Project 1 - Designing a Scheduler [A day in the life of a Minervan Part I]

Minerva University

CS110 - Problem Solving with Data Structures and Algorithms

Prof. Ribeiro

March 18, 2025

Project 1 - Designing a Scheduler [A day in the life of a Minervan Part I]

Setting up

id	description	duration	dependencies	time constraint	type of task
1	SS110 Session	90 minutes		10 AM	regular
2	Brunch	30 minutes			regular
3	SS110 Group Assignment Meeting	240 minutes	2	12 PM	regular
4	Take a bus/train to Nodeul Island	60 minutes		4 PM	LBA: Nodeul Island is a small artificial island in the Han River. It has been transformed into a cultural complex featuring music venues, bookstores, cafes, and green spaces. I like the fact that it is not tourist-heavy area, and you can actually meet many locals there.
5	Plan trip to Jeju	120 minutes	4		LBA: Jeju is South Korea's largest island, and I am planning to visit it for couple of days with my friends during the break. We want to create an itinerary beforehand, with a detailed daily to-do list, so that we won't miss out on important spots. Since Jeju has a distinctive local identity and is completely different from big cities like Busan and Seoul, we

					definitely have to be prepared for cultural immersion. Looking up must-visit places, landmarks, local festivals and activities in Jeju will definitely help me to expand my knowledge about South Korea.
6	Dinner at Korean BBQ restaurant	90 minutes	5	7 PM	LBA: I have never tried Korean BBQ before, and I always wanted to go there with my friends. I know that KBBQ places offer a variety of meats, side dishes and dipping sauces, and it's a social experience where you cook food for yourself. People usually go in groups to bond over a meal. I am excited to finally try something I have only seen in K-dramas!
7	Work Study	60 minutes			regular
8	Apply to internships	60 minutes			regular

Algorithmic strategy

https://drive.google.com/file/d/1kGNonWrlq4g5KZf4qY9UntMAKrDVFemu/view?usp=sharing

Python implementation

A. See Appendix III (A) for the implementation of MaxHeapq and Appendix III (B) for the implementation of MinHeapq. Test cases are in Appendix III (C) and Appendix III (D).

B. See Appendix III (E) for the activity scheduler implementation. Test cases are provided in Appendix III (F).

C. See Appendix III (G), where I showed how my scheduler prioritizes tasks based on their priority value and changing the order of the input tasks yields the same output.

D. The scheduler is designed to produce a feasible schedule based on the given constraints and dependencies. However, its feasibility depends heavily on the quality of the input data (e.g., realistic time constraints, valid dependencies). While test-driving the code, potential issues such as circular dependencies, unrealistic constraints, and performance bottlenecks may arise. Addressing these challenges through input validation, error handling, and performance optimization will improve the scheduler's robustness and practicality.

Test Drive

https://drive.google.com/file/d/12DmyZ5DSVPbEd3SmmmeCXr6J3uc76IBS/view?usp=sharing

The code for test drive is provided in Appendix III (H).

Output:

```
Running the scheduler:

🕰 t=10h00
        Started 'SS110 Session' for 90 mins...
        ✅ t=11h30, task completed!
🕰 t=11h30
        Started 'Brunch' for 30 mins...
        ✅ t=12h00, task completed!
🕰 t=12h00
        Started 'SS110 Group Assignment Meeting' for 210 mins...
        ✅ t=15h30, task completed!
⏳ Waiting 30 minutes until 16h00
🕰 t=16h00
        Started 'Take a bus/train to Nodeul Island' for 60 mins...
        ✅ t=17h00, task completed!
🕰 t=17h00
        Started 'Plan trip to Jeju' for 120 mins...
        ✅ t=19h00, task completed!
🕰 t=19h00
        Started 'Dinner at Korean BBQ restaurant' for 90 mins...
        ✅ t=20h30, task completed!
🕰 t=20h30
        Started 'Work Study' for 60 mins...
        ✅ t=21h30, task completed!
🕰 t=21h30
        Started 'Apply to internships' for 60 mins...
        ✅ t=22h30, task completed!

🏁 Completed all planned tasks in 12h30!
```

Analysis

A. The scheduler provided a clear and structured plan for executing tasks, ensuring that fixed-time tasks (e.g., “SS110 Session”, “SS110 Group Assignment Meeting”) are completed at their exact scheduled times. It also handled task dependencies well, because tasks requiring collaboration (e.g., planning a trip to Jeju, going to Nodeul Island, and having dinner with friends) were executed in sequence, preventing the interruptions in my plans with my friends. Using this scheduler during the packed days would be especially helpful because it minimizes total time spent by filling gaps between fixed-time tasks with flexible tasks. For instance, it scheduled “Brunch” immediately after the “SS110 Session” so that no time was wasted.

Despite its advantages, the scheduler has several failure modes during the test drive. When entering your list of tasks, you have to account for every task because forgetting to input only one task can drastically change the scheduler output. For example, I forgot to include necessary transitions between locations in my list such as traveling from Nodeul Island to the Korean BBQ restaurants, which led to delays in the schedule. We arrived at the Korean BBQ restaurant at 8 PM instead of 7 PM, shifting subsequent tasks forward. Consequently, work-study tasks began later than expected, pushing the internship application task beyond feasible working hours.

The schedule assumed that all tasks could be completed sequentially without accounting for physical and mental fatigue. As a result, the internship application task was skipped due to exhaustion.

One more potential failure could be the fact that the scheduler assumes that fixed-time tasks (e.g. taking a bus) will always be completed on time. However, real-world factors like missing a bus, bus delay can disrupt the schedule.

Also, the scheduler does not allow for real-time adjustments during execution. For example, if a task takes longer than expected, it cannot dynamically reschedule subsequent tasks.

B. We can use several metrics to evaluate the general efficiency of the scheduler:

- Idle time - the total time spent waiting between tasks. It measures how effectively the scheduler utilizes available time.

  $Idle\ time\ =\ total\ time\ available\ -\ total\ time\ spent\ on$

  Total time available: 12.5 hours (750 minutes)

  Total time spent on tasks: 90 + 30 + 210 + 60 + 120 + 90 + 60 + 60 = 720

  Idle time = 810 - 720 = 30 minutes

  Only 30 minutes of idle time is observed, indicating that the scheduler efficiently uses time and generates well-optimized schedule.

- Dependency satisfaction rate - the percentage of tasks that are executed only after their dependencies are satisfied.

  $Dependency\ satisfaction\ rate\ =\ (\frac{number\ of\ tasks\ with\ satisfied\ dependencies}{total\ number\ of\ tasks\ with\ dependencies})\ \times\ 100\%$

  We have 3 tasks with dependencies: SS110 Group Assignment Meeting , Plan trip to Jeju, and Dinner at Korean BBQ restaurant. All of them are executed only after their dependencies are satisfied, so dependency satisfaction rate is 100%

- Fixed-time task adherence - the percentage of fixed-time tasks executed at their exact scheduled times. It measures how well the scheduler handles fixed-time constraints.

  $Fixed\ -\ time\ task\ adherence\ =\ (\frac{number\ of\ fixed-time\ tasks\ executed\ on\ time}{total\ number\ of\ fixed-time\ tasks})\ \times\ 100\%$

  3 out of 3 fixed-time tasks (SS110 Session, SS110 Group Assignment Meeting, Dinner at Korean BBQ restaurant) are executed at their exact scheduled times, resulting in 100% fixed-time task adherence rate.

The scheduler demonstrates high efficiency based on the defined metrics. However, it could be improved by incorporating buffer time to account for delays and including real-time adjustments to handle unexpected disruptions.

C. Theoretical time complexity

1. *build_dependents_lookup:*
   - Outer loop: iterates over all tasks (n iterations).
   - Inner loop: iterates over the dependencies of each task. In the worst case, each task has d dependencies, where d is the maximum number of dependencies per task.
   - The *if dep in dependents* check and *append* operation are O(1)
   - Time complexity:
     - If there are n tasks and each task has on average d dependencies, the complexity is $O(n \times d)$. If d is a small constant, we can neglect it, so the complexity would be $O(n)$.
     - In the worst case, each task depends on all others ($d = n$), so the complexity is $O(n^2)$.
2. *calculate_priority:*
   - The *priority_cache* ensures that each task's priority is computed only once, reducing redundant calculations.
   - *find_earliest_fixed* recursively traverses the dependency graph to find the earliest fixed-time constraint.

- *get_dependent_depth* recursively computes the depth of the dependency chain.
- Base priority calculation involves simple arithmetic operations ($O(1)$).
- Time complexity:
  - Each task's priority is computed once due to caching, so the total number of recursive calls is $O(n)$.
  - For each tasks, the *find_earliest_fixed* and *get_dependent_depth* functions traverse the dependency graph, which has d dependencies in total.
    - In average case (assuming a shallower dependency tree), the complexity is $O(n \times d)$. If the dependency graph is sparse, it leads to $O(1)$ recursive calls per task, resulting in time complexity of $O(n)$
    - In worst case, recursive calls traverse all n tasks per task, leading to $O(n \times n) = O(n^2)$

3. *initialize_queues:*
   - my scheduler uses two heaps (a min-heap for fixed-time tasks and a max-heap for flexible tasks). For each heap, inserting an element, removing an element and heapifying the tree takes $O(log\ n)$ per operation (because the height of the tree is $log\ n$). Since every task is inserted and later extracted, th total cost of heap operations: $O(n\ log\ n)$
   - *calculate_priority* is called for flexible tasks.

- Total time complexity:
    - Average case: $O(n\, log\, n) + O(n) = O(n\, log\, n)$.
    - Worst case: $O(n\, log\, n) + O(n^2) = O(n^2)$.

4. *run_task_scheduler:*
    - Runs until all tasks in both queues are processed (n times).
    - *peek, heappop* and *heappush* are $O(log\, n)$ per operation
    - Total time complexity (both for average-case and worst-case scenarios): $O(n\, log\, n)$.

Overall, the theoretical time complexity of the scheduler is:

- Worst-case (dense inter-dependency):
    - $O(n^2) + O(n^2) + O(n^2) + O(n\, log\, n) = O(n^2 + n^2 + n^2 + n\, log\, n) = O(n^2)$
- Average-case (sparse inter-dependency):
    - $O(n) + O(n) + O(n\, log\, n) + O(n\, log\, n) = O(n\, log\, n)$

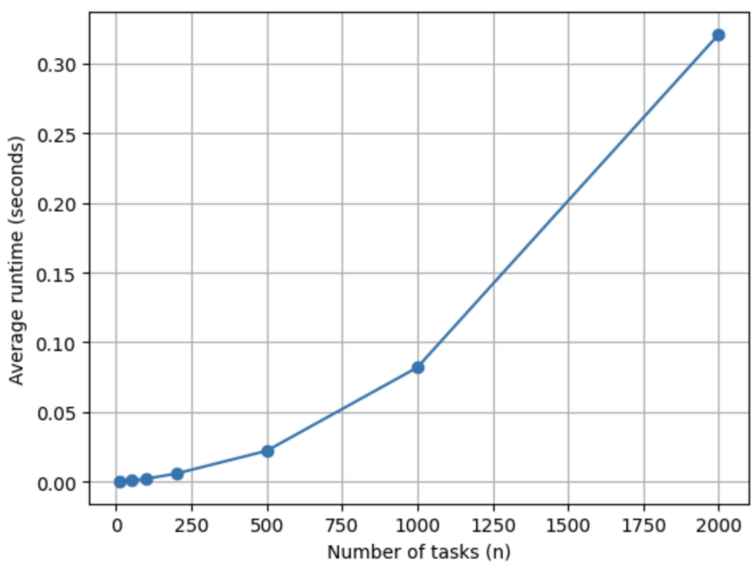

Figure 1. *Scheduler runtime vs input size*

I performed an experiment (See Appendix III (I)) to confirm my theoretical time complexity analysis (Fig. 1). As we can see from Fig. 2, the theoritical curve of $O(n^2)$ best approximates my experimental data. It means that the average runtime scales quadratically as the input size increases, which aligns with worst-case time complexity, $O(n^2)$. Even though the random task generator is designed to create some tasks with few dependencies, the observed quadratic behavior implies that the average or worst-case dependency chains are long enough to make the quadratic term $O(n^2)$ dominate. Essentially, the average number of dependencies per task is high enough that the cost of processing these dependencies (in both building the dependents lookup and in computing task priorities) is not negligible.

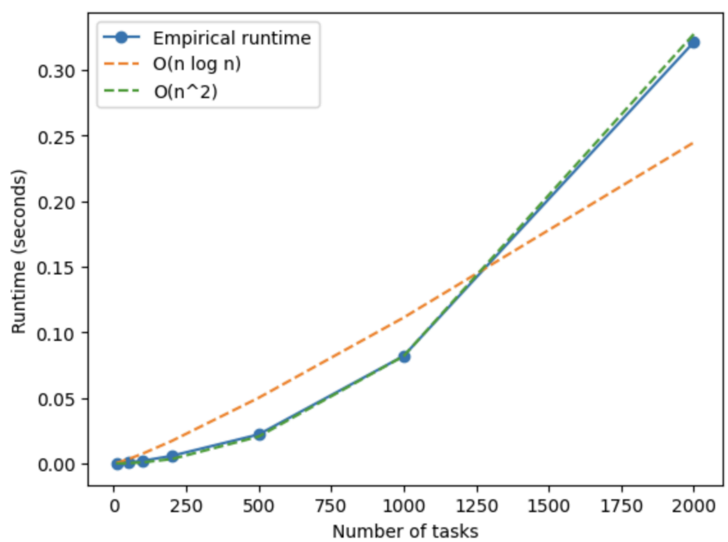

Figure 2. *Fitting empirical runtime into theoretical curves*

Getting back to the board. (Word count: 266 words)

A primary bottleneck in the current design is the recursive priority calculation and dependency management, which can lead to $O(n^2)$ time complexity in worst-case scenarios. One improvement is to replace current dependency representation with an adjacency list or graph data structure, which allows for efficient traversal and lookup of dependencies.

Moreover, currently, tasks are scheduled at the beginning and executed in order. However, real-world schedules change dynamically. Therefore, we should implement a dynamic scheduling mechanism to handle real-time changes in task dependencies or priorities, which guarantees that scheduler remains efficient in real-world scenarios.

Some tasks don't need your full attention. You could be listening to a podcast while cooking or downloading a file while writing an email. Right now, the scheduler treats everything as if it has to happen one after another. We can use parallel processing to execute multiple tasks simultaneously for tasks with no dependencies. It reduces the execution times, improving resource utilization.

Right now, if a long task is scheduled, it has to be finished all at once before moving on to something else. But what if a more urgent task pops up? The scheduler could use a pause-and-resume system, so if you need to stop working on a big project to take an important call, you can pick up where you left off later.

Moreover, if all tasks cannot fit within the given schedule, the scheduler should calculate and suggest an optimal wake-up time. The system can determine the earliest feasible wake-up time needed to complete all tasks before deadlines by summing up task durations and gaps between fixed event.

Appendices

Part I: Learning and Growth Reflection

The session on heap data structures and priority queues helped me a lot to complete this assignment. Understanding how max-heaps and min-heaps efficiently manage priorities helped me design a scheduler that dynamically assigns task priorities based on dependencies, duration, and fixed-time constraints. I am grateful for my professor for mentioning that the sessions on heaps are important because session activities allowed me to deeply engage with material (67 words)

Part II: LO and HC applications

A. **#cs110-AlgoStratDataStruct:** The scheduler applied algorithmic techniques and data structures by using priority queues (MinHeap and MaxHeap) to manage task scheduling. The min-heap manages fixed-time tasks efficiently, while the max-heap prioritizes flexible tasks based on dependencies. The scheduler uses greedy algorithms to prioritize tasks based on dependencies and fixed-time constraints. (48 words)

#cs110-ComplexityAnalysis: I determined the asymptotic behavior of my scheduler's core operations for average and worst case input, used Big-O notation to describe its time complexity and confirmed finding through empirical result. Addressing previous feedback, I corrected notation (O(n) instead of O(N)) used additional plot to validate my empirical result. (50 words)

#cs110-ComputationalCritique: I contrasted heap-based priority queues with alternative structures (e.g. lists) for task managements. Heaps were chosen for their efficient $O(log\, n)$

operations (insertion, deletion, heapifying), while lists require $O(n)$. I also included reflection on what we can improve on to reduce the average runtime for large inputs. (50 words)

#cs110-CodeReadability: The code was structured to be clear and concise, with meaningful comments explaining key logic (e.g., priority calculation, heap operations). Variable names like heap, heap_size and dependents_lookup were chosen for clarity. Consistent naming improve readability, while docstrings make the code understandable for external users. (45 words)

#cs110-PythonProgramming: Python was used to implement the scheduler, including algorithms for heap operations, dependency resolution, and priority calculation. The code also included functionality to plot performance metricsusing matplotlib. The visualization helped validate the theoretical time complexity ($O(n^2)$ for worst case) and demonstrated the scheduler's scalability for large task sets. (50 words)

#professionalism: The work was presented professionally, adhering to established guidelines for code formatting, documentation, and reporting. The code included detailed docstrings, and was organized into modular classes and functions. The analysis was presented clearly, with theoretical and empirical results contrasted in a structured manner, and visualizations were used to enhance understanding. (50 words)

B. **#algorithms:** Key algorithms included heap operations for task prioritization and a recursive priority calculation for flexible tasks. I have specified input, output and steps taken and justified the choice of using heap data structure in my video. I also proposed improvements like using adjacency lists, parallel processing, etc. (47 words)

Part III: Python code

A. MaxHeapq

```python
# This code was adapted from Session 13 - [7.2] Heaps and priority queues
PCWbook

class MaxHeapq:
   """
   A class that implements properties and methods
   that support a max priority queue data structure

   Attributes
   ----------
   heap : arr
       A Python list where key values in the max heap are stored
   heap_size: int
       An integer counter of the number of keys present in the max heap
   """

   def __init__(self):
       """
       Parameters
       ----------
       None
       """
       self.heap       = []
       self.heap_size  = 0

   def left(self, i):
       """
       Takes the index of the parent node
       and returns the index of the left child node

       Parameters
       ----------
       i: int
         Index of parent node
```

```
        Returns
        ----------
        int
          Index of the left child node
        """
        return 2 * i + 1

    def right(self, i):
        """
        Takes the index of the parent node
        and returns the index of the right child node

        Parameters
        ----------
        i: int
            Index of parent node

        Returns
        ----------
        int
            Index of the right child node
        """
        return 2 * i + 2

    def parent(self, i):
        """
        Takes the index of the child node
        and returns the index of the parent node

        Parameters
        ----------
        i: int
            Index of child node

        Returns
        ----------
```

```
        int
            Index of the parent node
        """

        return (i - 1)//2


    def heappush(self, task):
        """
        Insert a key into a priority queue

        Parameters
        ----------
        key: int
            The key value to be inserted

        Returns
        ----------
        None
        """
        if not isinstance(task, Task):
          raise TypeError("Only Task objects can be added to the heap")
        self.heap.append(task)
        self.heap_size+=1
        self.increase_key(self.heap_size - 1)

    def increase_key(self, i):
        """
        Modifies the value of a key in a max priority queue
        with a higher value

        Parameters
        ----------
        i: int
            The index of the key to be modified
        key: int
            The new key value
```

```
        Returns
        ----------
        None
        """
        while i > 0 and self.heap[self.parent(i)].priority < self.heap[i].priority:
            self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]
            i = self.parent(i)

    def heapify(self, i):
        """
        Creates a max heap from the index given

        Parameters
        ----------
        i: int
            The index of of the root node of the subtree to be heapify

        Returns
        ----------
        None
        """
        l = self.left(i)
        r = self.right(i)
        heap = self.heap
        largest = i
        if l <= (self.heap_size-1) and heap[l].priority > heap[i].priority:
            largest = l
        else:
            largest = i
        if r <= (self.heap_size-1) and heap[r].priority > heap[largest].priority:
            largest = r
        if largest != i:
            heap[i], heap[largest] = heap[largest], heap[i]
```

```
            self.heapify(largest)

    def heappop(self):
        """
        Returns the largest key in the max priority queue
        and removes it from the max priority queue

        Parameters
        ----------
        None

        Returns
        ----------
        int
            the max value in the heap that is extracted
        """
        if self.heap_size < 1:
                raise ValueError('Heap underflow: There are no keys in the
priority queue ')
        max_task = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self.heap_size-=1
        self.heapify(0)
        return max_task


    def is_empty(self):
      """
      Returns True if the heap is empty, otherwise False.
      """
      return self.heap_size == 0
```

B. MinHeapq

```
# This code was adapted from Session 13 - [7.2] Heaps and priority queues
PCWbook

class MinHeapq:
   """
   A class that implements properties and methods
   that support a min priority queue data structure.

   Attributes
   ----------
   heap : list
       A Python list where key values in the min heap are stored.
   heap_size : int
       An integer counter of the number of keys present in the min heap.
   """

   def __init__(self):
       """
       Parameters
       ----------
       None
       """
       self.heap = []
       self.heap_size = 0

   def left(self, i):
       """
       Takes the index of the parent node
       and returns the index of the left child node.

       Parameters
       ----------
       i: int
           Index of the parent node.

       Returns
       ----------
```

```
        int
            Index of the left child node.
        """
        return 2 * i + 1

    def right(self, i):
        """
        Takes the index of the parent node
        and returns the index of the right child node.

        Parameters
        ----------
        i: int
            Index of the parent node.

        Returns
        ----------
        int
            Index of the right child node.
        """
        return 2 * i + 2

    def parent(self, i):
        """
        Takes the index of the child node
        and returns the index of the parent node.

        Parameters
        ----------
        i: int
            Index of the child node.

        Returns
        ----------
        int
            Index of the parent node.
        """
```

```
        return (i - 1) // 2

    def heappush(self, task):
        """
        Insert a key into a priority queue.

        Parameters
        ----------
        task: Task
            The Task object to be inserted.

        Returns
        ----------
        None
        """
        if not isinstance(task, Task):
            raise TypeError("Only Task objects can be added to the heap")
        self.heap.append(task)
        self.heap_size += 1
        self.decrease_key(self.heap_size - 1)

    def decrease_key(self, i):
        """
        Modifies the value of a key in a min priority queue
        with a lower value to maintain the min-heap property.

        Parameters
        ----------
        i: int
            The index of the key to be modified.

        Returns
        ----------
        None
        """
        while i > 0 and self.heap[self.parent(i)].time_constraint >
self.heap[i].time_constraint:
```

```
                              self.heap[i],   self.heap[self.parent(i)]   =
self.heap[self.parent(i)], self.heap[i]
           i = self.parent(i)

   def heappop(self):
       """
       Returns the smallest key in the min priority queue
       and removes it from the min priority queue.

       Parameters
       ----------
       None

       Returns
       ----------
       Task
           The Task object with the smallest time constraint.
       """
       if self.heap_size < 1:
               raise ValueError("Heap underflow: No tasks in the priority
queue")
       min_task = self.heap[0]
       self.heap[0] = self.heap[-1]
       self.heap.pop()
       self.heap_size -= 1
       self._heapify_down(0)
       return min_task

   def _heapify_down(self, i):
       """
       Creates a min heap from the index given.

       Parameters
       ----------
       i: int
           The index of the root node of the subtree to be heapified.
```

```
        Returns
        ----------
        None
        """
        smallest = i
        l = self.left(i)
        r = self.right(i)

        if l < self.heap_size and self.heap[l].time_constraint <
self.heap[smallest].time_constraint:
            smallest = l
        if r < self.heap_size and self.heap[r].time_constraint <
self.heap[smallest].time_constraint:
            smallest = r
        if smallest != i:
            self.heap[i], self.heap[smallest] = self.heap[smallest],
self.heap[i]
            self._heapify_down(smallest)

    def is_empty(self):
        """
        Returns True if the heap is empty, otherwise False.

        Parameters
        ----------
        None

        Returns
        ----------
        bool
            True if the heap is empty, otherwise False.
        """
        return self.heap_size == 0

    def peek(self):
        """
```

```
        Returns the Task with the smallest time constraint without removing
it.

        Parameters
        ----------
        None

        Returns
        ----------
        Task
            The Task object with the smallest time constraint.
        """
        if self.heap_size < 1:
                raise ValueError("Heap underflow: No tasks in the priority
queue")
        return self.heap[0]
```

C. Test cases for MaxHeapq

```
# test cases for MaxHeapq

def test_max_heapq_basic():
    """
    Test basic functionality of MaxHeapq:
    - Insert elements and ensure the max element is always at the root.
    - Remove elements and ensure the heap maintains the max-heap property.
    """
    max_heap = MaxHeapq()
    tasks = [
            Task(id=1, description="Task A", duration=30, dependencies=[],
priority=5),
            Task(id=2, description="Task B", duration=20, dependencies=[],
priority=10),
            Task(id=3, description="Task C", duration=10, dependencies=[],
priority=3),
    ]
```

```
    for task in tasks:
        max_heap.heappush(task)

    # check the max element
    assert max_heap.heappop().priority == 10  # Task B
    assert max_heap.heappop().priority == 5   # Task A
    assert max_heap.heappop().priority == 3   # Task C
    assert max_heap.is_empty()  # heap should be empty

def test_max_heapq_heap_property():
    """
    Test that the max-heap maintains its structure:
    - Insert elements in random order.
    - Ensure the max element is always at the root after each insertion.
    """
    max_heap = MaxHeapq()
    tasks = [
            Task(id=1, description="Task A", duration=30, dependencies=[],
priority=7),
            Task(id=2, description="Task B", duration=20, dependencies=[],
priority=3),
            Task(id=3, description="Task C", duration=10, dependencies=[],
priority=10),
            Task(id=4, description="Task D", duration=15, dependencies=[],
priority=5),
    ]
    for task in tasks:
        max_heap.heappush(task)
         assert max_heap.heap[0].priority == max(task.priority for task in
max_heap.heap)  # max element at root

    # remove elements and check heap property
    assert max_heap.heappop().priority == 10  # Task C
    assert max_heap.heappop().priority == 7   # Task A
    assert max_heap.heappop().priority == 5   # Task D
    assert max_heap.heappop().priority == 3   # Task B
    assert max_heap.is_empty()  # heap should be empty
```

```
def test_max_heapq_single_element():
    """
    Test the max-heap with a single element:
    - Insert one element and ensure it is returned when popped.
    - Ensure the heap is empty after popping.
    """
    max_heap = MaxHeapq()
     task = Task(id=1, description="Task A", duration=30, dependencies=[],
priority=5)
    max_heap.heappush(task)

    assert max_heap.heappop().priority == 5  # Task A
    assert max_heap.is_empty()  # heap should be empty


test_max_heapq_basic()
test_max_heapq_heap_property()
test_max_heapq_single_element()
```

D. Test cases for MinHeapq

```
# test cases for MinHeapq

def test_min_heapq_basic():
    """
    Test basic functionality of MinHeapq:
    - Insert elements and ensure the min element is always at the root.
    - Remove elements and ensure the heap maintains the min-heap property.
    """
    min_heap = MinHeapq()
    tasks = [
            Task(id=1, description="Task A", duration=30, dependencies=[],
time_constraint=60),
            Task(id=2, description="Task B", duration=20, dependencies=[],
time_constraint=30),
```

```
        Task(id=3, description="Task C", duration=10, dependencies=[],
time_constraint=90),
    ]
    for task in tasks:
        min_heap.heappush(task)

    # check the min element
    assert min_heap.heappop().time_constraint == 30  # Task B
    assert min_heap.heappop().time_constraint == 60  # Task A
    assert min_heap.heappop().time_constraint == 90  # Task C
    assert min_heap.is_empty()  # heap should be empty

def test_min_heapq_heap_property():
    """
    Test that the min-heap maintains its structure:
    - Insert elements in random order.
    - Ensure the min element is always at the root after each insertion.
    """
    min_heap = MinHeapq()
    tasks = [
        Task(id=1, description="Task A", duration=30, dependencies=[],
time_constraint=60),
        Task(id=2, description="Task B", duration=20, dependencies=[],
time_constraint=30),
        Task(id=3, description="Task C", duration=10, dependencies=[],
time_constraint=90),
        Task(id=4, description="Task D", duration=15, dependencies=[],
time_constraint=45),
    ]
    for task in tasks:
        min_heap.heappush(task)
        assert min_heap.heap[0].time_constraint == min(task.time_constraint
for task in min_heap.heap)  # Min element at root

    # remove elements and check heap property
    assert min_heap.heappop().time_constraint == 30  # Task B
    assert min_heap.heappop().time_constraint == 45  # Task D
```

```
    assert min_heap.heappop().time_constraint == 60  # Task A
    assert min_heap.heappop().time_constraint == 90  # Task C
    assert min_heap.is_empty()  # heap should be empty


def test_min_heapq_single_element():
    """
    Test the min-heap with a single element:
    - Insert one element and ensure it is returned when popped.
    - Ensure the heap is empty after popping.
    """
    min_heap = MinHeapq()
     task = Task(id=1, description="Task A", duration=30, dependencies=[],
time_constraint=60)
    min_heap.heappush(task)

    assert min_heap.heappop().time_constraint == 60  # Task A
    assert min_heap.is_empty()  # heap should be empty

test_min_heapq_basic()
test_min_heapq_heap_property()
test_min_heapq_single_element()
```

E. Activity scheduler implementation

```
# This code was adapted from Session 13 - [7.2] Heaps and priority queues
Breakout Workbook

class Task:
    """
    - id: Task id (a reference number)
    - description: Task short description
    - duration: Task duration in minutes
    - dependencies: List of task ids that need to precede this task
    - status: Current status of the task
    - time_constraint: Scheduled start time for fixed-time tasks
```

```
    - priority: Computed priority value for flexible tasks
    """
    def __init__(self, id, description, duration, dependencies, status="N",
time_constraint=None, priority=None):
        self.id = id
        self.description = description
        self.duration = duration
        self.dependencies = dependencies
        self.status = status
        self.time_constraint = time_constraint
        self.priority = priority

    def __lt__(self, other):
         # for fixed-time tasks (MinHeapq): earlier tasks have higher
priority.
        if self.time_constraint is not None:
            return self.time_constraint < other.time_constraint
         # for flexible tasks (MaxHeapq): higher computed priority means
higher priority.
        else:
            return self.priority > other.priority


class TaskScheduler:
    """
     A scheduler that manages tasks with stict execution times for fixed
tasks
    and priority-based scheduling for flexible tasks.

    Attributes
    ----------
    tasks : list
        List of Task objects to be scheduled
    task_lookup: dict
        Dictionary mapping task IDs to Task objects
    dependents_lookup: dict
        Dictionary mapping task IDs to lists of tasks that depend on them
```

```
    priority_cache: dict
        Cache for storing computed priority values of tasks
    fixed_queue: MinHeapq
        Min-heap priority queue for fixed-time tasks
    flexible_queue: MaxHeapq
        Max-heap priority queue for flexible tasks
    """

    NOT_STARTED = 'N'
    IN_PROGRESS = 'I'
    COMPLETED = 'C'

    def __init__(self, tasks):
        self.tasks = tasks
        self.task_lookup = {task.id: task for task in tasks}
        self.dependents_lookup = self.build_dependents_lookup()
        self.priority_cache = {}
        self.fixed_queue = MinHeapq()
        self.flexible_queue = MaxHeapq()
        self.initialize_queues()

    def build_dependents_lookup(self):
        """
         Builds a mapping from task ID to the list of tasks that depend on
it.

        Returns
        -------
        dist
               Dictionary where keys are task IDs and values are lists of
dependent
            tasks.
        """
        dependents = {task.id: [] for task in self.tasks}
        for task in self.tasks:
            for dep in task.dependencies:
                if dep in dependents:
```

```
                dependents[dep].append(task)
        return dependents

    def initialize_queues(self):
        """
         Sorts tasks into fixed or flexible queues based on their time
constraints.
         Computes priorities for flexible tasks and pushes them into the
appropriate
        queues.
        """
        for task in self.tasks:
            if task.time_constraint:
                self.fixed_queue.heappush(task)
            else:
                task.priority = self.calculate_priority(task)
                self.flexible_queue.heappush(task)

    def calculate_priority(self, task):
        """
        Recursively computes the priority value for a flexible task.
        The priority value is calculated as:
                        priority_value(task) = base_priority(task) +
sum(priority_value(dependent)
            for dependent in dependents)
        where:
            base_priority(task) = (P_weight * P + L_weight * L)
           P: fixed-time dependency boost – 1/(earliest fixed time among
dependents + 1)
            L: depth of the dependency chain (i.e. how many layers of
dependents)

        Parameters
        ----------
        task
            The task for which to calculate the priority
```

```
        Returns
        -------
            int
                The computed priority value for the given task
        """
        if task.id in self.priority_cache:
            return self.priority_cache[task.id]

        P_weight, L_weight = 100, 10  # weights can be adjusted if needed

        # find the earliest fixed-time among all dependents (direct or indirect)
        def find_earliest_fixed(t):
            times = []
            for dep in self.dependents_lookup.get(t.id, []):
                if dep.time_constraint:
                    times.append(dep.time_constraint)
                else:
                    t_dep = find_earliest_fixed(dep)
                    if t_dep != float("inf"):
                        times.append(t_dep)
            return min(times) if times else float("inf")
        earliest_fixed = find_earliest_fixed(task)
        P = 1 / (earliest_fixed + 1) if earliest_fixed != float("inf") else 0

        # compute the depth of dependents: if no dependents, depth is 0; else 1 + max(depth(dependent))
        def get_dependent_depth(t):
            deps = self.dependents_lookup.get(t.id, [])
            if not deps:
                return 0
            return 1 + max(get_dependent_depth(dep) for dep in deps)
        L = get_dependent_depth(task)

        base_priority = (P_weight * P + L_weight * L)
```

```
        # final prioririty value includes the priorities of all dependents
        priority_value = base_priority
        for dep in self.dependents_lookup.get(task.id, []):
            priority_value += self.calculate_priority(dep)
        self.priority_cache[task.id] = priority_value
        return priority_value

    def get_next_fixed_time(self):
        """
        Returns the time of the next fixed-time task.

        Returns
        -------
        int
             The time of the next fixed-time task, or None if there are no
fixed-time tasks.
        """
                return  self.fixed_queue.peek().time_constraint  if  not
self.fixed_queue.is_empty() else None

    def remove_dependency(self, id):
        """
        Removes a completed task from other tasks' dependencies.

        Parameters
        ----------
        id: int
            The ID of the completed task.
        """
        for task in self.tasks:
            if id in task.dependencies:
                task.dependencies.remove(id)

    def run_task_scheduler(self, starting_time):
        """
```

```
        Executes tasks while ensuring fixed tasks are run at their exact scheduled
        times.

        Parameters
        ----------
        starting_time: int
            The starting time for the scheduler in minutes.

        """
        current_time = starting_time
        print("Running the scheduler:\n")

        while not self.fixed_queue.is_empty() or not self.flexible_queue.is_empty():
            next_fixed_time = self.get_next_fixed_time()

            if not self.fixed_queue.is_empty() and self.fixed_queue.peek().time_constraint == current_time:
                task = self.fixed_queue.heappop()
            elif not self.flexible_queue.is_empty() and (next_fixed_time is None or current_time + self.flexible_queue.heap[0].duration <= next_fixed_time):
                task = self.flexible_queue.heappop()
            else:
                wait_duration = next_fixed_time - current_time
                print(f"⏳ Waiting {wait_duration} minutes until {self.format_time(next_fixed_time)}")
                current_time = next_fixed_time
                continue

            print(f"🕰️ t={self.format_time(current_time)}")
            print(f"\tStarted '{task.description}' for {task.duration} mins...")
            current_time += task.duration
            print(f"\t✅ t={self.format_time(current_time)}, task completed!")
```

```
            task.status = self.COMPLETED
            self.remove_dependency(task.id)

                        print(f"\n🏁   Completed   all   planned   tasks   in
{self.format_time(current_time - starting_time)}!\n")

    def format_time(self, time):
        """
            Formats a time in minutes into a human-readable string (e.g.
"1h30")

        Parameters
        ----------
        time: int
            Time in minutes.

        Returns
        -------
        str
            Formatted time string.

        """
        return f"{time // 60}h{time % 60:02d}"
```

F. Test cases for the activity scheduler

```
# test cases for Scheduler

def test_case_1():
    """
    Simple dependency chain with one fixed-time task.
    """
    tasks = [
        Task(id=1, description="Wake up", duration=15, dependencies=[]),
```

```
        Task(id=2, description="Morning routine", duration=20, dependencies=[1]),
        Task(id=3, description="Breakfast", duration=30, dependencies=[2]),
        Task(id=4, description="PCW for 11 AM class", duration=90, dependencies=[3]),
        Task(id=5, description="Class at 11 AM", duration=90, dependencies=[4], time_constraint=11*60), # 11:00 class
        Task(id=6, description="Cooking", duration=60, dependencies=[]),
        Task(id=7, description="Lunch", duration=45, dependencies=[6])
    ]
    scheduler = TaskScheduler(tasks)
    scheduler.run_task_scheduler(starting_time=8*60)  # 8:00 AM


def test_case_2():
    """
    All tasks are flexible and have interdependent relationships.
    Expected: The root task "Research" should have the highest priority value.
    """
    tasks = [
        Task(id=1, description="Research", duration=120, dependencies=[]),
        Task(id=2, description="Design", duration=90, dependencies=[1]),
        Task(id=3, description="Prototype", duration=180, dependencies=[1, 2]),
        Task(id=4, description="Testing", duration=60, dependencies=[2,3]),
        Task(id=5, description="Documentation", duration=90, dependencies=[3])
    ]
    scheduler = TaskScheduler(tasks)
    scheduler.run_task_scheduler(starting_time=12*60)  # 12:00 PM

def test_case_3():
    """
    Test with multiple fixed-time tasks back-to-back and flexible tasks fitting
    in between.
```

```
    """
    tasks = [
        Task(id=1, description="Wake Up", duration=15, dependencies=[]),
        Task(id=2, description="Breakfast", duration=30, dependencies=[1]),
               Task(id=3,  description="Work-study  meeting",  duration=60,
dependencies=[2], time_constraint=9*60),
               Task(id=4,  description="11  AM  class  PCW",  duration=60,
dependencies=[]),
               Task(id=5,  description="Class  at  11  AM",  duration=60,
dependencies=[4], time_constraint=660),
                Task(id=6,  description="Reading  a  book",  duration=30,
dependencies=[]),
    ]
    scheduler = TaskScheduler(tasks)
    scheduler.run_task_scheduler(starting_time=8*60)  # 8:00 AM



def test_case_4():
    """
    Every task is a fixed-time task.
      Expected: The scheduler should simply execute them at their exact
specified times.
    """
    tasks = [
                Task(id=1,  description="Morning  meeting",  duration=30,
dependencies=[], time_constraint=480),
                Task(id=2,  description="Project  update",  duration=45,
dependencies=[], time_constraint=540),
                  Task(id=3,  description="Brunch  break",  duration=60,
dependencies=[], time_constraint=600),
       Task(id=4, description="Client call", duration=30, dependencies=[],
time_constraint=660),
          Task(id=5, description="Wrap-up", duration=15, dependencies=[],
time_constraint=690)
    ]
    scheduler = TaskScheduler(tasks)
```

```
    scheduler.run_task_scheduler(starting_time=8*60)  # 8:00 AM


def test_case_5():
    """
    Edge case: an empty task list
    """
    tasks = []
    scheduler = TaskScheduler(tasks)
    scheduler.run_task_scheduler(starting_time=1)



# UNCOMMENT TO RUN EACH TEST CASE
# test_case_1()
# test_case_2()
# test_case_3()
# test_case_4()
# test_case_5()
```

G. Test to show correct task prioritization

```
# test that the scheduler prioritizes tasks correctly and that the order
of
# input tasks does not affect the output.
def test_order_1():

    tasks_order_1 = [
        Task(id=1, description="Wake Up", duration=15, dependencies=[]),
        Task(id=2, description="Breakfast", duration=30, dependencies=[1]),
               Task(id=3,  description="Work-study  meeting",  duration=60,
dependencies=[2], time_constraint=9*60),  # 9:00 AM
               Task(id=4,  description="11  AM  class  PCW",  duration=60,
dependencies=[]),
               Task(id=5,  description="Class  at  11  AM",  duration=60,
dependencies=[4], time_constraint=660),  # 11:00 AM
```

```
                Task(id=6,  description="Reading  a  book",  duration=30,
dependencies=[]),
   ]
   scheduler = TaskScheduler(tasks_order_1)
   scheduler.run_task_scheduler(starting_time=8*60)  # 8:00 AM

def test_order_2():

   tasks_order_2 = [
              Task(id=3,  description="Work-study  meeting",  duration=60,
dependencies=[2], time_constraint=9*60),  # 9:00 AM
               Task(id=5,  description="Class  at  11  AM",  duration=60,
dependencies=[4], time_constraint=660),  # 11:00 AM
               Task(id=4,  description="11  AM  class  PCW",  duration=60,
dependencies=[]),
       Task(id=2, description="Breakfast", duration=30, dependencies=[1]),
       Task(id=1, description="Wake Up", duration=15, dependencies=[]),
                Task(id=6,  description="Reading  a  book",  duration=30,
dependencies=[]),
   ]
   scheduler = TaskScheduler(tasks_order_2)
   scheduler.run_task_scheduler(starting_time=8*60)  # 8:00 AM

assert test_order_1() == test_order_2(), "The output is not the same for
the same test cases with different order."
print("Test  passed:  Scheduler  prioritizes  tasks  correctly  and  is
order-independent.")
```

H. Code for test drive

```
def my_schedule():

   tasks = [
                 Task(id=1,  description="SS110  Session",  duration=90,
dependencies=[], time_constraint=10*60),
       Task(id=2, description="Brunch", duration=30, dependencies=[]),
```

```
            Task(id=3, description="SS110 Group Assignment Meeting", duration=210, dependencies=[2], time_constraint=12*60),
            Task(id=4, description="Take a bus/train to Nodeul Island", duration=60, dependencies=[], time_constraint=16*60),
            Task(id=5, description="Plan trip to Jeju", duration=120, dependencies=[4]),
            Task(id=6, description="Dinner at Korean BBQ restaurant", duration=90, dependencies=[5], time_constraint=19*60),
            Task(id=7, description="Work Study", duration=60, dependencies=[]),
            Task(id=8, description="Apply to internships", duration=60, dependencies=[]),
    ]
    scheduler = TaskScheduler(tasks)
    scheduler.run_task_scheduler(starting_time=10*60)  # 10 AM

my_schedule()
```

I. Experiment (Figure 1 code)

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

def generate_random_tasks(n):
    """
    Generates a list of random tasks with dependencies, durations, and fixed-time constraints.

    Parameters
    ----------
    n: int
        Number of tasks to generate.

    Returns
    -------
    list
```

```
        List of Task objects.
    """
    tasks = []
    for i in range(n):
        duration = random.randint(5, 240)  # random duration between 10 and
240 minutes
        dependencies = random.sample(range(i), min(i, random.randint(0,
5))) if i > 0 else []  # random dependencies
        time_constraint = random.choice([None, random.randint(480, 1020)])
# some tasks have fixed time
        tasks.append(Task(id=i, description=f"Task {i}", duration=duration,
dependencies=dependencies, time_constraint=time_constraint))
    return tasks

def measure_runtime(n):
    """
    Measures the runtime of the scheduler for a given number of tasks.

    Parameters
    ----------
    n: int
        Number of tasks.

    Returns
    -------
    float
        Runtime in seconds.
    """
    tasks = generate_random_tasks(n)
    scheduler = TaskScheduler(tasks)
    start_time = time.time()
    scheduler.run_task_scheduler(starting_time=480)  # Start at 8:00 AM
    end_time = time.time()
    return end_time - start_time

# experiment parameters
input_sizes = [10, 50, 100, 200, 500, 1000, 2000]  # input sizes to test
```

```
num_trials = 10  # number of trials for each input size
runtimes = []  # list to store average runtimes

# run experiments
for n in input_sizes:
   trial_runtimes = []
   for _ in range(num_trials):
       runtime = measure_runtime(n)
       trial_runtimes.append(runtime)
   average_runtime = sum(trial_runtimes) / num_trials
   runtimes.append(average_runtime)
      print(f"Input  size:  {n},  Average  Runtime:  {average_runtime:.4f}
seconds")

# plot results
plt.plot(input_sizes, runtimes, marker='o')
plt.xlabel("Number of tasks (n)")
plt.ylabel("Average runtime (seconds)")
plt.grid(True)
plt.show()
```

J. Validation (Figure 2 code)

```
import numpy as np
#  theoretical curves for comparison
n_vals = np.array(input_sizes)
n_log_n = n_vals * np.log(n_vals)
n_sq = n_vals ** 2

scale_factor_nlogn = np.mean(runtimes) / np.mean(n_log_n)
scale_factor_nsqr = np.mean(runtimes) / np.mean(n_sq)

plt.figure()
plt.plot(input_sizes, runtimes, 'o-', label="Empirical runtime")
plt.plot(n_vals, scale_factor_nlogn * n_log_n, '--', label="O(n log n)")
plt.plot(n_vals, scale_factor_nsqr * n_sq, '--', label="O(n^2)")
plt.xlabel("Number of tasks")
```

```
plt.ylabel("Runtime (seconds)")
plt.legend()
plt.show()
```

Part IV: AI statement:

I used Grammarly for proofreading.