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

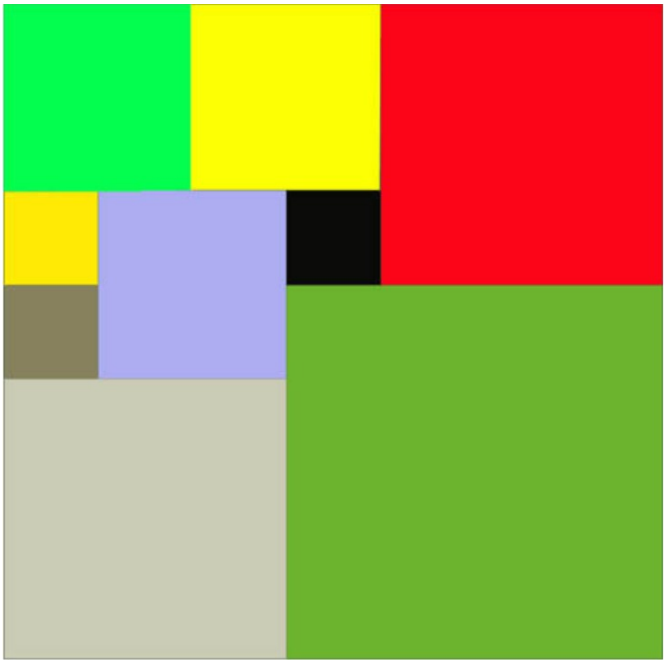
Цель работы.

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 *Benchmark.hpp* содержит класс *Benchmark*, необходимый для подсчета времени выполнения алгоритма. Использует структуру *BenchmarkResult*, которая содержит размер доски и время замощения этой доски. В конструкторе класс *Benchmark* принимает число — размер выборки для проведения исследования.
- В файле *main.cpp* происходит обработка параметров, поданных на вход через терминал, выбор режима работы: обычный (самый быстрый), подсчет времени выполнения (для исследования), визуализации результатов для разных размеров доски, режим поэтапной визуализации работы замощения.
- Файл *Visualizer.cpp* необходим для визуализации замощения. Содержит функции для отрисовки и создания gif- и .mp4 файлов. Для визуализации использовались библиотеки *matplotplusplus* и *ffmpeg*.
- Файл *JsonHandler.hpp* содержит клас для работы с файлами, удовлетворяя идиоме RAII: время жизни выделенного ресурса совпадает с временем жизни объекта.
- Файлы *Serializer.hpp* и *Deserializer.hpp* содержат классы для сериализации и десериализации квадратов и досок для замощения. Нужны для сохранения в json файл результата исследования работы

алгоритма и использования этих данных для построения визуализации.

Оценка сложности алгоритма.

- По времени: $O(c^{p^2})$, где c — положительная константа, p — минимальный простой делитель стороны полотна (для p^2 ячеек можно принять с решений).
- По памяти: $O(p^2 \log(p))$, где p — минимальный простой делитель стороны полотна.

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 **простым делителем** размера полотна. Иначе: квадраты в замощении могут быть с нецелой стороной.

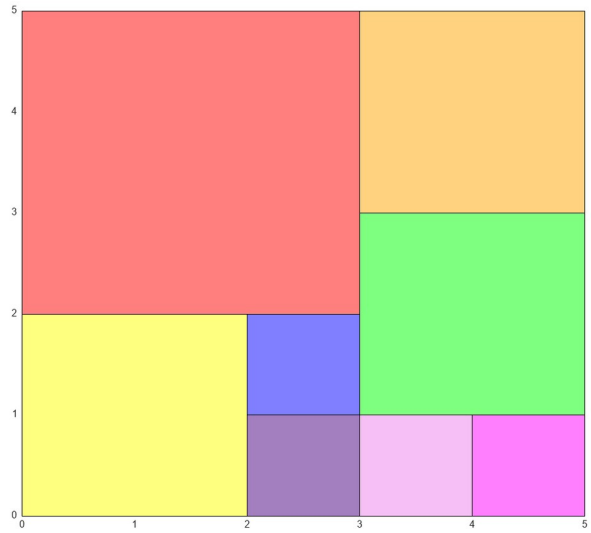


Рисунок 1 — результат замощения квадрата 5x5

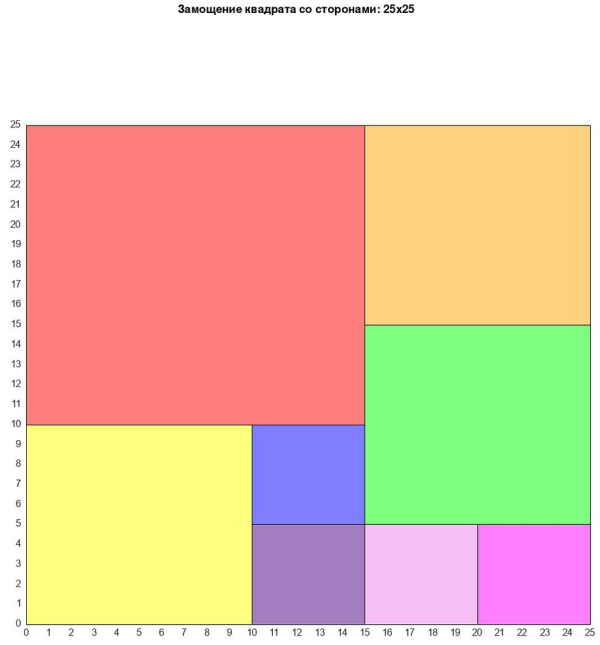


Рисунок 2 — результат замощения квадрата 25x25

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

Рисунок 3 — результат замощения квадрата 2x2

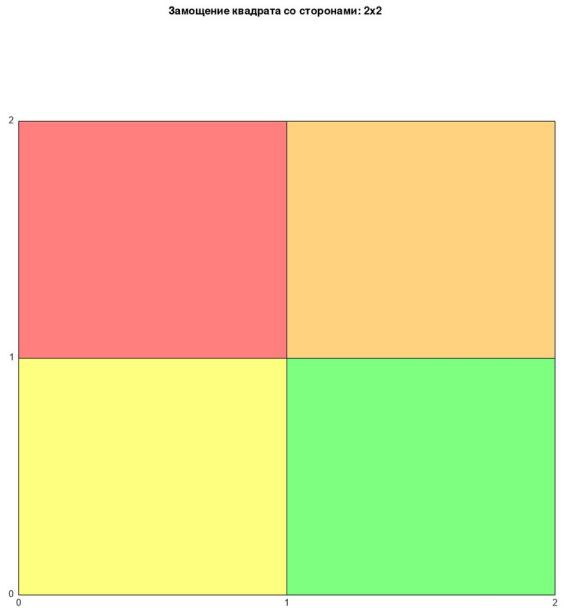
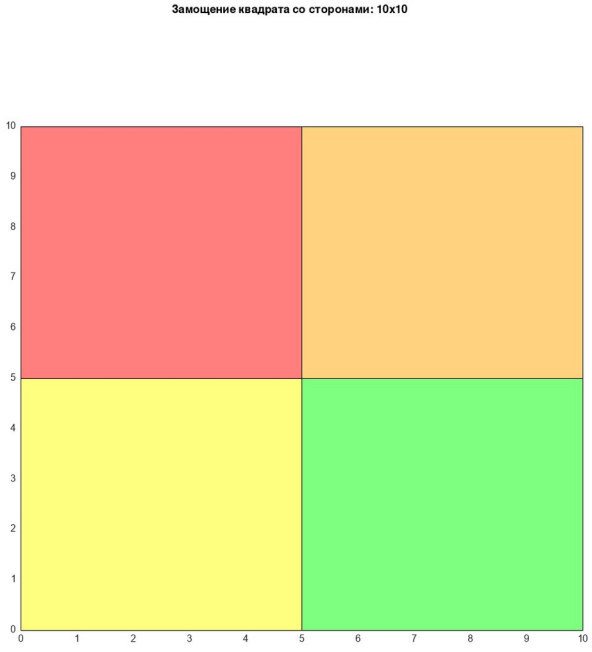


Рисунок 4 — результат замощения квадрата 10x10

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.

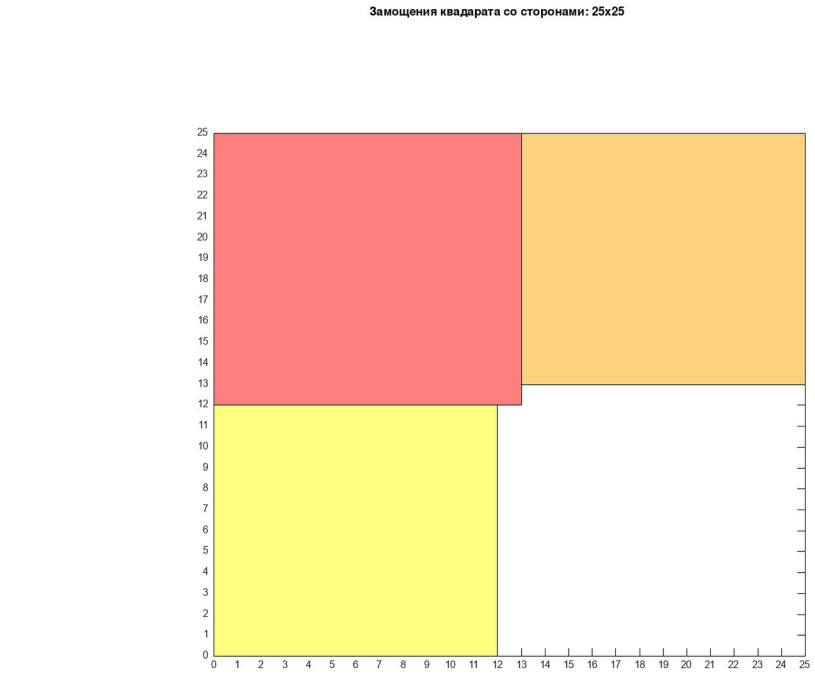


Рисунок 5 — пример начальной расстановки

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.

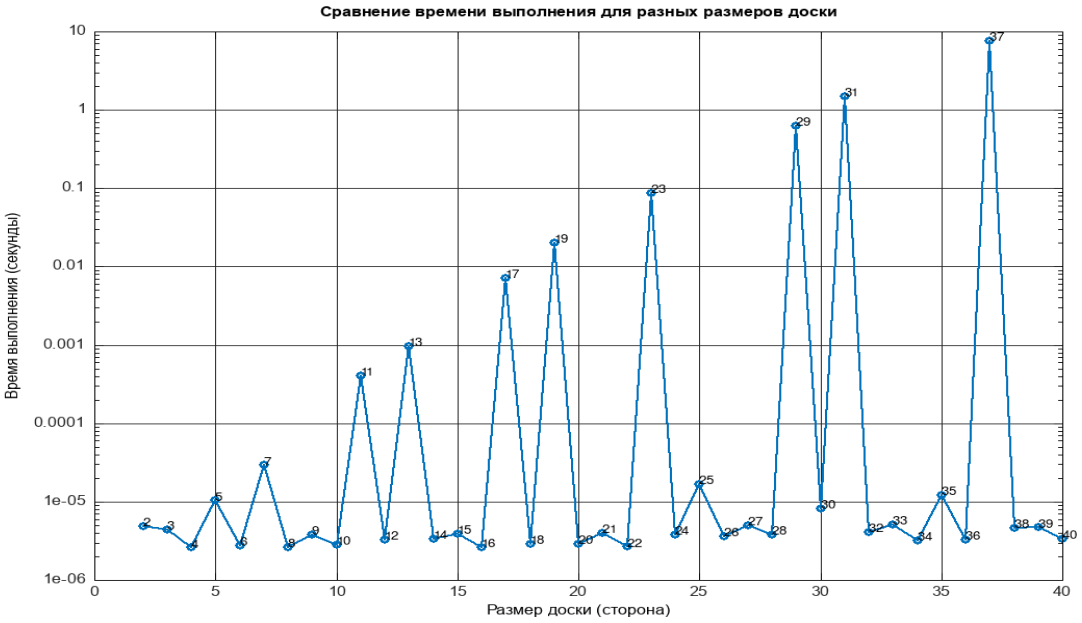


Рисунок 6 — зависимость времени выполнения алгоритма от размера доски (по логарифмической шкале)

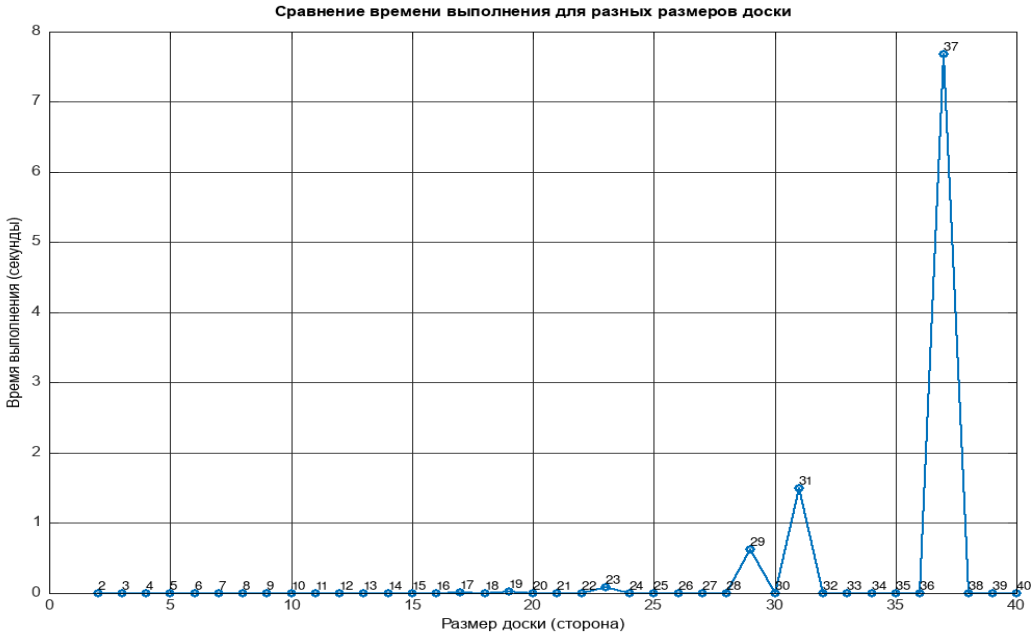


Рисунок 7 — зависимость времени выполнения алгоритма от размера доски (по линейной шкале)

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 <exception>
#include <functional>
#include <iostream>
#include <map>
#include <string>

#include "../include/Benchmark.hpp"
#include "../include/Board.hpp"
#include "../include/Deserializer.hpp"
#include "../include/JsonHandler.hpp"
#include "../include/Square.hpp"
#include "../include/Tiling.hpp"
#include "../include/Visualizer.hpp"

#define _MIN_SIZE 2
#define _MAX_SIZE 40

#define _JSON_FILE "benchmark/benchmark.json"
#define _ITERATIONS 3

#define _VISUAL_DIR "visual-output/"

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

  if (n < _MIN_SIZE || n > _MAX_SIZE) {
      std::cerr << "Invalid input size (" << _MIN_SIZE << "-" <<
_MAX_SIZE << ")!"
              << std::endl;
    return;
  }

  Board board = tiling::solve(n, false).first;

  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;
  }
}

void benchmarkMode() {
  Benchmark benchmark(_ITERATIONS);
  benchmark.run(_MIN_SIZE, _MAX_SIZE);
  benchmark.save(_JSON_FILE);

   std::cout << "Benchmarks completed and saved to " << _JSON_FILE <<
std::endl;
}

void visualizeMode() {
  nlohmann::json benchmark = nlohmann::json::object();
  try {
```

```
    JsonHandler handler(_JSON_FILE);
    benchmark = handler.read();
  } catch (const std::exception& e) {
    benchmarkMode();
    JsonHandler handler(_JSON_FILE);
    benchmark = handler.read();
  }

  std::vector<int> sizes;
  std::vector<double> times;

  for (const auto& result : benchmark["benchmarks"]) {
    Board board = deserialize::get_board(result["board"]);

    sizes.push_back(board.getSize());
    times.push_back(result["time"].get<double>());

    std::string title =
                        "Замощение квадрата со стороной: " +
std::to_string(board.getSize()) +
        "x" + std::to_string(board.getSize());
     std::string filename = "tiling_" + std::to_string(board.getSize())
+ ".png";

    std::string path =
                  visualize::visualizeTiling(board, title, filename,
_VISUAL_DIR);
    std::cout << "Generated: " << path << std::endl;
  }

  std::string path = visualize::visualizeExecutionTimes(
      sizes, times, "execution_times.png", _VISUAL_DIR);
    std::cout << "Generated execution time graph: " << path <<
std::endl;
}

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

  if (n < _MIN_SIZE || n > _MAX_SIZE) {
      std::cerr << "Invalid input size (" << _MIN_SIZE << "-" <<
_MAX_SIZE << ")!"
              << std::endl;
    return;
  }

  auto [board, steps] = tiling::solve(n, true);
  steps.push_back(board);

  std::string path = visualize::visualizeSteps(steps, _VISUAL_DIR);

  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;
  }
```

```
  std::cout << "Generated steps animation: " << path << std::endl;
}

int main(int argc, char* argv[]) {
  if (argc < 2) {
    return 1;
  }

  std::string mode = argv[1];
  const std::map<std::string, std::function<void()>> modes = {
      {"solve", solveMode},
      {"benchmark", benchmarkMode},
      {"visualize", visualizeMode},
      {"steps", stepsMode}};

  auto it = modes.find(mode);
  if (it != modes.end()) {
    it->second();
  }

  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 <utility>
#include <vector>

#include "Board.hpp"

namespace tiling {

std::pair<Board, std::vector<Board>> backtrack(Board startBoard, bool
record);
std::pair<Board, std::vector<Board>> solve(int size, bool record);

}  // namespace tiling
#endif  // TILING_HPP_
```

Файл *Benchmark.hpp*:

```
#include <string>
#include <vector>

#include "BenchmarkResult.hpp"

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

 public:
  Benchmark(int iterations);

  void run(size_t minSize, size_t maxSize);
  void save(const std::string& ouput) const;
```

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

Файл *JsonHandler.hpp*:

```
#ifndef JSONHANDLER_HPP_
#define JSONHANDLER_HPP_

#include <fstream>
#include <nlohmann/json.hpp>
#include <string>

class JsonHandler {
 private:
  std::fstream file;

 public:
  JsonHandler(const std::string& filename);

  void write(nlohmann::json& json);
  nlohmann::json read();

  ~JsonHandler();
};

#endif  // JSONHANDLER_HPP_
```

Файл *Serializer.hpp*:

```
#ifndef SERIALIZER_HPP_
#define SERIALIZER_HPP_

#include <nlohmann/json.hpp>

struct Square;
class Board;

namespace serialize {

nlohmann::json to_json(const Square& square);
nlohmann::json to_json(const Board& board);

}  // namespace serialize

#endif  // SERIALIZER_HPP_
```

Файл *Deserializer.hpp*:

```
#ifndef DESERIALIZER_HPP_
#define DESERIALIZER_HPP_

#include <nlohmann/json.hpp>

struct Square;
class Board;

namespace deserialize {

Square get_square(const nlohmann::json& square_json);
```

```
Board get_board(const nlohmann::json& board_json);

}  // namespace deserialize

#endif  // DESERIALIZER_HPP_
```

Файл *Visualizer.hpp*:

```
#ifndef VISUALIZER_HPP_
#define VISUALIZER_HPP_

#include <string>
#include <vector>

class Board;

namespace visualize {

std::string visualizeTiling(const Board& board, const std::string&
title,
                            const std::string& filename,
                            const std::string& output_dir,
                            const std::string& additional="");
std::string visualizeExecutionTimes(const std::vector<int>& sizes,
                                    const std::vector<double>& times,
                                    const std::string& filename,
                                    const std::string& output_dir);
std::string visualizeSteps(const std::vector<Board>& steps,
                           const std::string& output_dir);

}  // namespace visualize

#endif  // VISUALIZER_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 <set>
#include <stack>

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

namespace tiling {

namespace details {

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 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;
}

}  // namespace details
```

```
std::pair<Board, std::vector<Board>> backtrack(Board startBoard, bool
record) {
  int minimalCount = INT32_MAX;
  Board bestBoard(startBoard.getSize());

  std::stack<Board> stack;
  stack.push(startBoard);

  std::vector<Board> steps;

  while (!stack.empty()) {
    Board board = stack.top();
    stack.pop();
    if (record) steps.push_back(board);

    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);
        if (record) steps.push_back(newBoard);
      }
    }
    steps.pop_back();
  }

  return {bestBoard, steps};
}

std::pair<Board, std::vector<Board>> solve(int size, bool record) {
  std::set<int> factors = details::baseFactorize(size);
  int minimal = *factors.begin();
  int coef = size / minimal;

  Board startBoard = details::initPrimeBoard(minimal);
  auto [board, steps] = backtrack(startBoard, record);
  board.scale(coef);
```

```
  if (record) {
    for (Board& step : steps) {
      step.scale(coef);
    }
  }

  return {board, steps};
}

}  // namespace tiling
```

Файл *Benchmark.cpp*:

```
#include "../include/Benchmark.hpp"

#include <chrono>
#include <iostream>
#include <nlohmann/json.hpp>

#include "../include/JsonHandler.hpp"
#include "../include/Serializer.hpp"
#include "../include/Tiling.hpp"

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

void Benchmark::run(size_t minSize, size_t maxSize) {
  benchmarkResults.clear();

  for (size_t n = minSize; n < maxSize + 1; n++) {
         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 = tiling::solve(n, false).first;
      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(averageTime, result);

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

```
void Benchmark::save(const std::string& output) const {
  nlohmann::json results = nlohmann::json::object();
  nlohmann::json benchmarks = nlohmann::json::array();

  for (const auto& result : benchmarkResults) {
    nlohmann::json entry = nlohmann::json::object();
    entry["time"] = result.time;
    entry["board"] = serialize::to_json(result.board);
    benchmarks.push_back(entry);
  }
  results["benchmarks"] = benchmarks;
  results["iterations"] = iterations;

  JsonHandler handler(output);
  handler.write(results);

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

Файл *JsonHandler.cpp*:

```
#include "../include/JsonHandler.hpp"

JsonHandler::JsonHandler(const std::string& filename) {
  file.open(filename);
  if (!file.is_open()) {
    throw std::runtime_error("Couldn't open a json-file!");
  }
}

void    JsonHandler::write(nlohmann::json&    json)    {    file    <<
json.dump(2); }

nlohmann::json JsonHandler::read() {
  nlohmann::json json = nlohmann::json::object();
  file >> json;
  return json;
}

JsonHandler::~JsonHandler() {
  if (file.is_open()) {
    file.close();
  }
}
```

Файл *Serializer.cpp*:

```
#include "../include/Serializer.hpp"

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

namespace serialize {

  nlohmann::json to_json(const Square& square) {
    nlohmann::json square_json = nlohmann::json::object();
    nlohmann::json coordinates = nlohmann::json::array();

    coordinates[0] = square.x;
```

```
    coordinates[1] = square.y;

    square_json["size"] = square.size;
    square_json["coordinates"] = coordinates;

    return square_json;
  }

  nlohmann::json to_json(const Board& board) {
    nlohmann::json board_json = nlohmann::json::object();

    nlohmann::json squares_json = nlohmann::json::array();
    for (const auto& square : board.getSquares()) {
      nlohmann::json square_json = to_json(square);
      squares_json.push_back(square_json);
    }

    board_json["size"] = board.getSize();
    board_json["squares"] = squares_json;

    return board_json;
  }

} // namespace serialize
```

Файл *Deserializer.cpp*:

```
#include "../include/Deserializer.hpp"

#include <stdexcept>

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

namespace deserialize {

namespace detail {

void validate_field(const nlohmann::json& json, const std::string&
field_name,
                    const std::string& object_type) {
  if (!json.contains(field_name)) {
       throw std::runtime_error("Invalid " + object_type + " JSON:
missing '" +
                                field_name + "' field in '" + object_type
+ "'");
  }
}

}  // namespace detail

Square get_square(const nlohmann::json& square_json) {
  detail::validate_field(square_json, "coordinates", "square");
  detail::validate_field(square_json, "size", "square");

  int x = square_json["coordinates"][0];
  int y = square_json["coordinates"][1];
  int size = square_json["size"];

  return Square({x, y}, size);
```

```
}

Board get_board(const nlohmann::json& board_json) {
  detail::validate_field(board_json, "size", "board");
  detail::validate_field(board_json, "squares", "board");

  int size = board_json["size"];
  Board board(size);

  for (const auto& square_json : board_json["squares"]) {
    Square square = deserialize::get_square(square_json);
    board.addSquare(square);
  }

  return board;
}

}  // namespace deserialuze
```

Файл *Visualizer.cpp*:

```
#include "../include/Visualizer.hpp"

#include <matplot/matplot.h>

#include <array>
#include <filesystem>
#include <vector>

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

#define _TEMP_DIR "temp/"
#define _DELAY 100

namespace visualize {

namespace colors {

using color_t = std::array<float, 3>;

const float _alpha = 0.5f;
const float _thickness = 0.5f;

const color_t _red = {1.0f, 0.0f, 0.0f};
const color_t _orange = {1.0f, 0.65f, 0.0f};
const color_t _yellow = {1.0f, 1.0f, 0.0f};
const color_t _green = {0.0f, 1.0f, 0.0f};
const color_t _blue = {0.0f, 0.0f, 1.0f};
const color_t _indigo = {0.29f, 0.0f, 0.51f};
const color_t _violet = {0.93f, 0.51f, 0.93f};
const color_t _magenta = {1.0f, 0.0f, 1.0f};
const color_t _cyan = {0.0f, 1.0f, 1.0f};
const color_t _coral = {1.0f, 0.5f, 0.31f};
const color_t _lime = {0.75f, 1.0f, 0.0f};
const color_t _brown = {0.65f, 0.16f, 0.16f};
```

```
const color_t _black = {0.0f, 0.0f, 0.0f};
const color_t _white = {1.0f, 1.0f, 1.0f};

std::vector<color_t> _colors = {_red,  _orange, _yellow, _green,
                                _blue, _indigo, _violet, _magenta,
                                _cyan, _coral,  _lime,   _brown};

color_t _getTransparentColor(const color_t& foreground,
                             const color_t& background, float alpha) {
  return {foreground[0] * alpha + background[0] * (1 - alpha),
          foreground[1] * alpha + background[1] * (1 - alpha),
          foreground[2] * alpha + background[2] * (1 - alpha)};
}

}  // namespace colors

namespace details {

void _configureAxes(matplot::axes_handle ax, int width, int height) {
  ax->xlim({0.0, static_cast<double>(width)});
  ax->ylim({0.0, static_cast<double>(height)});

  auto x_ticks = matplot::linspace(0, width, width + 1);
  auto y_ticks = matplot::linspace(0, height, height + 1);

  ax->x_axis().tick_values(x_ticks);
  ax->y_axis().tick_values(y_ticks);
}

void _drawSquare(const Square& square, colors::color_t fillCollor,
                 colors::color_t borderColor, float thickness) {
  auto rectangle =
                matplot::rectangle(square.x,   square.y,   square.size,
square.size);
  rectangle->color(fillCollor).fill(true);

  auto border =
                matplot::rectangle(square.x,   square.y,   square.size,
square.size);
  border->color(borderColor).fill(false).line_width(thickness);
}

}  // namespace details

std::string  visualizeTiling(const  Board&  board,  const  std::string&
title,
                             const std::string& filename,
                             const std::string& output_dir,
                             const std::string& additional) {
  matplot::figure_handle figure = matplot::figure(true);
  figure->size(1200, 800);
  figure->position({0, 0, 1200, 800});
  figure->title(title);

  details::_configureAxes(figure->current_axes(), board.getSize(),
                          board.getSize());

  if (additional.length() > 0) {
    figure->current_axes()->title(additional);
```

```
  }

  std::vector<Square> squares = board.getSquares();
  for (size_t i = 0; i < squares.size(); ++i) {
    Square& square = squares[i];

    colors::color_t fillColor = colors::_getTransparentColor(
        colors::_colors[i % colors::_colors.size()], colors::_white,
        colors::_alpha);

    square.y = board.getSize() - square.y - square.size;
           details::_drawSquare(square,   fillColor,   colors::_black,
colors::_thickness);
    square.y = (board.getSize() - square.size) / 2;
  }

  std::string output_path = output_dir + filename;

  matplot::save(output_path, "png");
  return output_path;
}

std::string visualizeExecutionTimes(const std::vector<int>& sizes,
                                    const std::vector<double>& times,
                                    const std::string& filename,
                                    const std::string& output_dir) {
  matplot::figure_handle figure = matplot::figure(true);
  figure->size(1200, 1200);
  figure->position({0, 0, 1200, 1200});

  std::vector<double> sizes_double(sizes.begin(), sizes.end());

  auto p = matplot::semilogy(sizes_double, times, "o-");
  p->line_width(2);
  p->marker_size(8);

  for (size_t i = 0; i < sizes.size(); ++i) {
               matplot::text(sizes_double[i],    times[i]    *    1.1,
std::to_string(sizes[i]));
  }

  auto ax = figure->current_axes();
  ax->grid(true);
  ax->title("Сравнение времени выполнения для разных размеров доски");
  ax->x_axis().label("Размер доски (сторона)");
  ax->y_axis().label("Время выполнения (секунды)");

  std::string output_path = output_dir + filename;
  matplot::save(output_path, "png");

  return output_path;
}

std::string visualizeSteps(const std::vector<Board>& steps,
                           const std::string& output_dir) {
  std::filesystem::create_directory(_TEMP_DIR);

  std::vector<std::string> frame_paths;
```

```
     std::string boardSize = std::to_string(steps[steps.size() -
1].getSize());
  int bestCount = INT32_MAX;

  for (size_t i = 0; i < steps.size(); i++) {
    std::string filename = "frame_" +
                                                    std::string(10 -
std::to_string(i).length(), '0') +
                           std::to_string(i) + ".png";
    std::string title =
                         "step " + std::to_string(i) + "/" +
std::to_string(steps.size() - 1);

    Board step = steps[i];

    if (step.getEmptyCells() == 0) {
      if (step.getSquaresCount() <= bestCount) {
        bestCount = step.getSquaresCount();
      }
    }

    std::string additional =
        "current count: " + std::to_string(step.getSquaresCount()) + "
" +
        "current best: ";

    if (bestCount != INT32_MAX)
      additional += std::to_string(bestCount);
    else
      additional += "NS";

    std::string frame_path =
        visualizeTiling(step, title, filename, _TEMP_DIR, additional);
    frame_paths.push_back(frame_path);
  }

  int fps = 2;

  std::string ouput_filename = "steps_" + boardSize + "x" + boardSize;

    auto getPath = [](const std::string& dir, const std::string&
filename,
                      const std::string& extenton) -> std::string {
    return dir + filename + extenton;
  };

  std::string gifCommand = "ffmpeg -y -loglevel quiet -framerate " +
                           std::to_string(fps) + " -start_number 0 -i
" +
                           std::string(_TEMP_DIR) + "frame_%010d.png "
+
                                   getPath(output_dir, ouput_filename,
".gif");

  std::system(gifCommand.c_str());

  std::string mp4Command = "ffmpeg -y -loglevel quiet -i " +
                           getPath(output_dir, ouput_filename, ".gif")
+ " " +
```

```
                                        getPath(output_dir, ouput_filename,
".mp4");

  std::system(mp4Command.c_str());

  std::filesystem::remove_all(_TEMP_DIR);

  return getPath(output_dir, ouput_filename, "");
}

}  // namespace visualize
```