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»
Вариант 3р

Студент гр. 3343 Жучков О.Д.

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.

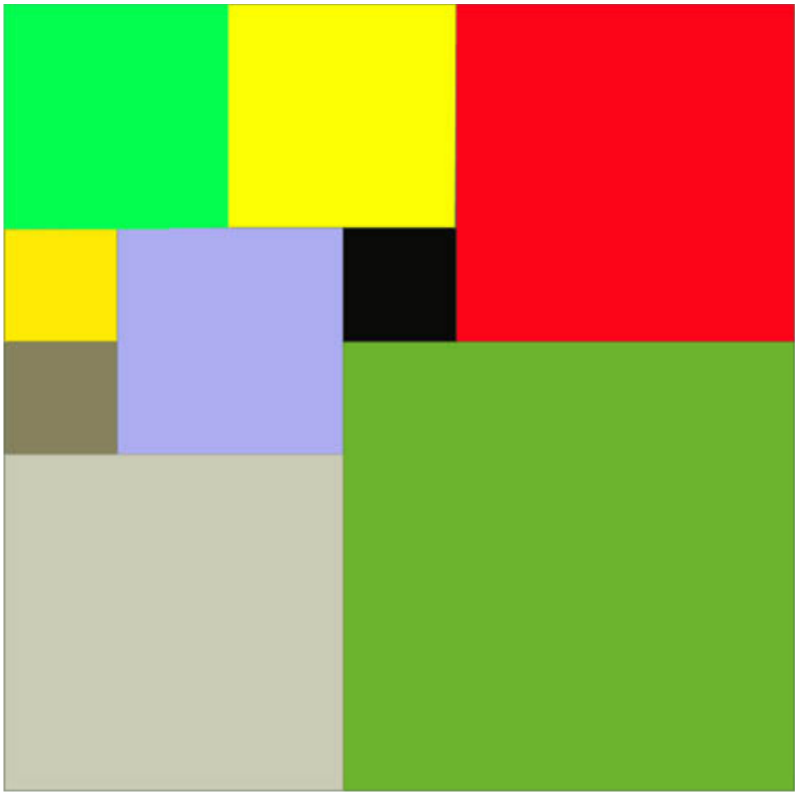

Рисунок 1 – Пример разбиения столешницы

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

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 ($1 \leq x, y \leq N$) 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

Вариант 3р

Рекурсивный бэктрекинг. Исследование кол-ва операций O(n) от размера квадрата.

Выполнение работы

Для решения задачи использован алгоритм рекурсивного бэктрекинга. Для каждой точки (x,y) проверяется, можно ли разместить там квадрат. Для каждой точки вычисляется максимальный размер квадрата, который можно разместить без перекрытия с уже размещенными квадратами. Рекурсивно вызывается функция backtrack для каждого возможного размера квадрата. Если текущее количество квадратов превышает лучшее найденное решение, ветка отсекается. Также на каждом шаге производится оценка нижней границы длины расстановки на основе не заполненной площади и верхней границы размера квадрата.

Для оптимизации решения задачи для составных чисел находится p – наибольший простой делитель. Тогда решение задачи сводится к решению для квадрата стороной n/p, полученное решение масштабируется в p раз.

Новый квадрат всегда устанавливается в максимально верхнюю левую клетку, размеры перебираются от большего к меньшему, чтобы сократить количество неэффективных расстановок.

Класс *Square* – представление одного квадрата в расстановке, имеет поля координат и размера.

Класс *Solution* – содержит информацию о лучшем найденном решении и лог работы алгоритма для вывода в графическом интерфейсе.

Функция *find_free_point* ищет ближайшую свободную точку для нового квадрата (сверху вниз, слева направо)

Функции *add_square* и *remove_square* добавляют и удаляют квадраты в square_map – массив nxn для проверки коллизий.

Функция *init_squares* ставит квадрат со стороной (n+1)/2 в точку (0, 0) и два квадрата размером n/2 в точки (0, (n+1)/2) и ((n+1)/2, 0).

Функция *greatest_divisor* находит наибольший делитель числа для масштабирования и оптимизации алгоритма.

Функция *run_algorithm* отвечает за запуск алгоритма и передачу результата в графический интерфейс

Графический интерфейс описан в interface.py классом TkWindow с использованием библиотеки tkinter. Пользователь может запустить алгоритм с введенным числом, увидеть графическое представление результата, пошагово смотреть работу алгоритма.

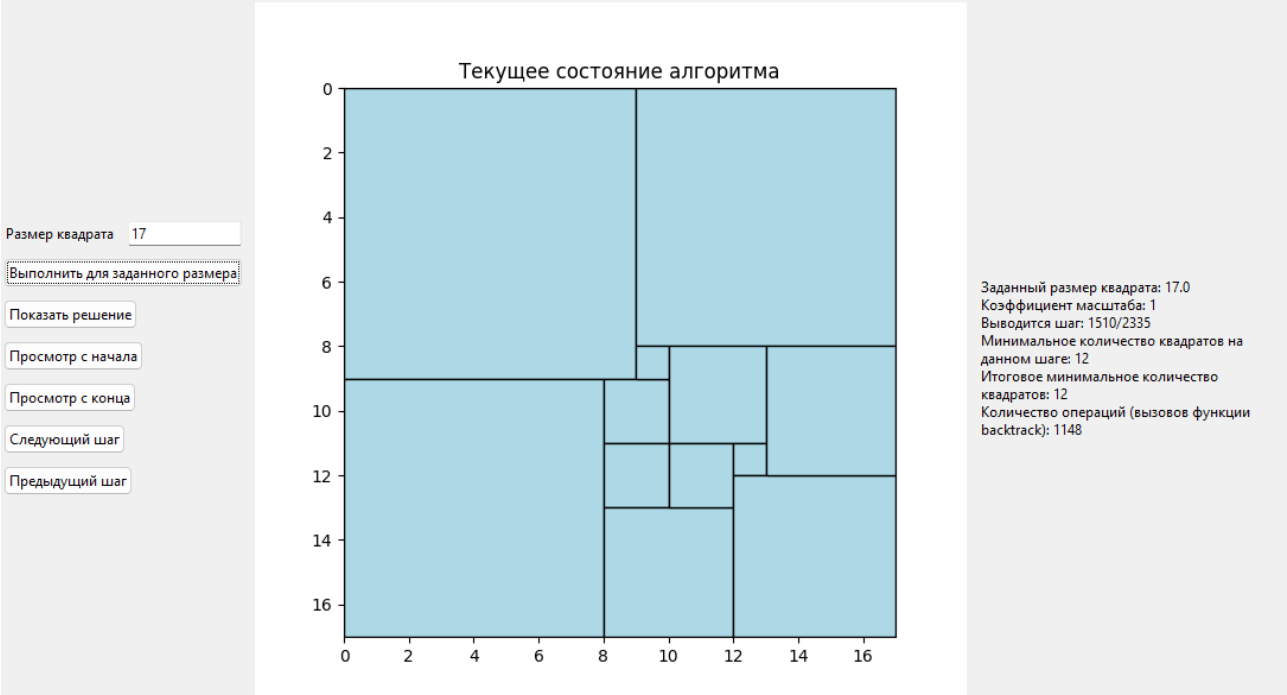

Рисунок 1 – Пример работы алгоритма

Оценка сложности алгоритма.

По времени:

В худшем случае для простого числа для каждой из n*n позиций перебирается n размеров квадрата, алгоритм экспоненциальный O(n^{n^2}).

Для составных чисел оценка аналогичная, но вместо n используется q = n / p – размер квадрата, для которого ищется масштабируемое решение.

По памяти:

O(n^2) – хранится доска размером n * n.

Исследование кол-ва операций от размера квадрата.

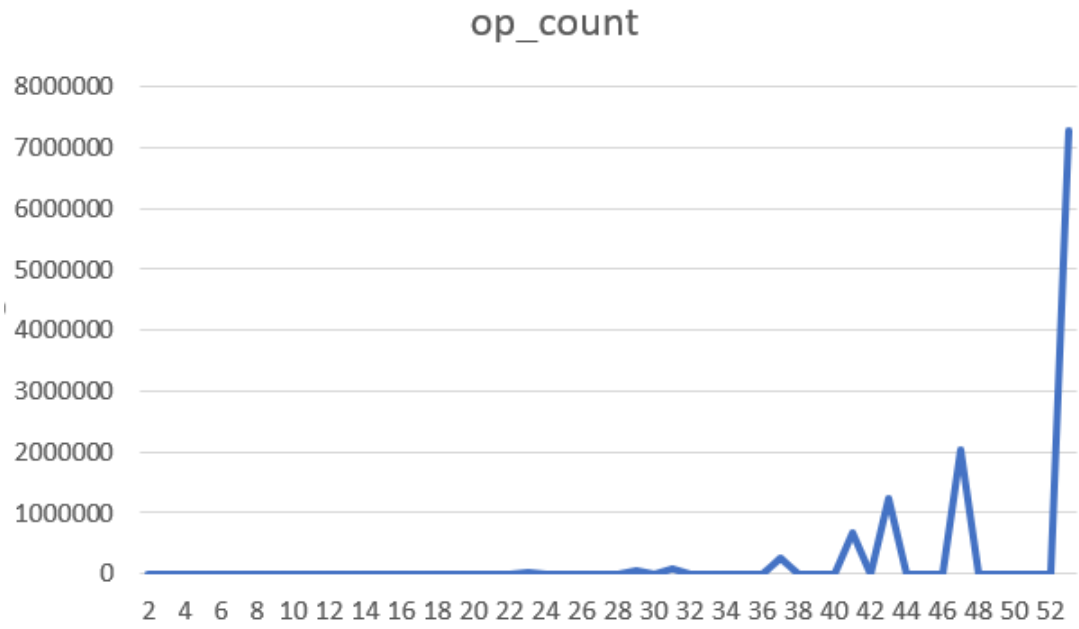

Рисунок 2 – зависимость количества операций от размера квадрата

Кол-во итераций экспоненциально возрастает с каждым следующим простым числом.

Тестирование

Таблица 1 – Тестирование алгоритма

№ п/п	Входные данные	Выходные данные	Комментарии
1	4	4 1 1 2 1 3 2 3 1 2 3 3 2	Верно, кол-во итераций равно единице из-за оптимизации
2	7	9 1 1 4 1 5 3 5 1 3 4 5 1 4 6 2 5 4 2 6 6 2 7 4 1 7 5 1	Верно
3	23	13 1 1 12 1 13 11 13 1 11 12 13 1 12 14 3 12 17 7 13 12 2 15 12 5 19 17 2 19 19 5 20 12 4 20 16 1 21 16 3	Верно

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. Оптимизация с использованием нижней оценки решения значительно влияет на количество операций и скорость выполнения алгоритма

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from modules.interface import TkWindow
from math import ceil


class Square:
    def __init__(self, x, y, size):
        self.x = x
        self.y = y
        self.right = x + size - 1
        self.bottom = y + size - 1
        self.size = size

    def __str__(self):
        return f"({self.x},{self.y}): {self.size}"


class Solution:
    def __init__(self, count, squares, scale, grid_size):
        self.count = count
        self.squares = squares
        self.operation_count = 0
        self.log = list()
        self.scale = scale
        self.grid_size = grid_size
        self.solution_index = -1

    def print(self, scale):
        print(self.count)
        print("\n".join([f"{square.x * scale + 1} {square.y * scale
+ 1} {square.size*scale}" for square in self.squares]))

    def add_log(self, squares):
        self.log.append((tuple((square.x, square.y, square.size)
for square in squares),
                         len(squares), self.count))


def find_free_point(square_map, size, x0, y0):
    for x in range(x0, size):
        for y in range(y0, size):
            if square_map[y][x] == 0:
                return x, y
        y0 = 0


def remove_square(square_map, square):
    for i in range(square.x, square.right+1):
        for j in range(square.y, square.bottom+1):
            square_map[j][i] = 0


def init_squares(squares, square_map, size):
    half = (size + 1) // 2
```

```
        small_half = size // 2
        squares.append(Square(0,0, half))
        squares.append(Square(0, half, small_half))
        squares.append(Square(half, 0, small_half))
        add_square(square_map, Square(0,0, half))
        add_square(square_map, Square(0, half, small_half))
        add_square(square_map, Square(half, 0, small_half))
        return half * half + small_half * small_half * 2


    def greatest_divisor(n):
        divisor = 1
        for i in range(1, n//2+1):
            if n % i == 0:
                divisor = i
        return divisor


    def backtrack(squares: list, square_map:list, count: int,
filled_area: int, x0: int, y0: int, size: int, best: Solution):
        best.operation_count += 1
        x, y = find_free_point(square_map, size, x0, y0)
        max_size = min(size - x, size - y)
        for i in range(y, size + 1):
            if i == size:
                break
            if square_map[i][x] == 1:
                break
        max_size = min(max_size, i - y)
        for n in range(max_size, 0, -1):
            remaining_area = size * size - filled_area - n*n
            #print(f"Попытка поставить квадрат на {x}, {y} размером
{n}; ", end="")
            if remaining_area > 0:
                max_possible_size = min(size - x, size - y)
                min_squares_needed = remaining_area / (max_possi-
ble_size * max_possible_size)
                lower_bound = ceil(count + 1 + min_squares_needed)
                #print(f"нижняя граница кол-ва квадратов для заполне-
ния: {lower_bound}; ", end="")
                if lower_bound > best.count:
                    #print(f"больше {best.count}, пропускаем")
                    continue

            new_square = Square(x, y, n)

            squares.append(new_square)
            add_square(square_map, new_square)
            if filled_area + n * n == size * size:
                #print("квадрат заполнен; ", end='')
                if count + 1 < best.count:
                    best.count = count + 1
                    best.squares = squares.copy()
                    best.solution_index = len(best.log)
                    #print(f"\nНовое решение ({count+1}):", ' |
'.join([str(square) for square in squares]))
                best.add_log(squares)
                #print("удаляем последний квадрат")
```

```
                squares.pop(-1)
                remove_square(square_map, new_square)
                best.add_log(squares)
                continue
            elif count + 1 < best.count:
                best.add_log(squares)
                #print()
                backtrack(squares, square_map, count + 1, filled_area +
n * n, x, y, size, best)
                #print(f"Удаляем квадрат {new_square}")
                squares.pop(-1)
                best.add_log(squares)
            #else:
                #print("превышен минимум, возвращаемся")
            remove_square(square_map, new_square)


    def add_square(square_map, square):
        for i in range(square.x, square.right+1):
            for j in range(square.y, square.bottom+1):
                square_map[j][i] = 1


    def run_algorithm(n):
        scale = greatest_divisor(n)
        grid_size = n // scale
        squares = list()
        square_map = [[0] * grid_size for i in range(grid_size)]
        filled = init_squares(squares, square_map, grid_size)
        solution = Solution(grid_size * grid_size + 1, [], scale,
grid_size)
        solution.add_log(squares)
        #print("Начало, заполняем угол:", ' | '.join([str(square) for
square in squares]))
        backtrack(squares, square_map, 3, filled, 0, 0, grid_size, so-
lution)
        solution.print(scale)
        #print("Backtrack function called:", solution.operation_count)
        return solution


    window = TkWindow({"run_alg": run_algorithm})
    window.mainloop()
```

Название файла: interface.py

```
    import tkinter as tk
    from tkinter import ttk
    from matplotlib.figure import Figure
    from matplotlib.patches import Rectangle
    from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
    from sys import exit


    class TkWindow(tk.Tk):
        def finish(self):
            self.destroy()
            exit()
```

```
        def __init__(self, functions):
            super().__init__()
            self.geometry("1100x600")
            self.resizable(False, False)
            self.solution = None
            self.current_step = -1
            self.params = dict()
            self.functions = functions
            self.finished = False
            frame = ttk.Frame(self)
            entry_frame = ttk.Frame(frame)
            ttk.Label(entry_frame, text="Размер квадрата").grid(padx=5,
pady=5, column=0, row=0, sticky=tk.W)
            param1 = ttk.Entry(entry_frame, width=15)
            param1.grid(padx=5, pady=5, column=1, row=0, sticky=tk.W)
            entry_frame.grid(column=0, row=0, sticky=tk.W)
            self.param_fields = {"size": param1}

            ttk.Button(frame, text="Выполнить для заданного размера",
command=self.execute_button).grid(padx=5, pady=5, column=0, row=1,
sticky=tk.W)
            ttk.Button(frame, text="Показать решение", com-
mand=self.show_solution).grid(padx=5, pady=5, column=0, row=2,
sticky=tk.W)
            ttk.Button(frame, text="Просмотр с начала", com-
mand=self.show_start).grid(padx=5, pady=5, column=0, row=3, sticky=tk.W)
            ttk.Button(frame, text="Просмотр с конца", com-
mand=self.show_end).grid(padx=5, pady=5, column=0, row=4, sticky=tk.W)
            ttk.Button(frame, text="Следующий шаг", com-
mand=self.step_forward).grid(padx=5, pady=5, column=0, row=5,
sticky=tk.W)
            ttk.Button(frame, text="Предыдущий шаг", com-
mand=self.step_back).grid(padx=5, pady=5, column=0, row=6, sticky=tk.W)

            frame.grid(column=0, row=0)

            plot_frame = ttk.Frame(self)
            plot_frame.grid(padx=5, pady=5, column=1, row=0)

            figure1 = Figure(figsize=(6,6))
            plot1 = figure1.add_subplot(1, 1, 1)
            plot1.set_title("Текущее состояние алгоритма")
            ax1 = figure1.gca()
            ax1.xaxis.get_major_locator().set_params(integer=True)
            ax1.yaxis.get_major_locator().set_params(integer=True)
            self.canvas_process = FigureCanvasTkAgg(figure1, mas-
ter=plot_frame)
            self.canvas_process.get_tk_widget().grid(row=0,column=0)

            self.info_text = ttk.Label(self, wraplength=250, text="")
            self.info_text.grid(row=0, column=2, padx=5, pady=5,
sticky=tk.NSEW)

            self.protocol("WM_DELETE_WINDOW", self.finish)

        def update_params(self):
            for param in self.param_fields:
```

```
                self.params[param] =
float(self.param_fields[param].get())

        def draw_solution(self, solution, size):
            ax = self.canvas_process.figure.gca()
            ax.cla()
            ax.set_title("Текущее состояние алгоритма")
            ax.xaxis.get_major_locator().set_params(integer=True)
            ax.yaxis.get_major_locator().set_params(integer=True)
            ax.set_ylim(size, 0)
            ax.set_xlim(0, size)
            for square in solution:
                rect = Rectangle((square[0], square[1]), square[2],
square[2], linewidth=1, edgecolor='black',
                                 facecolor='lightblue')
                ax.add_patch(rect)
            self.canvas_process.draw()

        def execute_button(self):
            self.finished = False
            self.update_params()
            self.solution = self.func-
tions["run_alg"](int(self.params["size"]))
            self.finished = True
            self.show_solution()

        def show_start(self):
            if not self.finished:
                return
            self.current_step = 0
            self.update_window()

        def show_end(self):
            if not self.finished:
                return
            self.current_step = len(self.solution.log)-1
            self.update_window()

        def step_forward(self):
            if not self.finished:
                return
            if self.current_step == len(self.solution.log) - 1 or
self.current_step == -1:
                return
            self.current_step += 1
            self.update_window()

        def step_back(self):
            if not self.finished:
                return
            if self.current_step == 0:
                return
            self.current_step -= 1
            self.update_window()

        def show_solution(self):
            if not self.finished:
                return
```

```
            self.current_step = self.solution.solution_index
            self.update_window()

        def update_window(self):
            info = self.solution.log[self.current_step]
            text = (f"Заданный размер квадрата:
{self.params["size"]}\n"
                    f"Коэффициент масштаба: {self.solution.scale}\n"
                    f"Выводится шаг: {self.cur-
rent_step+1}/{len(self.solution.log)}\n"
                    f"Минимальное количество квадратов на данном шаге:
{info[2]}\n"
                    f"Итоговое минимальное количество квадратов:
{self.solution.count}\n"
                    f"Количество операций (вызовов функции backtrack):
{self.solution.operation_count}")
            self.draw_solution(info[0], self.solution.grid_size)
            self.info_text.config(text=text)
            self.update()
```