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 _____ Никишин С.А,

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

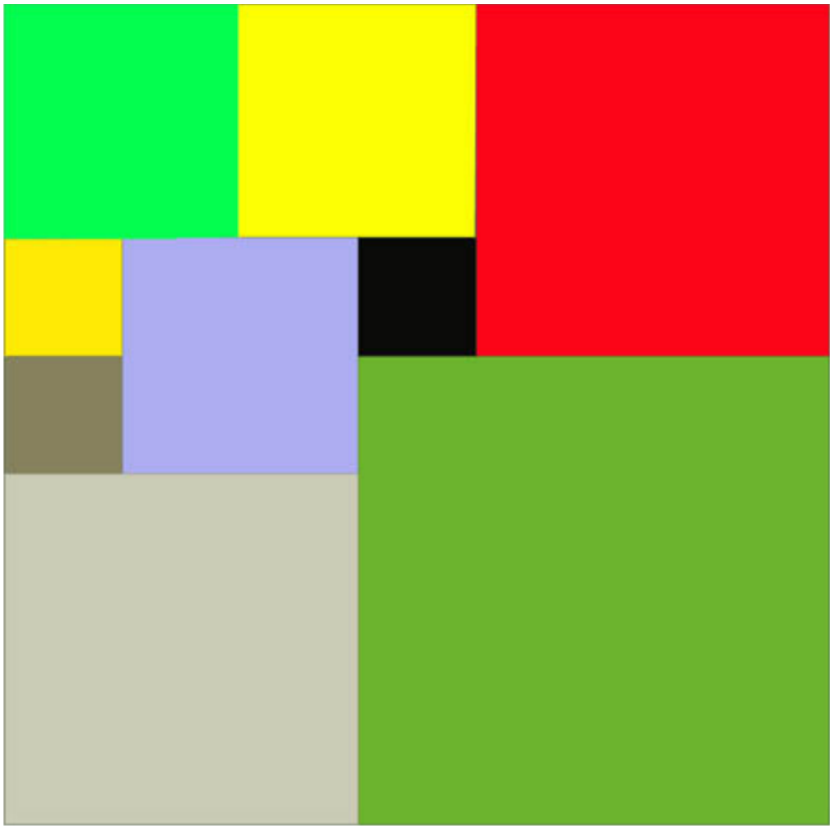
Цель работы.

Изучить алгоритм поиска с возвратом (бэктрекинг). Решить поставленную задачу при помощи данного алгоритма.

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≤N≤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

Вар. 3р. Рекурсивный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Описание алгоритма.

Общее описание

Единственной точкой входа программы является метод split_square(), который возвращает оптимальное разбиение квадрата на минимальное количество квадратов. Основной алгоритм программы — бэктрекинг (backtracking) с оптимизациями для специальных случаев. Алгоритм находит

первую свободную клетку в сетке, начиная с верхнего левого угла. Пытается разместить квадрат максимально возможного размера. Если квадрат помещается, заполняет соответствующую область в сетке и добавляет квадрат в текущее решение. Рекурсивно вызывает себя для оставшейся части сетки. После возврата из рекурсии убирает последний добавленный квадрат и пробует меньший размер. Когда вся сетка заполнена, сравнивает количество квадратов с лучшим найденным решением

Хранение частичных решений

- grid - матрица размером n×n для отслеживания занятых/свободных клеток
- current_squares - список объектов Square для хранения текущего частичного решения
- best_answer_squares - список с лучшим найденным решением
- best_answer - минимальное количество квадратов в оптимальном разбиении

Используемые методы оптимизации

- Для четных размеров: квадрат делится на 4 равные части размером n/2 × n/2
- Для простых нечетных чисел: используется специальная стратегия с одним большим квадратом (n/2 + 1) и двумя квадратами размером n/2
- Отсечение по границе: если текущее решение уже хуже лучшего, дальнейший перебор прекращается

Описание функций

- split_square() - Основная точка входа, которая выбирает стратегию разбиения в зависимости от свойств размера квадрата. Для четных размеров возвращает оптимальное разбиение на четыре части, для простых нечетных использует специализированный алгоритм, в остальных случаях применяет полный перебор.
- backtrack() - Реализует алгоритм поиска с возвратом, последовательно размещая квадраты максимального размера в первой свободной клетке и рекурсивно обрабатывая оставшуюся область. При полном заполнении сетки сравнивает полученное решение с текущим лучшим результатом и обновляет его при улучшении.
- can_place() - Проверяет возможность размещения квадрата заданного размера в указанной позиции, учитывая границы основного квадрата и занятость клеток. Возвращает истину только если вся целевая область свободна и находится within границ.
- place_square() - Заполняет или очищает область сетки, соответствующую размещаемому квадрату, устанавливая всем клеткам заданное значение. Используется как для добавления квадратов в процессе поиска, так и для отката изменений при backtracking.
- _prime_size_solution() - Реализует оптимизированную стратегию для простых нечетных размеров, размещая большой квадрат и два средних в угловых позициях. Оставшуюся область разбивает рекурсивно с соответствующим смещением координат.
- _backtracking_solution() - Служит оберткой для запуска алгоритма перебора, инициализируя поиск с пустым решением и возвращая лучший найденный результат после завершения работы backtracking.

Оценка сложности

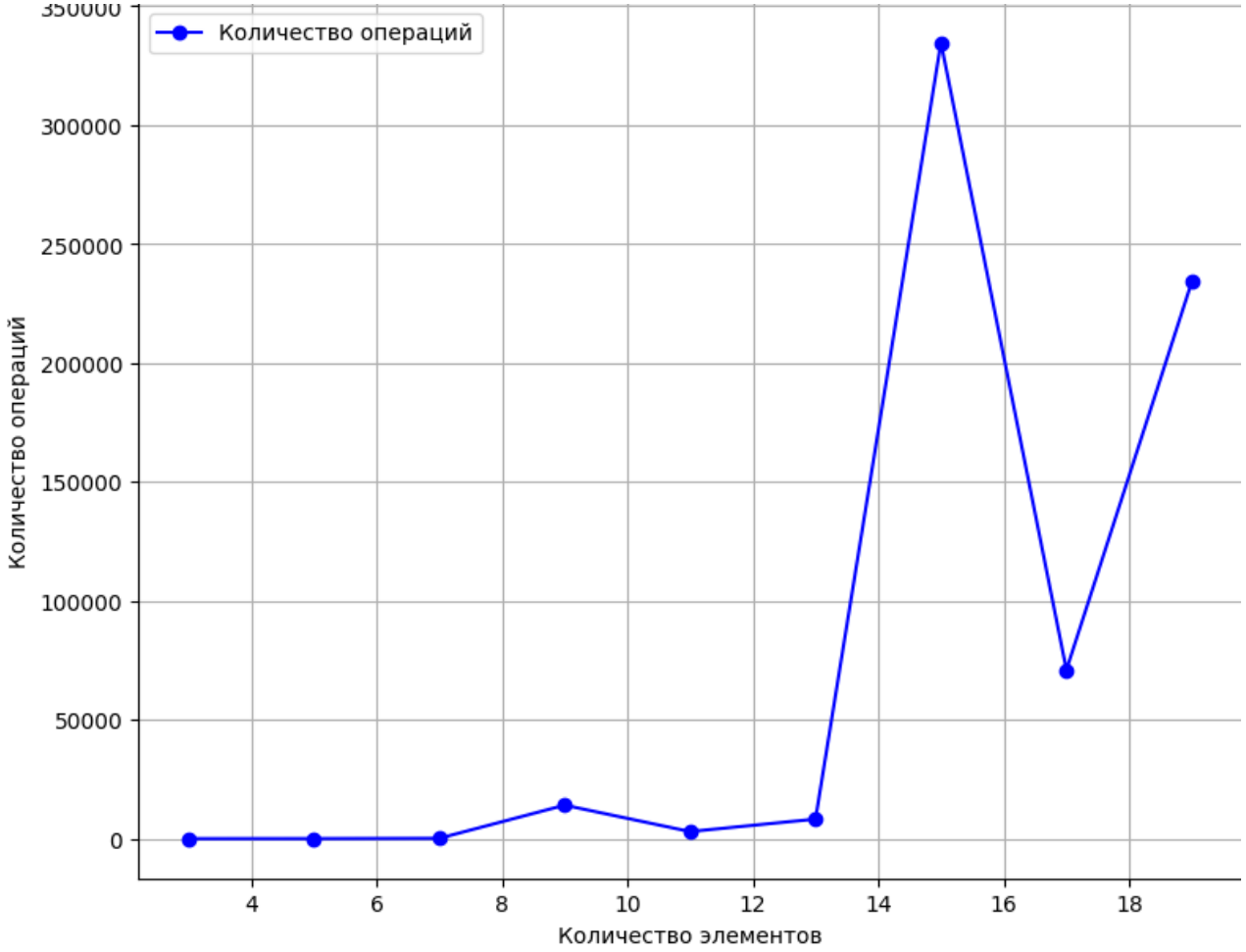
В худшем случае сложность алгоритма оценивается как O(n^(n²)), однако благодаря оптимизациям на практике время работы значительно сокращается. Для четных n: O(1). Для простых нечетных n рекурсивное разбиение уменьшает задачу. Отсечение неоптимальных ветвей сокращает пространство поиска.

Исследование.

Время выполнения и количество операций в зависимости от размера столешницы приведено в таблице.

Размер столешницы	Результат	Кол-во операций	Время, с.
3	6	3	± 0ms
5	8	26	± 0ms
7	9	137	± 0ms
9	6	14093	± 4ms
11	11	2985	± 2ms
13	11	8240	± 3ms
15	6	334700	± 22ms
17	12	70967	± 12ms
19	13	234223	± 19ms

График зависимости количества операций от размера столешницы:



Выводы.

В ходе работы успешно реализован алгоритм решения задачи квадрирования квадрата. Программа эффективно сочетает специализированные стратегии для частных случаев с общим алгоритмом backtracking. Экспериментально подтверждена работоспособность алгоритма для размеров до 19×19. Наибольшая эффективность достигается для размеров, допускающих регулярные разбиения.

ПРИЛОЖЕНИЯ

КОД ПРОГРАММЫ

```
import math
import tkinter as tk
from tkinter import ttk, messagebox, filedialog
from typing import List
from PIL import Image, ImageTk, ImageDraw
import time

class Square:
    def __init__(self, size: int, x: int, y: int):
        self.size = size
        self.x = x
        self.y = y

class SquareCutter:
    def __init__(self, size: int):
        self.size = size
        self.grid = [[0] * size for _ in range(size)]
        self.best_answer = float('inf')
        self.best_answer_squares: List[Square] = []
        self.operations_amount = 0

    @staticmethod
    def is_prime(num: int) -> bool:
        if num < 2:
            return False
        for i in range(2, int(math.sqrt(num)) + 1):
            if num % i == 0:
                return False
        return True

    def split_square(self) -> List[Square]:
        if self.size % 2 == 0:
            half_size = self.size // 2
            return [
                Square(half_size, 0, 0),
                Square(half_size, 0, half_size),
                Square(half_size, half_size, 0),
                Square(half_size, half_size, half_size),
            ]

        if self.is_prime(self.size) and self.size % 2 != 0:
            half_size = self.size // 2
            large_size = half_size + 1

            self.place_square(0, 0, large_size, 1)
            self.best_answer_squares.append(Square(large_size, 0, 0))

            self.place_square(large_size, 0, half_size, 1)
            self.best_answer_squares.append(Square(half_size, large_size, 0))

            self.place_square(0, large_size, half_size, 1)
            self.best_answer_squares.append(Square(half_size, 0, large_size))

            self.backtrack(self.best_answer_squares)
        else:
            self.backtrack([])
```

```
        return self.best_answer_squares

    def can_place(self, x: int, y: int, square_size: int) -> bool:
        if x + square_size > self.size or y + square_size > self.size:
            return False
        for i in range(y, y + square_size):
            for j in range(x, x + square_size):
                self.operations_amount += 1
                if self.grid[i][j] == 1:
                    return False
        return True

    def place_square(self, x: int, y: int, square_size: int, value: int) -> None:
        for i in range(y, y + square_size):
            for j in range(x, x + square_size):
                self.grid[i][j] = value

    def backtrack(self, current_squares: List[Square]) -> None:
        if len(current_squares) >= self.best_answer:
            return

        next_x, next_y = -1, -1
        for i in range(self.size):
            for j in range(self.size):
                if self.grid[i][j] == 0:
                    next_x, next_y = j, i
                    break
            if next_x != -1:
                break

        if next_x == -1:
            if len(current_squares) < self.best_answer:
                self.best_answer = len(current_squares)
                self.best_answer_squares = current_squares.copy()
            return

        max_size = min(self.size - 1, self.size - next_x, self.size - next_y)
        for size in range(max_size, 0, -1):
            if self.can_place(next_x, next_y, size):
                self.place_square(next_x, next_y, size, 1)
                current_squares.append(Square(size, next_x, next_y))
                self.backtrack(current_squares)
                current_squares.pop()
                self.place_square(next_x, next_y, size, 0)

class SquareCutterGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Square Cutter - Разбиение квадрата")
        self.root.geometry("1000x700")

        # Переменные
        self.size_var = tk.IntVar(value=5)
        self.result_squares = []

        self.setup_ui()

    def setup_ui(self):
        # Main frame
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
```

```
# Input section
input_frame = ttk.LabelFrame(main_frame, text="Параметры разбиения", padding="10")
input_frame.grid(row=0, column=0, columnspan=2, sticky=(tk.W, tk.E), pady=(0, 10))

ttk.Label(input_frame, text="Размер квадрата (n):").grid(row=0, column=0, sticky=tk.W)
size_entry = ttk.Entry(input_frame, textvariable=self.size_var, width=10)
size_entry.grid(row=0, column=1, sticky=tk.W, padx=(10, 0))

ttk.Button(input_frame, text="Выполнить разбиение",
        command=self.execute_cutting).grid(row=0, column=2, padx=(20, 0))

ttk.Button(input_frame, text="Сохранить результат",
        command=self.save_result).grid(row=0, column=3, padx=(10, 0))

ttk.Button(input_frame, text="Сохранить изображение",
        command=self.save_image).grid(row=0, column=4, padx=(10, 0))

# Results section
results_frame = ttk.LabelFrame(main_frame, text="Результаты", padding="10")
results_frame.grid(row=1, column=0, sticky=(tk.W, tk.E, tk.N, tk.S), pady=(0, 10))

# Text widget for results
self.results_text = tk.Text(results_frame, height=8, width=50)
results_scrollbar = ttk.Scrollbar(results_frame, orient="vertical", command=self.results_text.yview)
self.results_text.configure(yscrollcommand=results_scrollbar.set)

self.results_text.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
results_scrollbar.grid(row=0, column=1, sticky=(tk.N, tk.S))

# Visualization section
viz_frame = ttk.LabelFrame(main_frame, text="Визуализация", padding="10")
viz_frame.grid(row=1, column=1, rowspan=2, sticky=(tk.W, tk.E, tk.N, tk.S), padx=(10, 0))

# Canvas for visualization
self.canvas = tk.Canvas(viz_frame, width=400, height=400, bg='white')
self.canvas.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

# Info section
info_frame = ttk.LabelFrame(main_frame, text="Информация", padding="10")
info_frame.grid(row=2, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

self.info_text = tk.Text(info_frame, height=6, width=50)
info_scrollbar = ttk.Scrollbar(info_frame, orient="vertical", command=self.info_text.yview)
self.info_text.configure(yscrollcommand=info_scrollbar.set)

self.info_text.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
info_scrollbar.grid(row=0, column=1, sticky=(tk.N, tk.S))

# Configure grid weights
main_frame.columnconfigure(0, weight=1)
main_frame.columnconfigure(1, weight=1)
main_frame.rowconfigure(1, weight=1)
main_frame.rowconfigure(2, weight=1)

results_frame.columnconfigure(0, weight=1)
results_frame.rowconfigure(0, weight=1)

viz_frame.columnconfigure(0, weight=1)
viz_frame.rowconfigure(0, weight=1)

info_frame.columnconfigure(0, weight=1)
```

```
    info_frame.rowconfigure(0, weight=1)

def execute_cutting(self):
    try:
        size = self.size_var.get()
        if size < 1:
            messagebox.showerror("Ошибка", "Размер квадрата должен быть положительным числом")
            return
        if size > 20:
            if not messagebox.askyesno("Предупреждение",
                                       "Большие размеры могут работать медленно. Продолжить?"):
                return

        # Clear previous results
        self.results_text.delete(1.0, tk.END)
        self.info_text.delete(1.0, tk.END)
        self.canvas.delete("all")

        # Execute algorithm
        start_time = time.time()
        cutter = SquareCutter(size)
        self.result_squares = cutter.split_square()
        end_time = time.time()

        # Display results
        self.display_results(cutter, end_time - start_time)
        self.visualize_grid()

    except Exception as e:
        messagebox.showerror("Ошибка", f"Произошла ошибка: {str(e)}")

def display_results(self, cutter, execution_time):
    # Display results in text widget
    result_text = f"Размер квадрата: {cutter.size}\n"
    result_text += f"Количество квадратов: {len(self.result_squares)}\n"
    result_text += f"Время выполнения: {execution_time:.3f} секунд\n"
    result_text += f"Операций: {cutter.operations_amount}\n\n"
    result_text += "Координаты квадратов (x, y, размер):\n"

    for i, square in enumerate(self.result_squares):
        result_text += f"{i+1}: ({square.x + 1}, {square.y + 1}, {square.size})\n"

    self.results_text.insert(1.0, result_text)

    # Display info
    info_text = f"Алгоритм завершен успешно!\n"
    info_text += f"Оптимальное разбиение найдено.\n"
    info_text += f"Квадрат разбит на {len(self.result_squares)} частей.\n"

    if cutter.size % 2 == 0:
        info_text += "Использовано разбиение для четного размера.\n"
    elif cutter.is_prime(cutter.size):
        info_text += "Использовано разбиение для простого нечетного числа.\n"
    else:
        info_text += "Использован алгоритм backtracking.\n"

    self.info_text.insert(1.0, info_text)

def visualize_grid(self):
    if not self.result_squares:
        return
```

```
    size = self.size_var.get()
    cell_size = min(350 // size, 30)  # Adaptive cell size
    canvas_size = size * cell_size

    # Adjust canvas size
    self.canvas.config(width=canvas_size + 20, height=canvas_size + 20)

    colors = ['red', 'green', 'blue', 'yellow', 'magenta', 'cyan', 'orange',
              'purple', 'brown', 'pink', 'gray', 'darkgreen', 'navy']

    # Draw grid
    for i in range(size + 1):
        x = i * cell_size + 10
        self.canvas.create_line(x, 10, x, canvas_size + 10, fill='black')
        y = i * cell_size + 10
        self.canvas.create_line(10, y, canvas_size + 10, y, fill='black')

    # Draw squares
    for idx, square in enumerate(self.result_squares):
        color = colors[idx % len(colors)]

        x1 = square.x * cell_size + 10
        y1 = square.y * cell_size + 10
        x2 = (square.x + square.size) * cell_size + 10
        y2 = (square.y + square.size) * cell_size + 10

        # Draw filled square
        self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline='black', width=2)

        # Draw square number
        center_x = (x1 + x2) // 2
        center_y = (y1 + y2) // 2
        self.canvas.create_text(center_x, center_y, text=str(idx + 1),
                                font=('Arial', 10, 'bold'))

def save_result(self):
    if not self.result_squares:
        messagebox.showwarning("Предупреждение", "Нет данных для сохранения")
        return

    filename = filedialog.asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Text files", "*.txt"), ("All files", "*.*")],
        title="Сохранить результаты"
    )

    if filename:
        try:
            with open(filename, 'w', encoding='utf-8') as f:
                f.write(f"Разбиение квадрата размером {self.size_var.get()}\n")
                f.write(f"Количество квадратов: {len(self.result_squares)}\n\n")
                for i, square in enumerate(self.result_squares):
                    f.write(f"{square.x + 1} {square.y + 1} {square.size}\n")
            messagebox.showinfo("Успех", f"Результаты сохранены в {filename}")
        except Exception as e:
            messagebox.showerror("Ошибка", f"Не удалось сохранить файл: {str(e)}")

def save_image(self):
    if not self.result_squares:
        messagebox.showwarning("Предупреждение", "Нет данных для сохранения")
        return
```

```
        filename = filedialog.asksaveasfilename(
            defaultextension=".png",
            filetypes=[("PNG files", "*.png"), ("All files", "*.*")],
            title="Сохранить изображение"
        )

        if filename:
            try:
                self.create_png_image(filename)
                messagebox.showinfo("Успех", f"Изображение сохранено в {filename}")
            except Exception as e:
                messagebox.showerror("Ошибка", f"Не удалось сохранить изображение: {str(e)}")

    def create_png_image(self, filename: str):
        size = self.size_var.get()
        cell_size = 40
        image_size = size * cell_size

        img = Image.new('RGB', (image_size, image_size), 'white')
        draw = ImageDraw.Draw(img)

        colors = [
            (255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0),
            (255, 0, 255), (0, 255, 255), (255, 165, 0), (128, 0, 128),
            (165, 42, 42), (255, 192, 203), (128, 128, 128), (0, 128, 0), (0, 0, 128)
        ]

        # Draw grid
        for i in range(size + 1):
            draw.line([(i * cell_size, 0), (i * cell_size, image_size)], fill='black', width=2)
            draw.line([(0, i * cell_size), (image_size, i * cell_size)], fill='black', width=2)

        # Draw squares
        for idx, square in enumerate(self.result_squares):
            color = colors[idx % len(colors)]

            x1 = square.x * cell_size
            y1 = square.y * cell_size
            x2 = (square.x + square.size) * cell_size
            y2 = (square.y + square.size) * cell_size

            draw.rectangle([x1, y1, x2, y2], fill=color, outline='black', width=3)

            # Add number
            center_x = (x1 + x2) // 2
            center_y = (y1 + y2) // 2
            draw.text((center_x - 5, center_y - 8), str(idx + 1), fill='black',
                      font=ImageDraw.ImageFont.load_default())

        img.save(filename)

def main():
    root = tk.Tk()
    app = SquareCutterGUI(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

7 [redacted]