МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 3343 _____ Бондаренко Ф. А.
Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2025

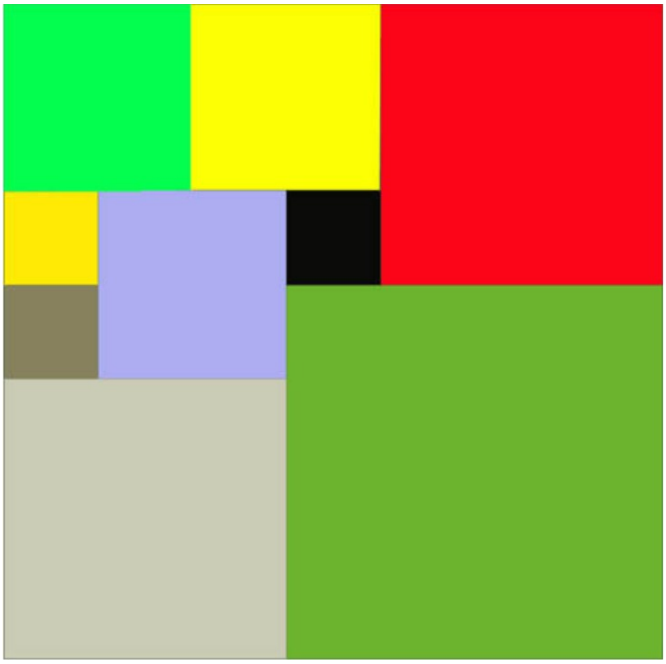
Цель работы.

Изучение алгоритма бэктрекинга (поиска с возвратом) для решения задачи о квадрировании квадрата.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу – квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков (квадратов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N $(2 \le N \le 20)$.

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат)

заданного размера *N*. Далее должны идти *K* строк, каждая из которых должна содержать три целых числа *x,y* и *w*, задающие координаты левого верхнего угла ($1 \le x,y \le N$) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Выполнение работы.

Вар. 2и. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Для решения задачи был применен алгоритм итеративного бэктрекинга.

Описание реализованных структур, классов, методов и функций:

- *struct Square* — структура, отвечающая за работу с квадратом: хранит такие поля, как:
  - *int x*, *int y* — целочисленные координаты квадрата.
  - *int size* — размер квадрата.
- *class Board* — класс отвечающий за работу с доской для замощения. Содержит:
  - *int size* — размер поля для замощения.
  - *int emptyCells* — число свободных (пустых) клеток.

- *int startX, int startY* — координаты первой теоретически незанятой (пустой) клетки (необходимы для ускорения поиска первой свободной клетки (см. п. Оптимизации)).
- *std::vector<Square> squares* — вектор квадратов, участвующих в замощении.
- *Board(int size)* — конструктор, принимающий размер полотна.
- *Board(const Board& other)* — конструктор копирования.
- *Board& operator=(const Board& other)* — оператор присваивания.
- *int getSize() const* — метод, который возвращает размер полотна.
- *int getEmptyCells() const* — метод, который возвращает количество пустых клеток.
- *const std::vector<Square>& getSquares() const* — метод, возвращающий вектор квадратов, участвующих в замощении.
- *void setStartCoordinates(int x, int y)* — метод, устанавливающий *startX* и *startY*.
- *bool isPointOccupied(int x, int y) const* — метод, проверяющий, является ли данная клетка поля пустой.
- *bool isValid(std::pair<int, int> coordinates, int squareSize) const* — метод, проверяющий валидность расстановки квадрата в данной области.
- void addSquare(const Square& square) — метод добавления квадрата в вектор квадратов.
- std::pair<int, int> findEmptyCell() const — метод поиска первой свободной клетки.
- void scale(int scaleFactor) — метод для увеличения размера полотна (вместе с размером квадратов) на определенный целый коэффициент.

• В файле *Tiling.hpp* и *Tiling.cpp* приведены реализации алгоритма итеративного backtracking вместе с оптимизациями. Отведение для реализации алгоритма отдельного файла необходимо для

корректного подключения через *include* функции *solve()* в файле *benchmarks.cpp*.

- *Board backtrack(Board initBoard)* — итеративная функция поиска с возвратом. Вначале определяются такие поля, как: *int minimalCount* — минимальное число квадратов, участвующее в замощении полотна, *Board bestBoard* — лучший результат замощения, *std::stack<Board> stack* — стек состояний доски.

  Далее в стек заносится начальная доска *startBoard* и начинает работу цикл *while*. В цикле *while* достается первая доска *board* и обрабатывается: проверяется число квадратов, участвующее в замощении (если больше *minimalCount*, то сразу такую доску отметаем), если же число свободных клеток в такой доске равняется 0 и число квадратов меньше лучшего замощения (было проверено в прошлом *if*), то делаем данную доску лучшим вариантом.

  После начальной проверки производится поиск первой свободной клетки. Затем в цикле *for* пробегаемся по возможным размерам квадратов, что можно установить в данной клетке. Если данный квадрат можно установить, то создаем новую доску *newBoard*, устанавливаем *startX* и *startY* для данной доски: если установленный квадрат не доходит до границы области полотна, то смещаемся вправо на размер квадрата, иначе спускаемся по оси Oy вниз на одну ячейку. Добавляем *newBoard* в стек.
- *Board solve(int size)* — функция, которая использует оптимизацию для замощения полотна: берет не начальный входной размер, а находит его первый простой делитель. Затем применяет функцию *initPrimeBoard()* (устанавливает т. н. "штаны") и алгоритм backtracking к данной доске, и увеличивает размер полотна (и, соответственно, квадратов в полотне) на коэффициент масштабирования.

- *std::set<int> baseFactorize(int n)* — функция, находящая основания в факторизации целого числа.
- *Board initPrimeBoard(int size)* — функция, устанавливающая три начальных квадрата на полотно, размер которого — простое число. Сначала создает доску определенного размера, затем устанавливает три квадрата на нее: одни в левый верхний угол (со стороной *(size + 1) / 2*), и два других: один под верхний квадрат (со стороной *size / 2*), другой сбоку справа (также со стороной *size / 2*). Возвращает полученную доску.

- Файл *benchmarks.cpp* содержит класс *Benchmark* и дополнительную структуру *BenchmarkResult*, необходимый для подсчета времени выполнения алгоритма.
- В файле *main.cpp* происходит считывание пользовательского значения размера полотна, запуск алгоритма квадрирования, а также вывод полученных результатов.
- Python-скрипт *visual.py* необходим для визуализации результата замощения полотна.

Оценка сложности алгоритма.

- По времени: $O(c^{p^2})$, где c — положительная константа, p — минимальный делитель стороны полотна (для p^2 ячеек можно принять с решений).
- По памяти: $O(k \cdot n)$, где k — максимальный размер стека, n — число квадратов в векторе квадратов данной доски.

Оптимизации.

Основная идея оптимизации заключается в замощении меньшего полотна (чем то, что было дано на вход), а затем умножение на коэффициент масштабирования. Данная идея значительно уменьшает время работы алгоритма, так как сокращается число возможных расстановок.

По условию, квадраты, участвующие в замощении, имеют целочисленную сторону. Следовательно мы должны выбрать такой подполотно в нашем полотне, чтобы при умножении на коэффициент мы получили квадраты с целочисленной стороной.

Какой размер подполотна брать? **Первое требование**: размер подполотна должен быть **делителем** размера полотна. Иначе: квадраты в замощении могут быть с нецелой стороной.

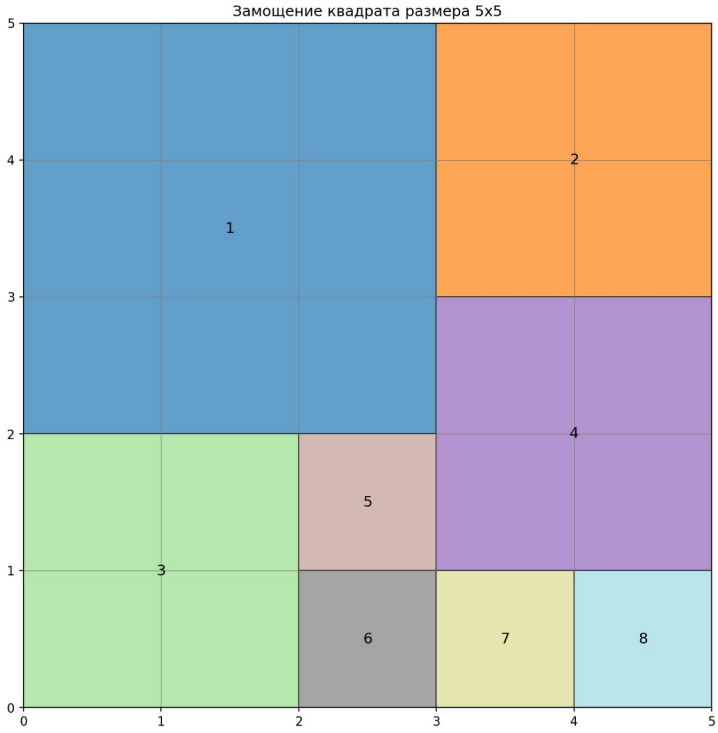


Рисунок 1 — результат замощения квадрата 5x5

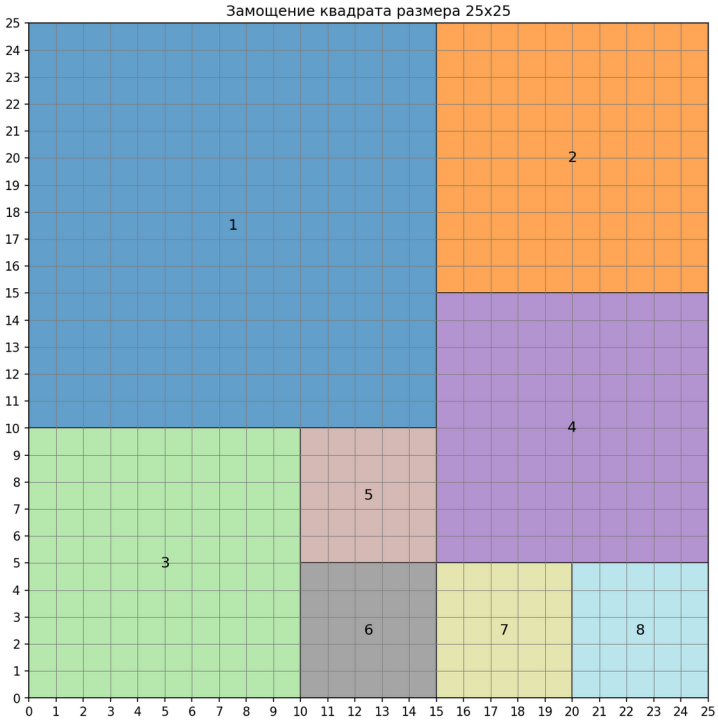


Рисунок 2 — результат замощения квадрата 25x25

Второе требование: размер подполотна должен быть **минимальным делителем** размера полотна. Иначе число квадратов замощении будет не минимальным.

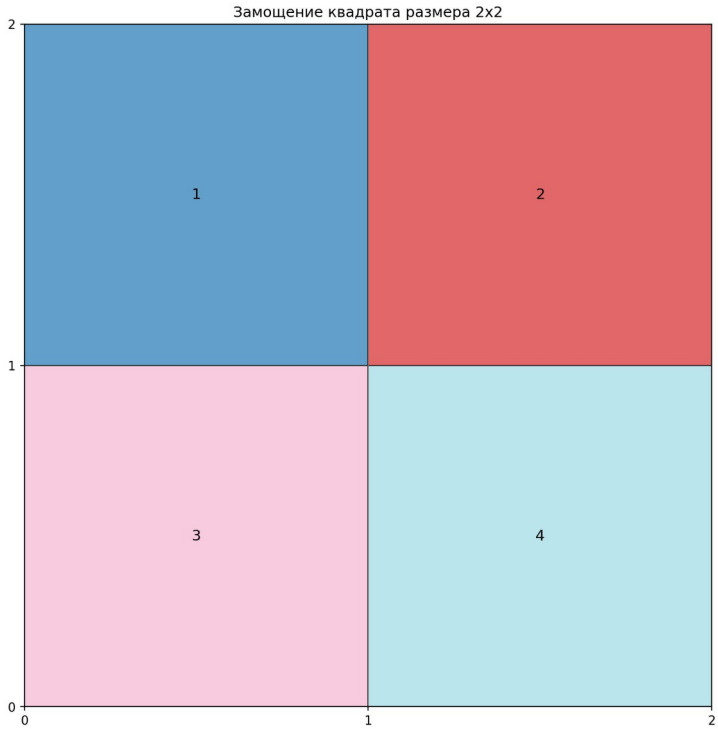


Рисунок 3 — результат замощения квадрата 2x2

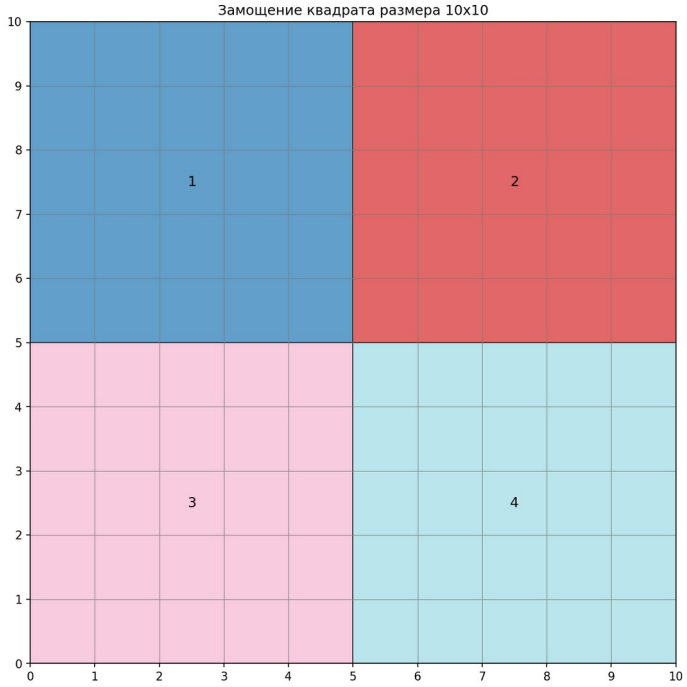


Рисунок 4 — результат замощения квадрата 10x10

Как видно из примера: квадрат со стороной 10 имеет в своем минимальном замощении 4 квадрата (как у подполотна 2x2), а не 8 (как у подполотна 5x5). Хотя 10 кратно и 2, и 5.

Итого, задача о замощении квадрата свелся к замощению подквадрата, размер которого — минимальный делитель размера начального квадрата.

В ходе проверок результатов, было эмпирически выведено, что для полотен с простой стороной p в замощении обязательно участвуют три квадрата: один размера $[\frac{p+1}{2}]$, два размерами $[\frac{p}{2}]$, где [x] — целая часть от числа x. Для удобства восприятия и обработки, данные квадраты устанавливаются по координатам: 1) $(1,1)$ (самый большой), 2) $(1+[\frac{p+1}{2}],1)$ (первый из двух маленьких (справа от большого)), 3) $(1,1+[\frac{p+1}{2}])$ (второй из двух маленьких (под большим)). Таким образом, будет оптимально замощено примерно 70 - 75 % от изначального полотна.

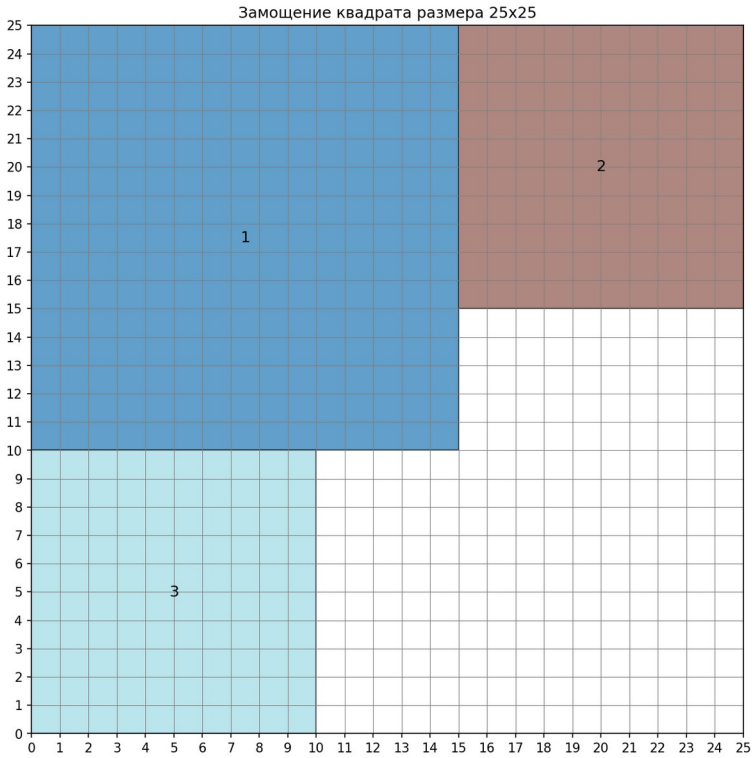


Рисунок 5 — пример начальной расстановки

В связи с такой начальной расстановкой, в класс *Board* были добавлены стартовые координаты для поиска свободной клетки (*startX*, *startY*). После инициализации подполотна они будут равны $(1+[\frac{p+1}{2}], 1+[\frac{p}{2}])$, соответственно. Далее в методе класса Board — findEmptyCell() проверка разбивается на две части: 1) нижняя часть полотна: если строка — первая (т. е. $y == startY$), то обход начинается с $x = startX$, иначе: $x = 0$ (т. е. полностью обходится строка). 2) верхняя часть полотна: если в нижней части полотна свободная клетка не нашлась. То проверяется верхняя (до $y == startY$).

Исследование.

Рисунок 6 — зависимость времени выполнения алгоритма от размера доски (по логарифмической шкале)

Рисунок 7 — зависимость времени выполнения алгоритма от размера доски (по линейной шкале)

По результатам графиков, можно сделать вывод, что время выполнения алгоритма сильно (экспоненциально) возрастает при возрастании минимального делителя стороны полотна.

Тестирование.

Входные данные	Выходные данные	Комментарий
13	11 1 1 7 8 1 6 1 8 6 8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2	Успех
26	4 1 1 13 14 1 13 1 14 13 14 14 13	Успех
37	15 1 1 19 20 1 18 1 20 18 20 19 2 22 19 5 27 19 11 19 20 1 19 21 3 19 24 8	Успех

	27 30 3 30 30 8 19 32 6 25 32 1 26 32 1 25 33 5	

Вывод.

По ходу данной лабораторной работы была написана программа, решающая задачу о квадрировании квадрата с использованием итеративного алгоритма бэктрекинг (поиск с возвратом). Было проведено исследование зависимости времени работы алгоритма от начального размера квадрата.

ПРИЛОЖЕНИЕ А

Файл *main.cpp*:

```
#include <iostream>

#include "include/Board.hpp"
#include "include/Square.hpp"
#include "include/Tiling.hpp"

#define MIN_SIZE 2
#define MAX_SIZE 40

int main() {
  int n = 0;
  std::cin >> n;

  if (n < MIN_SIZE || n > MAX_SIZE) {
    std::cerr << "Invalid input size!!"<< std::endl;
    return 1;
  }

  Board board = solve(n);

  std::cout << board.getSquares().size() << std::endl;
  for (const Square& square : board.getSquares()) {
      std::cout << square.x + 1 << " " << square.y + 1 << " " <<
square.size
              << std::endl;
  }

  return 0;
}
```

Файл *Board.hpp*:

```
#ifndef BOARD_HPP_
#define BOARD_HPP_

#include <utility>
#include <vector>

#include "Square.hpp"

class Board {
 private:
  int size;
  int emptyCells;
  int startX, startY;
  std::vector<Square> squares;

  bool isPointOccupied(int x, int y) const;

 public:
  Board(int size);
  Board(const Board& other);
  Board& operator=(const Board& other);

  int getSize() const { return size; }
  int getEmptyCells() const { return emptyCells; }
```

```
  const std::vector<Square>& getSquares() const { return squares; }
  int getSquaresCount() const { return squares.size(); }

  void setStartCoordinates(int x, int y) {
    startX = x;
    startY = y;
  }

  bool isValid(std::pair<int, int> coordinates, int squareSize) const;

  void addSquare(const Square& square);

  std::pair<int, int> findEmptyCell() const;

  void scale(int scaleFactor);
};

#endif  // BOARD_HPP_
```

Файл *Square.hpp*:

```
#ifndef SQUARE_HPP_
#define SQUARE_HPP_

#include <utility>

struct Square {
  int x, y;
  int size;

  Square() = default;
  Square(std::pair<int, int> coordinates, int size)
      : x(coordinates.first), y(coordinates.second), size(size) {}
};

#endif  // SQUARE_HPP_
```

Файл *Tiling.hpp*:

```
#ifndef TILING_HPP_
#define TILING_HPP_

#include <set>

#include "Board.hpp"

std::set<int> baseFactorize(int n);
Board backtrack(Board startBoard);
Board solve(int size);

#endif  // TILING_HPP_
```

Файл Board.cpp:

```
#include "../include/Board.hpp"

Board::Board(int size) : size(size), emptyCells(size * size) {}

Board::Board(const Board& other)
    : size(other.size),
      emptyCells(other.emptyCells),
      startX(other.startX),
```

```
      startY(other.startY),
      squares(other.squares) {}

Board& Board::operator=(const Board& other) {
  if (this != &other) {
    size = other.size;
    emptyCells = other.emptyCells;
    startX = other.startX;
    startY = other.startY;
    squares = other.squares;
  }
  return *this;
}

bool Board::isPointOccupied(int x, int y) const {
  for (const Square& square : squares) {
     if (x >= square.x && x < square.x + square.size && y >= square.y
&&
        y < square.y + square.size) {
      return true;
    }
  }
  return false;
}

bool Board::isValid(std::pair<int, int> coordinates, int squareSize)
const {
  if (squareSize >= size) {
    return false;
  }

  int x = coordinates.first;
  int y = coordinates.second;

   if (x < 0 || y < 0 || x + squareSize > size || y + squareSize >
size) {
    return false;
  }

  for (const Square& square : squares) {
    if (x < square.x + square.size && x + squareSize > square.x &&
        y < square.y + square.size && y + squareSize > square.y) {
      return false;
    }
  }
  return true;
}

void Board::addSquare(const Square& square) {
  squares.push_back(square);
  emptyCells -= square.size * square.size;
}

std::pair<int, int> Board::findEmptyCell() const {
  for (int y = startY; y < size; y++) {
    int startX_pos = (y == startY ? startX : 0);
    for (int x = startX_pos; x < size; x++) {
      if (!isPointOccupied(x, y)) {
        return {x, y};
```

```
      }
    }
  }

  for (int y = 0; y < startY; y++) {
    for (int x = 0; x < size; x++) {
      if (!isPointOccupied(x, y)) {
        return {x, y};
      }
    }
  }

  return {-1, -1};
}

void Board::scale(int scaleFactor) {
  for (Square& square : squares) {
    square.x *= scaleFactor;
    square.y *= scaleFactor;
    square.size *= scaleFactor;
  }
  size *= scaleFactor;
}
```

Файл *Tiling.cpp*:

```
#include "../include/Tiling.hpp"

#include <cmath>
#include <cstdint>
#include <stack>
#include <utility>

#include "../include/Board.hpp"
#include "../include/Square.hpp"

std::set<int> baseFactorize(int n) {
  std::set<int> factors;
  for (int i = 2; i <= std::sqrt(n); i++) {
    if (n % i == 0) {
      factors.insert(i);
      while (n % i == 0) {
        n /= i;
      }
    }
  }
  if (n > 1) {
    factors.insert(n);
  }
  return factors;
}

Board backtrack(Board startBoard) {
  int minimalCount = INT32_MAX;
  Board bestBoard(startBoard.getSize());
  std::stack<Board> stack;
  stack.push(startBoard);

  while (!stack.empty()) {
    Board board = stack.top();
```

```
    stack.pop();

    if (board.getSquaresCount() >= minimalCount) {
      continue;
    } else if (board.getEmptyCells() == 0) {
      minimalCount = board.getSquaresCount();
      bestBoard = board;
      continue;
    }

    auto [x, y] = board.findEmptyCell();

    for (int squareSize = 1;
         squareSize <= std::min(board.getSize() - x, board.getSize() -
y);
         ++squareSize) {
      if (board.isValid({x, y}, squareSize)) {
        Board newBoard = board;
        Square square({x, y}, squareSize);
        newBoard.addSquare(square);

        if (x + squareSize < board.getSize()) {
          newBoard.setStartCoordinates(x + squareSize, y);
        } else {
          newBoard.setStartCoordinates(0, y + 1);
        }

        stack.push(newBoard);
      }
    }
  }

  return bestBoard;
}

Board initPrimeBoard(int size) {
  Board initBoard(size);

  int centralSize = (size + 1) / 2;
  int sideSize = size / 2;

  Square central({0, 0}, centralSize);
  Square rightSide({centralSize, 0}, sideSize);
  Square leftSide({0, centralSize}, sideSize);

  initBoard.addSquare(central);
  initBoard.addSquare(rightSide);
  initBoard.addSquare(leftSide);
  initBoard.setStartCoordinates(centralSize, sideSize);

  return initBoard;
}

Board solve(int size) {
  std::set<int> factors = baseFactorize(size);
  int minimal = *factors.begin();
  int coef = size / minimal;

  Board startBoard = initPrimeBoard(minimal);
```

```
  Board board = backtrack(startBoard);
  board.scale(coef);

  return board;
}
```

Файл *benchmarks.cpp*:

```
#include <chrono>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <nlohmann/json.hpp>
#include <vector>

#include "../include/Board.hpp"
#include "../include/Square.hpp"
#include "../include/Tiling.hpp"

#define _DEFAULT_ITERATIONS 3
#define _DEFAULT_OUTPUT_FILE "results.json"

#define _MIN_SIZE 2
#define _MAX_SIZE 40

struct BenchmarkResult {
  int size;
  double time;
  int iterations;
  Board board;

  BenchmarkResult(int size, double time, int iterations, Board result)
       : size(size), time(time), iterations(iterations), board(result)
{}
};

class Benchmark {
 private:
  int iterations;
  std::vector<BenchmarkResult> benchmarkResults;

 public:
  Benchmark(int iterations) : iterations(iterations) {}

  void runBenchmarks(const std::vector<int>& board_sizes) {
    benchmarkResults.clear();

    for (size_t idx = 0; idx < board_sizes.size(); idx++) {
      int n = board_sizes[idx];
          std::cout  <<  "----------------------------------------"  <<
std::endl;
      std::cout << "Testing board size " << n << "..." << std::endl;

      double totalTime = 0.0;
      Board result(n);

      for (int i = 0; i < iterations; i++) {
        auto start = std::chrono::high_resolution_clock::now();
        result = solve(n);
        auto end = std::chrono::high_resolution_clock::now();
```

```
        std::chrono::duration<double> elapsed = end - start;
        totalTime += elapsed.count();

        std::cout << "  Iteration " << (i + 1) << "/" << iterations <<
": "
                  << elapsed.count() << "s" << std::endl;
      }

      double averageTime = totalTime / iterations;
        std::cout << "Average time for board size " << n << ": " <<
averageTime
                << " seconds" << std::endl;

            benchmarkResults.emplace_back(n,  averageTime,  iterations,
result);

          std::cout  <<  "----------------------------------------"  <<
std::endl;
    }
  }

  void saveResultsToJson(const std::string& output_file) const {
    nlohmann::json results = {};
    results["benchmarks"] = nlohmann::json::array();

    for (const auto& result : benchmarkResults) {
      nlohmann::json entry = {};
      entry["size"] = result.size;
      entry["time"] = result.time;
      entry["iterations"] = result.iterations;

      entry["squares"] = nlohmann::json::array();
      std::vector<Square> squares = result.board.getSquares();
      for (size_t i = 0; i < squares.size(); i++) {
        Square square = squares[i];
        nlohmann::json info = {};

        info["size"] = square.size;
        info["x"] = square.x;
        info["y"] = square.y;
        entry["squares"].push_back(info);
      }

      results["benchmarks"].push_back(entry);
    }

    std::ofstream outputFile(output_file);
    outputFile << std::setw(2) << results << std::endl;
    outputFile.close();

    std::cout << "\nResults saved to " << output_file << std::endl;
  }

  const std::vector<BenchmarkResult>& getResults() const {
    return benchmarkResults;
  }
};
```

```
int main() {
  std::vector<int> board_sizes;
  for (int i = _MIN_SIZE; i <= _MAX_SIZE; i++) {
    board_sizes.push_back(i);
  }

  Benchmark benchmark(_DEFAULT_ITERATIONS);

  benchmark.runBenchmarks(board_sizes);
  benchmark.saveResultsToJson(_DEFAULT_OUTPUT_FILE);

  return 0;
}
```

Файл *visual.py*:

```
import json
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import os
from dataclasses import dataclass

_DEFAULT_OUTPUT_DIR = "visual/output"
_DEFAULT_JSON_PATH = "results.json"


@dataclass
class Square:
    x: int
    y: int
    size: int


@dataclass
class Board:
    size: int
    squares: list[Square]


def _load_data(json_path: str):
    with open(json_path, 'r') as file:
        data = json.load(file)
    file.close()

    if "benchmarks" not in data:
        raise ValueError(
                f"Invalid JSON format: 'benchmarks' key not found in
{json_path}")

    if not data["benchmarks"]:
        raise ValueError(f"No benchmark data found in {json_path}")

    return data["benchmarks"]


def _desserialize_square(square_info) -> Square:
                return    Square(square_info["x"],    square_info["y"],
square_info["size"])
```

```
def _deserialize_board(board_info) -> Board:
    return Board(board_info["size"], [_desserialize_square(square) for
square in board_info["squares"]])


def _configure_ax(ax: plt.Axes, board: Board) -> None:
    ax.set_xlim(0, board.size)
    ax.set_ylim(0, board.size)

    ax.set_xticks(np.arange(0, board.size + 1, 1))
    ax.set_yticks(np.arange(0, board.size + 1, 1))
    ax.grid(True, color='gray', linestyle='-', linewidth=0.5)

                    ax.set_title(f"Замощение     квадрата     размера
{board.size}x{board.size}")


def      visualize_tiling(board_data,      output_dir:      str      =
_DEFAULT_OUTPUT_DIR) -> str:
    os.makedirs(output_dir, exist_ok=True)

    board = _deserialize_board(board_data)

    fig, ax = plt.subplots(figsize=(10, 10))
    _configure_ax(ax, board)

    colors = plt.cm.tab20(np.linspace(0, 1, len(board.squares)))

    for i, square in enumerate(board.squares):
            rect  =  patches.Rectangle((square.x,  board.size-square.y-
square.size), square.size, square.size,
                                   linewidth=1, edgecolor='black',
                                   facecolor=colors[i], alpha=0.7)
        ax.add_patch(rect)

           ax.text(square.x + square.size/2, board.size - square.y -
square.size/2, str(i+1),
                fontsize=12, ha='center', va='center')

    output_path = os.path.join(output_dir, f"tiling_{board.size}.png")
    plt.savefig(output_path, dpi=150, bbox_inches='tight')
    plt.close()

    return output_path


def plot_times(results, output_dir: str = _DEFAULT_OUTPUT_DIR) -> str:
    sizes = [result["size"] for result in results]
    times = [result["time"] for result in results]

    plt.figure(figsize=(12, 6))
    plt.semilogy(sizes, times, 'o-', linewidth=2, markersize=8)

    for size, time in zip(sizes, times):
            plt.annotate(f'{size}',  (size,  time),  textcoords="offset
points",
                    xytext=(0, 5), ha='center')
```

```
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
        plt.title('Сравнение времени выполнения для разных размеров
доски')
    plt.xlabel('Размер доски (сторона)')
    plt.ylabel('Время выполнения (секунды)')

    output_path = os.path.join(output_dir, "execution_times.png")
    plt.savefig(output_path, dpi=150, bbox_inches='tight')
    plt.close()

    return output_path


def   main(path:   str   =   _DEFAULT_JSON_PATH,   output:   str   =
_DEFAULT_OUTPUT_DIR):
    results = _load_data(path)
    print(f"Loaded data for {len(results)} board sizes")
    os.makedirs(output, exist_ok=True)

    for result in results:
        path = visualize_tiling(result, output)
        print(
            f"Created visualization for board size {result['size']} at
{path}")

    times_path = plot_times(results, output)
    print(f"Created execution time plot at {times_path}")


if __name__ == "__main__":
    try:
        main()
    except ValueError as e:
        print(f"Error: {e}")
```