EXAM SEAT ALLOCATION SYSTEM

AN ACADEMIC MINI PROJECT REPORT

This document is an adapted version of an academic mini project report, shared for learning and documentation purposes.

Submitted by

MELVIN SABU

LINOJ KURIAN

AARSHA SURESH

ABEL MANOJ CHACKO

ANJO JOSE

2025

ABSTRACT

Creating seating arrangements for examinations or large classes, especially with students from multiple departments, years, and subjects, is a complex and time-consuming task when done manually. Manual methods are prone to errors, inconsistencies, and inefficiencies. This project proposes an **Automated Student Seating Arrangement System** designed to streamline this process. The system utilizes a graphical user interface (GUI) built with Python's Tkinter library, allowing administrators to easily input details about different branches (departments/years), student counts, subjects offered, student enrollment in subjects (supporting whole branch or specific ranges), and classroom specifications (name, capacity, rows, allowed subjects). The core logic employs a round-robin algorithm to assign students to available seats in designated classrooms, aiming for a balanced distribution of subjects within each room. The system generates a clear, printable seating arrangement report in HTML format, detailing the placement of each student in specific classrooms, rows, and columns. It also lists any students who could not be placed due to capacity constraints. This automated system significantly reduces manual effort, minimizes errors, ensures adherence to basic allocation rules, and provides an organized output, enhancing the efficiency of managing student seating logistics.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Educational institutions frequently need to organize seating arrangements for various purposes, most notably examinations, workshops, or large combined classes. Managing hundreds or thousands of students from different branches, years, and subject groups poses a significant logistical challenge. The traditional method of manually creating seating plans is laborious, time-consuming, and highly susceptible to human error. Errors might include assigning two students to the same seat, placing students in the wrong room, or failing to adequately separate students taking the same subject if required.

As the number of students and the complexity of constraints (multiple subjects per room, varying classroom capacities) increase, the manual process becomes exponentially more difficult. This often leads to delays, confusion on the day of the event, and potential compromises in examination integrity or classroom management.

Recognizing these challenges, there is a clear need for an automated solution. This project, the Automated Student Seating Arrangement System, leverages programming to address these issues. By providing a user-friendly interface for data input and employing a systematic algorithm for seat allocation, the system aims to automate the entire process. It takes into account various parameters like department, year, student enrollment per subject, classroom capacities, and subject restrictions per classroom. The system generates a structured and easily accessible seating plan, significantly improving efficiency, accuracy, and organization compared to manual methods. This report details the design, implementation, and features of this automated system.

OBJECTIVE

The primary objective of this project is to design and develop an efficient and user-friendly Automated Student Seating Arrangement System. The specific goals are:

Automation: To automate the process of assigning seats to students based on predefined inputs, eliminating the need for manual planning.

User-Friendly Interface: To provide a simple and intuitive Graphical User Interface (GUI) for administrators to input data regarding branches, students, subjects, and classrooms.

Flexibility: To handle students from multiple branches (departments/years) and accommodate various subjects within the same seating arrangement.

Classroom Management: To manage multiple classrooms with varying seating capacities, row structures, and subject allowances.

Efficient Allocation: To implement a systematic round-robin allocation algorithm that assigns students to appropriate classrooms based on allowed subjects and available seats.

Clear Output: To generate a well-formatted and easy-to-understand HTML report displaying the final seating arrangement for each classroom, including row and column placement.

Error Reduction: To minimize errors commonly associated with manual seating plan creation.

Efficiency: To significantly reduce the time and effort required to generate seating plans.

Reporting: To identify and report any students who could not be assigned a seat due to capacity limitations.

By achieving these objectives, the system aims to provide educational institutions with a reliable tool for managing student seating arrangements effectively.

CHAPTER 2

LITERATURE SURVEY

Algorithm For Efficient Seating Plan For Centralized Exam System

Author : Prosanta Kumar Chaki , Shikha Anirban

Proper seating arrangements in examination halls are essential for maintaining academic integrity and ensuring a fair environment for all students. Educational institutions face the dual challenge of using available space efficiently while preventing cheating, such as copying or using unauthorized materials. Traditionally, seat allocation has been done manually, following basic rules and experience. However, this method is time-consuming, often inefficient, and not suitable for large exams involving many students and multiple subjects. Common issues with manual planning include too many empty seats, students from the same exam sitting too close together, or mixing students from different exams in ways that create confusion and opportunities for cheating.

To improve exam logistics, researchers have developed computational tools, especially for automating exam timetables and assigning exams to rooms. For instance, studies by Kahar and Kendall, and Ayob and Malik, introduced models to assign exams to rooms, although these often rely on simplified conditions, like one exam per room. A project at Prince of Songkla University also offered useful insights. While these methods improve higher-level planning, they do not address how students are seated within each room — a key factor in reducing cheating and using space wisely.

This reveals a clear research gap. Few systems exist that generate detailed seating layouts tailored to prevent cheating while optimizing room use. This research aims to fill that gap by developing algorithms that create smart, automated seating plans for large-scale exams, ensuring both fairness and efficiency at the seat level.

CHAPTER 3

PROPOSED WORK

3.1. PROBLEM STATEMENT

Manually allocating seats for hundreds of students takes a lot of time and effort from administrators. Since the process is done by hand, it's easy to make mistakes—like assigning the same seat to more than one student, placing someone in the wrong room, or missing important rules and constraints. The task becomes even more complicated when handling students from different departments, academic years, and subjects, especially when classrooms have varying capacities and subject-specific limitations. Manual methods often lack a uniform system, which can result in unfair or inefficient seating arrangements. Additionally, it's challenging to clearly visualize how students are distributed across all rooms when relying on physical charts or basic spreadsheets. Making any last-minute changes to the plan can also be difficult and time-consuming, as each update might require reworking large portions of the layout. The proposed system aims to address these issues by providing an automated, efficient, and reliable solution.

3.2. PROPOSED METHODOLOGY

The system employs a combination of a graphical user interface for data management and a core algorithmic engine for seat allocation.

The system includes a user-friendly graphical interface built with Tkinter that allows structured data entry across multiple tabs. Users can enter details for branches, such as department name, academic year, and the number of students. Based on this information, the system automatically generates unique student IDs in a specific format. For subjects, users can specify the subject name and choose how to select the students—either by assigning an entire branch or selecting specific ranges of student IDs. Classroom information includes the name, total seating capacity, number of rows, and a list of subjects permitted in that room.

All of the input data is stored efficiently using Python data structures. Dictionaries are used to associate branch names and subject names with lists of student IDs, while classroom information is kept in a list of objects. Each classroom object contains its own attributes along with a seating grid represented as a NumPy array.

The core seat allocation logic is handled by a function called assign_students_round_robin, located in the main Python file. This function loops through each classroom and checks which allowed subjects have students to be placed. It then moves through the seating grid column by column. Within each column, students are placed in a round-robin fashion from the list of eligible subjects. For every subject, the function attempts to place the next student in the next available seat in that column. To keep the process seamless, it uses a global tracking system to remember which student was last placed for each subject, allowing it to continue smoothly across different classrooms and columns.

Once all classrooms have been processed, the program generates a report in the form of an HTML file, neatly organizing the seating arrangement for each classroom using tables and simple CSS styling. Each seat displays the student ID, or remains blank if unassigned. There's also a section listing students who couldn't be seated, grouped by their subject.

To enhance user experience, the interface displays confirmation messages after data is successfully entered and once seating has been allocated. It also opens the HTML report automatically in the default browser so users can review the results immediately. Additionally, the GUI includes a "Results" tab that provides a summary of all the input data for quick reference.

3.3. SYSTEM DESIGN

3.3.1. SYSTEM ARCHITECTURE

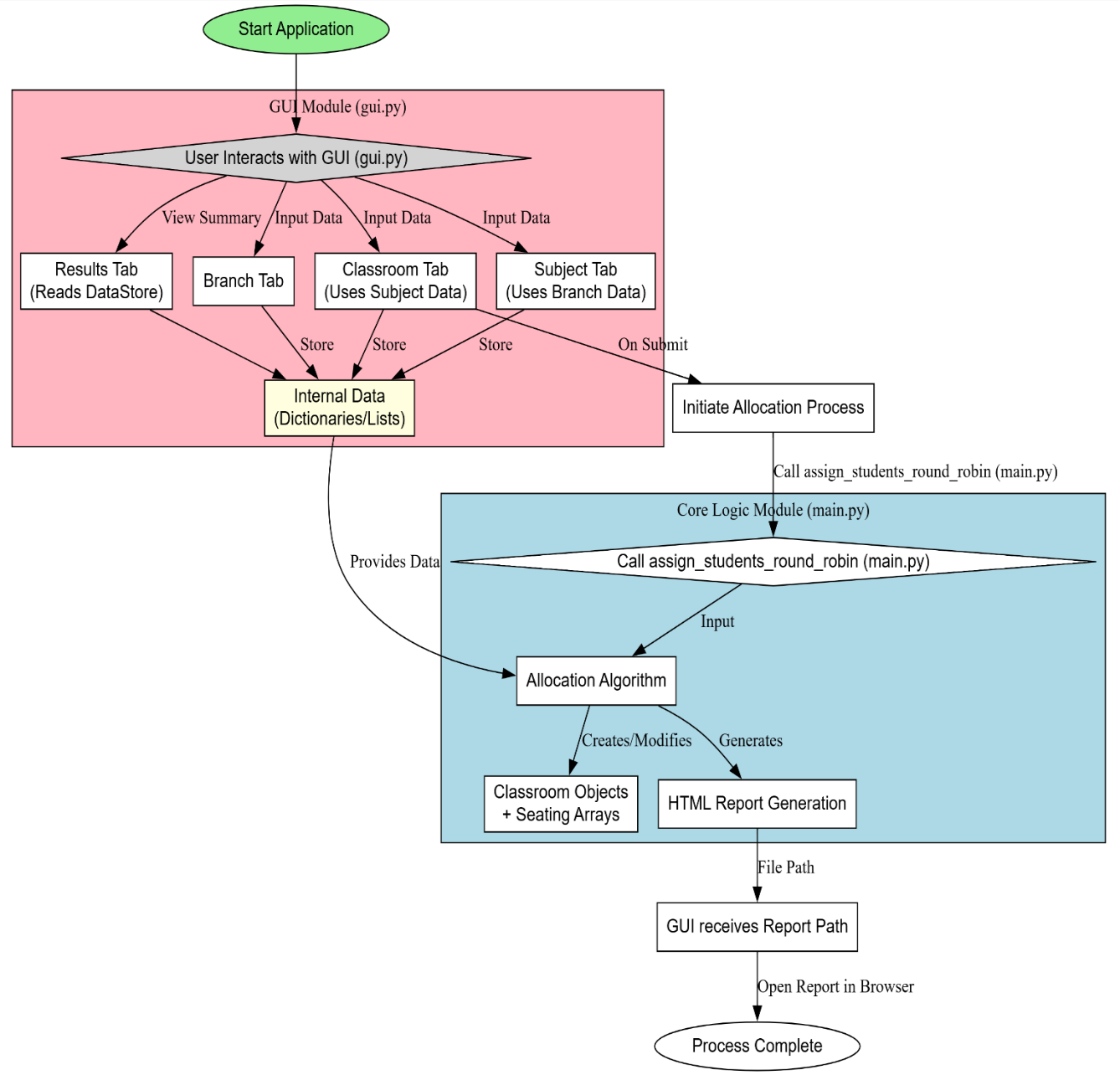


Fig 3.3.1. System Architecture of the Automated Seating Arrangement System

The system is designed with modularity, separating the user interface from the core processing logic.

1. GUI Module (gui.py):

* **Technology:** Python with Tkinter and ttk widgets for a modern look.

* **Structure:**

* StudentRegistrationGUI class encapsulates the entire interface.

* Uses ttk.Notebook to create a tabbed interface for different input stages: Branches, Subjects, Classrooms, and a Results summary tab.

* **Functionality:**

* **Branch Tab:** Takes the number of branches, then dynamically creates input fields (Department Name, Year, Student Count) for each. Validates input and stores branch data (generating student IDs) in the self. Branch dictionary upon submission.

* **Subject Tab:** Takes the number of subjects. For each subject, provides fields for name and selection type (Whole Branch or Dynamic). If "Whole Branch", a dropdown lists available branches. If "Dynamic", fields appear for selecting a branch and start/end student range numbers. Stores subject-to-student mappings in self. subjects dictionary.

* **Classroom Tab:** Takes the number of classrooms. For each classroom, provides fields for name, total seats, number of rows, and number of allowed subjects. Dynamically creates dropdowns to select the allowed subjects from the list defined previously. Stores Classroom objects (defined in main.py) in the self.classrooms list.

* **Results Tab:** Displays the processed data (branches, subjects, classrooms) in a text area for user review.

* **Interaction:** Buttons trigger actions like dynamic field creation (Set) and data processing/storage (Submit). Uses messagebox for user feedback (errors, success messages). Initiates the allocation process by calling assign_students_round_robin from main.py after classroom submission. Opens the resulting HTML file.

* **Data Handling:** Uses tk.StringVar, tk.IntVar to link GUI widgets with data variables.

2. Core Logic Module (main.py):

* **Classroom Class:**

* **Attributes:** name, seats, rows, subjects (list of allowed subject names), cols (calculated: (seats + rows - 1) // rows), seating_arrangement (initialized as an empty NumPy array of strings).

* **Purpose:** Represents a physical classroom and its properties, including the grid where students will be placed.

* **assign_students_round_robin Function:**

* **Inputs:** subjects (dict), classrooms (list of Classroom objects).

* **Processing:** Implements the round-robin allocation algorithm described in the methodology. Uses a global dictionary global_subject_indices to track the next student to be assigned for each subject across all

classrooms. Fills the seating_arrangement NumPy array within each Classroom object.

* **Output:** Creates the seating_arrangements directory if needed (using os module). Generates the seating_arrangements.html file with detailed tables for each classroom and a list of unplaced students.

* **Return Value:** Returns the path to the generated HTML file.

* **Dependencies:** NumPy for array manipulation, os for directory handling.

3. Data Structures:

* **self.branch (in GUI):** Dictionary {branch_name: [student_id_1, student_id_2, ...]}. Stores the generated student IDs for each branch.

* **self.subjects (in GUI):** Dictionary {subject_name: [student_id_1, student_id_2, ...]}. Stores the list of students enrolled in each subject.

* **self.classrooms (in GUI):** List [Classroom_obj1, Classroom_obj2, ...]. Stores instances of the Classroom class.

* **seating_arrangement (in Classroom):** NumPy array (rows, cols) storing student IDs or empty strings.

* **global_subject_indices (in assign_students_round_robin):** Dictionary {subject_name: index}. Tracks the index of the next student to assign for each subject.

4. Output Generation:

* Uses basic HTML string formatting within the assign_students_round_robin function.

* Generates <table>, <tr>, <th>, <td> tags to structure the seating plan.

* Includes embedded CSS (<style>) for basic table borders, alignment, padding, and alternating student ID colors for readability.

* Creates a separate for unplaced students, listing them by subject using <h3> and / tags.

3.4. HARDWARE REQUIREMENTS

The system is lightweight and does not require specialized hardware.

1. Processor: Standard Intel or AMD processor (e.g., Core i3 / Ryzen 3 or higher recommended for smoother GUI experience).
2. RAM: 4 GB or higher (8 GB recommended).
3. Storage: Minimal disk space required for Python installation, libraries, and the generated HTML report (typically less than 100 MB).
4. Display: Standard monitor.

3.5. SOFTWARE REQUIREMENTS

1. **Operating System:** Windows, macOS, or Linux (any OS that supports Python).
2. **Python:** Python 3.6 or higher.
3. **Python Libraries:**
    - **Tkinter:** Usually included with standard Python installations. Used for the GUI.
    - **NumPy:** Required for array manipulation in the core logic. Install using pip install numpy.
4. **Web Browser:** Any modern web browser (e.g., Chrome, Firefox, Edge, Safari) to view the generated HTML report.

CHAPTER 4

SOFTWARE TESTING

4.1. UNIT TESTING

Unit testing focuses on verifying individual components or functions of the code in isolation.

main.py - Classroom Class:

Test initialization: Verify attributes (name, seats, rows, subjects) are set correctly.

Test cols calculation: Ensure the number of columns is calculated accurately for different seat/row combinations (including edge cases like seats perfectly divisible by rows).

Test seating_arrangement initialization: Check if the NumPy array is created with the correct dimensions (rows, cols) and is initially empty (dtype=object).

main.py - assign_students_round_robin (Conceptual Units): While testing the whole function is more integration/system level, conceptually test parts:

Correct identification of available_subjects for a given classroom.

Accurate calculation of indices within the seating_arrangement array.

Correct updating of global_subject_indices.

Proper handling of unplaced students list.

gui.py - Input Validation:

Test validation for numeric inputs (number of branches, students, subjects, classrooms, seats, rows). Ensure non-numeric or non-positive inputs are handled (e.g., show error messages).

Test student ID generation logic: Verify IDs are generated in the expected format MBC<year><dept_code><roll_no>.

Test dynamic range validation in the Subjects tab: Ensure start/end ranges are numeric, valid (1 <= start <= end <= total_students), and within the bounds of the selected branch's student count.

gui.py - Data Handling Functions: Test functions responsible for collecting data from entry fields and storing it (submit_branches, add_dynamic_selection, submit_subjects, create_subject_dropdowns, submit_classrooms). Mock dependencies where necessary.

4.2. SYSTEM TESTING

System testing evaluates the integrated system as a whole, checking the interaction between the GUI and the core logic.

- End-to-End Workflow: Test the complete process: Launch GUI -> Input Branch Data -> Input Subject Data (Whole Branch) -> Input Subject Data (Dynamic) -> Input Classroom Data -> Submit Classrooms

-> Verify HTML report generation -> Verify report content.

- Data Consistency: Ensure data entered in one tab (e.g., Branches) is correctly reflected and usable in subsequent tabs (e.g., Branch dropdowns in Subjects tab, Subject dropdowns in Classrooms tab).
- Boundary Conditions:
    - Test with a small number of students/classrooms (e.g., 1 branch, 1 subject, 1 classroom).
    - Test with a large number of students/classrooms (approaching capacity limits).
    - Test with zero students in a branch.
    - Test scenarios where the number of students exactly matches or exceeds total classroom capacity.
- Error Handling: Test how the system handles invalid inputs across different stages (e.g., invalid numbers, non-existent branches selected, missing required fields). Verify appropriate error messages are displayed via messagebox.
- File System Interaction: Verify the seating_arrangements directory is created correctly and the HTML file is saved with the correct name and content. Test permissions issues (if applicable). Check behavior if the file is already open.

4.3. ALGORITHM TESTING

This focuses specifically on the correctness of the assign_students_round_robin allocation logic, often driven by specific test cases constructed manually or through system testing.

- **Correct Placement:** Verify students are placed *only* in classrooms where their subject is allowed.
- **Round-Robin Distribution:** Check if students of different subjects are interleaved column by column as expected. Manually trace the algorithm for a small, controlled dataset with multiple subjects and classrooms.
- **Capacity Limits:** Ensure no more students are placed in a classroom than its calculated capacity (rows * cols, though the actual limit is seats). Ensure the number of rows is respected.
- **Global Index Tracking:** Verify that the global_subject_indices correctly tracks the next student for each subject, ensuring continuity across columns and different classrooms.
- **Unplaced Students Report:** Test scenarios where students *should* remain unplaced (due to lack of seats or no suitable classroom) and verify they are correctly listed in the HTML report's "Unplaced Students" section, grouped by subject.
- **Edge Cases:** Test with subjects having zero students. Test with classrooms allowing zero subjects (should not place anyone). Test when a classroom is exactly filled. Test scenarios with many subjects and few rows/cols.

CHAPTER 5

SOURCE CODE

```python
import numpy as np
import os


class Classroom:  2 usages  linoj-k
    def __init__(self, name, seats, rows, subjects):  linoj-k
        self.name = name
        self.seats = seats
        self.rows = rows
        self.subjects = subjects
        self.cols = (seats + rows - 1) // rows
        self.seating_arrangement = np.full((rows, self.cols), '', dtype=object)


def assign_students_round_robin(subjects, classrooms):  2 usages  linoj-k
    # Create output directory if it doesn't exist
    output_dir = 'seating_arrangements'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    #keep a single subject_indices accross all classrooms
    global_subject_indices = {sub: 0 for sub in subjects}

    unplaced_students = {subject: [] for subject in subjects}
    html_content = []

    # Start the HTML document
    html_content.append("""
    <!DOCTYPE html>
    <html>
    <head>
        <title>Classroom Seating Arrangements</title>
        <style>
            body {
                font-family: Arial, sans-serif;
                margin: 20px;
                background-color: #f5f5f5;
            }
```

```
}
.classroom {
    margin-bottom: 30px;
    page-break-inside: avoid;
}
table {
    border-collapse: collapse;
    width: 100%;
    margin-bottom: 20px;
    background-color: white;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
}
.class-title {
    background-color: white;
    text-align: center;
    font-size: 24px;
    font-weight: bold;
    padding: 10px;
    border: 2px solid black;
    border-bottom: none;
}
th {
    background-color: #f2f2f2;
    text-align: center;
    font-weight: bold;
    padding: 10px;
    border: 2px solid black;
}
td {
    padding: 10px;
    border: 2px solid black;
    text-align: center;
    vertical-align: top;
}
.student {
    margin: 5px 0;
}
```

```
        }
        .student:nth-child(even) {
            color: #0000aa;
        }
        .student:nth-child(odd) {
            color: #8b4513;
        }
        .unplaced {
            margin-top: 30px;
        }
        h2 {
            color: #333;
        }
        @media print {
            body {
                background-color: white;
            }
            table {
                box-shadow: none;
            }
            .classroom {
                page-break-inside: avoid;
            }
        }
    </style>
</head>
<body>
<h1>Classroom Seating Arrangements</h1>
""")

# Process each classroom
for classroom in classrooms:
    available_subjects = [sub for sub in classroom.subjects if sub in subjects]
    #subject_indices = {sub: 0 for sub in available_subjects}

    col_index = 0
    # Fill the seating arrangement
```

```
    # Fill the seating arrangement
    while col_index < classroom.cols:
        for subject in available_subjects:
            if col_index >= classroom.cols:
                break

            students = subjects[subject]
            start_index = global_subject_indices[subject]

            row_index = 0
            while row_index < classroom.rows and start_index < len(students):
                classroom.seating_arrangement[row_index, col_index] = students[start_index]
                start_index += 1
                row_index += 1

            global_subject_indices[subject] = start_index
            col_index += 1

    # Create HTML for this classroom
    html_content.append(f'<div class="classroom">')
    html_content.append(f'<div class="class-title">{classroom.name}</div>')
    html_content.append('<table>')

    # Column headers
    html_content.append('<tr>')
    for col in range(classroom.cols):
        html_content.append(f'<th>column {col + 1}</th>')
    html_content.append('</tr>')

    # Rows with student IDs
    for row in range(classroom.rows):
        html_content.append('<tr>')
        for col in range(classroom.cols):
            student_id = classroom.seating_arrangement[row, col]
            if student_id:
                html_content.append(f'<td><div class="student">{student_id}</div></td>')
            else:
```

```
                else:
                    html_content.append('<td></td>')
            html_content.append('</tr>')

        html_content.append('</table>')
        html_content.append('</div>')


    # Track unplaced students - THIS SECTION MOVED HERE
    for subject in subjects:
        if subject in global_subject_indices:
            remaining = subjects[subject][global_subject_indices[subject]:]
            if remaining:
                unplaced_students[subject].extend(remaining)



    # Add unplaced students to HTML if any
    if any(students for students in unplaced_students.values()):
        html_content.append('<div class="unplaced">')
        html_content.append('<h2>Unplaced Students</h2>')

        for subject, students in unplaced_students.items():
            if students:
                html_content.append(f'<h3>{subject}</h3>')
                html_content.append('<ul>')
                for student in students:
                    html_content.append(f'<li>{student}</li>')
                html_content.append('</ul>')

        html_content.append('</div>')

    # Close HTML document
    html_content.append('</body></html>')

    # Write combined HTML file
    output_file = os.path.join(output_dir, 'seating_arrangements.html')
```

```
with open(output_file, 'w') as f:
    f.write('\n'.join(html_content))

print(f"HTML file has been created successfully in the '{output_dir}' folder!")
print(f"Open 'seating_arrangements.html' to view all classroom seating arrangements.")

return output_file
```

```python
from main import assign_students_round_robin, Classroom
import tkinter as tk
from tkinter import ttk, messagebox
import numpy as np


class StudentRegistrationGUI:  1 usage  linoj-k
    def __init__(self, root):  linoj-k
        self.root = root
        self.root.title("Student Registration System")
        self.root.geometry("800x600")

        # Main data storage
        self.branch = {}
        self.subjects = {}
        self.classrooms = []

        # Class variables that were causing "defined outside __init__" errors
        self.branch_entries = []
        self.subject_entries = []
        self.classroom_entries = []
        self.branch_entries_frame = None
        self.branch_submit_btn = None
        self.subject_entries_frame = None
        self.subject_submit_btn = None
        self.subject_notebook = None
        self.dynamic_frame = None
        self.classroom_entries_frame = None
        self.classroom_submit_btn = None
        self.classroom_notebook = None
        self.results_frame = None
        self.results_text = None

        # Create notebook for tabs
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(fill='both', expand=True, padx=10, pady=10)

        # Create tabs
```

```
        self.branch_tab = ttk.Frame(self.notebook)
        self.subjects_tab = ttk.Frame(self.notebook)
        self.classroom_tab = ttk.Frame(self.notebook)
        self.results_tab = ttk.Frame(self.notebook)

        self.notebook.add(self.branch_tab, text="Branches")
        self.notebook.add(self.subjects_tab, text="Subjects")
        self.notebook.add(self.classroom_tab, text="Classrooms")
        self.notebook.add(self.results_tab, text="Results")

        # Variable initialization
        self.num_branch_var = tk.StringVar()
        self.subjects_count_var = tk.StringVar()
        self.no_of_classes_var = tk.StringVar()

        # Initialize each tab
        self.setup_branch_tab()
        self.setup_subjects_tab()
        self.setup_classroom_tab()
        self.setup_results_tab()

    def setup_branch_tab(self):  1 usage  linoj-k
        frame = ttk.LabelFrame(self.branch_tab, text="Add Branches")
        frame.pack(fill="both", expand=True, padx=10, pady=10)

        # Number of branches
        ttk.Label(frame, text="Number of branches:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
        ttk.Entry(frame, textvariable=self.num_branch_var).grid(row=0, column=1, padx=5, pady=5)
        ttk.Button(frame, text="Set", command=self.create_branch_entries).grid(row=0, column=2, padx=5, pady=5)

        # Container for dynamic branch entries
        self.branch_entries_frame = ttk.Frame(frame)
        self.branch_entries_frame.grid(row=1, column=0, columnspan=3, padx=5, pady=5, sticky="nsew")

        # Submit button
        self.branch_submit_btn = ttk.Button(frame, text="Submit Branches", command=self.submit_branches)
```

```
        self.branch_submit_btn.grid(row=2, column=0, columnspan=3, padx=5, pady=5)
        self.branch_submit_btn.grid_remove()  # Hide initially

    def create_branch_entries(self):  1 usage  linoj-k
        # Clear any existing entries
        for widget in self.branch_entries_frame.winfo_children():
            widget.destroy()

        try:
            num_branches = int(self.num_branch_var.get())
            if num_branches <= 0:
                messagebox.showerror( title: "Invalid Input",  message: "Number of branches must be positive.")
                return

            self.branch_entries = []

            # Create headers
            ttk.Label(self.branch_entries_frame, text="Department Name").grid(row=0, column=0, padx=5, pady=5)
            ttk.Label(self.branch_entries_frame, text="Year").grid(row=0, column=1, padx=5, pady=5)
            ttk.Label(self.branch_entries_frame, text="Number of Students").grid(row=0, column=2, padx=5, pady=5)

            # Create entry fields for each branch
            for i in range(num_branches):
                branch_entry = {
                    "name": tk.StringVar(),
                    "year": tk.StringVar(),
                    "students": tk.StringVar()
                }

                ttk.Entry(self.branch_entries_frame, textvariable=branch_entry["name"]).grid(row=i + 1, column=0,
                                                                                              padx=5, pady=5)
                ttk.Entry(self.branch_entries_frame, textvariable=branch_entry["year"]).grid(row=i + 1, column=1,
                                                                                              padx=5, pady=5)
                ttk.Entry(self.branch_entries_frame, textvariable=branch_entry["students"]).grid(row=i + 1, column=2,
                                                                                                 padx=5, pady=5)
```

```
                self.branch_entries.append(branch_entry)

            # Show the submit button
            self.branch_submit_btn.grid()

        except ValueError:
            messagebox.showerror( title: "Invalid Input",  message: "Please enter a valid number.")

    def submit_branches(self):  1 usage  linoj-k
        # Process branch entries
        self.branch = {}
        for i, entry in enumerate(self.branch_entries):
            try:
```

```
            branch_name = entry["name"].get()
            year = int(entry["year"].get())
            no_of_students = int(entry["students"].get())

            # Map department codes according to original code
            dep = ""
            if branch_name == "CS":
                dep = "CS"
            elif branch_name == "EC":
                dep = "EC"
            elif branch_name == "CE":
                dep = "CE"
            elif branch_name == "ME":
                dep = "ME"
            elif branch_name == "EE":
                dep = "EE"
            else:
                messagebox.showerror( title: "Invalid Department",  message: f"Invalid department: {branch_name}")
                return

            # Generate student IDs
            self.branch[branch_name] = [f"MBC{year}{dep}{g + 1:02d}" for g in range(no_of_students)]

        except ValueError:
            messagebox.showerror( title: "Invalid Input",  message: f"Please enter valid numbers for branch {i + 1}")
            return

    # Update the results tab with branch information
    self.update_results()

    # Move to the next tab
    self.notebook.select(1)  # Select the Subject tab
    messagebox.showinfo( title: "Success",  message: "Branches added successfully!")

def setup_subjects_tab(self):  1 usage  linoj-k
```

```
    frame = ttk.LabelFrame(self.subjects_tab, text="Add Subjects")
    frame.pack(fill="both", expand=True, padx=10, pady=10)

    # Number of subjects
    ttk.Label(frame, text="Number of subjects:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
    ttk.Entry(frame, textvariable=self.subjects_count_var).grid(row=0, column=1, padx=5, pady=5)
    ttk.Button(frame, text="Set", command=self.create_subject_entries).grid(row=0, column=2, padx=5, pady=5)

    # Container for dynamic subject entries
    self.subject_entries_frame = ttk.Frame(frame)
    self.subject_entries_frame.grid(row=1, column=0, columnspan=3, padx=5, pady=5, sticky="nsew")

    # Submit button
    self.subject_submit_btn = ttk.Button(frame, text="Submit Subjects", command=self.submit_subjects)
    self.subject_submit_btn.grid(row=2, column=0, columnspan=3, padx=5, pady=5)
    self.subject_submit_btn.grid_remove()  # Hide initially
```

```
def create_subject_entries(self):  1 usage  linoj-k
    # Clear any existing entries
    for widget in self.subject_entries_frame.winfo_children():
        widget.destroy()

    try:
        subjects_count = int(self.subjects_count_var.get())
        if subjects_count <= 0:
            messagebox.showerror( title: "Invalid Input",  message: "Number of subjects must be positive.")
            return

        self.subject_entries = []

        # Create a notebook for each subject
        self.subject_notebook = ttk.Notebook(self.subject_entries_frame)
        self.subject_notebook.pack(fill='both', expand=True)

        for i in range(subjects_count):
            subject_frame = ttk.Frame(self.subject_notebook)
            self.subject_notebook.add(subject_frame, text=f"Subject {i + 1}")

            subject_entry = {
                "name": tk.StringVar(),
                "selection_type": tk.IntVar(value=1),
                "branch_selection": tk.StringVar(),
                "range_start": tk.StringVar(),
                "range_end": tk.StringVar(),
                "dynamic_selections": [],
                "dynamic_frame": None  # Store the frame reference here
            }

            # Subject name
            ttk.Label(subject_frame, text="Subject Name:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
            ttk.Entry(subject_frame, textvariable=subject_entry["name"]).grid(row=0, column=1, padx=5, pady=5,
                                                                              columnspan=2)

            # Selection type
```

```
            ttk.Label(subject_frame, text="Selection Type:").grid(row=1, column=0, padx=5, pady=5, sticky="w")

            # Using lambda with default arguments to avoid closure issues
            ttk.Radiobutton(subject_frame, text="Whole Branch", variable=subject_entry["selection_type"], value=1,
                            command=lambda s=subject_frame, e=subject_entry: self.toggle_selection_type(s, e,
                                                                                                        selection_type:
                row=1, column=1, padx=5, pady=5)
            ttk.Radiobutton(subject_frame, text="Dynamic Selection", variable=subject_entry["selection_type"],
                            value=2,
                            command=lambda s=subject_frame, e=subject_entry: self.toggle_selection_type(s, e,
                                                                                                        selection_type:
                row=1, column=2, padx=5, pady=5)

            # Branch selection (for whole branch)
            ttk.Label(subject_frame, text="Branch:").grid(row=2, column=0, padx=5, pady=5, sticky="w")
            branch_dropdown = ttk.Combobox(subject_frame, textvariable=subject_entry["branch_selection"])
            branch_dropdown.grid(row=2, column=1, padx=5, pady=5, columnspan=2)
            branch_dropdown['values'] = list(self.branch.keys())
```

```
            # Dynamic selection controls
            dynamic_frame = ttk.LabelFrame(subject_frame, text="Dynamic Selection")
            dynamic_frame.grid(row=3, column=0, columnspan=3, padx=5, pady=5, sticky="nsew")
            dynamic_frame.grid_remove()  # Hide initially

            subject_entry["dynamic_frame"] = dynamic_frame  # Store the reference

            ttk.Label(dynamic_frame, text="Branch:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
            dynamic_branch = ttk.Combobox(dynamic_frame, textvariable=subject_entry["branch_selection"])
            dynamic_branch.grid(row=0, column=1, padx=5, pady=5, columnspan=2)
            dynamic_branch['values'] = list(self.branch.keys())

            ttk.Label(dynamic_frame, text="Range Start:").grid(row=1, column=0, padx=5, pady=5, sticky="w")
            ttk.Entry(dynamic_frame, textvariable=subject_entry["range_start"]).grid(row=1, column=1, padx=5,
                                                                                     pady=5)

            ttk.Label(dynamic_frame, text="Range End:").grid(row=2, column=0, padx=5, pady=5, sticky="w")
            ttk.Entry(dynamic_frame, textvariable=subject_entry["range_end"]).grid(row=2, column=1, padx=5, pady=5)

            # Add dynamic selection button
            ttk.Button(dynamic_frame, text="Add Selection",
                       command=lambda e=subject_entry: self.add_dynamic_selection(e)).grid(row=3, column=0,
                                                                                           columnspan=3, padx=5,
                                                                                           pady=5)

            # Add to entries
            self.subject_entries.append(subject_entry)

        # Show the submit button
        self.subject_submit_btn.grid()

    except ValueError:
        messagebox.showerror( title: "Invalid Input",  message: "Please enter a valid number.")

def toggle_selection_type(self, subject_frame, subject_entry, selection_type):  2 usages  linoj-k
```

```
        # Use the stored reference instead of trying to find the widget
        dynamic_frame = subject_entry["dynamic_frame"]

        if selection_type == 1:  # Whole Branch
            dynamic_frame.grid_remove()
        else:  # Dynamic Selection
            dynamic_frame.grid()

    def add_dynamic_selection(self, subject_entry):  1 usage  linoj-k
        try:
            branch_name = subject_entry["branch_selection"].get()
            range_start = int(subject_entry["range_start"].get())
            range_end = int(subject_entry["range_end"].get())

            if branch_name not in self.branch:
                messagebox.showerror( title: "Invalid Branch",  message: f"Branch {branch_name} does not exist.")
                return

            # Validate range
            if range_start < 1 or range_end > len(self.branch[branch_name]) or range_start > range_end:
                messagebox.showerror( title: "Invalid Range",  message: "Please enter a valid range.")
                return
```

```
            # Add selection to the entry
            subject_entry["dynamic_selections"].append((branch_name, range_start, range_end))
            messagebox.showinfo( title: "Success",  message: f"Added students {range_start}-{range_end} from {branch_name}")

        except ValueError:
            messagebox.showerror( title: "Invalid Input",  message: "Please enter valid numbers for range.")

    def submit_subjects(self):  1 usage  linoj-k
        # Process subject entries
        self.subjects = {}
        for subject_entry in self.subject_entries:
            subject_name = subject_entry["name"].get()
            selection_type = subject_entry["selection_type"].get()

            if selection_type == 1:  # Whole Branch
                branch_name = subject_entry["branch_selection"].get()
                if branch_name in self.branch:
                    self.subjects[subject_name] = self.branch[branch_name].copy()
                else:
                    messagebox.showerror( title: "Invalid Branch",  message: f"Branch {branch_name} does not exist.")
                    return
            else:  # Dynamic Selection
                student_list = []
                for branch_name, range_start, range_end in subject_entry["dynamic_selections"]:
                    if branch_name in self.branch:
                        student_list.extend(self.branch[branch_name][range_start - 1:range_end])

                self.subjects[subject_name] = student_list

        # Update the results tab with subject information
        self.update_results()

        # Move to the next tab
        self.notebook.select(2)  # Select the Classroom tab
        messagebox.showinfo( title: "Success",  message: "Subjects added successfully!")
```

```
    def setup_classroom_tab(self):  1 usage  linoj-k
        frame = ttk.LabelFrame(self.classroom_tab, text="Add Classrooms")
        frame.pack(fill="both", expand=True, padx=10, pady=10)

        # Number of classrooms
        ttk.Label(frame, text="Number of classrooms:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
        ttk.Entry(frame, textvariable=self.no_of_classes_var).grid(row=0, column=1, padx=5, pady=5)
        ttk.Button(frame, text="Set", command=self.create_classroom_entries).grid(row=0, column=2, padx=5, pady=5)

        # Container for dynamic classroom entries
        self.classroom_entries_frame = ttk.Frame(frame)
        self.classroom_entries_frame.grid(row=1, column=0, columnspan=3, padx=5, pady=5, sticky="nsew")

        # Submit button
        self.classroom_submit_btn = ttk.Button(frame, text="Submit Classrooms", command=self.submit_classrooms)
        self.classroom_submit_btn.grid(row=2, column=0, columnspan=3, padx=5, pady=5)
        self.classroom_submit_btn.grid_remove()  # Hide initially
```

```
def create_classroom_entries(self):  1 usage  linoj-k
    # Clear any existing entries
    for widget in self.classroom_entries_frame.winfo_children():
        widget.destroy()

    try:
        no_of_classes = int(self.no_of_classes_var.get())
        if no_of_classes <= 0:
            messagebox.showerror( title: "Invalid Input",  message: "Number of classrooms must be positive.")
            return

        self.classroom_entries = []

        # Create a notebook for each classroom
        self.classroom_notebook = ttk.Notebook(self.classroom_entries_frame)
        self.classroom_notebook.pack(fill='both', expand=True)

        for i in range(no_of_classes):
            classroom_frame = ttk.Frame(self.classroom_notebook)
            self.classroom_notebook.add(classroom_frame, text=f"Classroom {i + 1}")

            classroom_entry = {
                "name": tk.StringVar(),
                "seats": tk.StringVar(),
                "rows": tk.StringVar(),
                "subject_count": tk.StringVar(),
                "subjects": [],
                "subject_frame": None
            }

            # Classroom details
            ttk.Label(classroom_frame, text="Classroom Name:").grid(row=0, column=0, padx=5, pady=5, sticky="W")
            ttk.Entry(classroom_frame, textvariable=classroom_entry["name"]).grid(row=0, column=1, padx=5, pady=5,
                                                                                  columnspan=2)

            ttk.Label(classroom_frame, text="Number of Seats:").grid(row=1, column=0, padx=5, pady=5, sticky="W")
            ttk.Entry(classroom_frame, textvariable=classroom_entry["seats"]).grid(row=1, column=1, padx=5, pady=5,
```

```
                                                                                  columnspan=2)

            ttk.Label(classroom_frame, text="Number of Rows:").grid(row=2, column=0, padx=5, pady=5, sticky="w")
            ttk.Entry(classroom_frame, textvariable=classroom_entry["rows"]).grid(row=2, column=1, padx=5, pady=5,
                                                                                  columnspan=2)

            # Subjects allowed
            ttk.Label(classroom_frame, text="Number of Subjects:").grid(row=3, column=0, padx=5, pady=5, sticky="w")
            ttk.Entry(classroom_frame, textvariable=classroom_entry["subject_count"]).grid(row=3, column=1, padx=5,
                                                                                           pady=5)

            # Use lambda with default argument to avoid closure issues
            ttk.Button(classroom_frame, text="Set",
                       command=lambda c_entry=classroom_entry: self.create_subject_dropdowns(c_entry)).grid(row=3,
                                                                                                            column=2,
                                                                                                            padx=5,
                                                                                                            pady=5)
```

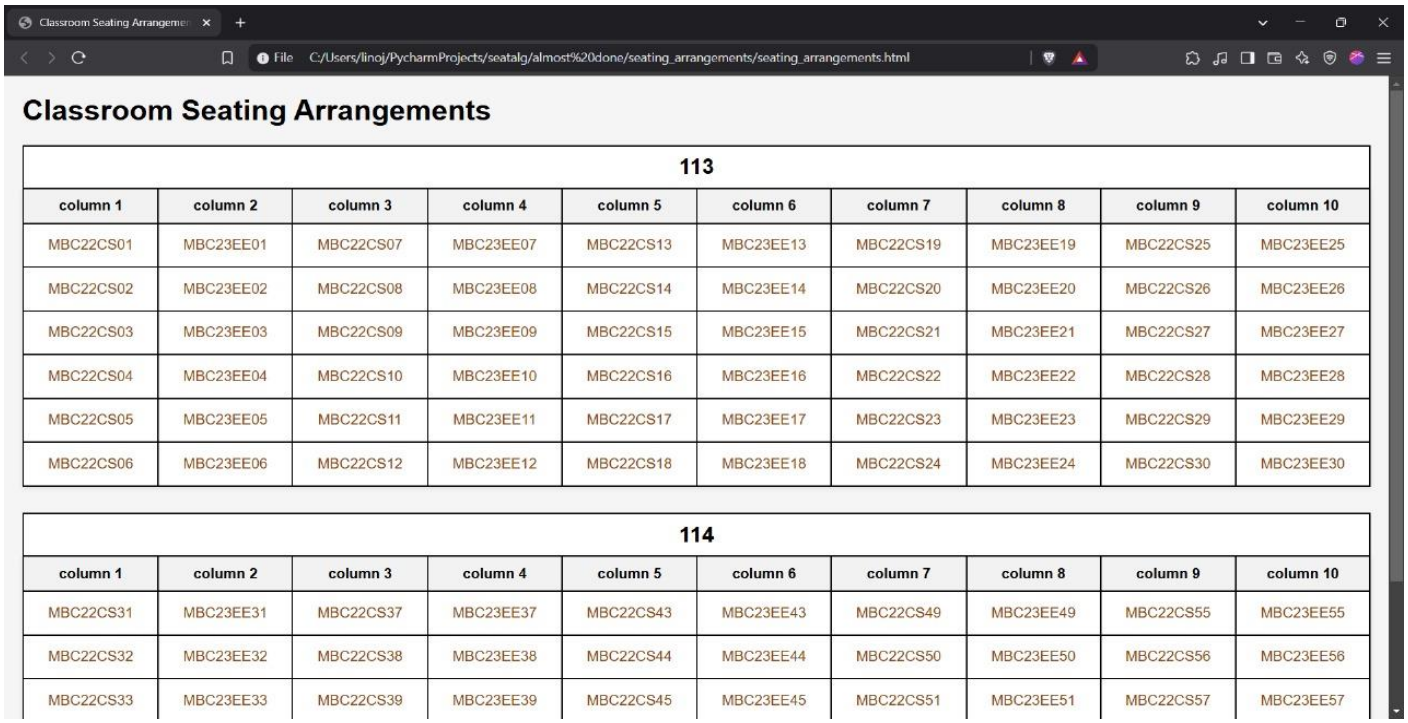
```
            # Container for subject dropdowns
            subject_frame = ttk.Frame(classroom_frame)
            subject_frame.grid(row=4, column=0, columnspan=3, padx=5, pady=5, sticky="nsew")
            classroom_entry["subject_frame"] = subject_frame

            self.classroom_entries.append(classroom_entry)

        # Show the submit button
        self.classroom_submit_btn.grid()

    except ValueError:
        messagebox.showerror( title: "Invalid Input",  message: "Please enter a valid number.")

def create_subject_dropdowns(self, classroom_entry):  1 usage  linoj-k
    # Clear any existing dropdowns
    subject_frame = classroom_entry["subject_frame"]
    for widget in subject_frame.winfo_children():
        widget.destroy()

    try:
        subject_count = int(classroom_entry["subject_count"].get())
        if subject_count <= 0:
            messagebox.showerror( title: "Invalid Input",  message: "Number of subjects must be positive.")
            return

        classroom_entry["subjects"] = []
        subject_list = list(self.subjects.keys())

        for i in range(subject_count):
            subject_var = tk.StringVar()
            ttk.Label(subject_frame, text=f"Subject {i + 1}:").grid(row=i, column=0, padx=5, pady=5, sticky="w")
            subject_dropdown = ttk.Combobox(subject_frame, textvariable=subject_var)
            subject_dropdown.grid(row=i, column=1, padx=5, pady=5)
            subject_dropdown['values'] = subject_list

            classroom_entry["subjects"].append(subject_var)
```

```
    except ValueError:
        messagebox.showerror( title: "Invalid Input",  message: "Please enter a valid number.")

def submit_classrooms(self):  1 usage  linoj-k
    # Process classroom entries
    self.classrooms = []


    for classroom_entry in self.classroom_entries:
        try:
            name = classroom_entry["name"].get()
            seats = int(classroom_entry["seats"].get())
            rows = int(classroom_entry["rows"].get())

            subjects_list = [var.get() for var in classroom_entry["subjects"]]

            classroom = Classroom(name, seats, rows, subjects_list)
            self.classrooms.append(classroom)

        except ValueError:
            messagebox.showerror( title: "Invalid Input",  message: "Please enter valid numbers for classroom details.")
            return
```

```
        # Update the results tab with classroom information
        self.update_results()

        try:
            output_file = assign_students_round_robin(self.subjects, self.classrooms)
            messagebox.showinfo( title: "Success",  message: f"Seating arrangement created: {output_file}")
            import webbrowser
            webbrowser.open(output_file)
        except Exception as e:
            messagebox.showerror( title: "Error",  message: f"An error occurred: {str(e)}")



        # Move to the results tab
        self.notebook.select(3)  # Select the Results tab
        messagebox.showinfo( title: "Success",  message: "Classrooms added successfully!")

    def setup_results_tab(self):  1 usage  linoj-k
        self.results_frame = ttk.Frame(self.results_tab)
        self.results_frame.pack(fill="both", expand=True, padx=10, pady=10)

        # Add a scrollable text widget for results
        self.results_text = tk.Text(self.results_frame, wrap=tk.WORD)
        scrollbar = ttk.Scrollbar(self.results_frame, command=self.results_text.yview)
        self.results_text.configure(yscrollcommand=scrollbar.set)

        self.results_text.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

    def update_results(self):  3 usages  linoj-k
        # Clear current results
        self.results_text.delete(1.0, tk.END)

        # Display branch information
        self.results_text.insert(tk.END, "=== BRANCH INFORMATION ===\n\n")
        for name, items in self.branch.items():
            self.results_text.insert(tk.END, f"Branch name: {name}\n")
```

```
            self.results_text.insert(tk.END, f"Students: {items}\n\n")

        # Display subject information
        self.results_text.insert(tk.END, "=== SUBJECT INFORMATION ===\n\n")
        for name, items in self.subjects.items():
            self.results_text.insert(tk.END, f"Subject: {name}\n")
            self.results_text.insert(tk.END, f"Students: {items}\n\n")

        # Display classroom information
        self.results_text.insert(tk.END, "=== CLASSROOM INFORMATION ===\n\n")
        for classroom in self.classrooms:
            self.results_text.insert(tk.END, f"Classroom name: {classroom.name}\n")
            self.results_text.insert(tk.END, f"Subjects: {classroom.subjects}\n")
            self.results_text.insert(tk.END, f"Rows: {classroom.rows}\n")
            self.results_text.insert(tk.END, f"Columns: {classroom.cols}\n\n")



# Main application
if __name__ == "__main__":
    root = tk.Tk()
    app = StudentRegistrationGUI(root)
    root.mainloop()
```

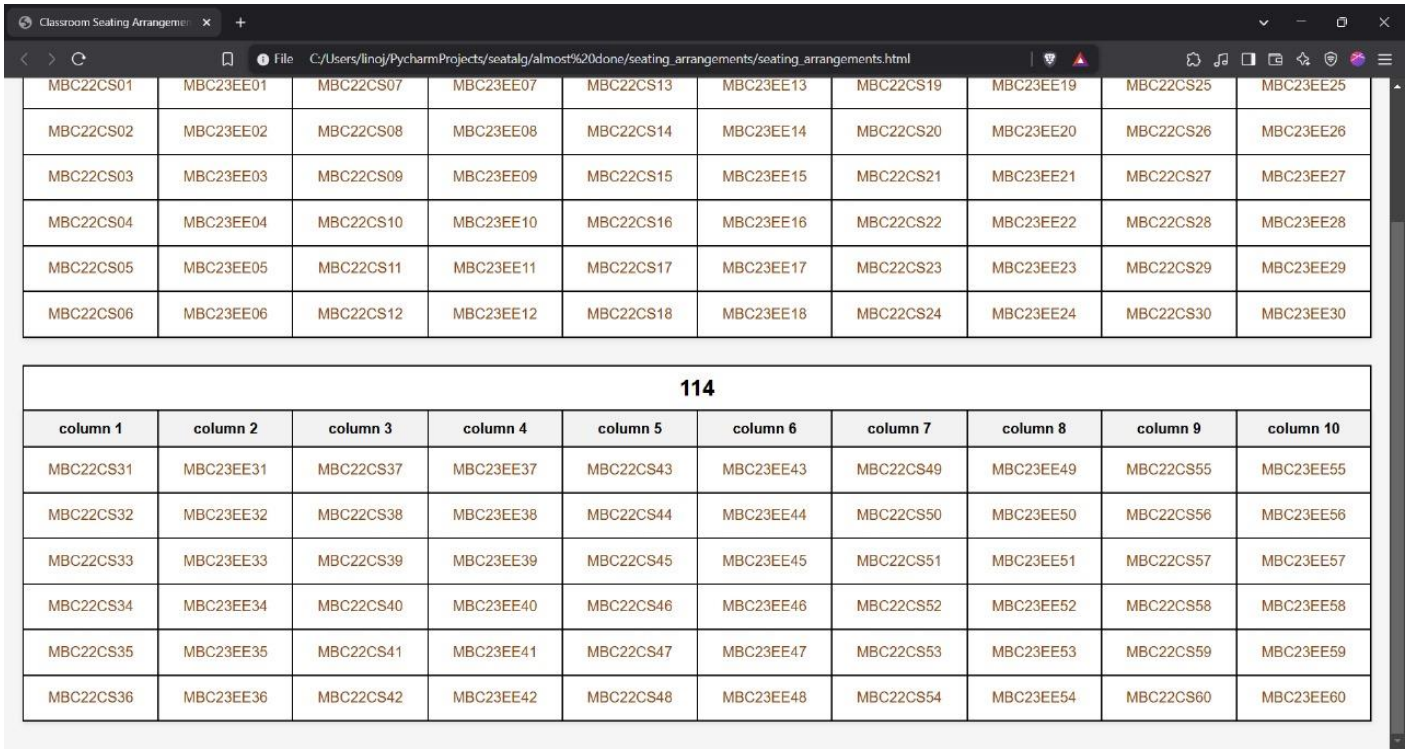
OUTPUT



Classroom Seating Arrangements

113									
column 1	column 2	column 3	column 4	column 5	column 6	column 7	column 8	column 9	column 10
MBC22CS01	MBC23EE01	MBC22CS07	MBC23EE07	MBC22CS13	MBC23EE13	MBC22CS19	MBC23EE19	MBC22CS25	MBC23EE25
MBC22CS02	MBC23EE02	MBC22CS08	MBC23EE08	MBC22CS14	MBC23EE14	MBC22CS20	MBC23EE20	MBC22CS26	MBC23EE26
MBC22CS03	MBC23EE03	MBC22CS09	MBC23EE09	MBC22CS15	MBC23EE15	MBC22CS21	MBC23EE21	MBC22CS27	MBC23EE27
MBC22CS04	MBC23EE04	MBC22CS10	MBC23EE10	MBC22CS16	MBC23EE16	MBC22CS22	MBC23EE22	MBC22CS28	MBC23EE28
MBC22CS05	MBC23EE05	MBC22CS11	MBC23EE11	MBC22CS17	MBC23EE17	MBC22CS23	MBC23EE23	MBC22CS29	MBC23EE29
MBC22CS06	MBC23EE06	MBC22CS12	MBC23EE12	MBC22CS18	MBC23EE18	MBC22CS24	MBC23EE24	MBC22CS30	MBC23EE30

114									
column 1	column 2	column 3	column 4	column 5	column 6	column 7	column 8	column 9	column 10
MBC22CS31	MBC23EE31	MBC22CS37	MBC23EE37	MBC22CS43	MBC23EE43	MBC22CS49	MBC23EE49	MBC22CS55	MBC23EE55
MBC22CS32	MBC23EE32	MBC22CS38	MBC23EE38	MBC22CS44	MBC23EE44	MBC22CS50	MBC23EE50	MBC22CS56	MBC23EE56
MBC22CS33	MBC23EE33	MBC22CS39	MBC23EE39	MBC22CS45	MBC23EE45	MBC22CS51	MBC23EE51	MBC22CS57	MBC23EE57



MBC22CS01	MBC23EE01	MBC22CS07	MBC23EE07	MBC22CS13	MBC23EE13	MBC22CS19	MBC23EE19	MBC22CS25	MBC23EE25
MBC22CS02	MBC23EE02	MBC22CS08	MBC23EE08	MBC22CS14	MBC23EE14	MBC22CS20	MBC23EE20	MBC22CS26	MBC23EE26
MBC22CS03	MBC23EE03	MBC22CS09	MBC23EE09	MBC22CS15	MBC23EE15	MBC22CS21	MBC23EE21	MBC22CS27	MBC23EE27
MBC22CS04	MBC23EE04	MBC22CS10	MBC23EE10	MBC22CS16	MBC23EE16	MBC22CS22	MBC23EE22	MBC22CS28	MBC23EE28
MBC22CS05	MBC23EE05	MBC22CS11	MBC23EE11	MBC22CS17	MBC23EE17	MBC22CS23	MBC23EE23	MBC22CS29	MBC23EE29
MBC22CS06	MBC23EE06	MBC22CS12	MBC23EE12	MBC22CS18	MBC23EE18	MBC22CS24	MBC23EE24	MBC22CS30	MBC23EE30

114									
column 1	column 2	column 3	column 4	column 5	column 6	column 7	column 8	column 9	column 10
MBC22CS31	MBC23EE31	MBC22CS37	MBC23EE37	MBC22CS43	MBC23EE43	MBC22CS49	MBC23EE49	MBC22CS55	MBC23EE55
MBC22CS32	MBC23EE32	MBC22CS38	MBC23EE38	MBC22CS44	MBC23EE44	MBC22CS50	MBC23EE50	MBC22CS56	MBC23EE56
MBC22CS33	MBC23EE33	MBC22CS39	MBC23EE39	MBC22CS45	MBC23EE45	MBC22CS51	MBC23EE51	MBC22CS57	MBC23EE57
MBC22CS34	MBC23EE34	MBC22CS40	MBC23EE40	MBC22CS46	MBC23EE46	MBC22CS52	MBC23EE52	MBC22CS58	MBC23EE58
MBC22CS35	MBC23EE35	MBC22CS41	MBC23EE41	MBC22CS47	MBC23EE47	MBC22CS53	MBC23EE53	MBC22CS59	MBC23EE59
MBC22CS36	MBC23EE36	MBC22CS42	MBC23EE42	MBC22CS48	MBC23EE48	MBC22CS54	MBC23EE54	MBC22CS60	MBC23EE60

CONCLUSION AND FUTURE WORK

The Automated Student Seating Arrangement System successfully addresses the challenges associated with manually organizing seating plans for educational institutions. By providing a user-friendly graphical interface built with Tkinter, the system allows administrators to efficiently input complex data regarding multiple student branches, subjects with flexible enrollment criteria, and diverse classroom configurations.

The core round-robin allocation algorithm implemented in the system provides a systematic and repeatable method for assigning students to seats, distributing different subjects across columns within each classroom. While simpler than advanced constraint satisfaction or optimization algorithms, this approach significantly reduces the manual effort and potential for errors inherent in traditional methods.

The generation of a clear, well-formatted HTML report offers an immediate and easily shareable visualization of the seating arrangement for each classroom, along with a crucial list of any students who could not be accommodated. This output facilitates better planning and communication on the day of the examination or event.

Future enhancements could include:

- Implementing more sophisticated allocation algorithms (e.g., minimizing adjacent students from the same subject).
- Adding constraints like keeping specific groups together or apart.
- Allowing import/export of data from/to common formats like CSV or Excel.
- Integrating with existing student information systems.
- Providing options for different output formats (e.g., PDF).
- Visualizing the seating plan directly within the GUI.

In its current form, the project provides a valuable, practical tool that significantly improves the efficiency, accuracy, and organization of student seating arrangement, demonstrating the effective application of basic programming principles and GUI design to solve a common administrative problem.

REFERENCES

[1]. P. K. Chaki and S. Anirban, "Algorithm for efficient seating plan for centralized exam system," 2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT), New Delhi, India, 2016, pp. 320-325,

[2]. R. D. Danielsen, A. F. Simon, and R. Pavlick, “The Culture of Cheating: From the Classroom to the Exam Room,” Journal of Physician Assistant Education, vol. 17(1), pp. 23-29, 2006.

[3]. A. Dammak, A. Elloumi, and H. Kamoun. "Classroom assignment for exam timetabling," Advances in Engineering Software, pp. 659 -666, 2006.

[4]. M. Ayob and A. Malik, “A New Model for an Examination-Room Assignment Problem,” IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.10, 2011 .

[5]. S.Vasupongayya, W.Noodam, and P.Kongyong, “Developing Examination Management System: Senior Capstone Project, a Case Study,” World Academy of Science, Engering and Technology, Vol:7 2013.