Project 1 - Designing a Scheduler

1. Tasks

The selected tasks are indicative of a well-rounded daily schedule that includes vital activities for education, cultural exploration, personal growth, and physical and mental health. Every task has a distinct significance that benefits education, personal relationships, health, and cultural enrichment. Setting these tasks in order of importance guarantees a holistic outlook on life, supporting mental and physical health as well as personal development.

ID	Description	Duration (min)	Dependencies	Status	Starting time	Deadline	Location	Category	Time sensitivity
0	Prepare and have breakfast	30	None	not_yet_started	8	10	Indoor	Health	Flexible
1	Call family	30	None	not_yet_started	12	22	Indoor	Personal	Flexible
2	Work on CS110 Assignment	180	None	not_yet_started	10	17	Indoor	Education	Flexible
3	Exercise at the gym	45	0	not_yet_started	15	17	Outdoor	Health	Flexible
4	Cook and have lunch	45	0, 3	not_yet_started	16	18	Indoor	Health	Flexible
5	Visit National Museum of Korea	120	None	not_yet_started	17	22	Outdoor	Cultural	Flexible
6	Do a city walking tour	90	None	not_yet_started	20	23	Outdoor	Cultural	Flexible
7	Have dinner at a new local restaurant	60	0, 3, 4	not_yet_started	18	22	Outdoor	Cultural	Flexible

We can break down the relevance of each of them:

1- Prepare and Have Breakfast (ID: 0):

Gives me the energy I need for the day and supports my general health and well-being. It is essential to initialize the day and consider its starting time the initialized current time.

2- Call Family (ID: 1):

Sustains mental well-being, fortifies bonds between family members, offers emotional support, and cultivates a feeling of inclusion. I made this task with a very long range of time, so you have 10 hours to do it.

3- Work on CS110 Assignment (ID: 2):

Essential for increasing my knowledge, advancing academically, and expanding my professional prospects. This task has a long duration so that we can test the effect of duration on the priority.

4- Exercise at the Gym (ID: 3):

Increases mental clarity, builds cardiac strength, encourages physical activity, and benefits general well-being. This is the first task that has dependencies, so we can make sure it will only be executed after having breakfast.

5- Cook and Have Lunch (ID: 4):

Promotes independence, permits the consumption of healthier foods, gives me energy for the day, and develops practical life skills. This was the second task with dependencies, and I increased their number

to see the effect on the priority.

6- Visit National Museum of Korea (ID: 5):

Expands perspectives, fosters involvement with the city's cultural legacy, and improves the enjoyment of art and history. This task had a relatively small time range of only three hours, and the duration is already an hour and a half, so we can test finishing tasks in a small range.

7- Do a City Walking Tour (ID: 6):

Deepens the connection between me and the city while educating me about its history, architecture, and culture. This task has a late deadline to test if the algorithm will both respect the deadline and respect the 24 hours and not go over them.

8- Have Dinner at a New Local Restaurant (ID: 7):

Enhances gastronomic experiences, appreciates regional food culture, and investigates a variety of flavors and cuisines. This task had more dependencies with a late deadline, which may be an important task that we need to make sure the algorithm will know how to deal with.

I also added some task qualifiers to help me with the implementation:

1- Starting Time:

Vital because it aids in determining the start time of each task, reducing improper start times and overlapping. This feature makes sure that the tasks are arranged logically, following the normal course of the day. For example, it makes sense that tasks like "Prepare and have breakfast" should begin early in the morning.

2- Deadline:

Essential for setting priorities when working under time restrictions. Due dates should be factored into the scheduling process so that tasks can be completed on time and without delay. For example, a task with a stringent deadline such as "Submit work report by 5 PM" would require early prioritization.

3- Location:

Important to distinguish between tasks that are done indoors and outdoors. Tasks can be efficiently grouped by the scheduler using knowledge about the location. For optimal movement, tasks such as "Call family" (inside) and "Visit National Museum of Korea" (outdoor) might be planned sequentially.

4- Category:

Gives a more comprehensive context that makes it possible to arrange jobs according to their nature. Similar tasks in the same category allow the scheduler to efficiently manage several facets of life. To provide a balanced and prioritized routine that supports general health and wellness, tasks labeled as "Health" may include those pertaining to physical well-being.

5- Time Sensitivity:

Has a significant impact on how the scheduler approaches the task. A task that has been designated as "flexible" can be scheduled in a flexible manner, maximizing overall efficiency by easily fitting into available time slots. A "fixed" task, on the other hand, must be completed exactly at the appointed time;

hence, the scheduler must give it top priority right away to avoid delays. This differentiation guarantees that the tasks are in line with their assigned time limits, resulting in a smooth workflow within the timetable.

These task qualifiers provide the scheduler with a comprehensive grasp of the tasks, allowing for intelligent organization and prioritization. Every feature contributes in a different way to increasing the scheduler's effectiveness and guaranteeing that jobs are scheduled logically, on time, and in a manner that optimizes resources and time.

2. The algorithmic strategy

Because of its built-in effectiveness at handling constantly changing priorities, a priority queue continues to be a great option when it comes to assigning tasks to the daily activity scheduler. A priority queue, as opposed to a static sorted list, allows tasks to be added and removed in real-time, allowing for quick adjustments in response to changing priorities. In a daily schedule, when task priorities might change quickly in response to contextual factors, deadlines, and dependencies, this real-time adaptability is essential.

Tasks from both the time-flexible and fixed-time categories are unified in a single priority queue in our improved algorithm. By using a single, cohesive strategy, all tasks are prioritized according to their own set of requirements, including starting times, locations, dependencies, and deadlines. Both fixed-time tasks with strict deadlines and time-flexible tasks that can be completed at different times during the day belong in this one queue.

The flexibility of the algorithm is improved by the use of a single priority queue. Time-sensitive tasks are given priority and finished on time as the algorithm advances by carefully choosing them from this single queue. Time-flexible tasks are simultaneously optimized in response to shifting dependencies or contextual factors, enabling effective use of available time slots.

To sum up, our algorithm's foundation is the use of a single, dynamic priority queue. This unified method guarantees the algorithm's efficient adaptation to changing task priorities and limitations, in conjunction with the nuanced utility function and task qualifiers. It ensures that daily tasks are dynamic and prioritized differently, resulting in an ideal and well-structured schedule that fits nicely with life.

Let's explain at a very high level to you:

Think of your day as a puzzle, with each task representing a distinct piece. Our scheduler puts these parts together for you with the dexterity of a puzzle solver. This is how it goes:

Task Collection: Initially, we compile every task you've completed, each with unique attributes. While certain tasks, like attending courses, have fixed times, others, like studying, are flexible. A web of links is created when certain tasks rely on earlier tasks that we call dependencies.

Task Prioritization: Our scheduler's core function is to dynamically assign tasks a priority value. Imagine that the urgency of each task is represented by a number; the higher the number, the more urgent the task is. A complex utility function is used to calculate these priorities, which take into consideration a number of variables including duration, dependencies, task status, starting time, proximity to deadline, location, category, and time sensitivity. The particular priority value of each task is determined by its own combination of these elements.

Organizing by Priority: Based on their computed priority values, tasks are grouped into a single priority queue. All tasks are taken into account for prioritization, regardless of their dependencies or time limits, thanks to this cohesive methodology. Within this single queue, tasks with comparable priorities are put together.

Scheduling Algorithm: The first step of our scheduling algorithm is to choose the task with the highest priority and its starting time began from the unified queue. The necessity for explicit dependency checks during scheduling is eliminated by the careful calculation of these priorities, which takes task dependencies into account. The algorithm skillfully arranges tasks in a priority-based order, intuitively respecting dependencies along the way.

Dynamic Adaptation: Unexpected tasks may come up during the course of the day. Our scheduler quickly determines the new tasks' priority inside the current puzzle in order to dynamically accommodate them. The algorithm reorders tasks as needed to fit these tasks into your schedule in an easy manner. This flexibility ensures that even with last-minute additions, your plan stays structured.

Optimizing Idle Time: During idle periods, where you're waiting for a fixed-time task to start (because it has the highest priority), the scheduler proactively scans the remaining tasks in the queue. If it identifies a task with a duration equal to or less than the remaining time for the fixed-time task to begin, it executes that task immediately even though its priority is less than the fixed-time task because this will be better than time going unused.

Task Execution: During the day, you carry out the tasks in the single priority queue in the correct order. The scheduler acts as your guide, telling you which task to work on right now and once you finish a task its status is changed and moved from the queue to a new list directing your attention to what matters most at any given time, thanks to the carefully calculated priorities.

Completed Tasks Record: Completed tasks are shifted out of the queue and added to an additional list. You may maintain track of the tasks you've completed using this record, which gives you a comprehensive picture of your daily accomplishments.

An important note for anyone implementing this algorithm in Python:

In Python, dictionaries are used to describe tasks, and each dictionary is given a unique ID. Task qualifiers including description, duration, dependencies, and priority are captured in these dictionaries. When implemented as a list, a priority queue makes it easier to arrange tasks according to their calculated priorities. This queue is iterated through by a loop that carefully considers the utility function's priority for scheduling tasks.

There is a flowchart visualizating the Algorithm at the begining of the Appendix.

3. Utility Function Design

The utility function has been meticulously crafted to dynamically compute the priority values of tasks, ensuring a balanced and efficient scheduling process. Here's how we derived the final utility function based on assigned weights and values for the task qualifiers:

Task Dependencies:

Tasks with fewer dependencies are prioritized. The weight assigned to dependencies - which is -5 is a negative value and is multiplied by the number of dependencies so when their number increases their overall value decreases - ensures tasks with no dependencies receive higher utility values.

Task Duration:

Shorter tasks are favored, emphasizing efficiency. The duration weight assigned to minutes of duration - which is -0.1 is a negative value and is multiplied by the task duration minutes - gives higher utility values to tasks of shorter duration.

Starting Time:

Tasks are arranged logically within the day's timeline. The closer the starting time the more prioritized the task so it can be finished once its starting time begins. The weight assigned to the remaining time until the starting time - which is -0.5 is a negative value and is multiplied by the number of hours of the remaining time until the starting time - ensures tasks begin at appropriate times and as soon as their starting time begins.

Deadline Proximity:

Tasks closer to their deadlines receive higher priority. The weight assigned to the remaining time until the deadline - which is -3 is a negative value and is multiplied by the number of hours of the remaining time until the deadline - ensures tasks nearing their deadlines are prioritized. Importantly, to maintain logical prioritization, the negative sign for the deadline weight is essential. The further the deadline, the more prioritized the task.

Location:

Tasks at indoor locations are preferred initially. Positive weights - which are 3 for indoors and 1 for outdoors - are assigned to indoor and outdoor tasks, ensuring tasks at convenient locations are prioritized. This preference is vital for ensuring tasks in comfortable or nearby places receive positive prioritization.

Category:

Different positive weights are assigned based on the task category: 4 for "Health," 3 for "Education," 2 for "Cultural," and 1 for "Personal." This aligns with user preferences, ensuring tasks in preferred categories receive higher utility values. The preference for specific categories ensures tasks align with the user's interests and priorities, making the scheduling process more personalized and user-centric.

Time Sensitivity:

Tasks are categorized as "flexible" or "fixed time." Positive weights are assigned based on time sensitivity: 5 for "fixed time" and 1 for "flexible time." Time-sensitive tasks are given higher utility values, ensuring they are appropriately prioritized in the scheduling process.

Final Utility Calculation:

The final utility value is calculated by summing up the contributions from all factors using the provided weights and values, ensuring tasks are prioritized dynamically based on their dependencies, duration, deadline proximity, time sensitivity, location, category, and user preferences. The min-heap priority queue ensures that the task with the minimum priority value (highest utility) is always at the top and can

be executed next. The dynamic adaptation of the utility function guarantees optimal scheduling outcomes, considering various task attributes and user preferences.

Final Utility Function:

Utility = Base Utility + (Duration Weight × Duration) + (Dependency Weight × Number of Dependencies) + (Starting Time Weight × Starting Time Proximity) + (Deadline Weight × Deadline Proximity) + Location Value + Category Value + Time Sensitivity Value

We can plug in the weight values in the function:

Utility = Base Utility + (-0.1 × Duration) + (-5 × Number of Dependencies) + (-0.5 × Starting Time Proximity) + (-3 × Deadline Proximity) + Location Value + Category Value + Time Sensitivity Value

1. Code testing manually

First of All, we will define the current time which is in other words the start of the day for the schedular which is the earliest starting time in the tasks input

As we can see Task C has the earliest starting time at 9:30 AM or 9.5 * 60 = 570 minutes

let's dive in right now:

Task A:

Duration: 60 minutes

Dependencies: None

Starting Time: 10:00 AM (600 minutes)

Deadline: 12:00 PM (720 minutes)

Location: Indoor (3 points)

Category: Health (4 points)

Time Sensitivity: Flexible (1 point)

Calculating Utility for Task A:

Utility = (−0.1 × 60) + (−5 × 0) + (−0.5 × (600−570)) + (−3 × (720−570) + 3 + 4 + 1

Utility = −6 + 0 + -15 + -450 + 3 + 4 + 1

Utility = -463

Task B:

Duration: 45 minutes

Dependencies: None

Starting Time: 11:00 AM (660 minutes)

Deadline: 1:00 PM (780 minutes)

Location: Indoor (3 points)

Category: Personal (1 points)

Time Sensitivity: Fixed (5 points)

Calculating Utility for Task B:

Utility = (−0.1 × 45) + (−5 × 0) + (−0.5 × (660−570)) + (−3 × (780−570) + 3 + 1 + 5

Utility = −4.5 + 0 + −45 + −630 + 3 + 1 + 5

Utility = -670.5

Task C:

Duration: 30 minutes

Dependencies: None

Starting Time: 9:30 AM (570 minutes)

Deadline: 11:30 AM (690 minutes)

Location: Outdoor (1 point)

Category: Education (3 points)

Time Sensitivity: Flexible (1 point)

Calculating Utility for Task C:

Utility = (−0.1 × 30) + (−5 × 0) + (−0.5 × (570−570)) + (−3 × (690−570)) + 1 + 3 + 1

Utility = −3 + 0 + 0 + −360 + 1 + 3 + 1

Utility = −358

Task D:

Duration: 90 minutes

Dependencies: [A]

Starting Time: 1:30 PM (810 minutes)

Deadline: 4:00 PM (960 minutes)

Location: Indoor (3 points)

Category: Cultural (2 points)

Time Sensitivity: Flexible (1 point)

Calculating Utility for Task D:

Utility = −0.1 × 90 + (−5×1) + (−0.5 × (810−660)) + (−3 × (960−660)) + 3 + 2 + 1

Utility = −9 + −5 + −75 + −900 + 3 + 2 + 1

Utility = −983

From these calculations, Task C has the highest priority value (-358), followed by Task A (-463), Task B (-670.5), and Task D (-983).

So, the tasks will be ordered as follows:

1- C

2- A

3- B

4- D

We will need also to remember the current time which was the earliest starting time which was the starting time of task C which was 9:30 AM.

Now, we will begin by looking at the starting time of C because it is supposed to be executed right now. The starting time of C is 9:30 AM so it is equal to the current time so we will execute it.

Now we should update the current time based on the Duration taken to execute C which was 30 minutes so now the current time is 570 + 30 = 600 minutes which is 10 AM

We will now move to the next in the queue which is Task A Looking at the starting time of A, we will see it is 10 AM and the current time right now is 10 AM so we can execute it right now.

We have to update the current time again so it is 10 AM + Duration of task A = (10*60) + 60 = 660 minutes or 11 AM

We will now move to the next in the queue which is Task B Looking at the starting time of B, we will see it is 11 AM and the current time right now is 11 AM so we can execute it right now.

We have to update the current time again, so it is 11 AM + Duration of task B = (11*60) + 45 = 705 minutes or 11:45 AM

We will now move to the last in the queue which is Task D Looking at the starting time of D, we will see it is 1:30 PM and the current time right now is 11:45 AM so we are supposed to skip and look at the task with the highest priority and is ready to be executed right now but as D is the last one in the queue,

we will wait until its starting time come so we can execute it right now.

We must update the current time again, so it is 11:45 AM + Duration of task D = (11.75*60) + 90 = 795 minutes or 01:15 PM

Using this scheduler algorithm, we executed all four tasks based on their priority and in their appropriate time slots in 3:45 hours or 225 minutes.

We can look at the test case which had the input from the table at the begining (the complete working code is in the appendix after the flowchart)

```
⏰ 08:00 AM
        ✅ Prepare and have breakfast till 08:30 AM ,⏱️ 30 min
⏰ 10:00 AM
        ✅ Work on CS110 Assignment till 01:00 PM ,⏱️ 3 hr
⏰ 01:00 PM
        ✅ Call family till 01:30 PM ,⏱️ 30 min
⏰ 03:00 PM
        ✅ Exercise at the gym till 03:45 PM ,⏱️ 45 min
⏰ 04:00 PM
        ✅ Cook and have lunch till 04:45 PM ,⏱️ 45 min
⏰ 05:00 PM
        ✅ Visit National Museum of Korea till 07:00 PM ,⏱️ 2 hr
⏰ 07:00 PM
        ✅ Have dinner at a new local restaurant till 08:00 PM ,⏱️ 1 hr
⏰ 08:00 PM
        ✅ Do a city walking tour till 09:30 PM ,⏱️ 1 hr 30 min
```

5. Analysis

Benefits of the Algorithmic Strategy:

Priority-Based Scheduling:

Tasks are prioritized by the scheduler using a priority-based method, taking into account a number of variables, including duration, dependencies, starting time, deadline, location, category, and time sensitivity. This method guarantees that the most important tasks are scheduled first, which results in effective resource use.

Flexibility and Customizability:

Tasks with flexible start and finish times, dependencies, and time sensitivity are made possible by the algorithm. Due to its adaptability, it can be used in a variety of real-world situations where the requirements and dependencies of the tasks differ.

Dynamic Scheduling:

The scheduler makes decisions in real-time based on the status of tasks and dynamically adapts to changing circumstances. It is flexible enough to accommodate a wide range of scheduling needs since it can adjust to varying input sizes and job arrangements.

Failure Modes and Limitations:

Task Dependency Complexity:

Task dependencies are managed by the scheduler, although extremely complicated dependence structures may provide difficulties. Difficulties in identifying the best scheduling sequence may arise from complex interdependencies or circular dependencies. **Scalability:**

The scheduling algorithm's time complexity may become a constraint when the number of tasks rises. The current implementation schedules tasks using a priority queue with a logarithmic insertion and deletion time called MaxHeap. This could cause a discernible delay for a lot of tasks.

Detailed Time Complexity Analysis:

1. Priority Queue Operations:

Insertion (Heappush):

Inserting a task into the MaxHeap takes O(log N) time, where N is the number of tasks.

Extraction (Heappop):

Extracting the task with the highest priority takes O(log N) time as well.

2. Task Priority Calculation:

Priority Calculation:

The calculate_priority function evaluates various attributes and computes the task's priority. This function's complexity depends on the number of attributes considered, but it generally takes constant time for each task, i.e., O(1) per task.

3. Time Sensitivity Handling:

Time Sensitivity Check:

Evaluating time sensitivity attributes incurs a constant time complexity, i.e., O (1) per task.

4. Overall Time Complexity:

The complexity of forming a heap of n elements is O (n). To preserve the heap property, this procedure entails first inserting each element into the heap and then carrying out "up-heap" operations. Thus, the heap's construction has an O (n) time complexity.

However, it requires O (log n) time to fix the heap structure after each removal of elements from the heap. Because each removal operation requires swapping the final element with the root and then executing "down-heap" operations to preserve the heap property, the answer is O (log n). A binary heap's height is equal to log n, where n is the total number of elements in the heap.

Our technique basically performs a sequence of O (n log n) operations (removals) and O (n) operations (heap construction) for each input size by continually deleting things from the heap and then rebuilding it. Thus, taking into account both heap constructions and deletions, the overall complexity for our scheduling simulation is O (n log n) for each input size.

Graphical Illustration of Time Complexity:

There are two graphs (at the end of the appendix) the first plots input size VS running time and the second one plots input size VS average time. I plotted two graphs to show that although there is some noise in the second graph, it would have been like the first graph if we didn’t take an average. In the second graph, the x-axis shows the number of tasks scheduled and the y-axis shows the average time of iterating 200 times for each input. In the graph, as the number of tasks (n) increases, the scheduling

time scale with O (n log n) time complexity. It is not obvious, and it seems linear, but it needs more range data. To check or make sure about the complexity graphically, I plotted the best line fit for the same input data which checks the O complexity so to check if it is O (n log n) I checked what number when plugged in for N in O (n log n) will give almost 1 and 3 was very close so basically I plotted the best-fit line as an upper bound to check if the data is growing linearly or with complexity O (n log n) and it was obvious that the upper bound is n log n.

We are using Big O to express the complexity because we are interested in making sure that this is the most time it can take and will not exceed especially since our algorithm can have a best and worst case.

Real-World Deployment Considerations:

Use Case Suitability:

The scheduler works effectively in situations where tasks have different priorities, dependencies, and constraints. It works well in dynamic settings when the order of tasks is determined by factors that change in real-time.

User Preference Consideration:

The calculate_priority function becomes essential when taking user preferences into account. The priority values are influenced by the users' priorities, which include short durations, indoor tasks, and health. Because of the function's flexibility, users can tailor the scheduler to meet their unique requirements by customizing the order in which tasks are prioritized.

Continuous Improvement:

To solve constraints, boost effectiveness, and manage changing requirements, frequent updates and optimizations are required. Real-world usage statistics and user input ought to direct these enhancements.

6. Video

https://drive.google.com/file/d/113_XJwWEYoMMJA-v7cDbPtcoaZ2lJIK3/view?usp=sharing

7. Los and HCs

#professionalism: The project was presented in an orderly and straightforward manner, the Python code was well-documented and followed recognized norms, and the audience's comprehension level was duly taken into consideration. The application of error handling and ethical coding practices, along with the thoughtful acknowledgment of the constraints of the implemented algorithms, demonstrate a holistic approach to professionalism in computer science work. Keeping a neatly organized and formatted PDF was essential to using this LO effectively. Furthermore, the assignment's overall professionalism was further improved by the skillful use of visualizations like flowcharts and the creation of an understandable, well-lit film explaining the algorithm. Word Count: 101

#cs110_AlgoStratDataStruct: With the help of a custom max-heap implementation, the code efficiently arranges tasks according to priority levels, displaying an understanding of algorithmic techniques and data structures. Task information is encapsulated in the task class, and high-priority tasks are done first

because of the max-heap's effective insertion, extraction, and sorting features. When it comes to dynamic priorities, the max-heap performs better than other scheduling algorithms. Additionally, I computed the time scaling for several operations, such as insertion, initialization, and deletion and subsequently employed this information to determine the overall scheduler complexity. Word Count: 90

#cs110_CodeReadability: With the use of clear and simple variable names, reliable naming practices, insightful docstrings, and insightful comments, the code is incredibly readable. Added to the code's user-friendliness, helpful error messages also facilitate debugging. The code's overall readability and lack of obvious flaws highlight that it conforms to the LO #cs110_CodeReadability. Word Count: 50

#cs110_ComplexityAnalysis: I carefully calculated and thoroughly assessed the scheduling algorithm's time complexity. I successfully dissected the method into its individual operations and explained how heap operations affect the total O (n log n) complexity. The analysis proved that I had a firm grasp of algorithmic complexity and could use the LO to evaluate the effectiveness of scheduling algorithms. It was also made abundantly evident how to comprehend Big-O notation, why we use it specifically instead of any other notation, and how to use it to examine the behaviors of the algorithms. Word Count: 91

#cs110_ComputationalCritique: In this project, the LO was implemented at the beginning by comparing the priority queue-based scheduler and the list-based scheduler, two distinct scheduling algorithms. We found that when dealing with a high number of tasks, the priority queue-based scheduler is more effective for scheduling tasks with different priorities and dependencies. Additionally, we had to decide to use one or two priority queues. Taking into account the quantity of tasks, the intricacy of task relationships, and the urgency of the tasks, we analyzed the resources that each one needs. We decided to have a single queue in order to increase efficiency based on this analysis. Word Count: 104

#cs110_PythonProgramming: I used the LO in this project by using Python to create the priority queue-based scheduling method. To confirm that the implementation is proper and to show that it can handle tasks with different priorities, dependencies, and time limitations, I also created test cases for the MaxHeapq class and the scheduler. In order to assess the performance of the method, I also performed a temporal complexity study and used Python plotting tools to visualize the findings. These initiatives show that I can successfully implement, evaluate, and visualize algorithms using my Python programming talents. Word Count: 93

#constraints: I used the #constraints HC in this project to solve the task scheduling issue. I determined the intrinsic limitations of the issue, including time constraints, and task dependencies. We were able to methodically investigate the priority and find workable schedules that met every requirement by encoding those constraints into a utility function problem. We were able to efficiently manage the scheduling problem's complexity and improve task execution thanks to this strategy. We ensured that no task is completed before its start time or after its deadline by using the HC. Word Count: 90

8. Collaborators and AI statement

Name: Eiad Hamdy Collaborators: Ali Osman and ChatGPT Details: I discussed with Ali about plotting the best-fit line for the graph to ensure the time complexity and how can I improve the output of the scheduler code and make it more detailed and professional (like adding emojis). I used ChatGPT to help comment on some codes to specify codes including the plotting code and test cases of maxheap class. I also used ChatGPT to improve the method turning hours to minutes.

I acknowledge that all of the work included in this workbook is my own, and I have not shared (or received) any working solutions with (or from) anyone, not even an AI engine or tool.

9. Resources

Breakout, Session. 7. 2. (2023, October 18). Python Implementation of the Activity Scheduler Breakout. Structured learning exercise document. https://sle-collaboration.minervaproject.com/?id=1f1839fc-678b-429a-b05c-a27dbe6b137d&userId=12385&name=Eiad+Hamdy&avatar=https%3A//s3.us-east-1.amazonaws.com/picasso.fixtures/Eiad_Hamdy_12385_2022-05-13T12%3A32%3A53.031Z&isGrading=1&readOnly=1&isInstructor=0&signature=570f452d213199673b375(

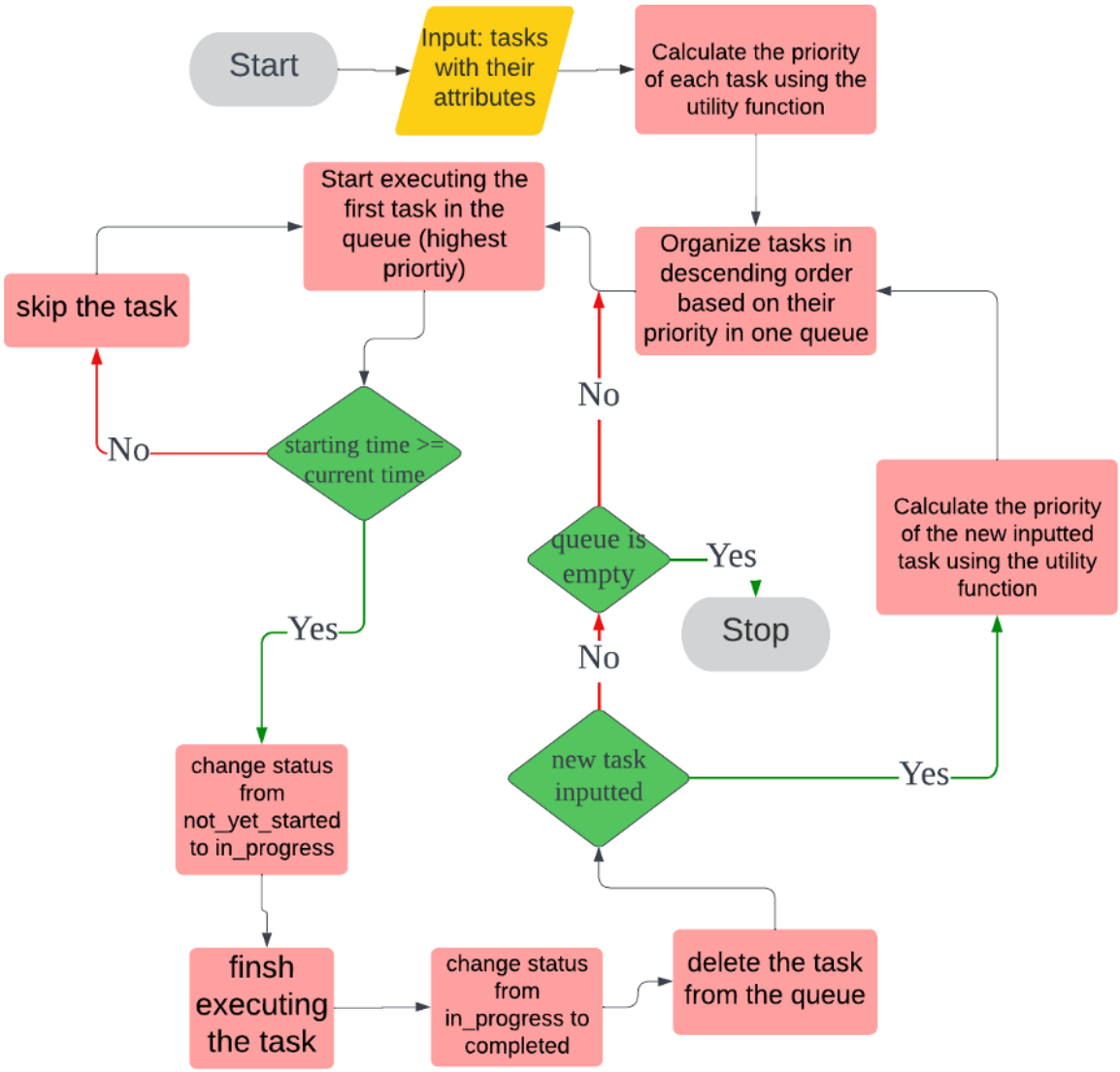
I used this resource which is the code we worked on in the first code cell in breakouts of session 7.2 from CS110 as a skeleton, starting point, or a template that I improved and polished to reach the final code. It acted more as a guide for me to know how my thinking process should go.

Megamind. (2020, March 7). Python program to convert hours to minutes. OneCompiler.https://onecompiler.com/posts/3thnznwd4/python-program-to-convert-hours-to-minutes

Simply I used this resource to have an idea of how I can implement a method to convert hours to minutes and this method was then improved or polished by ChatGPT.

Daily Scheduler algorithm flowchart

Eiad Hamdy | November 5, 2023



In [ ]:

```python
import random

class MaxHeap:
    """
    MaxHeap class represents a max-heap data structure for tasks.
    """

    def __init__(self):
        """
        Initializes an empty heap.
        """
        self.heap = []

    def heappush(self, task):
        """
        Adds a task to the heap and maintains the heap property.

        Args:
            task (tuple): A tuple containing priority value and task object.
        """
```

```
        self.heap.append(task)
        self._sift_up(len(self.heap) - 1)

    def get_max(self):
        """
        Returns the task with the maximum priority value without removing it.

        Returns:
            Task: Task object with the maximum priority.
        """
        return self.heap[0][1]

    def heappop(self):
        """
        Removes and returns the task with the maximum priority value from the heap.

        Returns:
            Task: Task object with the maximum priority.
        """
        if len(self.heap) == 0:
            raise IndexError("Heap is empty")
        task = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._sift_down(0)
        return task

    def _sift_up(self, i):
        """
        Maintains the heap property by moving the element at index 'i' up the heap.

        Args:
            i (int): Index of the element to be moved up.
        """
        while i > 0:
            parent = (i - 1) // 2
            if self.heap[i] > self.heap[parent]:
                self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]
                i = parent
            else:
                break

    def _sift_down(self, i):
        """
        Maintains the heap property by moving the element at index 'i' down the heap.

        Args:
            i (int): Index of the element to be moved down.
        """
        while 2 * i + 1 < len(self.heap):
            left_child = 2 * i + 1
            right_child = 2 * i + 2
            largest = left_child
            if right_child < len(self.heap) and self.heap[right_child] > self.heap[left_
                largest = right_child
            if self.heap[i] < self.heap[largest]:
                self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
                i = largest
            else:
                break
```

In [ ]:

```
import random

def test_max_heapq():
```

```
    """Tests the heappush(), heappop(), and heapify() methods of the MaxHeapq class."""

    # Test Case 1: Testing heappush method

    input_values = [random.randint(1, 100) for _ in range(10)]

    max_heap = MaxHeapq()

    # Push the elements of the random list to the heap
    for value in input_values:
        max_heap.heappush(value)

    expected_output = sorted(input_values, reverse=True)

    assert max_heap.heap == expected_output, "Heappush operation passed"


    # Test Case 2: Testing heappop method

    input_values = [random.randint(1, 100) for _ in range(10)]

    max_heap = MaxHeapq()

    # Push the elements of the random list to the heap
    for value in input_values:
        max_heap.heappush(value)

    expected_output = max(input_values)

    assert max_heap.heappop() == expected_output, "Heappop operation passed"


    # Test Case 3: Testing heapify method with a random list

    input_list = [random.randint(1, 100) for _ in range(10)]

    max_heap = MaxHeapq()

    # Heapify the random list
    max_heap.heap = input_list.copy()
    max_heap.heapify(0)

    # Expected output: A sorted list of integers in reverse order
    expected_output = sorted(input_list, reverse=True)

    assert max_heap.heap == expected_output, "Heapify operation passed for random list"


    # Test Case 4: Testing heapify method with an already sorted list

    input_list = sorted([random.randint(1, 100) for _ in range(10)], reverse=True)

    max_heap = MaxHeapq()

    # Heapify the sorted list
    max_heap.heap = input_list.copy()
    max_heap.heapify(0)

    expected_output = input_list.copy()

    assert max_heap.heap == expected_output, "Heapify operation passed for already sorte


    # Test Case 5: Testing heapify method with a reverse sorted list
```

```
    # Create a reverse sorted list of integers
    input_list = sorted([random.randint(1, 100) for _ in range(10)])

    max_heap = MaxHeapq()

    # Heapify the reverse sorted list
    max_heap.heap = input_list.copy()
    max_heap.heapify(0)

    # Expected output: A sorted list of integers in reverse order
    expected_output = sorted(input_list, reverse=True)

    assert max_heap.heap == expected_output, "Heapify operation passed for reverse sorte


    # Test Case 6: Testing heapify method with an empty list

    # Create an empty list
    empty_list = []

    max_heap = MaxHeapq()

    # Heapify the empty list
    max_heap.heap = empty_list.copy()
    max_heap.heapify(0)

    # Expected output: An empty list
    expected_output = empty_list.copy()

    assert max_heap.heap == expected_output, "Heapify operation passed for empty list"


    # Test Case 7: Testing heapify method with a list containing duplicates

    # Create a list containing duplicates
    input_list = [5, 2, 7, 2, 8, 5, 1, 9, 2, 3]

    max_heap = MaxHeapq()

    # Heapify the list containing duplicates
    max_heap.heap = input_list.copy()
    max_heap.heapify(0)

    # Expected output: A sorted list of integers in reverse order, with duplicates
    expected_output = [9, 8, 7, 5, 5, 2, 2, 2, 1, 3]

    assert max_heap.heap == expected_output, "Heapify operation passed for list with dup
```

In [ ]:

```
import math
import random
import datetime

class Task:
    """
    Task class represents a task with various attributes.
    """
    def __init__(self, id, description, duration, dependencies, status="not_yet_started"
                 starting_time=0, deadline=0, location="Indoor", category="Health", time
        """
        Initializes a Task object with the given attributes.

        parameters:
            id (int): Unique identifier for the task.
```

```
            description (str): Description of the task.
            duration (int): Duration of the task in minutes.
            dependencies (list): List of task IDs that this task depends on.
            status (str): Current status of the task (default is "not_yet_started").
            starting_time (int): Starting time of the task in minutes from midnight (def
            deadline (int): Deadline time of the task in minutes from midnight (default
            location (str): Location of the task (default is "Indoor").
            category (str): Category of the task (default is "Health").
            time_sensitivity (str): Time sensitivity of the task (default is "Flexible")
        """
        self.id = id
        self.description = description
        self.duration = duration
        self.dependencies = dependencies
        self.status = status
        self.starting_time = int(starting_time)  # Ensure starting_time is an integer
        self.deadline = int(deadline)  # Ensure deadline is an integer
        self.location = location
        self.category = category
        self.time_sensitivity = time_sensitivity
        self.utility = self.calculate_priority(self.starting_time)

    def __eq__(self, other):
        """
        Checks if two Task objects are equal.

        parameters:
            other (Task): Another Task object to compare with.

        Returns:
            bool: True if the objects are equal, False otherwise.
        """
        return (
            isinstance(other, Task) and
            self.id == other.id and
            self.description == other.description and
            self.duration == other.duration and
            self.dependencies == other.dependencies and
            self.status == other.status and
            self.starting_time == other.starting_time and
            self.deadline == other.deadline and
            self.location == other.location and
            self.category == other.category and
            self.time_sensitivity == other.time_sensitivity
        )

    def __lt__(self, other):
        # Compare tasks based on their priority values
        return self.utility < other.utility

    def calculate_priority(self, current_time):
        """
        Calculates the priority value of the task based on various factors.

        parameters:
            current_time (int): Current time in minutes from midnight.

        Returns:
            float: Priority value of the task.
        """
        base_utility = 0
        duration_weight = -0.1
        dependency_weight = -5
        starting_time_weight = -0.5
```

```
        deadline_weight = -3
        location_weight = 3 if self.location == "Indoor" else 1
        category_weight = 4 if self.category == "Health" else 3 if self.category == "Edu
        time_sensitivity_weight = 5 if self.time_sensitivity == "fixed" else 1

        utility = base_utility + (duration_weight * self.duration) + (dependency_weight
                  + starting_time_weight * (self.starting_time - current_time) \
                  + deadline_weight * (self.deadline - current_time) \
                  + location_weight + category_weight + time_sensitivity_weight

        return utility

class ActivityScheduler:
    """
    ActivityScheduler class schedules tasks based on their priority values.
    """
    def __init__(self, tasks):
        """
        Initializes the scheduler with a list of tasks.

        parameters:
            tasks (list): List of Task objects to be scheduled.
        """
        self.tasks = MaxHeap()
        for task in tasks:
            self.tasks.heappush((task.utility, task))

        self.completed_tasks = []

    def minutes_to_am_pm(self, minutes):
        """
        Converts minutes from midnight to AM/PM format.

        parameters:
            minutes (int): Time in minutes from midnight.

        Returns:
            str: Time in AM/PM format (e.g., "12:30 PM").
        """

        if minutes < 0 or minutes >= 24 * 60:
            return "Invalid input"

        hours = minutes // 60
        minutes %= 60

        if hours >= 12:
            period = "PM"
            if hours > 12:
                hours -= 12
        else:
            period = "AM"
            if hours == 0:
                hours = 12

        return "{:02d}:{:02d} {}".format(int(hours), int(minutes), period)

    def minutes_to_hours_minutes(self, minutes):
        """
        Converts minutes to hours and minutes format.

        parameters:
            minutes (int): Time in minutes.
```

```
        Returns:
            str: Time in hours and minutes format (e.g., "2 hr 30 min").
        """
        if minutes < 0:
            return "Invalid input"

        hours = minutes // 60
        remaining_minutes = minutes % 60

        if hours == 0:
            return f"{remaining_minutes} min"
        elif remaining_minutes == 0:
            return f"{hours} hr"
        else:
            return f"{hours} hr {remaining_minutes} min"


    def run_scheduler(self):
        """
        Runs the scheduler and prints the scheduled tasks along with their completion ti
        """
        self.times = MaxHeap()
        deleted = []
        for task in self.tasks.heap:
            self.times.heappush(- task[1].starting_time)
        current_time =  - self.times.heap[0]
        self.times.heappop()
        while self.tasks.heap:
          task = self.tasks.get_max()
          while current_time < task.starting_time:
            deleted.append(self.tasks.heap[0])
            self.tasks.heappop()
            if not self.tasks.heap:
              current_time = - self.times.heap[0]
              self.times.heappop()
              for delete in deleted:
                self.tasks.heappush((delete[0] , delete[1]))
              deleted = []
            task = self.tasks.get_max()

          print("⏰" , self.minutes_to_am_pm(current_time))
          current_time += task.duration
          print("\t" , "✅", task.description, "till" , self.minutes_to_am_pm(current_ti
          self.completed_tasks.append(task)
          self.tasks.heappop()
          for task in deleted:
            self.tasks.heappush(task)
          deleted = []









        print("\nAll tasks completed!")

tasks_data_1 = [
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=15 *
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
    Task(1, "Call family", 30, [], status="not_yet_started", starting_time=12 * 60, dead
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
```

```
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(5, "Visit National Museum of Korea", 120, [], status="not_yet_started", startin
    Task(2, "Work on CS110 Assignment", 180, [], status="not_yet_started", starting_time
]

# Create a scheduler and run it
scheduler = ActivityScheduler(tasks_data_1)
scheduler.run_scheduler()

tasks_data_2 = [
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
    Task(1, "Pre class work CS110", 30, [], status="not_yet_started", starting_time=12 *
    Task(2, "Attend on CS110 session", 90, [1], status="not_yet_started", starting_time=
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=13 *
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(5, "Visit Zoo", 90, [], status="not_yet_started", starting_time=14 * 60, deadli
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
]

# Create a scheduler and run it
scheduler = ActivityScheduler(tasks_data_2)
scheduler.run_scheduler()
```

```
⏰ 08:00 AM
        ✅ Prepare and have breakfast till 08:30 AM ,⏱ 30 min
⏰ 10:00 AM
        ✅ Work on CS110 Assignment till 01:00 PM ,⏱ 3 hr
⏰ 01:00 PM
        ✅ Call family till 01:30 PM ,⏱ 30 min
⏰ 03:00 PM
        ✅ Exercise at the gym till 03:45 PM ,⏱ 45 min
⏰ 04:00 PM
        ✅ Cook and have lunch till 04:45 PM ,⏱ 45 min
⏰ 05:00 PM
        ✅ Visit National Museum of Korea till 07:00 PM ,⏱ 2 hr
⏰ 07:00 PM
        ✅ Have dinner at a new local restaurant till 08:00 PM ,⏱ 1 hr
⏰ 08:00 PM
        ✅ Do a city walking tour till 09:30 PM ,⏱ 1 hr 30 min

All tasks completed!
⏰ 08:00 AM
        ✅ Prepare and have breakfast till 08:30 AM ,⏱ 30 min
⏰ 12:00 PM
        ✅ Pre class work CS110 till 12:30 PM ,⏱ 30 min
⏰ 01:00 PM
        ✅ Exercise at the gym till 01:45 PM ,⏱ 45 min
⏰ 02:00 PM
        ✅ Visit Zoo till 03:30 PM ,⏱ 1 hr 30 min
⏰ 04:00 PM
        ✅ Attend on CS110 session till 05:30 PM ,⏱ 1 hr 30 min
⏰ 05:30 PM
        ✅ Cook and have lunch till 06:15 PM ,⏱ 45 min
⏰ 06:15 PM
        ✅ Have dinner at a new local restaurant till 07:15 PM ,⏱ 1 hr
⏰ 08:00 PM
        ✅ Do a city walking tour till 09:30 PM ,⏱ 1 hr 30 min

All tasks completed!
```

In [ ]:

```
# Test case data inputs 1
tasks_data_1 = [
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=15 *
```

```
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
    Task(1, "Call family", 30, [], status="not_yet_started", starting_time=12 * 60, dead
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(5, "Visit National Museum of Korea", 120, [], status="not_yet_started", startin
    Task(2, "Work on CS110 Assignment", 180, [], status="not_yet_started", starting_time
]

# Create a scheduler and run it
scheduler = ActivityScheduler(tasks_data_1)

# Verify the order of completion is the same regardless of the input order
tasks_data_1_reordered = [
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
    Task(1, "Call family", 30, [], status="not_yet_started", starting_time=12 * 60, dead
    Task(2, "Work on CS110 Assignment", 180, [], status="not_yet_started", starting_time
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=15 *
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(5, "Visit National Museum of Korea", 120, [], status="not_yet_started", startin
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
]


# Create a new scheduler with reordered tasks and run it
scheduler_reordered = ActivityScheduler(tasks_data_1_reordered)

# Assert that the order of completed tasks is the same for both schedulers
assert scheduler.completed_tasks == scheduler_reordered.completed_tasks, "Order of compl
```

```
# Test case data inputs 2
tasks_data_2 = [
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
    Task(1, "Pre class work CS110", 30, [], status="not_yet_started", starting_time=12 *
    Task(2, "Attend on CS110 session", 90, [1], status="not_yet_started", starting_time=
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=13 *
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(5, "Visit Zoo", 90, [], status="not_yet_started", starting_time=14 * 60, deadli
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
]

# Create a scheduler and run it
scheduler = ActivityScheduler(tasks_data_2)

# Verify the order of completion is the same regardless of the input order
tasks_data_2_reordered = [
    Task(0, "Prepare and have breakfast", 30, [], status="not_yet_started", starting_tim
    Task(1, "Pre class work CS110", 30, [], status="not_yet_started", starting_time=12 *
    Task(3, "Exercise at the gym", 45, [0], status="not_yet_started", starting_time=13 *
    Task(2, "Attend on CS110 session", 90, [1], status="not_yet_started", starting_time=
    Task(5, "Visit Zoo", 90, [], status="not_yet_started", starting_time=14 * 60, deadli
    Task(4, "Cook and have lunch", 45, [0, 3], status="not_yet_started", starting_time=1
    Task(7, "Have dinner at a new local restaurant", 60, [0, 3, 4], status="not_yet_star
    Task(6, "Do a city walking tour", 90, [], status="not_yet_started", starting_time=20
]
```

```
# Create a new scheduler with reordered tasks and run it
scheduler_reordered = ActivityScheduler(tasks_data_2_reordered)

# Assert that the order of completed tasks is the same for both schedulers
assert scheduler.completed_tasks == scheduler_reordered.completed_tasks, "Order of compl
```

```
# Test case data inputs 3
tasks_data_3 = [
    Task(0, "Go for a morning jog", 45, [], status="not_yet_started", starting_time=6 *
    Task(1, "Read a book", 120, [], status="not_yet_started", starting_time=9 * 60, dead
    Task(2, "Attend online meeting", 60, [], status="not_yet_started", starting_time=12
    Task(3, "Cook and have lunch", 60, [0], status="not_yet_started", starting_time=12 *
    Task(4, "Take a nap", 30, [], status="not_yet_started", starting_time=14 * 60, deadl
    Task(5, "Work on coding project", 180, [2], status="not_yet_started", starting_time=
    Task(6, "Dinner with friends", 90, [1, 3], status="not_yet_started", starting_time=1
    Task(7, "Watch a movie", 150, [], status="not_yet_started", starting_time=21 * 60, d
]

# Create a scheduler and run it
scheduler = ActivityScheduler(tasks_data_3)

# Verify the order of completion is the same regardless of the input order
tasks_data_3_reordered = [
    Task(0, "Go for a morning jog", 45, [], status="not_yet_started", starting_time=6 *
    Task(1, "Read a book", 120, [], status="not_yet_started", starting_time=9 * 60, dead
    Task(2, "Attend online meeting", 60, [], status="not_yet_started", starting_time=12
    Task(5, "Work on coding project", 180, [2], status="not_yet_started", starting_time=
    Task(3, "Cook and have lunch", 60, [0], status="not_yet_started", starting_time=12 *
    Task(4, "Take a nap", 30, [], status="not_yet_started", starting_time=14 * 60, deadl
    Task(6, "Dinner with friends", 90, [1, 3], status="not_yet_started", starting_time=1
    Task(7, "Watch a movie", 150, [], status="not_yet_started", starting_time=21 * 60, d
]


# Create a new scheduler with reordered tasks and run it
scheduler_reordered = ActivityScheduler(tasks_data_3_reordered)

# Assert that the order of completed tasks is the same for both schedulers
assert scheduler.completed_tasks == scheduler_reordered.completed_tasks, "Order of compl
```

In [ ]:

```
import random
import time
import matplotlib.pyplot as plt

import math
import datetime

class Task:
    """
    Task class represents a task with various attributes.
    """
    def __init__(self, id, description, duration, dependencies, status="not_yet_started"
                 starting_time=0, deadline=0, location="Indoor", category="Health", time
        """
        Initializes a Task object with the given attributes.

        parameters:
            id (int): Unique identifier for the task.
            description (str): Description of the task.
```

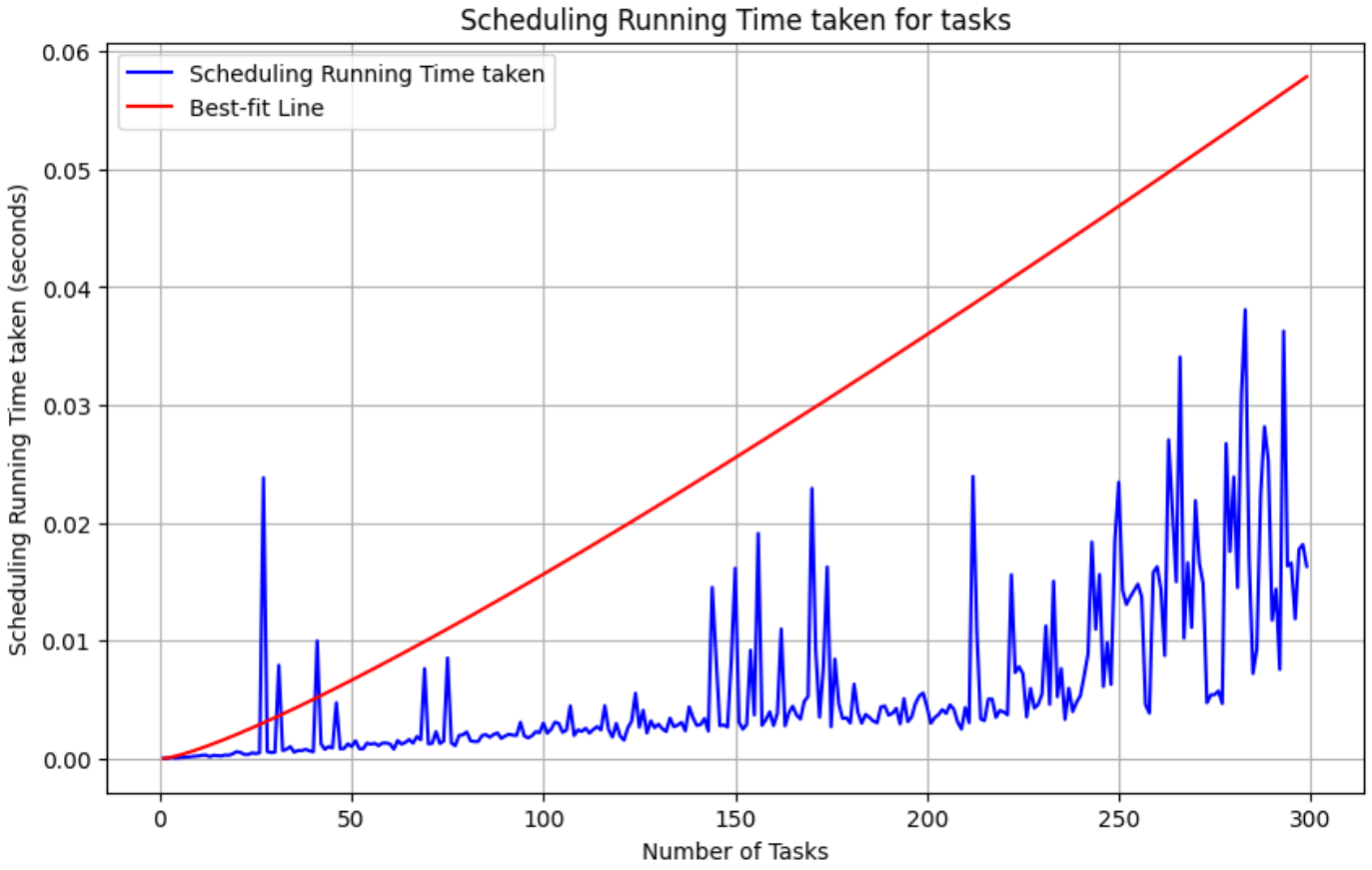
```
        duration (int): Duration of the task in minutes.
        dependencies (list): List of task IDs that this task depends on.
        status (str): Current status of the task (default is "not_yet_started").
        starting_time (int): Starting time of the task in minutes from midnight (def
        deadline (int): Deadline time of the task in minutes from midnight (default
        location (str): Location of the task (default is "Indoor").
        category (str): Category of the task (default is "Health").
        time_sensitivity (str): Time sensitivity of the task (default is "Flexible")
    """
    self.id = id
    self.description = description
    self.duration = duration
    self.dependencies = dependencies
    self.status = status
    self.starting_time = int(starting_time)  # Ensure starting_time is an integer
    self.deadline = int(deadline)  # Ensure deadline is an integer
    self.location = location
    self.category = category
    self.time_sensitivity = time_sensitivity
    self.utility = self.calculate_priority(self.starting_time)

def __lt__(self, other):
    # Compare tasks based on their priority values
    return self.utility < other.utility

def __eq__(self, other):
    """
    Checks if two Task objects are equal.

    parameters:
        other (Task): Another Task object to compare with.

    Returns:
        bool: True if the objects are equal, False otherwise.
    """
    return (
        isinstance(other, Task) and
        self.id == other.id and
        self.description == other.description and
        self.duration == other.duration and
        self.dependencies == other.dependencies and
        self.status == other.status and
        self.starting_time == other.starting_time and
        self.deadline == other.deadline and
        self.location == other.location and
        self.category == other.category and
        self.time_sensitivity == other.time_sensitivity
    )

def calculate_priority(self, current_time):
    """
    Calculates the priority value of the task based on various factors.

    parameters:
        current_time (int): Current time in minutes from midnight.

    Returns:
        float: Priority value of the task.
    """
    base_utility = 0
    duration_weight = -0.1
    dependency_weight = -5
    starting_time_weight = -0.5
    deadline_weight = -3
```

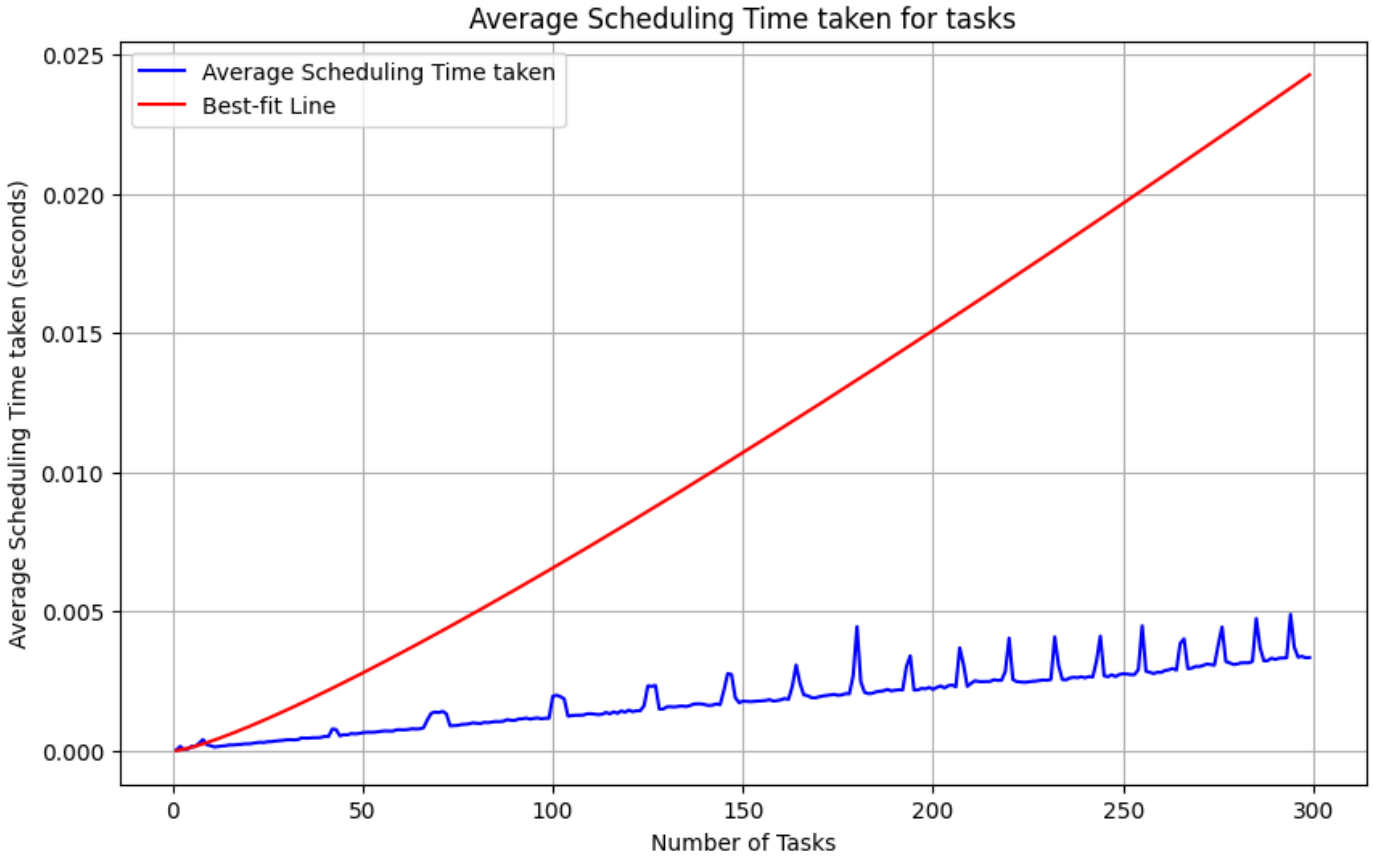
```
        location_weight = 3 if self.location == "Indoor" else 1
        category_weight = 4 if self.category == "Health" else 3 if self.category == "Edu
        time_sensitivity_weight = 5 if self.time_sensitivity == "fixed" else 1

        utility = base_utility + (duration_weight * self.duration) + (dependency_weight
                  + starting_time_weight * (self.starting_time - current_time) \
                  + deadline_weight * (self.deadline - current_time) \
                  + location_weight + category_weight + time_sensitivity_weight

        return utility

class ActivityScheduler:
    """
    ActivityScheduler class schedules tasks based on their priority values.
    """
    def __init__(self, tasks):
        """
        Initializes the scheduler with a list of tasks.

        parameters:
            tasks (list): List of Task objects to be scheduled.
        """
        self.tasks = MaxHeap()
        for task in tasks:
            self.tasks.heappush((task.utility, task))

        self.completed_tasks = []

    def minutes_to_am_pm(self, minutes):
        """
        Converts minutes from midnight to AM/PM format.

        parameters:
            minutes (int): Time in minutes from midnight.

        Returns:
            str: Time in AM/PM format (e.g., "12:30 PM").
        """

        if minutes < 0 or minutes >= 24 * 60:
            return "Invalid input"

        hours = minutes // 60
        minutes %= 60

        if hours >= 12:
            period = "PM"
            if hours > 12:
                hours -= 12
        else:
            period = "AM"
            if hours == 0:
                hours = 12

        return "{:02d}:{:02d} {}".format(int(hours), int(minutes), period)

    def minutes_to_hours_minutes(self, minutes):
        """
        Converts minutes to hours and minutes format.

        parameters:
            minutes (int): Time in minutes.

        Returns:
```

```
            str: Time in hours and minutes format (e.g., "2 hr 30 min").
        """
        if minutes < 0:
            return "Invalid input"

        hours = minutes // 60
        remaining_minutes = minutes % 60

        if hours == 0:
            return f"{remaining_minutes} min"
        elif remaining_minutes == 0:
            return f"{hours} hr"
        else:
            return f"{hours} hr {remaining_minutes} min"


    def run_scheduler(self):
        """
        Runs the scheduler and prints the scheduled tasks along with their completion ti
        """
        self.times = MaxHeap()
        deleted = []
        completed_tasks = []
        for task in self.tasks.heap:
            self.times.heappush(-task[1].starting_time)
        current_time = -self.times.heap[0]
        self.times.heappop()
        while self.tasks.heap:
            task = self.tasks.get_max()
            while current_time < task.starting_time:
                deleted.append(self.tasks.heap[0])
                self.tasks.heappop()
                if not self.tasks.heap:
                    current_time = -self.times.heap[0]
                    self.times.heappop()
                    for delete in deleted:
                        self.tasks.heappush((delete[0], delete[1]))
                        if delete[1].status != "completed":
                            completed_tasks.append(delete[1])
                    deleted = []
                task = self.tasks.get_max()
            current_time += task.duration
            self.completed_tasks.append(task)
            self.tasks.heappop()
            for task in deleted:
                self.tasks.heappush(task)
            deleted = []



def generate_random_tasks(num_tasks):
    tasks = []
    for i in range(num_tasks):
        task_id = i
        description = f"Task {i + 1}"
        duration = random.randint(30, 180)
        max_dependencies = min(i, 5)
        num_dependencies = random.randint(0, max_dependencies)
        dependencies = random.sample(range(i), num_dependencies)
        starting_time = random.randint(0, 1440 - duration)
        deadline = starting_time + duration + random.randint(0, 1440 - starting_time - d
        location = random.choice(["Indoor", "Outdoor"])
        category = random.choice(["Health", "Education", "Cultural", "Personal"])
        time_sensitivity = random.choice(["Flexible", "Fixed"])
```

```
        # Create Task instances and append them to the tasks list
        task_instance = Task(
            id=task_id,
            description=description,
            duration=duration,
            dependencies=dependencies,
            starting_time=starting_time,
            deadline=deadline,
            location=location,
            category=category,
            time_sensitivity=time_sensitivity,
            status="not_yet_started"
        )
        tasks.append(task_instance)
    return tasks




#plotting input size with running time without taking average by setting iterations to o

# Variables for storing input size (number of tasks) and corresponding time taken for sc
input_sizes = []
scheduling_times_taken = []
best_fit_line = []

# Number of iterations to calculate the average scheduling time for each input size
num_iterations = 1

# Generate tasks for different input sizes and calculate the average scheduling time tak
for num_tasks in range(1, 300):  # Generating tasks for 1 to 1000 tasks
    total_scheduling_time = 0

    # Run the scheduling process multiple times to calculate the average time
    for _ in range(int(num_iterations)):
        tasks = generate_random_tasks(num_tasks)  # Generate tasks
        scheduler = ActivityScheduler(tasks)

        start_time = time.time()  # Record the start time
        scheduler.run_scheduler()  # Run the scheduling algorithm
        end_time = time.time()  # Record the end time

        total_scheduling_time += end_time - start_time  # Accumulate the total schedulin

    # Calculate the average scheduling time for the current input size and append to the
    average_scheduling_time = total_scheduling_time / num_iterations
    input_sizes.append(num_tasks)
    scheduling_times_taken.append(average_scheduling_time)

for num_tasks in range(1, 300):
  best_fit_line.append(num_tasks * math.log10(num_tasks) * scheduling_times_taken[2])


# Plotting the graph
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, scheduling_times_taken, color='b', label='Scheduling Running Time
plt.plot(input_sizes, best_fit_line, color='r', label='Best-fit Line')
plt.xlabel('Number of Tasks')
plt.ylabel('Scheduling Running Time taken (seconds)')
plt.title('Scheduling Running Time taken for tasks')
plt.legend()
plt.grid(True)
```

```
plt.show()



#plotting input size with average running time by taking average by setting iterations t



# Variables for storing input size (number of tasks) and corresponding time taken for sc
input_sizes = []
scheduling_times_taken = []
best_fit_line = []

# Number of iterations to calculate the average scheduling time for each input size
num_iterations = 200

# Generate tasks for different input sizes and calculate the average scheduling time tak
for num_tasks in range(1, 300):  # Generating tasks for 1 to 1000 tasks
    total_scheduling_time = 0

    # Run the scheduling process multiple times to calculate the average time
    for _ in range(int(num_iterations)):
        tasks = generate_random_tasks(num_tasks)  # Generate tasks
        scheduler = ActivityScheduler(tasks)

        start_time = time.time()  # Record the start time
        scheduler.run_scheduler()  # Run the scheduling algorithm
        end_time = time.time()  # Record the end time

        total_scheduling_time += end_time - start_time  # Accumulate the total schedulin

    # Calculate the average scheduling time for the current input size and append to the
    average_scheduling_time = total_scheduling_time / num_iterations
    input_sizes.append(num_tasks)
    scheduling_times_taken.append(average_scheduling_time)

for num_tasks in range(1, 300):
  best_fit_line.append(num_tasks * math.log10(num_tasks) * scheduling_times_taken[2])


# Plotting the graph
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, scheduling_times_taken, color='b', label='Average Scheduling Time
plt.plot(input_sizes, best_fit_line, color='r', label='Best-fit Line')
plt.xlabel('Number of Tasks')
plt.ylabel('Average Scheduling Time taken (seconds)')
plt.title('Average Scheduling Time taken for tasks')
plt.legend()
plt.grid(True)
plt.show()
```





In [ ]:
```
#just for prinitng the file
!jupyter nbconvert --to html /content/Project_1_Designing_a_Schedulereiad.ipynb
```

```
[NbConvertApp] Converting notebook /content/Project_1_Designing_a_Schedulereiad.ipynb to
html
[NbConvertApp] Writing 1668836 bytes to /content/Project_1_Designing_a_Schedulereiad.htm
l
```