**Personal Scheduling Assistant Project**

**Introduction**

The Personal Scheduling Assistant is a Python-based tool designed to help users organize and manage their daily and academic tasks. By applying dynamic programming, optimized sorting, and searching techniques, the assistant efficiently categorizes and schedules tasks, provides reminders, and prioritizes high-value activities. The tool also includes visualizations to represent task distribution throughout the day, making it easy to track task density and visualize task schedules using a Gantt chart.

**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 effectively.

Quick Retrieval: Implement efficient retrieval of upcoming tasks through sorting and searching techniques.

Sorting by Priority and Deadline: Use merge sort to organize tasks by urgency, priority, and deadline.

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 even 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:

**Task Module:** Represents and manages task properties such as type, priority, duration, and deadline.

Scheduler Module: Handles task addition, scheduling, sorting, and deadline reminders.

Visualizer Module: Analyses and visually represents task density over time using Gantt charts.

**Core Components**

Task Module: Manages creation and storage of task properties such as type, priority, duration, and deadline.

**Task Class:**

Attributes:

name: Name of the task.

task\_type: Type of task (academic or personal).

priority: Integer value indicating task priority.

deadline: Deadline of the task (as a datetime object).

duration: Duration of the task in minutes.

Scheduler Module: Handles task management, sorting, and scheduling algorithms.

**TaskManager Class:**

add\_task: Adds tasks to the priority queue (min-heap) based on their priority and deadline.

merge\_sort: Sorts tasks using merge sort by deadline.

get\_upcoming\_tasks: Retrieves upcoming tasks, sorted by deadline.

schedule\_tasks: Schedules tasks by checking available time slots based on priority and deadlines.

analyse\_task\_density: Analyses task density over time and visualizes it.

remind\_tasks: Provides reminders for overdue and pending tasks.

mark\_task\_as\_completed: Marks a task as completed and removes it from the pending list.

view\_completed\_tasks: Displays tasks that have been marked as completed.

Visualizer Module: Handles the visualization of task schedules and density.

plot\_gantt\_chart: Plots tasks on a Gantt chart to visually represent their schedule.

Algorithms and Pseudocode

Task Class

Each task includes essential attributes for scheduling and prioritization.

class Task:

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

self.name = name

self.task\_type = task\_type

self.deadline = deadline

self.priority = priority

self.duration = duration

Scheduler Class (TaskManager)

**Adding Tasks:**

Tasks are added to the priority queue using a heap structure for efficient priority management.

python

def add\_task(self, task):

heappush(self.tasks, (task.priority, task.deadline, task))

**Merge Sort:**

The merge\_sort function sorts tasks by deadline, and the merge function merges two sorted halves.

def merge\_sort(self, tasks, key):

if len(tasks) <= 1:

return tasks

mid = len(tasks) // 2

left = self.merge\_sort(tasks[:mid], key)

right = self.merge\_sort(tasks[mid:], key)

return self.merge(left, right, key)

**Scheduling Tasks:**

The schedule\_tasks function checks available time slots and schedules tasks by priority and deadline.

def schedule\_tasks(self):

current\_time = datetime.now()

scheduled\_tasks = []

for priority, deadline, task in sorted(self.tasks):

if current\_time + timedelta(minutes=task.duration) <= deadline:

scheduled\_tasks.append(task)

current\_time += timedelta(minutes=task.duration)

return scheduled\_tasks

**Analysing Task Density:**

The analyse\_task\_density function creates a plot that shows the density of tasks over time.

def analyse\_task\_density(self):

deadlines = [task.deadline for \_, \_, task in self.tasks]

deadlines.sort()

density\_times = [deadlines[0] + timedelta(hours=i) for i in range(int((deadlines[-1] - deadlines[0]).total\_seconds() // 3600) + 1)]

density\_values = [sum(1 for deadline in deadlines if deadline <= time) for time in density\_times]

plt.plot(density\_times, density\_values)

**Visualization (Gantt Chart):**

The plot\_gantt\_chart function visualizes tasks on a Gantt chart based on their schedule.

def plot\_gantt\_chart(tasks):

fig, gnt = plt.subplots(figsize=(10, 5))

gnt.set\_ylim(0, 50)

gnt.set\_xlim(min(task.deadline for task in tasks), max(task.deadline for task in tasks) + timedelta(minutes=60))

for task in tasks:

start = task.deadline - timedelta(minutes=task.duration)

color = 'skyblue' if task.task\_type == "academic" else 'salmon'

gnt.broken\_barh([(start, timedelta(minutes=task.duration))], (15 if task.task\_type == "academic" else 25, 9), facecolors=color)

plt.show()

**Flow of Execution**

Initialization: The TaskManager instance is created to manage tasks.

Task Addition: Tasks are added through user input using the add\_task function.

Task Scheduling: Tasks are scheduled by calling schedule\_tasks, which considers task deadlines and priorities.

Upcoming Task Retrieval: The get\_upcoming\_tasks function retrieves tasks sorted by their deadlines.

Visualization: The plot\_gantt\_chart function displays the scheduled tasks in a Gantt chart, and analyse\_task\_density highlights busy periods.

**Conclusion**

This design document provides an overview of the Personal Scheduling Assistant project, demonstrating how tasks can be efficiently managed, prioritized, and scheduled. The use of heap structures, merge sort, dynamic scheduling, and Gantt chart visualizations ensure that tasks are organized effectively, making the assistant a helpful tool for managing personal and academic schedules.

**Pseudocode**

Class Task:

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

- Initialize task attributes: name, task\_type, deadline, priority, duration

Method \_\_repr\_\_():

- Return string representation of task for display

Class TaskManager:

Method \_\_init\_\_():

- Initialize empty lists: tasks (priority queue), schedule (list of scheduled tasks), completed\_tasks

Method add\_task(task):

- Add task to the priority queue (tasks list) using heappush with priority, deadline, and task

Method merge\_sort(tasks, key):

- If tasks length <= 1, return tasks

- Divide tasks into two halves

- Recursively merge sort left and right halves

- Merge sorted halves using merge method

Method merge(left, right, key):

- Merge two sorted lists (left and right) based on the key function

- Return merged list

Method get\_upcoming\_tasks():

- Sort tasks by deadline using merge\_sort

- Return list of tasks sorted by deadline

Method schedule\_tasks():

- Set current time to now

- Initialize empty list for scheduled tasks

- For each task, check if it can be scheduled before its deadline:

- Add to scheduled tasks and update current time with task's duration

- Return list of scheduled tasks

Method analyse\_task\_density():

- Get all deadlines from tasks

- Sort deadlines

- Generate a time series for hourly intervals and count tasks due by each time

- Plot task density over time using matplotlib

Method remind\_tasks():

- Set current time to now

- Initialize lists for overdue and pending tasks

- For each task, check if it is overdue or pending:

- Sort overdue tasks by priority

- Sort pending tasks by priority

- Print overdue and pending tasks

Method mark\_task\_as\_completed(task\_name):

- For each task, check if its name matches task\_name:

- Mark task as completed by appending to completed\_tasks and removing from tasks

- Print task marked as completed

- If task not found, print task not found

Method view\_completed\_tasks():

- Print list of completed tasks

Function plot\_gantt\_chart(tasks):

- If no tasks, print no tasks available

- Initialize a matplotlib figure for the Gantt chart

- Set x and y axis properties (time, task categories)

- For each task, plot a horizontal bar representing the task's duration

- Add legend and display chart

Function get\_user\_input(manager):

- Continuously prompt the user for task details until 'done' is entered:

- Task name, type (academic/personal), deadline (datetime format), priority, duration

- Create a Task object and add it to manager's task list

Function display\_menu(manager):

- Preload tasks for demonstration (e.g., Calculus Assignment, Project Report)

- Loop to display the menu with options:

- Add Task

- View Upcoming Tasks

- Schedule Tasks

- Analyse Task Density

- Task Reminders

- Mark Task as Completed

- View Completed Tasks

- Exit

- Based on user input, call corresponding methods (e.g., add\_task, get\_upcoming\_tasks, schedule\_tasks)

Function main():

- Create a TaskManager instance

- Call display\_menu with the TaskManager instance

If \_\_name\_\_ == "\_\_main\_\_":

- Call main()