САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

Отчет по практическому заданию №2
по дисциплине: «Функциональное программирование»

Студент:
группы 5130201/20102
_____ Салимли А.

Преподаватель:
к.т.н.
_____ Моторин Д.Е.

«____»_____2024 г.

Санкт-Петербург, 2024 г.

Содержание

Введение

В данном отчете, описаны результаты выполнения комплекса практических заданий, реализации фрактала кривой Гильберта, а так же реализация игры Ним.

В ходе выполнения практической работы, реализованы на языке Haskell:
Программа реализующая вывод списка пар координат для заданного фрактала.
Программа реализующая игру Ним, в которой из одной кучки M камней игрок может взять не более K камней (Взявший последний камень победитель).

Программы были реализованы в интегрируемой среде разработки - Visual Studio Code 1.94, на языке Haskell 9.4.8. Расширение программ - .hs.

1 Постановка задачи

В ходе прохождения практического задания №2, необходимо реализовать две программы на языке Haskell.

1. Вычислить все пары координат (x,y) для заданного фрактала на глубину ’n’ шагов и вывести в виде списка списков пар где каждый уровень рекурсии является списком пар. Вариант фрактала: **Кривая Гильберта.**
2. Реализовать заданную игру и стратегию следующим образом: В коде задать список, содержащий ходы пользователя; Реализовать функцию, определяющую работу стратегий и функцию, организующую игру (игра должна продолжатся не более 100 ходов). Вариант игры: **Ним** (одна кучка из M камней, игрок может взять не более K камней).

Для каждой части задачи необходимо сформировать (.hs) файл, содержащий весь необходимый код на языке Haskell.

2 Практическое задание

2.1 Фрактал: Кривая Гильберта

Фрактал - это геометрическая фигура, обладающая свойством самоподобия, что означает, что её структура повторяется на разных масштабах. Важной особенностью фракталов является то, что они обладают сложной формой при сравнительно простой генерации с помощью рекурсивных процедур. Математически, фракталы могут быть описаны с помощью рекурсивных алгоритмов, которые продолжают строить фигуру, добавляя всё новые и новые детали с каждым шагом. Фракталы обладают тремя свойствами - самоподобие, размерность, рекурсивная структура (алгоритм).

Кривая Гильберта - это один из известных примеров фракталов, который представляет собой непрерывную замкнутую линию, заполняющую пространство на плоскости. Каждое следующее приближение кривой строится по простому правилу: на основе предыдущей версии кривая копируется и поворачивается, формируя четыре квадранта, которые вместе образуют более сложную фигуру. Кривая Гильберта является примером двухмерного фрактала, используемого в различных приложениях, включая компьютерную графику, оптимизацию обработки изображений и алгоритмы обхода данных.

2.1.1 Математическое описание

Кривая Гильберта строится рекурсивно через деление квадрата на четыре подквадрата. Каждый подквадрат заполняется модифицированной версией предыдущего уровня кривой, состоящей из четырех частей. Математически это можно выразить следующим образом:

- **Базовый случай** (нулевая итерация): H_0 это единственная точка - $(0,0)$.
- **Рекурсивное построение**: Для уровня n (где $n \geq 1$) кривая H_n строится из кривой предыдущего уровня H_{n-1} следующим образом:
  1. Разделение текущего квадрата размером (2^n x 2^n) на четыре подквадрата размером: (2^{n-1} x 2^{n-1}).
  2. Кривая предыдущего уровня копируется и модифицируется для заполнения каждого из четырех подквадратов:
     - Верхний левый квадрант: кривая H_{n-1} поворачивается на 90° против часовой стрелки.
     - Верхний правый квадрант: кривая H_{n-1} перемещается вправо на 2^{n-1} единиц.
     - Нижний правый квадрант: кривая H_{n-1} перемещается вправо и вниз на 2^{n-1} единиц.
     - Нижний левый квадрант: кривая H_{n-1} поворачивается на 90° по часовой стрелке и затем перемещается вниз на 2^{n-1} единиц.

На рисунке 1, изображен начальный результат, при n = 2, когда формируется фрактал. Дальнейшее увеличение уровня рекурсии будет самоподобием этой.

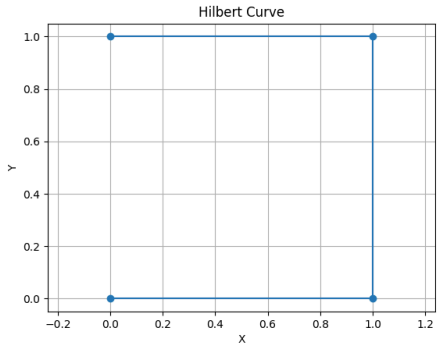

Рис.1. Кривая Гильберта при уровне 1

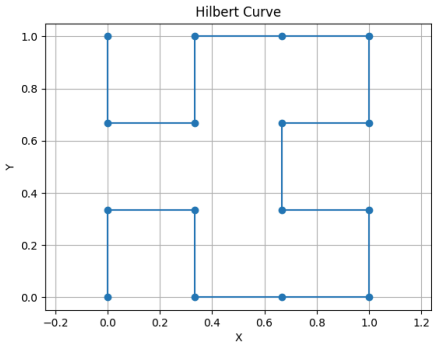

Рис.2. Кривая Гильберта при уровне 2

2.1.2 Реализация на языке Haskell

Ниже приведен код, на языке Haskell, выводящий списки списков пар координат в зависимости от числа printSingleLevelHilbertCurve, которая в реализации означает глубину рекурсии.

```haskell
type Point = (Double, Double)
type LevelPoints = [Point]

hilbertC :: Int -> [[Point]]
hilbertC 0 = [[(0, 0)]]
hilbertC n =
    let previousLevels = hilbertC (n - 1)
        levelNow = naNext (fromIntegral (2 ^ (n - 1))) (last previousLevels)
    in previousLevels ++ [levelNow]

scalePoints :: Double -> [Point] -> [Point]
scalePoints maxCoord points =
    [(x / maxCoord, y / maxCoord) | (x, y) <- points]

naNext :: Double -> [Point] -> [Point]
naNext size points =
    let
        tLeft  = [(y, x) | (x, y) <- points]
        tRight = [(x + size, y) | (x, y) <- points]
        bRight = [(x + size, y + size) | (x, y) <- points]
        bLeft  = [(size - 1 - y, size - 1 - x + size) | (x, y) <- points]
    in tLeft ++ tRight ++ bRight ++ bLeft

printSLHilbert :: Int -> IO ()
printSLHilbert n =
    let hilbertPoints = last (hilbertC n)
        maxCoord = fromIntegral (2 ^ n - 1)
        scaledPoints = scalePoints maxCoord hilbertPoints
    in print scaledPoints

main :: IO ()
main = printSLHilbert 5
```

Подробное описание кода:

Сначала мы определяем тип для точки. Так как мы не используем в коде числа с плавающей запятой, тип точки Point будет равен Int, Int. Затем нам нужно описать тип списка таких точек LevelPoints, его тип: [Point]. После мы определяем тип функции для глубины рекурсии generateHilbertCurve :: Int -> [[Point]], как обсуждали ранее для 0 базовый случий это (0,0). Далее generateHilbertCurve при принятии ’n’ образует рекурсивную функцию, где:

- let previousLevels = generateHilbertCurve (n - 1) — рекурсивно генерирует предыдущие уровни кривой до уровня n-1.
- currentLevel = createNextLevel (2 ^ (n - 1)) (last previousLevels) — создается текущий уровень, который строится на основе последнего уровня предыдущей рекурсии с помощью функции createNextLevel.
- in previousLevels ++ [currentLevel] — результатом работы функции является объединение всех предыдущих уровней с текущим.

- createNextLevel :: Int -> [Point] -> [Point] — это функция, которая на основе списка точек и размера квадрата создает следующий уровень кривой Гильберта.
- topLeft = [(y, x) | (x, y) <- points] — поворот предыдущего уровня на 90 градусов для создания левого верхнего квадранта нового уровня.
- topRight = [(x + size, y) | (x, y) <- points] — сдвиг точек на размер size вправо для создания правого верхнего квадранта.
- bottomRight = [(x + size, y + size) | (x, y) <- points] — сдвиг точек и вправо, и вниз для создания правого нижнего квадранта.
- bottomLeft = [(size - 1 - y, size - 1 - x + size) | (x, y) <- points] — отражение точек для создания левого нижнего квадранта.
- in topLeft ++ topRight ++ bottomRight ++ bottomLeft — объединение всех четырёх квадрантов для получения нового уровня кривой.

После чего определяем тип функции: printSingleLevelHilbertCurve с типом Int -> IO( ). Который будет выводить все в консоль. При принятии ’n’ функция выводит последний уровень, который был посчитан для глубины ’n’.

В ходе эксперимента было выявлено, что для оптимального изображения графика фрактала, достаточно указать фиксированную ось от 0 до 1.

Ниже представлены изображения для глубин рекурсии 3, 4, 6 соответственно.

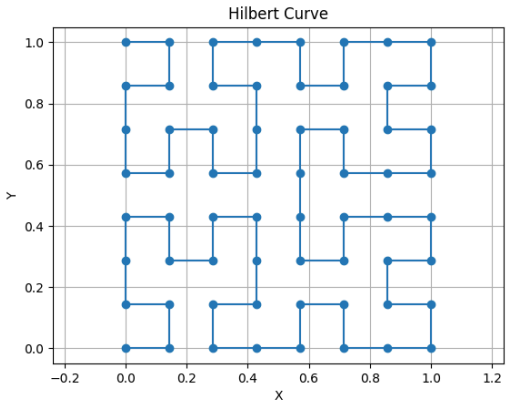

Рис.3. Кривая Гильберта при уровне 3

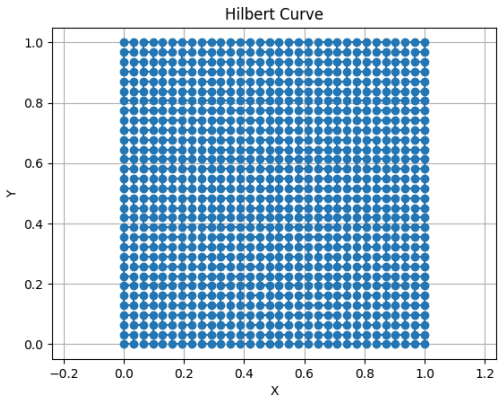

Рис.4. Кривая Гильберта при уровне 5

2.1.3 Результаты

Ниже приведены два результата. Первый результат, результат исполнения программы firstLab.hs, который выводит координаты точек кривой Гильберта при глубине = 4.

Для запуска программы, в терминале Visual Studio Code, нужно прописать команду -

- runhaskell firstLab.hs

[(0.0,0.0),(0.14285714285714285,0.0),(0.14285714285714285,0.14285714285714285),(0.0,0.14285714285714285),
(0.0,0.2857142857142857),(0.0,0.42857142857142855),(0.14285714285714285,0.42857142857142855),
(0.14285714285714285,0.2857142857142857),(0.2857142857142857,0.2857142857142857),
(0.2857142857142857,0.42857142857142855),(0.42857142857142855,0.42857142857142855),
(0.42857142857142855,0.2857142857142857),(0.42857142857142855,0.14285714285714285),
(0.2857142857142857,0.14285714285714285),(0.2857142857142857,0.0),(0.42857142857142855,0.0),
(0.5714285714285714,0.0),(0.5714285714285714,0.14285714285714285),(0.7142857142857143,0.14285714285714285),
(0.7142857142857143,0.0),(0.8571428571428571,0.0),(1.0,0.0),(1.0,0.14285714285714285),
(0.8571428571428571,0.14285714285714285),(0.8571428571428571,0.2857142857142857),(1.0,0.2857142857142857),
(1.0,0.42857142857142855),(0.8571428571428571,0.42857142857142855),(0.7142857142857143,0.42857142857142855),
(0.7142857142857143,0.2857142857142857),(0.5714285714285714,0.2857142857142857),
(0.5714285714285714,0.42857142857142855),(0.5714285714285714,0.5714285714285714),
(0.5714285714285714,0.7142857142857143),(0.7142857142857143,0.7142857142857143),
(0.7142857142857143,0.5714285714285714),(0.8571428571428571,0.5714285714285714),(1.0,0.5714285714285714),
(1.0,0.7142857142857143),(0.8571428571428571,0.7142857142857143),(0.8571428571428571,0.8571428571428571),
(1.0,0.8571428571428571),(1.0,1.0),(0.8571428571428571,1.0),(0.7142857142857143,1.0),
(0.7142857142857143,0.8571428571428571),(0.5714285714285714,0.8571428571428571),(0.5714285714285714,1.0),
(0.42857142857142855,1.0),(0.2857142857142857,1.0),(0.2857142857142857,0.8571428571428571),
(0.42857142857142855,0.8571428571428571),(0.42857142857142855,0.7142857142857143),
(0.42857142857142855,0.5714285714285714),(0.2857142857142857,0.5714285714285714),
(0.2857142857142857,0.7142857142857143),(0.14285714285714285,0.7142857142857143),
(0.14285714285714285,0.5714285714285714),(0.0,0.5714285714285714),(0.0,0.7142857142857143),
(0.0,0.8571428571428571),(0.14285714285714285,0.8571428571428571),(0.14285714285714285,1.0),(0.0,1.0)]

Результат 1. Выполнения firstLab.hs.

Ниже приведен второй результат (графический). Для реализации графической визуализации по координатам полученными с firstLab.hs, был использован язык Python 3.12.2, с использованием библиотеки MatPlotLib. Приведен график фрактала кривой Гильберта при n = 8.

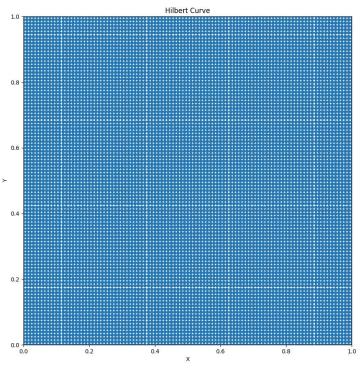

Рис.5. Кривая Гильберта при уровне 8

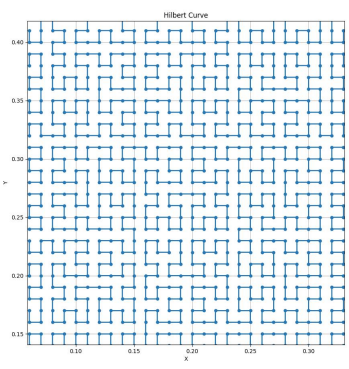

Рис.6. Увеличение изображения на рис.5.

Как видно из рисунка 6, при приближении фрактала, мы так же можем заметить его самоподобие к фигуре представленной ранее в рисунке 2. Таким образом, отображенный график по координатам полученными с реализацией программы действительно является фракталом «кривая Гильберта».

2.2. Работа 2: Игра «Ним»

2.3 Описание игры

Игра Ним - игра в которую играют два игрока, по очереди они берут предмет из кучи предметов (в нашем случае - кучка камней), За один ход можно взять n-ое число предметов (в нашем случае n=3). В результате побеждает игрок взявший последний предмет (камень).
В нашей версии, игрок будет играть с компьютером, где общее количество итераций игры i - варьируется в зависимости от n-го взятия камней игроком/компьютером, таким образом min(i) = 100, max(i) = 300. Так же в нашей реализации компьютер играет с применением стратегии равновесие по Нэшу (рассмотрено в пункте 2.5.1).

2.4 Равновесие по Нэшу

Равновесие Нэша — концепция решения, одно из ключевых понятий теории игр. Так называется набор стратегий в игре для двух и более игроков, в котором ни один участник не может увеличить выигрыш, изменив свою стратегию, если другие участники своих стратегий не меняют. Джон Нэш доказал существование такого равновесия в смешанных стратегиях в любой конечной игре.
Рассмотрим пример с дилеммой о заключенных:
Допустим два преступника арестованы и не имеют возможности общаться и сговориться, потому что сидят в разных камерах. Как полицейским получить показания? Полиция предлагает каждому заключенному возможность либо предать другого, рассказав о его преступлении, либо сохранять молчание.

		Заключенный № 1	
Заключенный № 2		Признаться	Не признаться
	Признаться	-5; -5	-1; -10
	Не признаться	-10; -1	-1; -1

Таб.1 к дилемме о заключенных

- Если оба заключенных предают друг друга, то каждый из них получает 5 лет тюрьмы: (-5; -5).
- Если заключенный № 1 предает заключенного № 2, а заключенный № 2 по-прежнему хранит молчание, то он окажется в худшем положении и получит 10 лет тюрьмы. А заключенный № 1 получит всего 1 год тюрьмы: (-10; -1).
- Если оба заключенных молчат и не признаются, то каждый из них получит только 1 год тюрьмы: (-1; -1).

Равновесие Нэша в этом примере состоит в том, что оба игрока друг друга предадут. Хотя однозначно, взаимное сотрудничество для обоих наиболее выгодный вариант.

2.5 Реализация на языке Haskell

Ниже приведен код, реализующий игру «Ним».

- Общее количество ходов - от 100 до 300.
- Начальное кол-во камней в кучке - 300.
- Можно взять камней за раз - (1-3).
- Стратегия игры - равновесие по Нэшу.

Код игры:

```
import Data.List (intercalate)

mInitial :: Int
mInitial = 300
kMax :: Int
kMax = 3
type Move = Int
type GameState = Int
type MoveHistory = [(String, Move)]
--ИгРы РаЗуМа--
nashStrategy :: GameState -> Move
nashStrategy m
  | remainder == 0 = 1
  | otherwise = remainder
  where remainder = m `mod` (kMax + 1)
moyaStrategiya :: GameState -> IO Move
moyaStrategiya m = do
  putStrLn $ "Осталось " ++ show m ++ " камней"
  putStrLn $ "Ваш ход: сколько камней вы возьмете? (1 до " ++ show (min kMax m) ++ ")"
  move <- getLine
  let move' = read move :: Int
  if move' > 0 && move' <= min kMax m
     then return move'
     else do
       putStrLn "Неверный ход!"
       moyaStrategiya m
-- комп
compik :: GameState -> IO Move
compik m = do
  let move = nashStrategy m
  putStrLn $ "Компьютер взял " ++ show move ++ " камней."
  return move
isGameOver :: GameState -> Bool
isGameOver = (== 0)
-- спк ходвв
printMoves :: MoveHistory -> IO ()
printMoves moves = do
  putStrLn "Ходы в игре:"
  putStrLn $ intercalate "\n" (map (\(player, move) -> player ++ " взял " ++ show move ++ " камней.") moves)
playNim :: GameState -> MoveHistory -> IO ()
playNim m moves
  | isGameOver m = do
      putStrLn "Game Over"
      printMoves moves
  | otherwise = do
      hodUser <- moyaStrategiya m
      let m' = m - hodUser
      let moves' = moves ++ [("Вы", hodUser)]
      if isGameOver m'
         then do
           putStrLn "Вы выиграли! Ура!"
           printMoves moves'
           putStrLn "Счет: 1-0 в вашу пользу."
         else do
           hodComp <- compik m'
           let m'' = m' - hodComp
```

```
          let moves'' = moves' ++ [("Компьютер", hodComp)]
          if isGameOver m''
             then do
               putStrLn "Компьютер выиграл(("
               printMoves moves''
               putStrLn "Счет: 0-1 в пользу компьютера."
             else playNim m'' moves''
main :: IO ()
main = do
  putStrLn "Ним игра"
  playNim mInitial []
```

Подробное описание кода:

Сначала мы подключаем библиотеку intercalate из Data.List, для того что бы потом работать смогли вывести список ходов игры.

Теперь нам нужно задать тип количества камней, это будет Int. После нужно указать количество (у нас 300). Также нужно задать ограничение на взятие k - камней. В моей реализации k = 3.

mInitial :: Int

mInitial = 100

kMax :: Int

kMax = 3

Теперь нужно определить тип для обозначения количества взятых камней (Move), тип для хранения текущего количества оставшихся камней (GameState) и список пар, где первая строка указывает игрока ("Вы" или "Компьютер"), а второе значение — количество взятых камней (MoveHistory).

type Move = Int

type GameState = Int

type MoveHistory = [(String, Move)]

По заданию нужно реализовать игру со стратегией - «равновесие по Нэшу». Для ее реализации нужно написать логику функции:

- Если остаток от деления текущего количества камней на kMax+1 = 0, то компьютер берет 1 камень
- Либо, компьютер берет столько камней, чтобы оставшееся количество было кратно 4

```
nashStrategy :: GameState -> Move
nashStrategy m
  | remainder == 0 = 1
  | otherwise = remainder
  where remainder = m `mod` (kMax + 1)
```

Теперь сама игра, нужно запросить у пользователя ход который проверяет его корректность, и если введено неправильное значение, повторно запрашивает ввод. Пользователь может взять

от 1 до 3 камней (но не больше, чем осталось в куче). И для компьютера тоже, но компьютер уже будет использовать стратегию Нэша, которую описали ранее.
Стратегия игрока:

```
moyaStrategiya :: GameState -> IO Move
moyaStrategiya m = do
  putStrLn $ "Осталось " ++ show m ++ " камней"
  putStrLn $ "Ваш ход: сколько камней вы возьмете? (1 до " ++ show (min kMax m) ++ ")"
  move <- getLine
  let move' = read move :: Int
  if move' > 0 && move' <= min kMax m
     then return move'
     else do
       putStrLn "Неверный ход!"
       moyaStrategiya m
```

Стратегия компьютера (по Нэшу):

```
compik :: GameState -> IO Move
compik m = do
  let move = nashStrategy m
  putStrLn $ "Компьютер взял " ++ show move ++ " камней."
  return move
```

Еще нам надо прописать окончание игры. Которое будет переводить тип GameState в булевое значение. Игра окончена если в кучке n = 0 камней.

```
isGameOver :: GameState -> Bool
isGameOver = (== 0)
```

После окончания игры выводим наш список ходов, используя модуль который подключали в начале.

```
printMoves :: MoveHistory -> IO ()
printMoves moves = do
  putStrLn "Ходы в игре:"
  putStrLn $ intercalate "\n" (map (\(player, move) -> player ++ " взял " ++ show move ++ "
камней.") moves)
```

Теперь нужен цикл игры, с такой логикой, что:

- Если игра окончена (то есть количество камней равно 0), выводится сообщение "Game Over" и история всех ходов.
- Если игра продолжается, сначала происходит ход игрока, уменьшается количество камней, и обновляется история ходов.
- Если после хода игрока игра не окончена, делает ход компьютер, снова обновляется количество камней и история ходов.

Этот процесс повторяется до тех пор, пока количество камней не станет 0.

```
playNim :: GameState -> MoveHistory -> IO ()
playNim m moves
  | isGameOver m = do
      putStrLn "Game Over"
      printMoves moves
  | otherwise = do
      hodUser <- moyaStrategiya m
      let m' = m - hodUser
      let moves' = moves ++ [("Вы", hodUser)]
      if isGameOver m'
        then do
```

```
      putStrLn "Вы выиграли! Ура!"
      printMoves moves'
      putStrLn "Счет: 1-0 в вашу пользу."
    else do
      hodComp <- compik m'
      let m'' = m' - hodComp
      let moves'' = moves' ++ [("Компьютер", hodComp)]
      if isGameOver m''
        then do
          putStrLn "Компьютер выиграл(("
          printMoves moves''
          putStrLn "Счет: 0-1 в пользу компьютера."
        else playNim m'' moves''
```

После чего вызываем main.

```
main :: IO ()
main = do
  putStrLn "Ним игра"
  playNim mInitial []
```

2.5.1 Равновесие Нэша в программной реализации

Игра Ним с n-камнями и максимальным количеством k-камней, которые можно взять за один ход, может быть проанализирована через стратегию Нэша, основанную на идее "нулевой позиции" — состояния игры, из которого любой ход приведет к выигрышной позиции для соперника. В нашем случае "нулевые позиции" — это состояния, когда оставшееся количество камней n кратно k+1.

Математическое описание:

Обозначим количество оставшихся камней как m, максимальное количество, которое можно взять за один ход = k и ход игрока = x.

Если у нас m = 0 mod (k+1) значит это нулевая позиция и следующий игрок не может использовать стратегию Нэша, так как любой его ход будет выгоден для соперника.

Если m != 0 mod(k+1) значит игрок должен взять x=m mod(k+1) камней что бы перевести все в нулевую позицию.

Ниже представлена таблица для m = 10 и k = 3:

m камней осталось	Нулевая позиция True/ False	Оптимальный ход (комп.)
10	F	2
8	T	Любой
7	F	3
6	F	2
5	F	1
4	T	Любой
3	F	3
2	F	2
1	F	1
Нету	T	-

Таблица 2. Стратегия при m=10, k = 3.

2.6 Результаты

Ниже приведены результаты игры Ним, со стратегией - равновесие по Нэшу. С начальным количеством камней = 300, и взятием камней от 1-3. Рисунки (7-13).

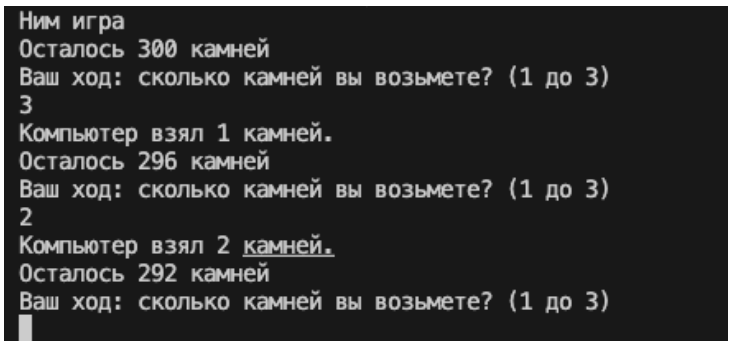

Рис. 7 Процесс игры

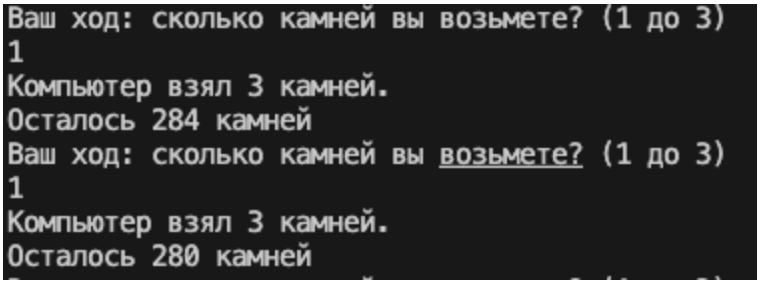

Рис. 8 Процесс игры

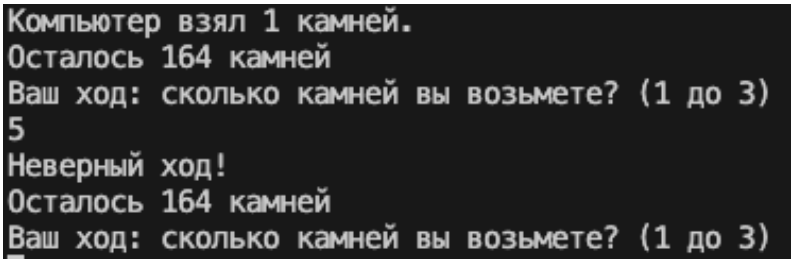

Рис. 9 Неверный ход игрока

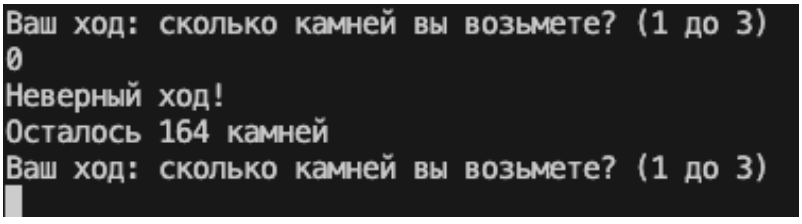

Рис. 10 Неверный ход игрока

Рис. 11 Результат игры

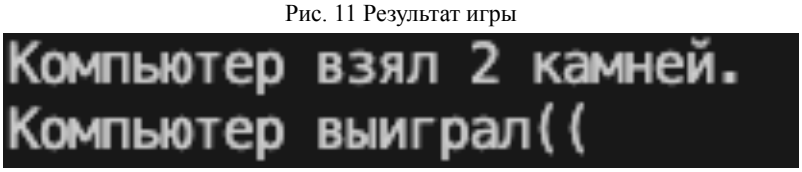

Рис. 12 Результат игры

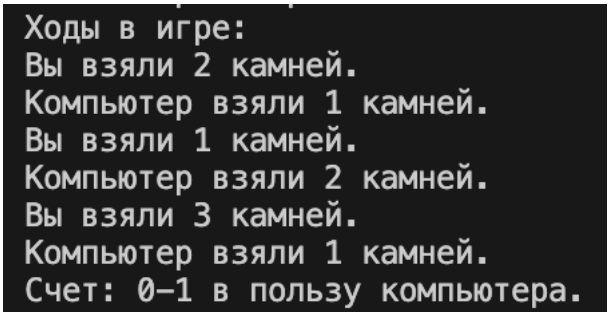

Рис. 13 Вывод списка хранящий ходы

3. Заключение

В результате выполнения практического задания №2, был выполнен комплекс работ, в процессе которого:

- Были изучены основы программирования на языке Haskell.
- Были изучены теоретические основы фракталов.
- Была изучена стратегия - равновесие по Нэшу.

В результате выполнения реализовано:

- Вычисления всех пар координат (x,y) для заданного фрактала на глубине n шагов. (кривая Гильберта)
- Изображен график заданного фрактала с помощью кода на Python 3.12.2 использующий библиотеку MatPlotLib.
- Игра «Ним», где была использована стратегия «равновесие по Нэшу», в игре настроено от 100 до 300 ходов. Максимальное количество взятых камней за 1 ход = 3. При желании параметры игры можно поменять.

Все программы (за исключением файла изображения графика графа), имеют формат (.hs), программы написаны на языке Haskell, в интегрируемой среде разработки Visual Studio Code 1.94.