {self.to_tower}"

    def render(self) -> None:
        backgroud_color = pygame.Color((130, 130, 130))
        screen_size = Size(1920, 1080)
        window = pygame.display.set_mode(screen_size)
        window.fill(backgroud_color)
        tower_color = pygame.Color((200, 50, 50))
        disk_color = pygame.Color((50, 200, 50))
        max_disks = self.total_disks()
        first_tower_x = screen_size.width / len(self.towers) / 3
        max_width_disk = 2 * first_tower_x
        towers_rects: list[pygame.Rect] = []
        disks_rects: list[pygame.Rect] = []
        for i, tower in enumerate(self.towers):
            tower_size = Size(
                max_width_disk / log(max_disks + 1, 2), screen_size.height
            )
            tower_position = Position(
                first_tower_x + screen_size.width / len(self.towers) * i,
                0,
            )
            center_of_tower = Position(
                tower_position.x + tower_size.width / 2, tower_position.y
```

```
            )
            towers_rects.append(
                tower.draw(
                    tower_position,
                    tower_size,
                )
            )
            for j, disk in enumerate(tower.disks):
                disk_size = Size(
                    max_width_disk / log(max_disks + 1, disk.weight + 1),
                    screen_size.height / max_disks,
                )
                disk_position = Position(
                    center_of_tower.x - disk_size.width / 2,
                    screen_size.height - disk_size.height * (j + 1),
                )
                disks_rects.append(disk.draw(disk_position, disk_size))
        for i, tower in enumerate(towers_rects):
            if i == self.selected_tower:
                pygame.draw.rect(window, pygame.Color(20, 20, 123), tower)
            elif i == self.to_tower:
                pygame.draw.rect(window, pygame.Color(20, 123, 123), tower)
            else:
                pygame.draw.rect(window, tower_color, tower)
        for disk in disks_rects:
            pygame.draw.rect(window, disk_color, disk)

    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_SPACE:
                        return MoveState(
                            self.move(self.selected_tower, self.to_tower).towers,
                            False,
                            self.to_tower,
                        )
                    case pygame.K_a | pygame.K_LEFT | pygame.K_h:
                        return SelectState(
                            self.towers,
                            False,
                            self.selected_tower,
                            (self.to_tower - 1) % len(self.towers),
                        )
                    case pygame.K_d | pygame.K_RIGHT | pygame.K_l:
                        return SelectState(
                            self.towers,
                            False,
                            self.selected_tower,
                            (self.to_tower + 1) % len(self.towers),
                        )
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,
                            "a - Move left\nd - Move right\nSPACE - select disk",
                            self,
                        )
```

```
394         return self
395
396
397 @dataclass()
398 class SolveState(GameState):
399     def poll(self):
400         for event in pygame.event.get():
401             if event.type == pygame.QUIT:
402                 return GameState(self.towers, True)
403             if event.type == pygame.KEYDOWN:
404                 match event.key:
405                     case pygame.K_ESCAPE:
406                         return GameState(self.towers, False)
407                     case pygame.K_r:
408                         return RecursiveSolveState(
409                             self.fresh_towers(self.total_disks()),
410                             False,
411                             GameState(
412                                 self.fresh_towers(self.total_disks()), False
413                             ).hanoi_recursive(),
414                         )
415                     case pygame.K_i:
416                         return IterativeSolveState(
417                             self.fresh_towers(self.total_disks()),
418                             False,
419                             GameState(
420                                 self.fresh_towers(self.total_disks()), False
421                             ).hanoi_iterative(),
422                         )
423                     case pygame.K_F1:
424                         return HelpState(
425                             self.towers,
426                             False,
427
                                ↪  "r - Recursive solve\ni - Iterative solve\nESC - Go back\nF1 - Help
428                             self,
429                         )
430         return self
431
432
433 @dataclass()
434 class RecursiveSolveState(GameState):
435     history: list[GameState]
436
437     def log(self) -> str:
438         return f"{self.history}"
439
440     def poll(self):
441         for event in pygame.event.get():
442             if event.type == pygame.QUIT:
443                 return GameState(self.towers, True)
444             if event.type == pygame.KEYDOWN:
445                 match event.key:
446                     case pygame.K_ESCAPE | pygame.K_SPACE:
447                         return GameState(self.towers, False)
448         return self
449
450     def next(self):
451         pygame.time.wait(1000)
452         if len(self.history) == 0:
```

```
453             return RestartState(self.towers, False)
454         return RecursiveSolveState(self.history[0].towers, False, self.history[1:])
455
456
457 @dataclass()
458 class IterativeSolveState(GameState):
459     history: list[GameState]
460
461     def log(self) -> str:
462         return f"{self.history}"
463
464     def poll(self):
465         for event in pygame.event.get():
466             if event.type == pygame.QUIT:
467                 return GameState(self.towers, True)
468             if event.type == pygame.KEYDOWN:
469                 match event.key:
470                     case pygame.K_ESCAPE | pygame.K_SPACE:
471                         return GameState(self.towers, False)
472         return self
473
474     def next(self):
475         pygame.time.wait(1000)
476         if len(self.history) == 0:
477             return RestartState(self.towers, False)
478         return IterativeSolveState(self.history[0].towers, False, self.history[1:])
479
480
481 @dataclass()
482 class MoveState(GameState):
483     selected_tower: int
484
485     def log(self) -> str:
486         return f"{self.selected_tower}"
487
488     def render(self) -> None:
489         backgroud_color = pygame.Color((130, 130, 130))
490         screen_size = Size(1920, 1080)
491         window = pygame.display.set_mode(screen_size)
492         window.fill(backgroud_color)
493         tower_color = pygame.Color((200, 50, 50))
494         disk_color = pygame.Color((50, 200, 50))
495         max_disks = self.total_disks()
496         first_tower_x = screen_size.width / len(self.towers) / 3
497         max_width_disk = 2 * first_tower_x
498         towers_rects: list[pygame.Rect] = []
499         disks_rects: list[pygame.Rect] = []
500         for i, tower in enumerate(self.towers):
501             tower_size = Size(
502                 max_width_disk / log(max_disks + 1, 2), screen_size.height
503             )
504             tower_position = Position(
505                 first_tower_x + screen_size.width / len(self.towers) * i,
506                 0,
507             )
508             center_of_tower = Position(
509                 tower_position.x + tower_size.width / 2, tower_position.y
510             )
511             towers_rects.append(
512                 tower.draw(
```

```
                    tower_position,
                    tower_size,
                )
            )
            for j, disk in enumerate(tower.disks):
                disk_size = Size(
                    max_width_disk / log(max_disks + 1, disk.weight + 1),
                    screen_size.height / max_disks,
                )
                disk_position = Position(
                    center_of_tower.x - disk_size.width / 2,
                    screen_size.height - disk_size.height * (j + 1),
                )
                disks_rects.append(disk.draw(disk_position, disk_size))
        for i, tower in enumerate(towers_rects):
            if i == self.selected_tower:
                pygame.draw.rect(window, pygame.Color(20, 20, 123), tower)
            else:
                pygame.draw.rect(window, tower_color, tower)
        for disk in disks_rects:
            pygame.draw.rect(window, disk_color, disk)

    def poll(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return GameState(self.towers, True)
            if event.type == pygame.KEYDOWN:
                match event.key:
                    case pygame.K_ESCAPE:
                        return GameState(self.towers, False)
                    case pygame.K_SPACE:
                        if len(self.towers[self.selected_tower].disks) == 0:
                            return self
                        return SelectState(
                            self.towers, False, self.selected_tower,
                            ↪  self.selected_tower
                        )
                    case pygame.K_a | pygame.K_LEFT | pygame.K_h:
                        return MoveState(
                            self.towers,
                            False,
                            (self.selected_tower - 1) % len(self.towers),
                        )
                    case pygame.K_d | pygame.K_RIGHT | pygame.K_l:
                        return MoveState(
                            self.towers,
                            False,
                            (self.selected_tower + 1) % len(self.towers),
                        )
                    case pygame.K_F1:
                        return HelpState(
                            self.towers,
                            False,
                            "space - Select\nleft - Move left\nright - Move right",
                            self,
                        )
        return self
```

Листинг 5: gamestate.py — множество классов, определяющее состояния

```python
from dataclasses import dataclass

import pygame

from src.drawable import Drawable
from src.position import Position
from src.printable import Printable
from src.size import Size


@dataclass()
class Disk(Drawable, Printable):
    # Should be unsigned and greater then zero
    weight: int

    def print(self) -> str:
        return self.weight * "#"

    def draw(self, position: Position, size: Size) -> pygame.Rect:
        return pygame.Rect(position, size)
```

Листинг 6: disk.py — класс, описывающий диск

```python
from dataclasses import dataclass

import pygame

from src.disk import Disk
from src.drawable import Drawable
from src.position import Position
from src.printable import Printable
from src.size import Size


@dataclass()
class Tower(Drawable, Printable):
    disks: list[Disk]

    def push(self, disk: Disk):
        if self.can_push(disk):
            return Tower(self.disks + [disk])
        return self

    def pop(self):
        if self.can_pop():
            return Tower(self.disks[:-1])
        return self

    def last(self) -> Disk:
        return self.disks[-1]

    def can_push(self, disk: Disk) -> bool:
        return len(self.disks) == 0 or self.last().weight > disk.weight

    def can_pop(self) -> bool:
        return len(self.disks) > 0

```

```
    def print(self) -> str:
        return super().print()

    def draw(self, position: Position, size: Size) -> pygame.Rect:
        return pygame.Rect(position, size)
```

Листинг 7: tower.py — класс, описывающий башню

```
from abc import ABC, abstractmethod


class Printable(ABC):
    @abstractmethod
    def print(self) -> str:
        """
        Create printable string for CLI.
        """
        pass
```

```
from abc import ABC, abstractmethod

import pygame

from src.position import Position
from src.size import Size


class Drawable(ABC):
    @abstractmethod
    def draw(self, position: Position, size: Size) -> pygame.Rect:
        """
        Create drawable element for pygame scene.
        """
        pass
```

```
from typing import NamedTuple


class Size(NamedTuple):
    width: float
    height: float
```

```
from typing import NamedTuple

from src.size import Size


class Position(NamedTuple):
    x: float
    y: float

    def add(self, other):
        return Size(self.x + other.x, self.y + other.y)
```

Листинг 8: Класс/интерфейсы, необходимые для отрисовки

Приложение В. Программа-тесты на языке Python

```python
from unittest import TestCase

from src.disk import Disk
from src.game import Game
from src.tower import Tower


class TestGame(TestCase):
    def test_move(self):
        game_expected = Game([Tower([]), Tower([]), Tower([])])
        game_expected.towers[0] = (
            game_expected.towers[0].push(Disk(3)).push(Disk(2)).push(Disk(1))
        )
        game_expected = game_expected.move(0, 1)
        game_actual = Game([Tower([Disk(3), Disk(2)]), Tower([Disk(1)]), Tower([])])
        self.assertEqual(game_expected, game_actual, "Game states aren't equal.")
        game_expected = game_expected.move(0, 1)
        self.assertEqual(game_expected, game_actual, "Game states aren't equal.")
        game_expected = game_expected.move(0, 2)
        game_actual = Game([Tower([Disk(3)]), Tower([Disk(1)]), Tower([Disk(2)])])
        self.assertEqual(game_expected, game_actual, "Game states aren't equal.")
        game_expected = game_expected.move(1, 2)
        game_actual = Game([Tower([Disk(3)]), Tower([]), Tower([Disk(2), Disk(1)])])
        self.assertEqual(game_expected, game_actual, "Game states aren't equal.")
        game_expected = game_expected.move(1, 2)
        game_actual = Game([Tower([Disk(3)]), Tower([]), Tower([Disk(2), Disk(1)])])
        self.assertEqual(game_expected, game_actual, "Game states aren't equal.")
```

Листинг 9: test_game.py — тестирование перемещения дисков

```python
from unittest import TestCase

from src.disk import Disk
from src.tower import Tower


class TestTower(TestCase):
    def test_push(self):
        tower_actual: Tower = Tower([])
        tower_actual: Tower = tower_actual.push(Disk(1))
        tower_expected: Tower = Tower([Disk(1)])
        self.assertEqual(tower_actual, tower_expected, "Towers aren't equal.")
        tower_actual: Tower = tower_actual.push(Disk(2))
        tower_expected: Tower = Tower([Disk(1)])
        self.assertEqual(tower_actual, tower_expected, "Towers aren't equal.")
        tower_actual: Tower = Tower([Disk(2), Disk(1)])
        tower_expected: Tower = Tower([Disk(2), Disk(1)])
        self.assertEqual(tower_actual, tower_expected, "Towers aren't equal.")

    def test_pop(self):
```

```
21          tower_actual: Tower = Tower([Disk(1)])
22          tower_actual: Tower = tower_actual.pop()
23          tower_expected: Tower = Tower([])
24          self.assertEqual(tower_actual, tower_expected, "Towers aren't equal.")
25          tower_actual: Tower = Tower([]).pop()
26          tower_expected: Tower = Tower([])
27          self.assertEqual(tower_actual, tower_expected, "Towers aren't equal.")
28
29      def test_last(self):
30          disk_actual = Tower([Disk(1)]).last()
31          disk_expected = Disk(1)
32          self.assertEqual(disk_actual, disk_expected, "Disks aren't equal.")
```

Листинг 10: test_tower.py — тестирование помещения и удаления дисков с башни