CS110 - A day in the life of a Minervan Part II

Minerva University

CS110: Problem Solving with Data Structures and Algorithms

December 10, 2022

1)

For this assignment, my utility calculation will consider the higher generated utility to list my tasks in order of importance. For this calculation, three main aspects will be considered: the number of dependencies a task has, its duration, and its type.

The number of dependencies a task has is important to determine if other tasks can be completed. For example, if “task_1” has no dependencies but “task_2” has “task_1” as its dependency, it means that we would need to do “task_1” first so then we would be able to do “task_2”. The number of dependencies is then crucial for the utility calculation that will order the list of tasks.

The duration of tasks is important so we can try and fit as many tasks as we can during our working hours in the day. Here we give preference to tasks that have less duration, maximizing the number of tasks we can complete.

The type of task is related to how I set different importance to different kinds of tasks. “Routine” has a heavier weight on my decision because they are the basis of my well-being and the negligence of those directly impacts my performance in any other tasks. “Routine” tasks include for example: eating well and exercising. “Study” tasks are related to the academic world, including personal projects, Minerva classes, and assignments. The downside related to them are: 1. If I do not complete personal projects I would not advance in learning new skills I take a liking to and would not have projects that could help me secure a summer internship; 2. If I am not prepared for Minerva classes or do not set aside enough time to work on assignments I would see my grades downfall. “Explore” tasks refer to city exploration activities. Those are important for me to connect culturally with the places I call home and to enjoy some leisure time. Since my focus right now is to work more “Explore” activities do not

hold much weight on my tasks' ordering. Lastly, the "Glue" tasks are the ones such as taking a bus, talking to a friend, and getting money—tasks that serve as a glue for other tasks, and that are not that important if not completed.

Example of the calculation:

```
def priority_calc_unfixed(self):
    """
    Method for the utility value calculation for unfixed tasks
    The utility value will be responsible for
    the ordering of the tasks

    Parameters
    ---------
    None

    Returns
    ----------
    None
    """
    scoring =  1/self.duration  - len(self.dependencies)
    if self.type == "Routine":
        self.priority = (1 + scoring)*3
    elif self.type == "Study":
        self.priority = (1 + scoring)*2
    else:
        self.priority = 1 + scoring

def priority_calc_fixed(self):
    """
    Method for the utility value calculation for fixed tasks

    Parameters
    ---------
    None

    Returns
    ----------
    None
    """
    self.priority = self.time
```

Observing how the utility calculation was set we can derive some conclusions based on prospect theory. Firstly, it is clear that I follow an aversion to lose approach. The dependencies aspect of the calculation is intuitive—I can not do a certain task if one has to be compulsorily done before it. However, taking into consideration the duration and type of tasks make clear my aversion to loss. I am constantly trying to maximize the number of tasks I can do in a window of time. For me, the satisfaction

of reaching the end of the day and having as few lasting tasks as possible is greater than the momentaneous satisfaction of completing a task. Also, the way the task types are ranked is based on how much I will lose if do not prioritize them. “Routine” tasks directly influence my physical and mental health, which impacts how I perform in any other tasks. “Study” tasks are connected to my academic performance and how they can influence my future working prospects. “Explore” tasks are mostly related to my ability to decompress and be able to have an outlet to stress, and other feelings that deteriorate my health and performance. “Glue” activities are less important but they still carry their weight for being necessary to the completion of other tasks.

Further analyzing my approach it is always possible to have a distorted view of the real probabilities of the impact each of the tasks would have on me if not completed. For academic-related tasks, for example, I would derive much less satisfaction from getting a good grade than dissatisfaction from a similar bad grade. The impact a bad grade may have on me distorts how I perceive the probability of failing a class, not learning a key concept, and further failing in other aspects of my life such as internships. In perspective, a few unperfect outcomes are not enough to jeopardize any class grades or future opportunities, however, it is clear I am biased to approach the decision-making process of ranking tasks under a “narrow frame”, which if changed to a “broad frame” could better show me the possibility of being more flexible with my schedule and find more windows for free time still getting things done.

2)

The algorithm for the LBA Part I assignment used a priority queue as its main data structure. A priority queue using OOP makes it more simple and more intuitive

to assign characteristics to an object, such as duration and dependencies in our scenario. However, the main aspect of using a priority queue is its characteristic of making it possible for the algorithm to receive streaming data. Here if we need to update our list of tasks, deleting, adding, or editing we can do so and reorder it easily. Priority queues use heaps as a "backbone" to operate. Rebalancing the queue then would take a runtime of O(logn), while if we used a list, for example, we would probably have to iterate through the whole list and reorder it, taking at best O(nlogn). This means that for an input "n" the priority queue's operations would make the algorithm scale to maximum O(logn), while the operations on a list would make the algorithm scale on O(nlogn) if used an optimized sorting algorithm such as Merge Sort.

For the first version, my utility calculation mistakenly took only the type of the activity as a variable on the utility calculation. I had assumed that the dependencies were already considered on other parts of the code and thus should not be inserted into the calculation. This would cause chunking on the code since the list would be ordered based on the dependencies and then reorganized based on the utility calculation. Both were being taken into account separately. The algorithm however still worked for the test cases generating the most optimal solution in the scenario. Other limitations existed given the solution. Firstly, all the tasks on the to-do list were inserted into the final output of the schedule. This means the algorithm could generate a schedule with more than 24 hours of activities, unrealistic given a normal working day. Another limiting assumption was that none of the tasks were considered to have a fixed time to happen—like classes, for example—meaning the algorithm would not guarantee that we would do any fixed task during its designated time. In case of a tie between tasks (same dependencies and type), the first task included on the to-do list

would be the one at the top of the schedule. This is not necessarily a big problem if there is no preference as to which task to start first, but for this assignment, we want to maximize the number of tasks we can do in a window of time, so the shortest task in duration would come first. Finally, the algorithm does not take into consideration multi-tasking—for example, a podcast can be listened to at the same time I take the bus.

Word count: 455 words.

3)

For the multi-tasking aspect to work we need to make a few changes in our code, especially when it comes to our priority calculation—the one to determine how the tasks will be ordered. First, we would need to assign a new characteristic to the tasks which would indicate if the task can be multi-tasked or not. Tasks such as CS110 classes can not be multi-tasked, but activities such as listening to a podcast and taking a bus can happen simultaneously with other tasks. However, some multitasking tasks cannot be paired together, such as listening to music and listening to a podcast. This requires the addition of another characteristic to differentiate tasks and facilitate their pairing. For that, each multitasking task will have a description of which of the 5 human senses it mainly uses, for example, music uses mainly hearing, eating taste, and reading the eyesight.

The set of multi-task tasks (considering all the improvements cited in question 2) will then be as the following simplified decision tree shows:

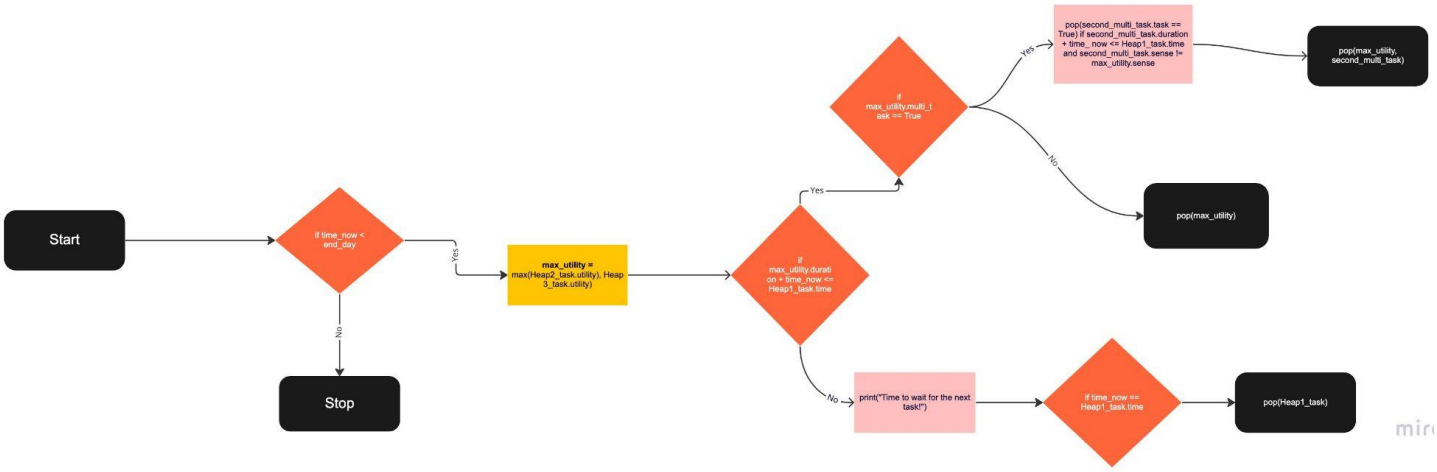


Figure 1: Simplified flowchart of how to deal with multi-task tasks on the scheduler algorithm.

Glossary:

- Heap 1: heap containing tasks with fixed time;
- Heap 2: heap containing tasks with no fixed time and that cannot be multi-tasked;
- Heap 3: heap containing tasks with no fixed time and that can be multi-tasked;

First, we need to get the task with the greatest utility value between both heaps with unfixed tasks, so we can compare it with the task with the greatest utility from the heap with fixed-time tasks. We then check to see if considering the time of the day and the duration of the tasks if we have enough time to complete an unfixed task until the start of the fixed one. If that's not possible we do nothing until we reach the time for the fixed task. However, if we do have time to complete unfixed tasks we need to check if this task is multi-tasked or not. If not, we only complete it and restart the process of finding the next task. If yes, through a recursive call on the heap with the multi-tasked tasks we will try to find another task to be completed with our original

one. This second task needs to respect two aspects: its duration needs to fit inside the time window we have until the next fixed task and its “sense” attribute needs to be different than the “sense” from the original task. For example:

original_task = Task(id = 0, description = 'Take the bus to Peter Cafe', duration = 20, type = 'Task', dependencies = [], multitask = True, sense = “Touch”)

second_task = Task(id = 1, description = 'Listen to Xadrez Verbal podcast', duration = 25, type = 'Task', dependencies = [], multitask = True, sense = “Hearing”)

next_fixed = Task(id = 2, description = 'CS110 class', duration = 90, type = 'Study', dependencies = [], multitask = False, time = 16*60)

time_now = 3 pm

In the example, the original_task and second_task both together fit on the time window available until the next fixed time. They also do not use the same human sense, so they can be combined in a multitasking scenario.

When the multitasking tasks are done we restart the process of looking for the next task. Only two tasks can be done at the same time—it is not ideal to put all five human senses to work on different tasks at the same time—so the recursion will end when we successfully find a task to combine with the original task or when we have gone through all the possible tasks and found no match (base cases).

4) a)

The greedy approach is quite similar to my application for the first LBA. However, here, as described in question 1, I have changed the utility calculation function to properly include dependencies, and to give priority to tasks smaller in

duration. I have also included fixed tasks to the general logic of the algorithm and a time limit for tasks to be done (check question1 for more details). This is a greedy algorithm because for each task considered we calculate its utility and decide if it is the greatest not considering combinations among tasks or similar. The generated output is not the global optimum of this scenario—the final ordered list of tasks is not the combination that will generate the greatest utility from all possible combinations between tasks. Fixed tasks will happen at their designated time, so the combinations rely on the unfixed tasks and their possibilities of being combined. This approach chooses these unfixed tasks by which ones will turn to have no dependencies and less duration first, while also taking into consideration their type. For example:

task_1 = Task(id=0, description=' ', duration=20, type='Task', dependencies=[])

task_2 = Task(id=1, description=' ', duration=30, type='Task', dependencies=[0])

task_3 = Task(id=2, description=' ', duration=20, type='Task', dependencies=[])

task_4 = Task(id=3, description=' ', duration=10, type='Task', dependencies=[2])

task_5 = Task(id=4, description=' ', duration=5, type='Task', dependencies=[3])

This approach will order the final output as [task_1, task_2, task_3, task_4, task_5]. Using the property of limiting the time to do tasks what will happen is that the tasks at the end of the list will be the first ones to be excluded from the final output. However, we can notice that the combination of task_3, task_4, and task_5 will generate a greater utility than task_1 and task_2 combined. All five of them have the same type and their combination generate the same duration (50 minutes), but doing three tasks should have greater priority than doing only two. With this example, it is clear that this algorithm does not generate the global optimal solution.

We also have another problem: a greedy approach to work efficiently needs to have, for each of its decisions, only one subproblem to solve. In our case, we have two: how much utility we gain from considering a task and how much we do from not considering it.

In this strategy, we still want to get which task will have the greater utility individually, so using a priority queue (heap) is still appropriate. This data structure as mentioned previously works well if we have streaming data for the to-do list while having a scaling of O(logn) for its operations.

For the dynamic programming approach instead of creating the final output based solely on individual utilities we go through all the possible combinations of unfixed tasks and generate the schedule with the global optimal utility value. We use dynamic programming exactly when we have subproblems overlapping—here the repetition of possible combinations between tasks. These combinations are computed only once, for example, the combination between task_1 with utility X and task_2 with utility Y is the same as the combination of task_3 with utility Y and task_4 with utility X. Storing these values (memoization) result on the hashtable “combinations” created on the code.

Explaining the code first, we create our to-do list normally. Then we get all the fixed tasks that are scheduled inside our time frame designated for work. Subtracting the duration of all the fixed tasks from the total time dedicated to working, we get the number of hours we can use for unfixed tasks. These hours will be the constraint used for us to select the best combination of unfixed tasks.

Similarly to the knapsack problem, we want to fit in the remaining hours as many tasks as we can, while maximizing our utility value. In our scenario the duration of the tasks are our “weight”, their utility the “value”, and the time window for activities the “capacity of the knapsack.” The optimal solution will be the combination of tasks given the constraints

that will generate the greater utility for us. These combinations are stored on a hashmap and we get the maximum one localized at the extreme right corner. From this, we can map back which tasks generate this utility value. By getting the task's id we could then from the to-do list identify which tasks to add to our final list to be inserted at the scheduler class. From this, the process of ordering the list is the same as the one we used before since there is no clear way of using dynamic programming to order the optimal to-do list.

For creating the dynamic approach the algorithm did not use a heap/a priority queue. Here, even if we have streaming data we are not interested in ordering the tasks but in getting the best combinations between them—which will take iterations through each of the tasks anyway. Also, our final goal here was not to pop one task per time for the schedule but to create a list with the precise tasks that would be done during the day from all of the tasks that were proposed on the to-do list.

b)

Tests Greedy

```
tasks = [
    Task(id=0, description='Wake up',
         duration=10, dependencies=[]),
    Task(id=1, description='Wash face',
         duration=5, dependencies=[0]),
    Task(id=2, description='Brush teeth',
         duration=5, dependencies=[0]),
    Task(id=3, description='Do skincare',
         duration=20, dependencies=[1], type='Routine'),
    Task(id=4, description='Go to Daan Park',
         duration=90, dependencies=[3,7]),
    Task(id=5, description='Do exercises',
         duration=10, dependencies=[4, 10]),
    Task(id=6, description='Do CS110 pre class work',
         duration=120, dependencies=[3], time= 11*60),
    Task(id=7, description='Take class',
         duration=90, dependencies=[6], time = 14*60),
    Task(id=8, description='Eat snack',
         duration=30, dependencies=[7],type='Study'),
    Task(id=9, description='Take nap',
         duration=80, dependencies=[8]),
    Task(id=10, description='Cook breakfast',
         duration=30, dependencies=[2],type='Routine'),
    Task(id=11, description='Eat breakfast',
         duration=20, dependencies=[10]),
    Task(id=12, description='Clean room',
         duration=10, dependencies=[11])
    ]

task_scheduler = TaskScheduler(tasks)


start_scheduler = 8*60
task_scheduler.run_task_scheduler(start_scheduler)
```

```
Running a simple scheduler:

🕗t=8h00
        started 'Wake up' for 10 mins...
        ✅ t=8h10, task completed!
🕗t=8h10
        started 'Wash face' for 5 mins...
        ✅ t=8h15, task completed!
🕗t=8h15
        started 'Brush teeth' for 5 mins...
        ✅ t=8h20, task completed!
🕗t=8h20
        started 'Do skincare' for 20 mins...
        ✅ t=8h40, task completed!
🕗t=8h40
        started 'Cook breakfast' for 30 mins...
        ✅ t=9h10, task completed!
🕗t=9h10
        started 'Eat breakfast' for 20 mins...
        ✅ t=9h30, task completed!
🕗t=9h30
        started 'Clean room' for 10 mins...
        ✅ t=9h40, task completed!
🕗t=11h00
        started 'Do CS110 pre class work' for 120 mins...
        ✅ t=13h00, task completed!
🕗t=14h00
        started 'Take class' for 90 mins...
        ✅ t=15h30, task completed!
🕗t=15h30
        started 'Eat snack' for 30 mins...
        ✅ t=16h00, task completed!

🏁 Completed all planned tasks in 8h00min!
```

Tests Dynamic Programming

```
tasks = [
    Task(id=0, description='Wake up',
         duration=10, dependencies=[]),
    Task(id=1, description='Wash face',
         duration=5, dependencies=[0]),
    Task(id=2, description='Brush teeth',
         duration=5, dependencies=[0]),
    Task(id=3, description='Do skincare',
         duration=20, dependencies=[1], type='Routine'),
    Task(id=4, description='Go to Daan Park',
         duration=90, dependencies=[3,7]),
    Task(id=5, description='Do exercises',
         duration=10, dependencies=[4, 10]),
    Task(id=6, description='Do CS110 pre class work',
         duration=120, dependencies=[3], time= 11*60),
    Task(id=7, description='Take class',
         duration=90, dependencies=[6], time = 14*60),
    Task(id=8, description='Eat snack',
         duration=30, dependencies=[7],type='Study'),
    Task(id=9, description='Take nap',
         duration=80, dependencies=[8]),
    Task(id=10, description='Cook breakfast',
         duration=30, dependencies=[2],type='Routine'),
    Task(id=11, description='Eat breakfast',
         duration=20, dependencies=[10]),
    Task(id=12, description='Clean room',
         duration=10, dependencies=[11])
    ]


optimal_fixed = OptimalTasks(tasks)
start_scheduler = 8*60
optimal_fixed.Order(start_scheduler)
```

```
    Running a simple scheduler:

    🕗t=8h00
            started 'Wake up' for 10 mins...
            ✅ t=8h10, task completed!
    🕗t=8h10
            started 'Wash face' for 5 mins...
            ✅ t=8h15, task completed!
    🕗t=8h15
            started 'Brush teeth' for 5 mins...
            ✅ t=8h20, task completed!
    🕗t=8h20
            started 'Do skincare' for 20 mins...
            ✅ t=8h40, task completed!
    🕗t=8h40
            started 'Cook breakfast' for 30 mins...
            ✅ t=9h10, task completed!
    🕗t=9h10
            started 'Eat breakfast' for 20 mins...
            ✅ t=9h30, task completed!
    🕗t=9h30
            started 'Clean room' for 10 mins...
            ✅ t=9h40, task completed!
    🕗t=11h00
            started 'Do CS110 pre class work' for 120 mins...
            ✅ t=13h00, task completed!
    🕗t=14h00
            started 'Take class' for 90 mins...
            ✅ t=15h30, task completed!
    🕗t=15h30
            started 'Eat snack' for 30 mins...
            ✅ t=16h00, task completed!

    🏁 Completed all planned tasks in 8h00min!
```

The outputs agree but that was somewhat expected for this case. The greedy approach outputs one of the local optima of the possible combinations of tasks given the to-do list. The dynamic approach outputs one of the global optima of the scenario. “One of the global optima” because for cases where we have on the to-do list combinations that result in the same maximum utility only one of these combinations will be outputted (mostly decided by

the order of the inputs). Here we have the same output for both because of the limited amount of tasks inside the task list, limiting the combinations that would differentiate both approaches' outputs.

c)

The time complexity of the greedy approach is O(n^2). Even though the priority queue is built using heaps and their operations scale bounded to O(logn), in our scheduler code we have other considerations that change our overall scaling. Specifically, every time we do a task we have to look into all the remaining ones to see if the completed task is a dependency or not. This process of looking on the list of tasks has a runtime of O(n). However, the call for the removing function is inside a while loop on the run_task_scheduler method, making the overall runtime of this process be O(n^2).

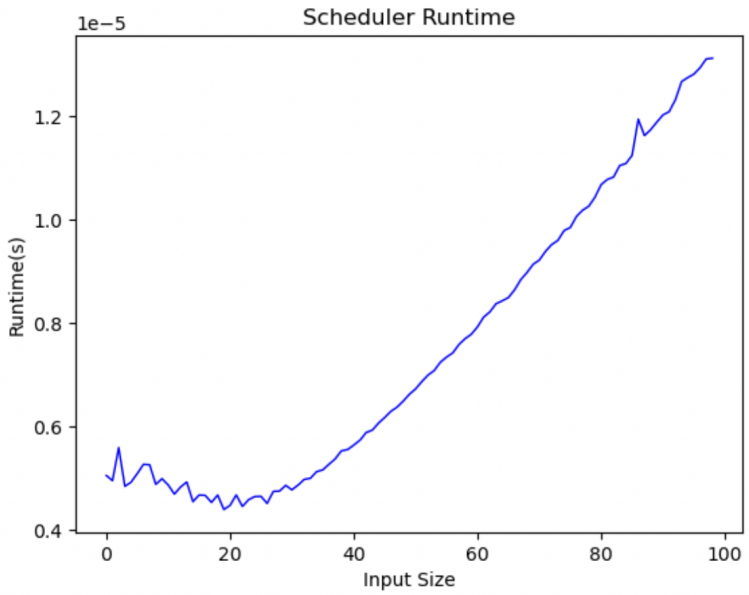


Figure 2: Plot showing the runtime average of the greedy approach.

The plot clearly shows this tendency of the algorithm, having the runtime scaling quadratically as the input size grows. (For the code, please check the Appendix).

For the dynamic approach, the new addition to the algorithm is the selection of optimal tasks before they are organized on the task scheduler. The process to create the hashtable to store the subproblems solutions takes O(number of tasks * time of execution), however since the time of execution is a constant, the number of tasks is the element that can escalate in our code, which is simply “n”. This means that to go through the DP process all it takes is O(n). However, outside of the hashtable creation, there is a nested loop that is responsible for appending the optimal tasks on the final output. Since this algorithm is the same as the greedy but doing the selection of tasks doing the DP before, we would then have O(n^2) + O(n^2) that is simply O(n^2).

I was not able to generate a visualization to experimentally validate my theoretical approach. The following images are my code and the error I kept getting.

```
### runtime experiment
# code based on my PCW and the breakout room Workbook from CS110 Session 10 and the Problem Set 2 Assignment

import time
import matplotlib.pyplot as plt
import numpy as np
import random

runtime_greedy = []
runtime_dp = []


num_tests = 500
lst_sizes = np.arange(1, 100, 1)

start_scheduler = 8*60

for lst_size in lst_sizes:
    # creating the list that will be used
    tasks = []
    types = ['Task', 'Explore', 'Study', "Routine"]

    for i in range(lst_size):
        task = Task(id = i, description= '', duration=random.randint(10, 180),
                    type = random.choice(types), dependencies=[])
        print(task.id, task.duration, task.type)
        tasks.append(task)

    greedy_scheduler = TaskScheduler(tasks)

    ## ERROR HAPPENS HERE
    dp_scheduler = OptimalTasks(tasks)


    # used for calculating the runtime
    end_greedy = 0
    end_dp = 0
```

```
    # test loops -> it will create the runtime average of how long it takes to run the algorithm for arr
    for test in range(num_tests):
        start_greedy = time.process_time()
        greedy_scheduler.run_task_scheduler(start_scheduler)
        final_greedy = time.process_time() - start_greedy
        end_greedy += final_greedy
    # average
    runtime_greedy.append(end_greedy/num_tests)


    for test in range(num_tests):
        start_dp = time.process_time()
        dp_scheduler.Order(start_scheduler)
        final_dp = time.process_time() - start_dp
        end_dp += final_dp
    runtime_dp.append(end_dp/num_tests)



# plotting
plt.plot(runtime_greedy, color="blue",  linewidth=1.0)
# plt.plot(runtime_dp, color="purple",  linewidth=1.0)

# labels
plt.title('Scheduler Runtime')
plt.xlabel('Input Size')
plt.ylabel('Runtime(s)')

# show
plt.show()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/872047371.py in <cell line: 18>()
     28
     29     greedy_scheduler = TaskScheduler(tasks)
---> 30     dp_scheduler = OptimalTasks(tasks)
     31
     32

/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/3888576308.py in __init__(self, tasks)
     28                 work_hours -= task.duration
     29
---> 30         ids_optimal = self.optimize(utilities, durations, work_hours, amount= len(utilities))
     31         for task in tasks:
     32             for ids in ids_optimal:

/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/3888576308.py in optimize(self, utilities, duration
s, work_hours, amount)
     57             if result <= 0:
     58                 break
---> 59             if result == combinations[j-1][k]:
     60                 continue
     61             else:

IndexError: list index out of range
```

When it comes to space complexity the greedy algorithm has a constant scaling. This can be explained by the fact that aside from the space the tasks occupy as objects there is not much being used by operations where there is a need to store values in new memory spaces. We can see this tendency analyzing the following plot:

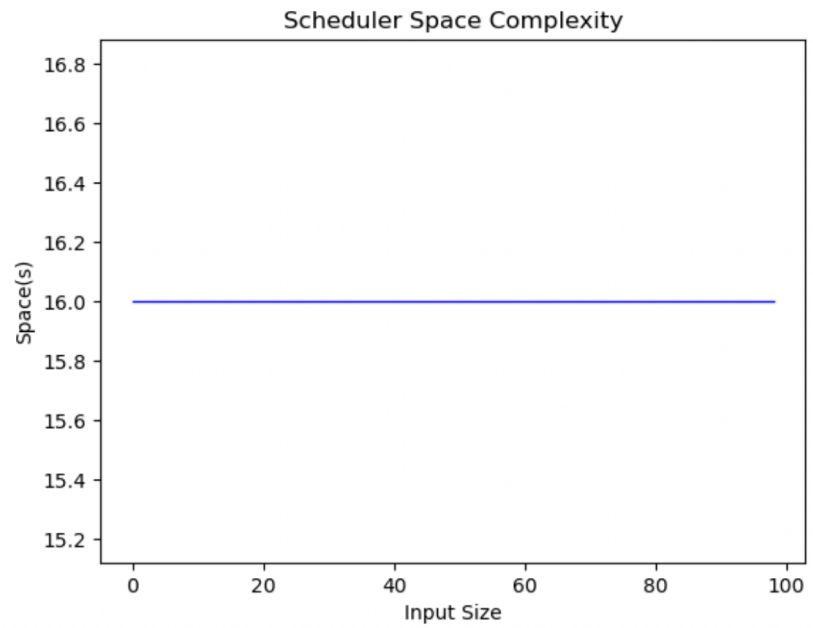


Figure 3: Plot of the average space occupied by the greedy approach algorithm.

The dynamic approach on the other hand will have a greater scaling since it requires to store the outputs of the subproblems in a hastable. The space complexity will then scale as the number of tasks grow and the amount of combinations between their utilities grow.

I was not able to generate a visualization to experimentally validate my theoretical approach. The following images are my code and the error I kept getting.

```
### runtime experiment
# code based on my PCW and the breakout room Workbook from CS110 Session 10 and the Problem Set 2 Assignment

import time
import matplotlib.pyplot as plt
import numpy as np
import random

space_dp = []



num_tests = 500
lst_sizes = np.arange(1, 100, 1)

start_scheduler = 8*60

for lst_size in lst_sizes:

    # creating the list that will be used
    tasks = []
    types = ['Task', 'Explore', 'Study', "Routine"]

    for i in range(lst_size):
        task = Task(id = i, description= '', duration=random.randint(10, 180),
                    type = random.choice(types), dependencies=[], time=random.randint(0, 16)*60)

        tasks.append(task)


    dp_scheduler = OptimalTasks(tasks)


    size_dp = 0

    # blocks prints form TaskScheduler
    blockPrint()
    # test loops -> it will create the runtime average of how long it takes to run the algorithm for arr
    for test in range(num_tests):
        size_dp += sys.getsizeof(dp_scheduler.Order(start_scheduler))
    # average
    space_dp.append(size_dp/num_tests)


# plotting
plt.plot(space_dp, color="blue",  linewidth=1.0)
# plt.plot(runtime_dp, color="purple",  linewidth=1.0)

# labels
plt.title('Scheduler Space Complexity')
plt.xlabel('Input Size')
plt.ylabel('Space(s)')

# show
plt.show()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/1681665026.py in <cell line: 18>()
     28
     29
---> 30     dp_scheduler = OptimalTasks(tasks)
     31
     32

/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/3888576308.py in __init__(self, tasks)
     28                 work_hours -= task.duration
     29
---> 30         ids_optimal = self.optimize(utilities, durations, work_hours, amount= len(utilities))
     31         for task in tasks:
     32             for ids in ids_optimal:

/var/folders/_y/pw2jz83j4zlf21ztmmzsqggr0000gp/T/ipykernel_63503/3888576308.py in optimize(self, utilities, duration
s, work_hours, amount)
     51                     combinations[j][k] = combinations[j-1][k]
     52
---> 53         result = combinations[amount][work_hours]
     54         k = work_hours
     55

IndexError: list index out of range
```

d)

The greedy approach will take a very similar direction to what was described in question 3. Since the greedy algorithm follows the same structure as the scheduler on the LBA I assignment and the multi-tasking in question 3 was explained based on this structure, both approaches will be the same. The greedy will have three heaps of which one will be for unfixed tasks, one for the fixed tasks that can not be multi-tasked, and the other for the unfixed tasks that can be multi-tasked.

For the Dynamic Approach algorithm a few things would have to be changed. First, similarly to the greedy approach the fixed tasks that fit inside the time window for work would be automatically added to the final optimal to-do list. For the remaining time after the duartion of the fixed tasks are counted we would need to find the best combination taking into consideration unfixed tasks that can be multitasked and tasks that cannot be multitasked. For that two lists would be created: one would be the output of the list of optimal unfixed non-multitasking tasks and the other would be the list of optimal unfixed multitasking tasks—meaning each list would already have the best combinations of their group. We can not concatanate both lists and the list with the fixed tasks to generate the final output because we would extrapolate our time window. What we can do is from both lists with unfixed tasks create a new one with the best combination **between** them and then put it together with the list with fixed tasks. To do so first we need to notice that multi-tasking tasks “go together” in pairs as we have assumed before. That means that the maximum utility created by these tasks is the combination of two tasks or a solo one. We need then to “store” the tasks that are in pairs in only one. We can do so by combining their durations, utilities (both used in the memoization step) and storing their ids so we can retrieve them back if needed for the final output. Having done that now we can combine both lists of unfixed tasks together and put this list again on the DP algorithm. Now the combination generated is taking into consideration the best between multi and non-multitasking tasks. When this list is generated it

can then be combined to the one with fixed tasks and the final output will be generated to be inserted into the TaskScheduler class for it to be ordered.

5)

Compared to the LBA I algorithm I would say the greedy and dynamic approaches from the LBA II are better suited to solve the problem.

Contrasting the LBA I with the greedy approach we have no difference in usage of data structures—priority queue/heaps were still suited as explained in question 2. However, the greedy algorithm produces more realistic outputs since its utility calculation takes into account dependencies, duration and type (as explained in question 1). It also takes into account humans normally do not work during all the twenty four hours of the day, setting then a limit of hours dedicated to activities.

Contrasting the LBA I with the dynamic approach we have not only the differences also present in the greedy algorithm but now we select the tasks that will generate the greatest utility before calling the scheduler code. For that we use different data structures such as hashtables for the memoization step, that facilitates the process of storing the outputs of subproblems and later identifying the optimal solution. Here then we would gt closer to the global optimal solution to the problem, ideal for a real world scenario where we want to maximize our utility values.

Both new formulations take into account the problems raised on question 1: the utility value is now calculated taking into account three main aspects: the number of dependencies a task has, its duration, and its type. Both algorithms also take into consideration the metrics set by the assignment instructions: the greatest utility is the biggest number among the utility

values, and there is a time constraint to when it comes to how many tasks can be completedn in one day.

All three algorithms have the same runtime—O(n^2), however we can define which is the best by other metrics. Considering the DP approach is the closest we have to the global optimal solution it is natural to point it as the best one. This approach generates the greatest utility, combining the tasks that will sum up to the greatest value given the time constraint. However, it takes more space to be done, as explained in question 4. In a scenario where computer memory is important the greedy approach would probably be better. The greedy algorithm has improvements if compared to the LBA I, takes less computer space and still generates an output good enough for daily activities. If the output does not need to be perfectly on point, and there is a certain flexibility to give up on a certain amount of utility value and gain on computer space, then the greedy algorithm is the way to go.

[I was not able to generate the visualization for the problem].

6)

#cs110-PythonProgramming: For the assignment I have written Python code to implement the scheduling algorithm, using the greedy and dynamic approach while applying data structures such as heaps and putting into practice structures such as Object-Oriented Programming (OOP). I have compared algorithms and data structures when asked (Question 4, for example) and I have also produced a plot to visualize the runtime metric of the algorithm (saved some errors). I have proposed strategic edge cases to be tested and show the code works. When relevant I pointed out when the algorithm could be improved and for what scenarios it would work as intended. **Word count: 101 words.**

#cs110-ComputationalCritique: Throughout the paper I have explained why using the heap data structure would be ideal for the scenario at hand, contrasting with other data structures such as lists (Question 2). I have pointed out possible operations that could be done using this data structure combined with OOP, while also analyzing metrics such as the runtime. To further support my choice I have performed experiments to analyze the algorithm's overall runtime and show its efficiency when the input grows in comparison to its time scaling. As indicated on the previous assignment, I have implemented tasks with fixed time and how they are dealt with when it comes to the utility calculation. **Word count: 110 words.**

#cs110-CodeReadability: In general, most of my code is limited to consistent docstrings explaining the overall functionality of a method, and minimal comments to add explanations to more complex concepts or strategies. Throughout the code, all variable names are meaningful or follow Python/coding conventions. I have also maintained helpful error messages on the code. This time I tried to make my code less repetitive as possible, using OOP on the DP approach, for example. **Word count: 73 words.**

#cs110-AlgoStratDataStruct: Throughout the assignment I have broken down the underlying steps to build the desired algorithms, for example in question 3. In plain English, I explained the techniques we should follow to generate the desired output given an input. These techniques could be translated to code language into data structures, algorithmic operations such as loops, sorting, and adding, and excluding elements from data structures. My explanation sets what are the possible scenarios an input can fall into (translated to if statements, utility calculations), and what are the specific termination conditions to the problem. I have also included dependencies on my utility calculation, as pointed on the last assignment, while further explaining why a priority queue should be used on the Task Schduler. **Word count: 121 words.**

#cs110-Professionalism: I have followed the submission guidelines for the assignment, submitting my work as a Jupyter pdf. and also as a word document. I have used auto-corrector software to correct any grammatical errors. My work follows a general logic—I do not jump from idea to idea, making it possible for me to reach clear conclusions. **Word count: 54 words.**

#algorithms: An algorithm is a set of steps that take a given input and generates a desired specific output. An algorithm is well-ordered, unambiguous, yields a result, and has a stop condition. Throughout the paper, I have identified, explained, and advocated which would be the most appropriate algorithm for the problem at hand. I tried to mplemented it using Python code, pointing out its possibilities and limitations. All steps were explained in plain English drawing out the underlying algorithmic strategies and later translated to code when needed. All code had docstrings and inline comments when necessary, which helped with the overall clarity of the program. Finally, the structure of the code can be used to solve other problems—we can further twist it to build more complex schedulers, or even use the internal OOP logic to build completely different solutions for other problems.

#utility: I built the ordering logic of my algorithm using the concept of utility in which we assume decision-making is based on pre-existing preferences that represent the amount of satisfaction a person derives from an outcome. I mapped and associated values for each type of task I normally do in a day, their duration and dependencies, and constructed a mathematical calculation to assign weights to each task. These weights played a major role in defining which task should be prioritized and resulted in a maximal expected utility. I have further explained my choices and the nature of my utility function on question 1, drawing on the fact I have risk-aversion tendencies that may bias me distorting the real probability of certain scenarios happening.

#breakitdown: To build the algorithm and especially code it, I first broke the problem (write the problem) into smaller subsets to make them tractable and solved those separately. In code, this decomposition can be seen in the different classes I created, how each of them has specific methods that come together (through object instantiation, use of its attributes and methods, etc.) to generate the desired output. The dynamic programming is a special case to this HC, where I have solved all the subproblems of the algorithm trying to find a global optima.

APPENDIX

Imports

```
import sys, os
import time
import matplotlib.pyplot as plt
import numpy as np
import random
```

```
# funtion to block prints
def blockPrint():
    sys.stdout = open(os.devnull, 'w')
```

Max Heap Class

```
# code based on Session's 13 PCW

class MaxHeap:
    """
    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 maxk(self):
        """
        Returns the highest key in the priority queue.

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

        Returns
        ----------
        int
            the highest key in the priority queue

        """
        return self.heap[0]


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

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

        Returns
        ----------
        None
        """
        self.heap.append(-float("inf"))
        self.increase_key(self.heap_size,key)
        self.heap_size+=1
```

```
    def increase_key(self, i, key):
        """
        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
        """
        if key < self.heap[i]:
            raise ValueError('new key is smaller than the current key')
        self.heap[i] = key
        while i > 0 and self.heap[self.parent(i)] < self.heap[i]:
            j = self.parent(i)
            holder = self.heap[j]
            self.heap[j] = self.heap[i]
            self.heap[i] = holder
            i = j


    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
        if l <= (self.heap_size-1) and heap[l]>heap[i]:
            largest = l
        else:
            largest = i
        if r <= (self.heap_size-1) and heap[r] > heap[largest]:
            largest = r
        if largest != i:
            heap[i], heap[largest] = heap[largest], heap[i]
            self.heapify(largest)

    def heappop(self):
        """
        returns the larest key in the max priority queue
        and remove 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 ')
        maxk = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self.heap_size-=1
        self.heapify(0)
        return maxk
```

Task Class

```
class Task:
    """
    Class for the tasks objects

    """
    #Initializes an instance of Task
    def __init__(self, id, description, duration, dependencies, type="Task",  time = 0, status="N"):
        """
        Attributes
        ---------
        - id: Task id (a reference number)
        - description: Task short description
        - duration: Task duration in minutes
        - dependencies: List of task ids that need to preceed this task
        - type: category of the task
        - status: Current status of the task
        - time: fixed time for fixed tasks

        """
        self.id = id
        self.description = description
        self.duration = duration
        self.dependencies = dependencies
        self.type = type
        self.status = status
        self.time = time

        # unfixed tasks
        if time == 0:
            self.priority_calc_unfixed()
        # fixed tasks
        else:
            self.priority_calc_fixed()
```

```
    def priority_calc_unfixed(self):
        """
        Method for the utility value calculation for unfixed tasks
        The utility value will be responsible for
        the ordering of the tasks

        Parameters
        ---------
        None

        Returns
        ----------
        None
        """
        scoring =  1/self.duration  - len(self.dependencies)
        if self.type == "Routine":
            self.priority = (1 + scoring)*3
        elif self.type == "Study":
            self.priority = (1 + scoring)*2
        else:
            self.priority = 1 + scoring

    def priority_calc_fixed(self):
        """
        Method for the utility value calculation for fixed tasks

        Parameters
        ---------
        None

        Returns
        ----------
        None
        """
        self.priority = self.time
```

```
    def __lt__(self, other):
        """
        Method for comparing tasks' utilities

        Parameters
        ---------
        Other: Object
            Next task in order

        Returns
        ----------
        None
        """
        # when other == -float("inf")
        if isinstance(other, Task):
            return self.priority < other.priority
        else:
            return self.priority < other
```

Task Scheduler Class

```
class TaskScheduler:
    """
    A Task Scheduler Using Priority Queues
    Outputs the tasks in order of which should be done first
    """

    # tasks' status
    NOT_STARTED = 'N'
    IN_PRIORITY_QUEUE = 'I'
    COMPLETED = 'C'

    def __init__(self, tasks):
        """
        Attributes
        ---------
        - tasks: list of tasks
        - priority_queue_unfixed: max heap with unfixed tasks
        - priority_queue_fixed: max heap with fixed tasks

        """
        self.tasks = tasks
        self.priority_queue_unfixed = MaxHeap()
        self.priority_queue_fixed = MaxHeap()

    def print_self(self):
        """
        Prints the list of all inputed tasks, their duration and
        their dependencies

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

        Returns
        ----------
        None
        """
```

```
        print("Tasks added to the simple scheduler:")
        print("-------------------------------------")
        for t in self.tasks:
            print(f"➡️'{t.description}', duration = {t.duration} mins.")
            if len(t.dependencies)>0:
                print(f"\t ⚠️ This task depends on others!")


    def remove_dependency(self, id):
        """
        Removes the completed tasks from
        the dependency list of other tasks

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

        Returns
        ----------
        None
        """
        for t in self.tasks:
            if t.id != id and id in t.dependencies:
                t.dependencies.remove(id)


    def get_tasks_ready(self):
        """
        Gets the tasks ready for scheduling
        Tasks are pushed into their designated queues

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

        Returns
        ----------
        None
        """
```

```
        for task in self.tasks:
            # Task has no dependencies and is not in the queue
            if task.status == self.NOT_STARTED and not task.dependencies:
                task.status = self.IN_PRIORITY_QUEUE
                task_time = task.time

                # pushes task to which heap depending if the task is fixed or not
                if task_time != 0:
                    self.priority_queue_fixed.heappush(task)
                else:
                    self.priority_queue_unfixed.heappush(task)

    def check_unscheduled_tasks(self):
        """
        Checks for any unscheduled tasks not in the queue

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

        Returns
        ----------
        None
        """
        for task in self.tasks:
            if task.status == self.NOT_STARTED:
                return True
        return False

    def format_time(self, time):
        """
        Formats the time into a 24 hour clock

        Parameters
        ----------
        time: int
            Inputted time

        Returns
```

```
        ----------
        String formatted in 24 hour clock
        """
        return f"{time//60}h{time%60:02d}"


    def run_task_scheduler(self, starting_time):
        """
        Runs the whole scheduler
        Orders the tasks by their utility values

        Parameters
        ----------
        starting_time: int
            the start time the user inputs in minutes

        Returns
        ----------
        Print of all of the tasks in order of execution
        """

        current_time = starting_time

        print("Running a simple scheduler:\n")

        # if there is a task in tasks list with status = NOT_STARTED or an element in one of the priority queues
        while self.check_unscheduled_tasks() or self.priority_queue_unfixed.heap or self.priority_queue_fixed.heap:

            # Identify tasks that are ready to execute
            self.get_tasks_ready()


            if len(self.priority_queue_unfixed.heap) > 0 and len(self.priority_queue_fixed.heap) > 0 :

                task_unfix = self.priority_queue_unfixed.heap[0]
                task_fix = self.priority_queue_fixed.heap[0]


                end_time = task_unfix.duration + current_time
```

```
                # do fixed task first
                if end_time > task_fix.time:
                    task = self.priority_queue_fixed.heappop()
                    current_time = task.time
                # do unfixed task first
                else:
                    task = self.priority_queue_unfixed.heappop()

            # cases where only one of the queues has elements
            elif len(self.priority_queue_fixed.heap) > 0:
                task = self.priority_queue_fixed.heappop()
                current_time = task.time
            else:
                task = self.priority_queue_unfixed.heappop()


            print(f"🕒t={self.format_time(current_time)}")
            print(f"\tstarted '{task.description}' for {task.duration} mins...")

            current_time += task.duration

            # reseting the clock
            if current_time > 24*60:
                current_time =  current_time - 24*60

            print(f"\t✅ t={self.format_time(current_time)}, task completed!")

            # If the task is done, remove it from the dependency list
            self.remove_dependency(task.id)
            task.status = self.COMPLETED

            # hour limit for completing activities
            if current_time >= 16*60:
                break

        total_time = current_time - starting_time

        print(f"\n🏁 Completed all planned tasks in {total_time//60}h{total_time%60:02d}min!")
```

Tests Greedy

```
tasks = [
    Task(id=0, description='Wake up',
         duration=10, dependencies=[]),
    Task(id=1, description='Wash face',
         duration=5, dependencies=[]),
    Task(id=2, description='Brush teeth',
         duration=5, dependencies=[0]),
    Task(id=3, description='Do skincare',
         duration=20, dependencies=[1], type='Routine'),
    Task(id=4, description='Go to Daan Park',
         duration=90, dependencies=[3,7]),
    Task(id=5, description='Do exercises',
         duration=10, dependencies=[4, 10]),
    Task(id=6, description='Do CS110 pre class work',
         duration=120, dependencies=[3], time= 11*60),
    Task(id=7, description='Take class',
         duration=90, dependencies=[6], time = 14*60),
    Task(id=8, description='Eat snack',
         duration=30, dependencies=[7],type='Study'),
    Task(id=9, description='Take nap',
         duration=80, dependencies=[8]),
    Task(id=10, description='Cook breakfast',
         duration=30, dependencies=[2],type='Routine'),
    Task(id=11, description='Eat breakfast',
         duration=20, dependencies=[10]),
    Task(id=12, description='Clean room',
         duration=10, dependencies=[11])
    ]

task_scheduler = TaskScheduler(tasks)


start_scheduler = 8*60
task_scheduler.run_task_scheduler(start_scheduler)
```

```
Running a simple scheduler:

🕗t=8h00
        started 'Wash face' for 5 mins...
        ✅ t=8h05, task completed!
🕗t=8h05
        started 'Wake up' for 10 mins...
        ✅ t=8h15, task completed!
🕗t=8h15
        started 'Brush teeth' for 5 mins...
        ✅ t=8h20, task completed!
🕗t=8h20
        started 'Do skincare' for 20 mins...
        ✅ t=8h40, task completed!
🕗t=8h40
        started 'Cook breakfast' for 30 mins...
        ✅ t=9h10, task completed!
🕗t=9h10
        started 'Eat breakfast' for 20 mins...
        ✅ t=9h30, task completed!
🕗t=9h30
        started 'Clean room' for 10 mins...
        ✅ t=9h40, task completed!
🕗t=11h00
        started 'Do CS110 pre class work' for 120 mins...
        ✅ t=13h00, task completed!
🕗t=14h00
        started 'Take class' for 90 mins...
        ✅ t=15h30, task completed!
🕗t=15h30
        started 'Eat snack' for 30 mins...
        ✅ t=16h00, task completed!

🏁 Completed all planned tasks in 8h00min!
```

Optimal Tasks Class (DP)

```
class OptimalTasks:

    def __init__(self, tasks):
        """
        Attributes
        ---------
        - tasks: list of tasks
        """

        # constraint
        work_hours = 7*60

        utilities = []
        durations = []

        global unfixed
        unfixed =[]

        # total_tasks will be the one inserted to be ordered as a heap
        global total_tasks
        total_tasks = []

        durations_tot = 0

        for task in tasks:
            # if unfixed
            if task.time == 0:
                unfixed.append(task)
                utilities.append(task.priority)
                durations.append(task.duration)
            # if fixed
            else:
                total_tasks.append(task)
                work_hours -= task.duration

        ids_optimal = self.optimize(utilities, durations, work_hours, amount= len(utilities))
```

```
        # appending unfixed optimal tasks
        for task in tasks:
            for ids in ids_optimal:
                if ids == task.id:
                    total_tasks.append(task)


    def optimize(self, utilities, durations, work_hours, amount):
        """
        DP method
        The combination of tasks that generate the greatest utility (subproblems)

        Parameters
        ----------
        utilities, durations: lst
            utilities: value we want to maximize
            durations: constraints

        Returns
        ----------
        id_list: lst
            List with the optimal tasks' ids
        """

        combinations = [[0 for k in range(work_hours + 1)] for j in range(amount + 1)]

        id_list = set()


        for j in range(amount + 1):
            for k in range(work_hours + 1):
                if j == 0 or k == 0:
                    combinations[j][k] = 0
                elif durations[j-1] <= k:
                    combinations[j][k] = max(utilities[j-1] + combinations[j-1][k-durations[j-1]],combinations[j-1][k])
                else:
                    combinations[j][k] = combinations[j-1][k]
```

```
        # optimal combination
        result = combinations[amount][work_hours]
        k = work_hours

        # get tasks that generate optima
        for j in range(amount, 0, -1):
            if result <= 0:
                break
            if result == combinations[j-1][k]:
                continue
            else:
                id_list.add(unfixed[j-1].id)
                result -= utilities[j - 1]
                k -= durations[j - 1]

        return id_list

    def Order(self,start_scheduler):
        """
        Call the Task Scheduler with the oprimal task list

        Parameters
        ----------
        start_scheduler: int
            time of start of the scheduler

        Returns
        ----------
        None
        """
        task_scheduler = TaskScheduler(total_tasks)
        task_scheduler.run_task_scheduler(start_scheduler)
```

Tests Dynamic Programming

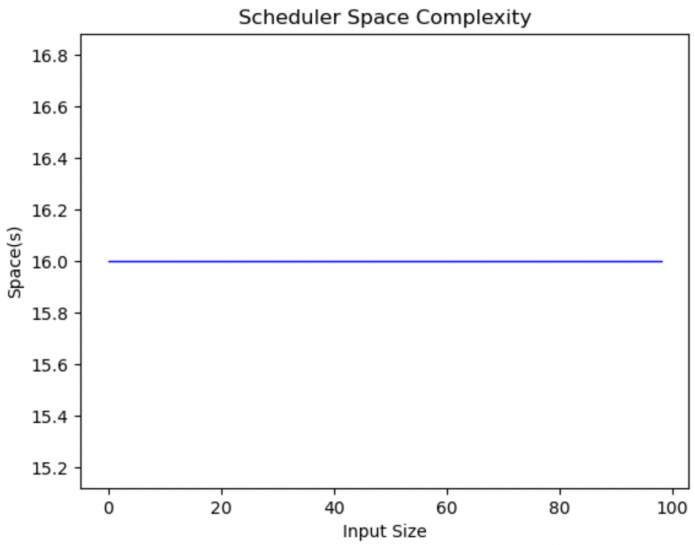
```
tasks = [
    Task(id=0, description='Wake up',
         duration=10, dependencies=[]),
    Task(id=1, description='Wash face',
         duration=5, dependencies=[0]),
    Task(id=2, description='Brush teeth',
         duration=5, dependencies=[0]),
    Task(id=3, description='Do skincare',
         duration=20, dependencies=[1], type='Routine'),
    Task(id=4, description='Go to Daan Park',
         duration=90, dependencies=[3,7]),
    Task(id=5, description='Do exercises',
         duration=10, dependencies=[4, 10]),
    Task(id=6, description='Do CS110 pre class work',
         duration=120, dependencies=[3], time= 11*60),
    Task(id=7, description='Take class',
         duration=90, dependencies=[6], time = 14*60),
    Task(id=8, description='Eat snack',
         duration=30, dependencies=[7],type='Study'),
    Task(id=9, description='Take nap',
         duration=80, dependencies=[8]),
    Task(id=10, description='Cook breakfast',
         duration=30, dependencies=[2],type='Routine'),
    Task(id=11, description='Eat breakfast',
         duration=20, dependencies=[10]),
    Task(id=12, description='Clean room',
         duration=10, dependencies=[11])
    ]


optimal_fixed = OptimalTasks(tasks)
start_scheduler = 8*60
optimal_fixed.Order(start_scheduler)
```

```
Running a simple scheduler:

🕰t=8h00
        started 'Wake up' for 10 mins...
        ✅ t=8h10, task completed!
🕰t=8h10
        started 'Wash face' for 5 mins...
        ✅ t=8h15, task completed!
🕰t=8h15
        started 'Brush teeth' for 5 mins...
        ✅ t=8h20, task completed!
🕰t=8h20
        started 'Do skincare' for 20 mins...
        ✅ t=8h40, task completed!
🕰t=8h40
        started 'Cook breakfast' for 30 mins...
        ✅ t=9h10, task completed!
🕰t=9h10
        started 'Eat breakfast' for 20 mins...
        ✅ t=9h30, task completed!
🕰t=9h30
        started 'Clean room' for 10 mins...
        ✅ t=9h40, task completed!
🕰t=11h00
        started 'Do CS110 pre class work' for 120 mins...
        ✅ t=13h00, task completed!
🕰t=14h00
        started 'Take class' for 90 mins...
        ✅ t=15h30, task completed!
🕰t=15h30
        started 'Eat snack' for 30 mins...
        ✅ t=16h00, task completed!

🏁 Completed all planned tasks in 8h00min!
```

Space Greedy

```
### runtime experiment
# code based on my PCW and the breakout room Workbook from CS110 Session 10 and the Problem Set 2 Assignment

space_greedy = []



num_tests = 500
lst_sizes = np.arange(1, 100, 1)

start_scheduler = 8*60

for lst_size in lst_sizes:
    # creating the list that will be used
    tasks = []
    types = ['Task', 'Explore', 'Study', "Routine"]

    for i in range(lst_size):
        task = Task(id = i, description= '', duration=random.randint(10, 180),
                    type = random.choice(types), dependencies=[], time=random.randint(0, 16)*60)
        # print(task.id, task.duration, task.type)
        tasks.append(task)


    greedy_scheduler = TaskScheduler(tasks)


    size_greedy = 0
    blockPrint()
    # test loops -> it will create the runtime average of how long it takes to run the algorithm for arr
    for test in range(num_tests):
        size_greedy += sys.getsizeof(greedy_scheduler.run_task_scheduler(start_scheduler))
    # average
    space_greedy.append(size_greedy/num_tests)
```

```
# plotting
plt.plot(space_greedy, color="blue",  linewidth=1.0)
# plt.plot(runtime_dp, color="purple",  linewidth=1.0)

# labels
plt.title('Scheduler Space Complexity')
plt.xlabel('Input Size')
plt.ylabel('Space(s)')

# show
plt.show()
```



Runtime Experiment Greedy

```
### runtime experiment
# code based on my PCW and the breakout room Workbook from CS110 Session 10 and the Problem Set 2 Assignment


runtime_greedy = []
runtime_dp = []


num_tests = 500
lst_sizes = np.arange(1, 100, 1)

start_scheduler = 8*60

for lst_size in lst_sizes:
    # creating the list that will be used
    tasks = []
    types = ['Task', 'Explore', 'Study', "Routine"]

    for i in range(lst_size):
        task = Task(id = i, description= '',
                    duration=random.randint(10, 180),
                    type = random.choice(types), dependencies=[], time=random.randint(0, 16)*60)
        # print(task.id, task.duration, task.type)
        tasks.append(task)


    greedy_scheduler = TaskScheduler(tasks)


    # used for calculating the runtime
    end_greedy = 0
    end_dp = 0

    blockPrint()
    # test loops -> it will create the runtime average of how long it takes to run the algorithm for arr
    for test in range(num_tests):
        start_greedy = time.process_time()
        greedy_scheduler.run_task_scheduler(start_scheduler)



        final_greedy = time.process_time() - start_greedy
        end_greedy += final_greedy
    # average
    runtime_greedy.append(end_greedy/num_tests)
```

```
# plotting
plt.plot(runtime_greedy, color="blue",  linewidth=1.0)
#plt.plot(runtime, color="purple",  linewidth=1.0)
# plt.plot(runtime_dp, color="purple",  linewidth=1.0)

# labels
plt.title('Scheduler Runtime')
plt.xlabel('Input Size')
plt.ylabel('Runtime(s)')

# show
plt.show()
```