**Personal Scheduling Assistant Project**

### **Introduction**

The **Personal Scheduling Assistant** is a Python-based tool that helps users organize and manage their daily and academic tasks. By applying dynamic programming and optimized sorting/searching techniques, the assistant efficiently categorizes and schedules tasks, provides reminders, and prioritizes high-value activities. It also includes visualizations to represent task distribution throughout the day.

### Design Requirements

### Functional Requirements

* **Task Management**: Add personal and academic tasks with specific deadlines, priorities, and durations.
* **Scheduling and Optimization**: Use dynamic programming to prioritize and schedule tasks.
* **Quick Retrieval**: Implement binary search for retrieving upcoming tasks efficiently.
* **Sorting by Priority and Deadline**: Use merge sort to organize tasks by urgency and type.
* **Visualization**: Provide visual output, including a Gantt chart and analysis of busy time slots.

#### Non-functional Requirements

* **Efficiency**: Apply optimal algorithms to ensure smooth operation with large data sets.
* **Usability**: Create a modular design that allows for easy updates and user-friendly interaction.

### System Architecture

The system is divided into three primary modules:

1. **Task Module**: Represents and manages task properties.
2. **Scheduler Module**: Manages task addition, scheduling, sorting, and deadline reminders.
3. **Visualizer Module**: Analyses and visually represents task density over time using Gantt charts.

### Core Components

### Task Module

Handles creation and storage of task properties such as type, priority, duration, and deadline.

#### Scheduler Module

Manages tasks through sorting and scheduling algorithms and prioritizes based on task attributes.

#### Visualizer Module

Analyses task density, identifies busy periods, and displays Gantt charts for scheduled tasks.

### Algorithms and Pseudocode

#### Task Class

Each task includes essential attributes for scheduling and prioritization.

Class Task:

Function \_\_init\_\_(self, name, task\_type, priority, duration, deadline):

Set self.name to name

Set self.task\_type to task\_type # 'academic' or 'personal'

Set self.priority to priority

Set self.duration to duration

Set self.deadline to deadline

#### Scheduler Class

The Scheduler class handles task management, including adding tasks, scheduling, and retrieving upcoming tasks.

**i). Adding Tasks**

The add\_task function adds tasks to the list, ordered by deadline using binary insertion.

Class Scheduler:

Function add\_task(self, name, task\_type, priority, duration, deadline):

Create new\_task as Task(name, task\_type, priority, duration, deadline)

Insert new\_task into self.tasks using binary insertion based on task deadline

**ii). Scheduling Tasks Using Dynamic Programming**

The schedule\_tasks function prioritizes tasks based on availability and priority.

Function schedule\_tasks(self):

Initialize scheduled\_tasks as an empty list

Set current\_time to the current datetime

Sort tasks by priority and deadline using merge sort

For each task in tasks:

If task deadline is after current\_time plus task duration:

Set task start\_time to current\_time

Add task to scheduled\_tasks

Update current\_time by adding task duration

Return scheduled\_tasks

**iii). Retrieving Upcoming Tasks with Binary Search**

The upcoming\_tasks function retrieves tasks with deadlines within a specified time range using binary search.

Function upcoming\_tasks(self, within\_hours):

Set cutoff\_time to current datetime plus within\_hours

Perform binary search to find tasks with deadlines before cutoff\_time

Return tasks found within cutoff time

#### Visualizer Class

**I). Busy Time Analysis**

Analyses time slots to identify periods with the highest task density

Class Visualizer(Scheduler):

Function busy\_time\_analysis(self, interval\_hours):

Set interval to interval\_hours in datetime format

Set start\_of\_day to the start of the current day

Initialize busy\_intervals as an empty dictionary

For each task in tasks:

Set interval\_start to max(task deadline minus task duration, start\_of\_day)

While interval\_start is less than task deadline:

Increment busy\_intervals at interval\_start by 1

Increment interval\_start by interval

Plot intervals and counts of busy\_intervals

**ii). Gantt Chart Visualization**

Displays tasks in a Gantt chart format to visually represent scheduled activities.

Function display\_gantt\_chart(self, scheduled\_tasks):

Create a new plot

For each task in scheduled\_tasks:

Plot task duration on the Gantt chart with different colors for 'academic' and 'personal' tasks

Show the plot

### Flow of Execution

1. **Initialization**:
   * Instances of the Scheduler and Visualizer classes are created at runtime.
2. **Task Addition**:
   * Users can add tasks with various attributes through the add\_task function.
3. **Task Scheduling**:
   * The schedule\_tasks function processes the tasks by priority and deadline to optimize available time blocks.
4. **Retrieving Upcoming Tasks**:
   * The upcoming\_tasks function allows quick access to tasks within a specific time range using binary search.
5. **Visualization**:
   * The display\_gantt\_chart function displays the tasks visually, while busy\_time\_analysis highlights high-density periods.

### Conclusion

This design document provides a comprehensive overview of the personal scheduling assistant project. By combining dynamic programming, binary search, and merge sort, the assistant efficiently manages and prioritizes tasks, provides timely reminders, and offers a visual representation of the task schedule. The modular design allows for flexibility and ensures efficient task management for users.

**AKUMU ANNA**

**S23B23/094**

**B24782**