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

ОТЧЕТ

по лабораторной работе № 1
по дисциплине «Построение и Анализ Алгоритмов»
Тема: «Поиск с возвратом»

Студент гр. 3343 Коршков А.А.

Преподаватель Жангиров Т. Р.

Санкт-Петербург
2025

Цель работы

Изучить общий метод нахождения решения задачи — поиск с возвратом на примере задачи о разбиении столешницы размером $N*N$ на минимальное количество квадратов.

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.

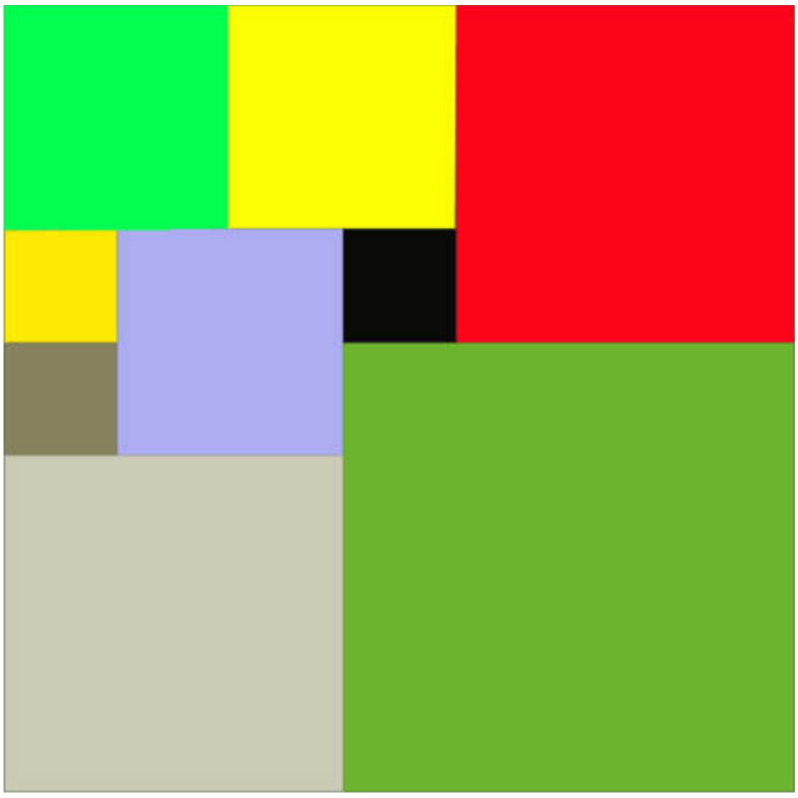

Рисунок 1 – Пример разбиения столешницы

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 ($2 \leq N \leq 20$).

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).

Индивидуализация для лабораторной работы (2и):

Итеративный бэктрекинг. Исследование времени выполнения от размера 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

Основные теоретические положения

Поиск с возвратом (backtracking) — это общий метод поиска решений задач, которые требуют полного перебора всех возможных вариантов из некоторого множества. Решение строится постепенно, начиная с частичного решения. Если на каком-то шаге расширить текущее решение не удается (например, оно не удовлетворяет ограничениям задачи), то происходит возврат к предыдущему шагу, и из более короткого частичного решения продолжается поиск.

Алгоритм поиска с возвратом позволяет найти всевозможные решения задачи, если они существуют, и является полезным в случаях, когда решение невозможно найти простым перебором всех вариантов сразу. Вместо этого, с помощью откатов, можно исключать неверные варианты и сосредотачиваться на тех, которые могут привести к решению.

Выполнение работы

Описание методов и структур данных

Для хранения частичных решений использовалась очередь (как список). В реализованном классе Board (файл Board.py) определены следующие методы:

__init__(self, size: int) -> None - конструктор класса. В качестве аргумента принимает число n, оно определяет длину стороны конструируемого стола.

size(self) -> int – возвращает размер доски

board(self) -> list[list[int]] – возвращает доску

square_list(self) -> list[list[int]] - возвращает список квадратов.

count_square(self) -> int - возвращает количество квадратов.

__deepcopy__(self, memodict: dict = None) -> 'Board' - создает копию объекта. (глубокое копирование при использовании функции deepcopy из библиотеки copy)

is_fill(self) -> bool - проверяет, заполнена ли доска.

get_empty_cell(self) -> tuple[int, int] - возвращает пустую ячейку.

check_possible_square(self, x: int, y: int, side: int) -> bool - проверяет, можно ли добавить квадрат на доску.

add_square(self, x: int, y: int, side: int) -> None - добавляет квадрат на доску.

place_squares_for_even_size(self) -> None - размещает квадраты на доске для четного размера.

place_squares_for_prime_size(self) -> None - размещает квадраты на доске для простого размера.

print_board(self) -> None – печатает доску

В файле backtracking.py находятся функции для выполнения итеративного бэктрекинга и оптимизации.

is_prime(n: int) -> bool - проверяет, является ли число простым.

get_divisors(n: int) -> tuple[int, int] - возвращает делители числа.

scale_board(board: Board, mult: int) -> Board - масштабирует доску.

backtracking_fill_board(board: Board) -> Board - возвращает доску с заполненными квадратами.

backtracking_algorithm(board: Board) -> list[list[int]] - реализует алгоритм поиска с возвратом.

silent_ backtracking(n: int) – заглушает промежуточные выводы функций, чтобы не влиять на подсчёт времени выполнения.

В файле main.py есть две функции:

main() -> None – основное задание

time_check() -> None – проверка на время в зависимости от размера квадрата

Применённые оптимизации

1) В случае, если размер стола N - простое число, заранее можно расставить один квадрат размером N // 2 + 1 и два смежных ему квадрата N // 2. Пример квадратов со сторонами 5,7:

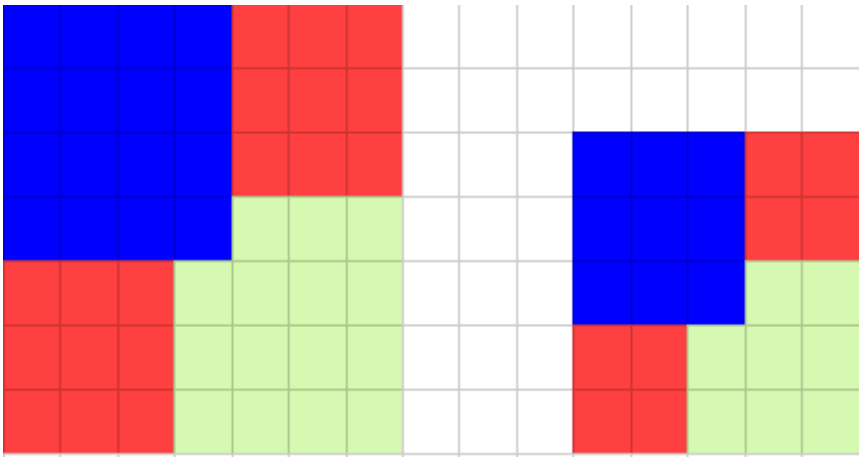

Рисунок 1 – Принцип размещения трёх квадратов 7 и 5

2) Новый квадрат всегда устанавливается в максимально верхнюю левую клетку, чтобы сократить кол-во одинаковых расстановок, но с разным порядком размещения квадратов.

3) Так как алгоритм при нахождении ответа в первую очередь расставляет самые большие квадраты, то расстановка, которую мы примем за итоговую будет найдена первой. Это позволяет нам не просчитывать все возможные варианты, а

прекратить выполнение алгоритма при нахождении первой расстановки, полностью заполняющей поле.

4) Если сторона квадрата чётное число, то минимальное разбиение всегда будет равно 4.

5) Если N - составное число, то число квадратов в оптимальном разбиении не превосходит аналогичного минимального для какого-либо из множителей числа. Нужно найти минимальный делитель числа, применить к нему алгоритм, затем умножить размер и координаты на оставшийся делитель для получения разбиения для размера N.

Оценка сложности алгоритма.

По времени:

В худшем случае алгоритм экспоненциальный (для простых чисел) O(n^{n^2}).

Для чётных чисел примерно O(1), потому что заранее известно кол-во квадратов и их положение, всё зависит от размера доски.

Для составного числа - O(T(small_div), т.к. время сводиться алгоритму над меньшим числом и к масштабированию, и зависит он от минимального делителя.

По памяти:

O(n^2) – т.к. необходимо хранить доску размером n * n.

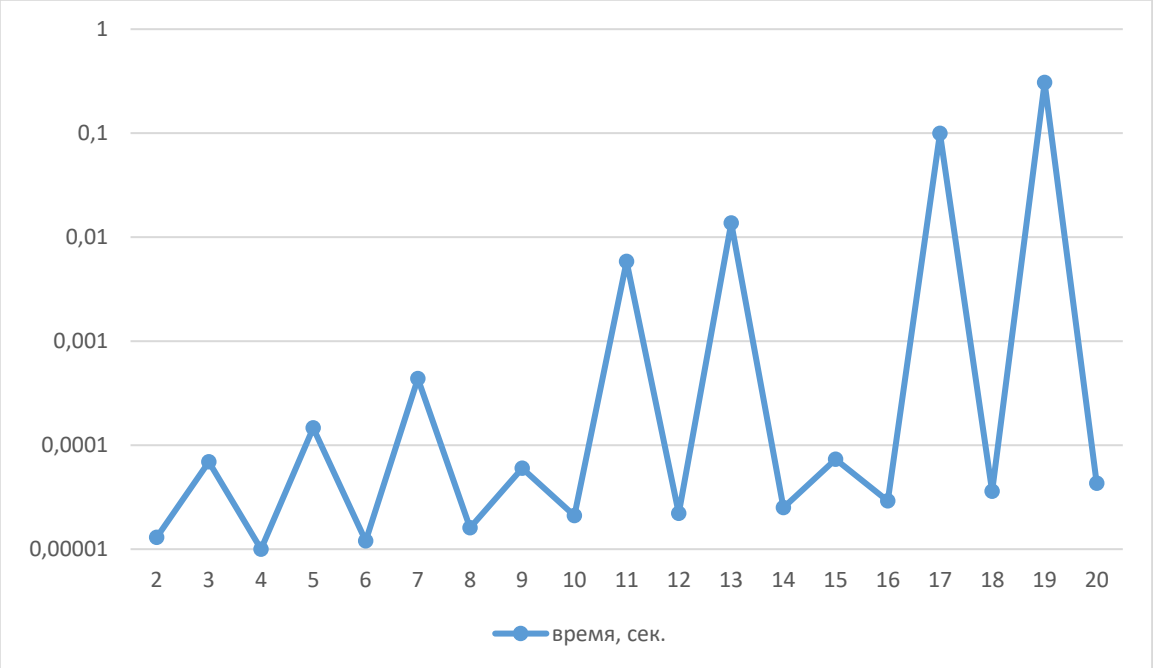

Рисунок 1 – график зависимости времени от размера квадрата (логарифмическая шкала)

По результату графика, можно сделать вывод, что время выполнения алгоритма сильно (экспоненциально) возрастает при возрастании минимального делителя стороны полотна (т.е. большее простое число).

Тестирование

Таблица 1 – Тестирование алгоритма

№ п/п	Входные данные	Выходные данные	Комментарии
1	4	4 1 1 2 3 1 2 1 3 2 3 3 2	Верно, оптимизация для чётных чисел.
2	7	9 1 1 4 1 5 3 5 1 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2	Верно, оптимизация для простых чисел.
3	11	11 1 1 6 1 7 5 7 1 5 6 7 3 6 10 2 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3	Верно, оптимизация для простых чисел.
4	15	6 1 1 10 1 11 5 11 1 5 6 11 5 11 6 5 11 11 5	Верно, оптимизация для составных чисел.
5	19	13 1 1 10 1 11 9 11 1 9 10 11 3 10 14 6 11 10 1 12 10 1 13 10 4 16 14 1 16 15 1 16 16 4 17 10 3	Верно, оптимизация для простых чисел.

		17 13 3	

Выводы

В ходе выполнения лабораторной работы было разработано решение задачи разбиения квадрата при помощи поиска с возвратом, а также проведено исследование зависимости времени работы алгоритма от размера квадрата.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
"""
Главный файл программы. Содержит функции для проверки времени выполнения алгоритма.

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


from modules.board import Board
from modules.backtracking import backtracking_algorithm, silent_backtracking

def time_check() -> None:
    """
    Эта функция проверяет время выполнения алгоритма поиска с возвратом.
    :return:
    """
    n_sizes: list[int] = list(range(2, 21))
    result_time: float = 0.0
    n: int
    for n in n_sizes:
        exec_time: float = timeit.timeit(lambda: silent_backtracking(n), number=1)
        result_time += exec_time
        print(f"Время выполнения для доски размером {n}*{n}:\t{exec_time:.6f} сек.".replace(".", ",", 1))
    print(f"\nОбщее время выполнения: {result_time:.6f} сек.")


def main() -> None:
    """
    Главная функция.
    :return:
    """
    # ввод размера доски
    while True:
        try:
            n: int = int(input())
            if not 2 <= n <= 60:
                print("Ошибка: размер доски должен быть натуральным целым числом в диапазоне от [2, 20].")
                continue
            break
        except ValueError:
            print("Ошибка: введено не целое натуральное число в диапазоне от [2, 20].")

    # алгоритм бэктрекинга вернёт список квадратов (x, y, w)
    result: list[list[int]] = backtracking_algorithm(Board(n))
    # преобразуем координаты (они от 1 до N)
```

```
        result: list[list[int]] = [[comp[0] + 1, comp[1] + 1, comp[2]]
for comp in result]
        print()
        # выведем кол-во квадратов
        print(len(result))
        # вывод квадратов
        for square in result:
            print(*square)


    if __name__ == "__main__":
        # главная функция
        main()

        # проверка времени выполнения алгоритма в зависимости от размера
квадратов
        # (раскомментировать при необходимости)
        # time_check()
```

Название файла: board.py

```
    """
    Этот модуль содержит класс Board, который представляет доску n*n
размером.
    """


    class Board:
        """
        Этот класс представляет доску n*n размером.
        """

        def __init__(self, size: int) -> None:
            """
            Эта функция инициализирует доску.
            :param size:
            """
            self.__size: int = size
            self.__board: list[list[int]] = [[0 for _ in range(size)] for
_ in range(size)]
            self.__square_list: list[list[int]] = []
            self.__count_square: int = 0

        @property
        def size(self) -> int:
            """
            Эта функция возвращает размер доски.
            :return:
            """
            return self.__size

        @property
        def board(self) -> list[list[int]]:
            """
            Эта функция возвращает доску.
            :return:
            """
            return self.__board
```

```
        @property
        def square_list(self) -> list[list[int]]:
            """
            Эта функция возвращает список квадратов.
            :return:
            """
            return self.__square_list

        @property
        def count_square(self) -> int:
            """
            Эта функция возвращает количество квадратов.
            :return:
            """
            return self.__count_square

        def __deepcopy__(self, memodict: dict = None) -> 'Board':
            """
            Эта функция создает копию объекта. (глубокое копирование)
            :param memodict:
            :return:
            """
            new_board: Board = Board(self.__size)
            new_board.__board: list[list[int]] = [row[:] for row in
self.__board]
            new_board.__square_list: list[list[int]] = [square[:] for
square in self.__square_list]
            new_board.__count_square: int = self.__count_square
            return new_board

        def is_fill(self) -> bool:
            """
            Эта функция проверяет, заполнена ли доска.
            :return:
            """
            row: list[int]
            for row in self.__board:
                if 0 in row:
                    return False
            return True

        def get_empty_cell(self) -> tuple[int, int]:
            """
            Эта функция возвращает пустую ячейку.
            :return:
            """
            for row in range(len(self.__board)):
                for col in range(len(self.__board[row])):
                    if self.__board[row][col] == 0:
                        return row, col
            return -1, -1

        def check_possible_square(self, x: int, y: int, side: int) ->
bool:
            """
            Эта функция проверяет, можно ли добавить квадрат на доску.
            :param x:
            :param y:
```

```
            :param side:
            :return:
            """
            if (x + side > self.__size) or (y + side > self.__size) or
side <= 0 or x < 0 or y < 0:
                return False
            for i in range(x, x + side):
                for j in range(y, y + side):
                    if self.__board[i][j] != 0:
                        return False
            return True

        def add_square(self, x: int, y: int, side: int) -> None:
            """
            Эта функция добавляет квадрат на доску.
            :param x:
            :param y:
            :param side:
            :return:
            """
            for i in range(x, x + side):
                for j in range(y, y + side):
                    self.__board[i][j] = self.__count_square + 1
            self.__square_list.append([x, y, side])
            self.__count_square += 1

        def place_squares_for_even_size(self) -> None:
            """
            Эта функция размещает квадраты на доске для четного размера.
            :return:
            """
            self.add_square(0, 0, self.__size // 2)
            self.add_square(self.__size // 2, 0, self.__size // 2)
            self.add_square(0, self.__size // 2, self.__size // 2)
            self.add_square(self.__size  //  2,  self.__size  //  2,
self.__size // 2)

        def place_squares_for_prime_size(self) -> None:
            """
            Эта функция размещает квадраты на доске для простого размера.
            :return:
            """
            self.add_square(0, 0, self.__size // 2 + 1,)
            self.add_square(0, self.__size // 2 + 1, self.__size // 2)
            self.add_square(self.__size // 2 + 1, 0, self.__size // 2)

        def render_board(self) -> None:
            """
            Эта функция отображает доску.
            :return:
            """
            row: list[int]
            for row in self.__board:
                print(*row, sep="\t")
```

Название файла: backtracking.py

```
"""
Этот модуль содержит функции для алгоритма поиска с возвратом.
```

```
    """
    from copy import deepcopy
    import sys
    import io

    from modules.board import Board


    def is_prime(n: int) -> bool:
        """
        Эта функция проверяет, является ли число простым.
        :param n:
        :return:
        """
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True


    def get_divisors(n: int) -> tuple[int, int]:
        """
        Эта функция возвращает делители числа.
        :param n:
        :return:
        """
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return i, n // i
        return 1, n


    def scale_board(board: Board, mult: int) -> Board:
        """
        Эта функция масштабирует доску.
        :param board:
        :param mult:
        :return:
        """
        print(f"Масштабируем доску {board.size}x{board.size} в {mult}
раз")
        new_board: Board = Board(board.size * mult)
        for square in board.square_list:
            new_board.add_square(square[0] * mult, square[1] * mult,
square[2] * mult)
        print(f"Новый размер доски: {new_board.size}x{new_board.size}")
        return new_board


    def backtracking_fill_board(board: Board) -> Board:
        """
        Эта функция возвращает доску с заполненными квадратами.
        :param board:
        :return:
        """
        iter_queue: list[Board] = [board]
```

```
        count_steps: int = 0
        while iter_queue:
            count_steps += 1
            current_board: Board = iter_queue.pop(0)
            print(f"\nШаг {count_steps}. Текущая доска:")
            current_board.render_board()
            if current_board.is_fill():
                print(f"Найдено полное заполнение за {count_steps} ша-
гов!")
                return current_board
            empty_x, empty_y = current_board.get_empty_cell()
            print(f"Пытаемся заполнить ячейку ({empty_x}, {empty_y})")
            for i in range(current_board.size, 0, -1):
                if current_board.check_possible_square(empty_x, empty_y,
i):
                    print(f"Пробуем квадрат размером {i}x{i} в позиции
({empty_x}, {empty_y})")
                    new_board: Board = deepcopy(current_board)
                    new_board.add_square(empty_x, empty_y, i)
                    new_board.render_board()
                    if new_board.is_fill():
                        print(f"Полное заполнение достигнуто на шаге
{count_steps}!")
                        return new_board
                    iter_queue.append(new_board)
        return board

    def backtracking_algorithm(board: Board) -> list[list[int]]:
        """
        Эта функция реализует алгоритм поиска с возвратом.
        :param board:
        :return:
        """
        if board.size % 2 == 0:
            print("Чётный размер - расставляем 4 квадрата")
            board.place_squares_for_even_size()
            board.render_board()
            return board.square_list
        if is_prime(board.size):
            print(f"Простой размер {board.size} - расставляем 3 квадрата")
            board.place_squares_for_prime_size()
            print("\nЗапуск поиска с возвратом для заполнения оставшихся
клеток")
            board = backtracking_fill_board(board)
            board.render_board()
            return board.square_list
        small_div, big_div = get_divisors(board.size)
        print(f"Составной размер {board.size} = {small_div} * {big_div}")
        small_board: Board = Board(small_div)
        if is_prime(small_div):
            print(f"Внутренний размер {small_div} простой - расставляем
3 квадрата")
            small_board.place_squares_for_prime_size()
        small_board: Board = backtracking_fill_board(small_board)
        print(f"\nМасштабирование результата в {big_div} раз")
        board = scale_board(small_board, big_div)
        board.render_board()
        return board.square_list
```

```
def silent_backtracking(n: int) -> None:
    """
    Обертка для подавления вывода алгоритма.
    :param n:
    :return:
    """
    original_stdout: sys = sys.stdout
    sys.stdout = io.StringIO()  # Перенаправляем вывод в буфер
    backtracking_algorithm(Board(n))
    sys.stdout.close()
    sys.stdout = original_stdout  # Восстанавливаем вывод
```