Final Computational Application

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

CS110

Prof. Ribeiro

14th December 2022

Q1 – Utility Values

In the previous assignment, we used the following formula to calculate our priority values for prioritization:

$$E(X) = cost(duration) * p(achieving\ a\ task)$$

The above formula came with constraints where we were just considering the duration of tasks as our main variable. This posed a problem where we are not accommodating our algorithm to fixed tasks and are assuming that a person likes to finish tasks with longer duration first which might not be true in all cases. For example, writing an email to a professor will take 10 minutes and has the utmost priority for the student but using the above function that task would not be prioritized. Therefore, we modified the function to fit fixed tasks based on upcoming due dates. The modified formula is

$$E(X) = 1/\left(((end\_time - duration) + 1) / 10000\right)$$

So, a simpler form of this function will be

$$E(X) = 10000 / (end\_time - duration) + 1$$

Firstly, the variable end-time refers to due dates or the checkpoint where a task must end. We can make this by giving time when the task will end. e.g. an assignment that is due in 6 hours and takes 2 hours to do the assignment, we will convert them into minutes as we want to keep the units the same, so end_time is 360 minutes and duration is 120 minutes. Secondly, The difference between end_time and duration will give us the time left before a task is due as in some cases a task would be added where the time left to finish is not equal to the duration, using

this priority value calculation even with smaller duration, those tasks will be prioritized. Thirdly, the difference between the end_time and duration is in the denominator as it has an inverse relationship to our priority value as the lower the difference, the higher priority that task will have as we will have less time to finish it. Lastly, we are dividing this by 10000 as we will get small utility values just by taking the difference of those two times but as that difference is in the denominator, we can divide it by 10000 to produce an increase in priority value by a factor of 1000 for a decent comparison and prioritization. Lastly, we are adding + 1 at the end as while making experimental plots we were getting zero in the denominator so this will prevent that error from coming again.

Q2 – Applicability of Task Scheduler

The task scheduler from the previous assignment had successes and failures. The success included the proper functioning of the scheduler which utilized the underlying priority values to order the tasks. The task scheduler worked on the assumption that our tasks will be added based on prior dependencies and so the user will be responsible for adding tasks that to a certain degree are prioritized. This assumption made it possible for our scheduler to do minimal work to prioritize tasks based on dependencies and durations (as mentioned above). However, our priority value model was not robust as we were assuming that a person focuses on longer tasks first but that might not be the case always.

The following were some failures of poor implementation of heaps and priority queues as our data structure. For instance, in terms of superseding effect and stream of data, our priority value model was considering high-duration tasks as priority tasks which meant that we are considering maximum priority queues but by only considering one heap we were not inclusive of

how a single heap will handle data stream (infinite inputs). Superseding effect of priority queues in heaps where the superseding effect means how we utilized priority queue as an intervention/supplementary data structure with heaps. By utilizing only one priority queue to handle our tasks we failed to utilize the principle of using strict-high priority queues which could have formed a layered structure and assisted in the management of traffic in terms of the data stream by utilizing lower priority queues to store some of the data and even provide potential to create a week-long task scheduler instead of a day-long (Juniper Networks, 2021). Therefore, we can use priority queues to supersede the problem data stream. However, priority queues are still problematic as a task scheduler adds and removes tasks in a continuous job, and in priority queues enqueuing and dequeuing have scaling growth of O(log n) which is quite slow for a continuous process.

Lastly, our task scheduler failed to consider multi-tasks as we didn't specify constraints and assignment of relevance or dependencies that can be allocated to tasks like eating and watching a movie or traveling in a bus while listening to music. We will discuss the method of modifying our tasks scheduler with multi-tasking property in the subsequent sections.

Word Count: 393 words

Q3 – Multitasking

We can make a few changes to our original algorithm to add the multitasking property. Firstly, now we have another attribute involved (multi-tasking), and we need to include this in our class for Tasks. Secondly, we need to utilize the status attribute in our algorithm and modify it so it can be used for multitasking. We can do this by creating a separate multitasking function for our multi-tasking property. In that function, we need to consider a few things. We need to keep track of the time that it takes for a task that can be done with another task and their end times and durations so that we can manage overlapping task intervals. We also need to provide a status to those tasks that can be multi-tasked. We can do this by using the attribute in the class(task) and subsequently using a second list or storage bin to keep all the tasks that can be multi-tasked by comparing the multi-tasking status and storing them in that bin.

Here, I will propose using more than one priority queue for efficiency as we should consider priority values for our organization as they act as a useful metric for organizing our tasks. By using the priority value function and end_time attribute as mentioned above, we would not need further improvement of priority values as multi-tasking needs duration and end_time as parameters. One queue will be responsible for storing tasks that can be multi-tasked and ordering them based on priorities as they are non-fixed tasks. The other one would be a priority queue for fixed tasks like taking a class or submitting tax returns, etc. When we are done with ordering the tasks in the multi-tasking priority queue we will be able to find tasks that can be done together so when we push those into our container holding our fixed-tasks (fixed-tasks priority queue), we will be adding multi-taskable tasks based on which of them can work together based on priority value evaluation. This will be efficient as one priority queue cannot organize both fixed and

non-fixed tasks as it might provide a superficial solution or might not be able to handle these operations based on space and multi-property functioning. We can further improve this application by merging two tasks that have close-by priority values so that once they are added to the main priority queue with fixed tasks, the scheduler will bring them to the top together so this way the tasks can take place around the same duration. With this feature, we also need to add a limitation: if a multi-taskable task has a priority value greater than the specified priority value, then it will not be merged with another task. This limitation will ensure that when it is added to the main priority queue, the task is considered an individual task that can be done with two smaller multi-taskable tasks. This is nuanced and might be a higher-level feature but considering this will make our scheduler robust and versatile.

Word Count: 500

Q4 – Greedy Algorithms and Dynamic Programming

A: Algorithmic Strategy:

Dynamic Programming

Dynamic programming (DP) is a recursive breakdown of a problem into simpler subproblems that can be solved individually to find optimal solutions for subproblems that can be joined or used as a source to find the global optimal solution. It is quite useful as we store the optimal solutions for subproblems and save time while recomputing the inputs. Therefore it is more efficient in terms of reusing inputs.

Although DP can be done in a bottom-up approach (Tabulation) or top-down approach (Memoization), for our DP approach we will use tabulation as it is faster than memoization and it is easier to access past solutions from the table compared to memoization (Kumar, 2017). We are still dealing with one nuance that memoization can offer which is that the values in the table are filled in demand but for our use, we will assume that one-by-one filling of tasks in the table is sufficient.

Building on how we will implement tabulation, we need to first discuss the consistency of using priority queues for our optimization. We believe that lists or arrays will be better to use here than a priority queue. Given that, using we can only get an item out of the priority queue if it has high or low priority values but in lists, we can get any item on the position in the list without removing one at a time.

Moreover, unlike memoization, where we fill up our table on demand and priority optimization is used, we will use end_time as a time constraint to optimize our algorithm. The reason is that a task scheduler's purpose is to assist users in time management, so by using time

as our objective function, we will find the profit gained in terms of time saved from our DP approach. However, it will be hard to use for bulk data or data steam as lists have a scaling growth of O(n) to add and remove items that are more costly than priority values, that is, O(log n), so we will have some setbacks (Gray, 2019).

Our implementation is simple, as tabulation can often lead to complicated code that might not be reusable and editable in the future so we will try to avoid additional nuances in our code to approach an optimal solution (Kumar, 2017).

DP requires two conditions to be implemented: overlapping subproblems and optimal substructure. To understand these two conditions, we need to imagine a swimming duck analogy. From the top, the duck seems still but from the bottom, it is paddling rigorously to stay afloat. The stillness, while floating, is the optimal substructure which is outside appearance or the optimal result, while each paddle is a subproblem that overlaps in terms of rhythm and sequence. Using this analogy for our task scheduler, the optimal substructure is increasing profit by reducing time spent while overlapping subproblems are problems that result in a sub-optimal solution but that can be used to produce a global optimal solution and in our case, we are using time constraints of the day and per task to reach our optimal solution.

This approach works by creating a table to store the results of the subproblems and then iterating through the days and tasks to fill in the table. It then reconstructs the solution by starting from the last day and working backward through the table to determine which tasks were included in the optimal solution. We will accomplish this by storing the time spent on each task in a table per activity and we will add those tasks in our table that yield profit in terms of time used to accomplish the tasks. Here by profit we mean the priority values obtained and the global

optimal solution would be in the case where in a time constraint, for instance, today, we can spend only 360 minutes on our tasks and we are using our task scheduler only for school work, then within the 360 minutes, the maximum number of tasks that can be completed in our list will provide us the optimal solution.

Greedy Algorithm

Similar to DP, a greedy algorithm is designed to use subproblems as well to find an optimal solution but it does not consider all the subproblems which is why it only provides a locally optimal solution. It only utilizes the information at hand to conclude the problem, unlike DP, which doesn't give an optimal solution until it has all the sub-optimal solutions from its overlapping subproblems.

Deconstructing our situation with a greedy approach will give us more insights into how we can implement this optimization. There are two characteristics of the greedy approach:

1. There is an ordered list of resources with associated costs or values. These quantify system constraints.
2. Using the maximum resources possible during the time a constraint is in effect (Walker, 2020).

In our problem, the ordered list of resources is the tasks and their relevant attributes that yield a cost in terms of a priority value and our constraint is the limited time in a day to accomplish maximum tasks.

With these characteristics underway, we can discuss the applicability of using a greedy algorithm for this problem. On one end, a greedy algorithm is a faster approach as it will give us the total priority value that can be accomplished under the constraint quickly but on the other

end, the value might need to be revised. Therefore, for this approach, we are assuming that we only need an approximate cost of doing the tasks in that constraint to check if it is worth it or not which means it is similar to the 0-1 knapsack problem than the greedy algorithm is not applicable to make a global optimal decision as Greedy algorithm does not provide the global optimal solution for all the problems. For implementation, we will still use the same limited time as our constraint and break down our problem into steps: the time taken to finish each task and compare the resulting priority values (profit). In terms of output, we are turning our tasks scheduler from a day-long to a week-long task scheduler, so we expect to see tasks on which day of the week will have the highest profit. Based on that day, the user can optimize or plan the level of productivity and which days to work more on compared to the others.

B: Python Implementation:

Fig. 1 & 2 shows output of day and week task scheduler. Code is added in Appendix B.

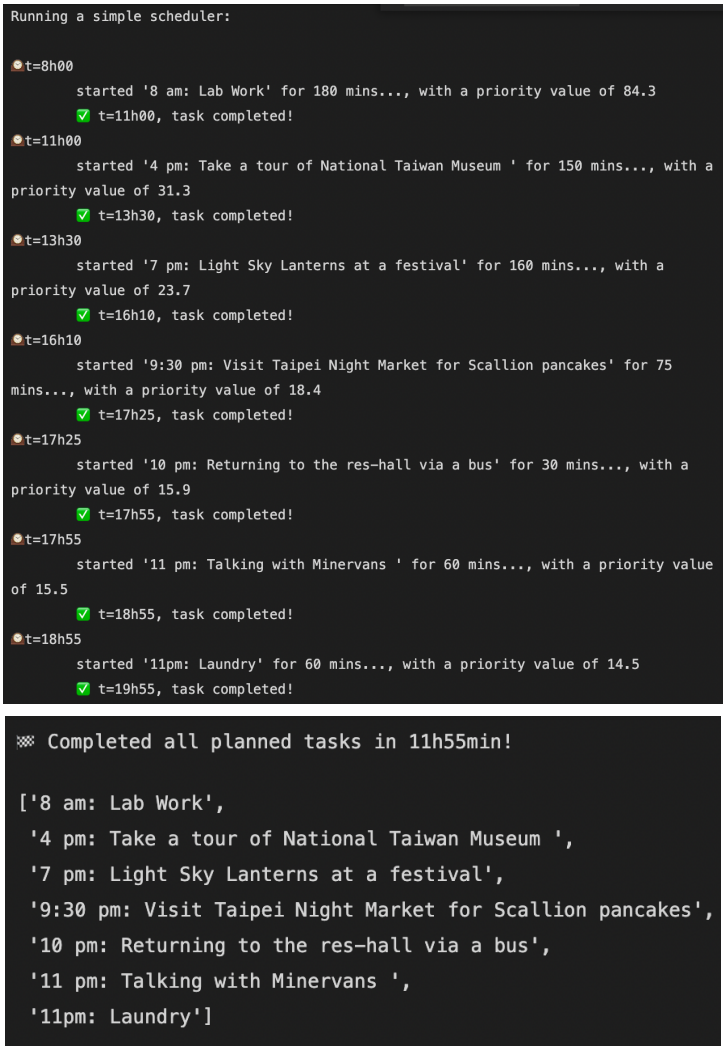

Fig. 1 shows the output generated from the modified simple task scheduler for a day.

```
Running a simple scheduler:

⏰t=8h00
    started '8 am: Lab Work' for 100 mins..., with a priority value of 101.0
    ✅ t=9h40, task completed!
⏰t=9h40
    started '4 pm: Take a bus to Sky Lantern Festival' for 50 mins..., with a priority value of 41.0
    ✅ t=10h30, task completed!
⏰t=10h30
    started '7 pm: Light Sky Lanterns at a festival' for 160 mins..., with a priority value of 42.7
    ✅ t=13h10, task completed!
⏰t=13h10
    started '9:30 pm: Visit Taipei Night Market for Scallion pancakes' for 75 mins..., with a priority value of 27.7
    ✅ t=14h25, task completed!
⏰t=14h25
    started '10 pm: Returning to the res-hall via a bus' for 30 mins..., with a priority value of 24.8
    ✅ t=14h55, task completed!

🏁 Completed all planned tasks in 6h55min!
Running a simple scheduler:

⏰t=8h00
    started '8 am: Lab Work' for 60 mins..., with a priority value of 72.4
    ✅ t=9h00, task completed!
⏰t=9h00
    started '4 pm: Take a tour of National Taiwan Museum ' for 100 mins..., with a priority value of 67.7
    ✅ t=10h40, task completed!
⏰t=10h40
    started '9:30 pm: Visit Taipei Night Market for Scallion pancakes' for 75 mins..., with a priority value of 45.4
    ✅ t=11h55, task completed!
⏰t=11h55
    started '10 pm: Returning to the res-hall via a bus' for 30 mins..., with a priority value of 33.3
    ✅ t=12h25, task completed!
⏰t=12h25
    started '11 pm: Talking with Minervans ' for 60 mins..., with a priority value of 30.4
    ✅ t=13h25, task completed!
⏰t=13h25
    started '11pm: Laundry' for 60 mins..., with a priority value of 30.4
    ✅ t=14h25, task completed!

🏁 Completed all planned tasks in 6h25min!
Running a simple scheduler:

⏰t=8h00
    started '8 am: Lab Work' for 100 mins..., with a priority value of 101.0
    ✅ t=9h40, task completed!
```

```
t=9h40
    started '4 pm: Take a tour of National Taiwan Museum ' for 50 mins..., with a priority value of 51.0
    ✅ t=10h30, task completed!
t=10h30
    started '7 pm: Light Sky Lanterns at a festival' for 100 mins..., with a priority value of 67.7
    ✅ t=12h10, task completed!

🏁 Completed all planned tasks in 4h10min!
Running a simple scheduler:

t=8h00
    started '8 am: Lab Work' for 180 mins..., with a priority value of 501.0
    ✅ t=11h00, task completed!
t=11h00
    started '11 pm: Talking with Minervans ' for 60 mins..., with a priority value of 38.0
    ✅ t=12h00, task completed!
t=12h00
    started '11pm: Laundry' for 60 mins..., with a priority value of 35.5
    ✅ t=13h00, task completed!

🏁 Completed all planned tasks in 5h00min!
Running a simple scheduler:

t=8h00
    started '8 am: Lab Work' for 180 mins..., with a priority value of 84.3
    ✅ t=11h00, task completed!
t=11h00
    started '4 pm: Take a tour of National Taiwan Museum ' for 150 mins..., with a priority value of 67.7
    ✅ t=13h30, task completed!
t=13h30
    started '11 pm: Talking with Minervans ' for 60 mins..., with a priority value of 30.4
    ✅ t=14h30, task completed!
t=14h30
    started '11pm: Laundry' for 60 mins..., with a priority value of 26.6
    ✅ t=15h30, task completed!

🏁 Completed all planned tasks in 7h30min!
```

Fig. 2 shows the output for using another class method to produce an initial output of the tasks for a week of tasks, their run time and priority values.

Test Case 1:

```
Day:Thursday | Running Time:300 | Profit:250
Day:Friday | Running Time:450 | Profit:220
Day:Tuesday | Running Time:385 | Profit:150
Day:Monday | Running Time:415 | Profit:100
Day:Wednesday | Running Time:250 | Profit:50
```

Test Case 2:

```
Day:Thursday | Running Time:300 | Profit:250
Day:Friday | Running Time:450 | Profit:220
```

Test Case 3:

```
Day:Friday | Running Time:450 | Profit:220
Day:Tuesday | Running Time:385 | Profit:150
Day:Monday | Running Time:415 | Profit:100
Day:Wednesday | Running Time:250 | Profit:50
Day:Thursday | Running Time:300 | Profit:50
```

Fig. 3 shows the outputs for the greedy scheduler, of different test cases with varying time constraints and duration of tasks in each day and changing profits.

From our greedy algorithm outputs, we see some constraints can be repurposed as features for the users. Firstly, the task scheduler orders the tasks of a week based on the days with the highest to lowest profit. This can be used by the users for managing productivity throughout the week and comparing how long they have to spend per day on their tasks. It

further provides a constraint to the user to manage their week by the time they can spend as it only shows the highly profitable day(s) that can be done during this limited time so the user can either change their time constraint to fit in all tasks or consider reorganizing some tasks to increase profit and ability to complete tasks in the time constraint. These features make it a robust week scheduler.

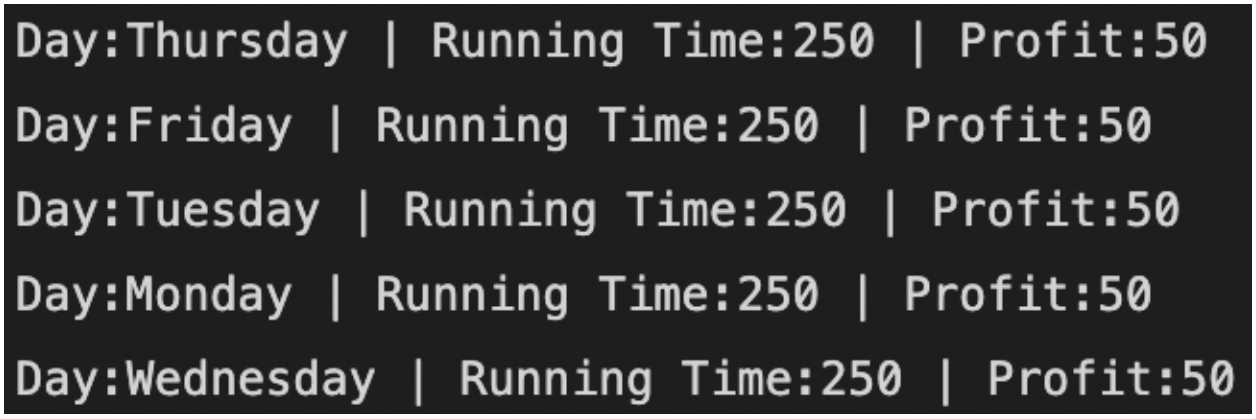

Fig. 4 shows the outputs for the dynamic scheduler with description, run-time and profits.

The dynamic scheduler still has some bugs, although it generates an output and similar to the greedy scheduler, prints days of the week based on highest to lowest priority, it prints the wrong running time and profit. Our primary guess is that it might be an index error where we are storing this information in the wrong position or our program is not iterating through the optimized tasks to select the smallest profit to print and second largest run-time to print. We can fix this by changing those comparisons with variables to give us an accurate print statement. However, apart from this minor bug, we are still getting similar results as the greedy algorithm in terms of optimizing our week. In

the Computational Critique section below, we will explain plausible faults and improvements for the dynamic scheduler.

C: Time and Space Complexity:

The code to make plots is in Appendix D. we will again make conjectures that we know that time and space complexity of dynamic programming and greedy algorithms will remain almost the same so finding one of the time or space complexity of both theoretically can provide us with answers to two complexities.

For dynamic programming, the time complexity is mainly determined by the number of subproblems that need to be solved and the time required to solve each subproblem. In the case of a task scheduler, the time complexity may be affected by the number of tasks to be scheduled and the specific constraints of the problem (e.g., end_time, dependencies, profit). In our case, we are using profit as the priority value for the organization while end_time as (time_limit) as our constraint.

Therefore, the time complexity of the dynamic_scheduler function should be O(M*N), where N is the number of elements in days and M is the value of the time_limit parameter. This is because the function has two nested for loops: one that iterates through the elements of days and another that iterates through the values from 1 to time_limit. The number of iterations for the inner loop is fixed at time_limit, and since this loop is inside the outer loop that iterates through days, the overall time complexity should be O(M*N). We expect the space complexity of this function to be O(M*N). This is because the function creates a two-dimensional table with MN elements. The size of this table will be (len(days) + 1) * (time_limit + 1), so the space complexity will be O(M*N) (AlgoDaily, n.d.).

However, experimentally, we observe O(M+1) time complexity and O(1) space complexity (Fig. 7 and Fig. 8). This is quite suspicious as we are tabulating the data and going through all the optimization of the sub-problems to check if we are reaching our global optimum. Initially, we thought that it might be because the schedulers are not working and it is background noise but by plotting each algorithm separately in Fig. 5 and 6, we found that it works completely fine. Therefore, we will trust the experimental plots with a grain of salt. It cannot be background noise as each task runs at 0 seconds and if the scheduler is not working, it should end at 0 seconds but it is taking some time to generate output so we will continue with our analysis.

For greedy algorithms, the time complexity is mainly determined by the number of iterations required to find a feasible solution. Based on the iterations the function has a scaling growth of O(N^2). This is because the function has two nested for loops: one that iterates through the elements of days and another that iterates through the elements of task_profit_obj.tasks. The number of iterations for the inner loop will depend on the size of task_profit_obj.tasks, and since this loop is inside the outer loop that iterates through days, the overall time complexity is O(N^2).

The space complexity of this algorithm is O(N) as it creates a new list “greedy” and appends one element to it for each iteration of the outer loop. Given that the size of this list will be equal to the number of elements in days, so the space complexity is O(N).

Experimentally, we observe an O(1) space and time complexity based on Fig. 7 and 8(code in Appendix D). We checked if there might be a problem with how input is scaling with time and space so we plotted each algorithm separately in Fig. 5 and 6 and found that it works

completely fine. Therefore, we will trust the experimental plots with a grain of salt. Although, experimentally, the plots show that it uses more space than a simple task scheduler but we will ignore that as a simple task scheduler only considers tasks from one day compared to a week in the greedy scheduler. Therefore, theoretically, considering the number of operations and their individual space and time complexity is the limit for assuming or calculating the space and time complexities of an algorithm.

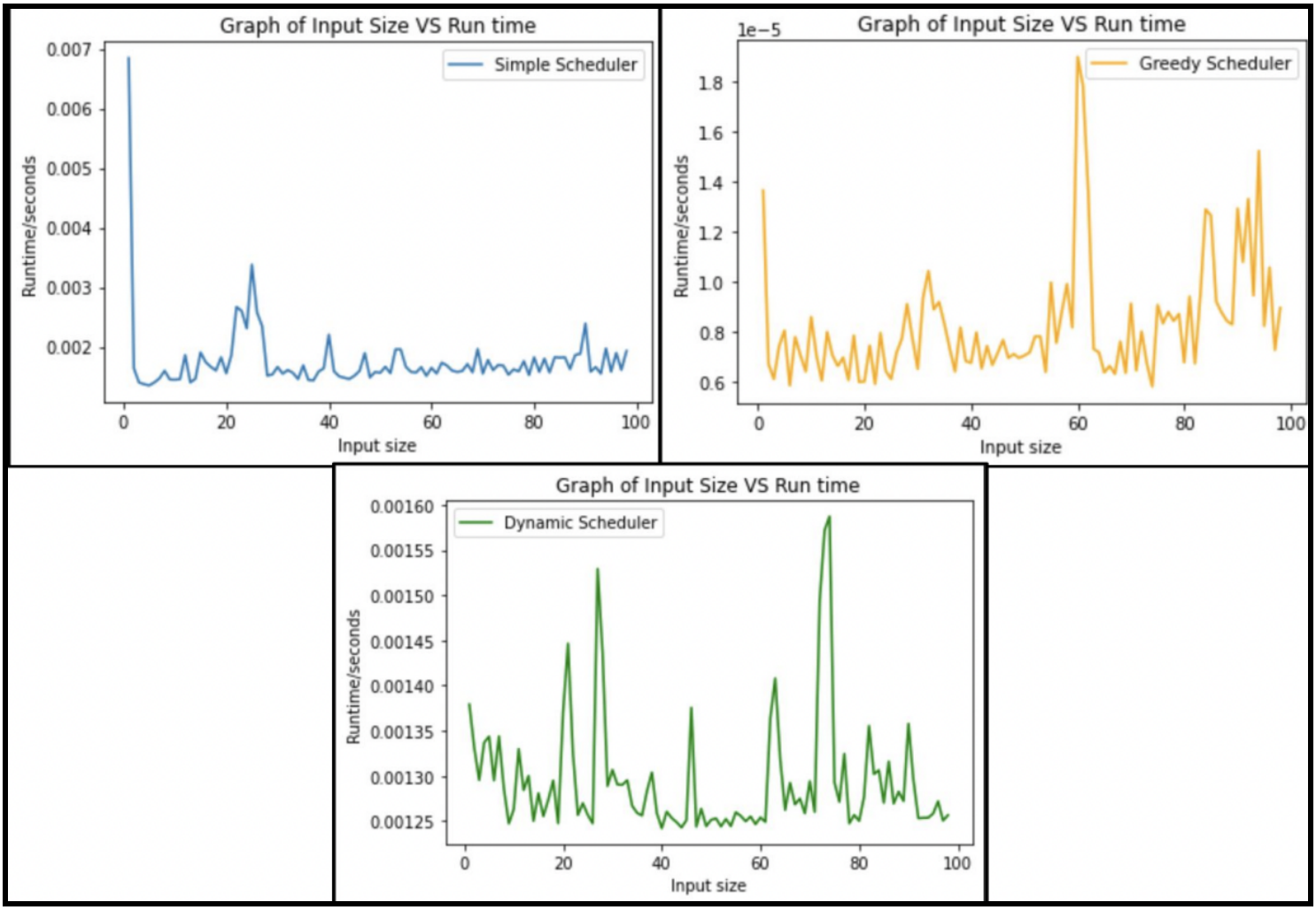

Fig. 5 shows the experimental plots of the three schedulers (simple, greedy, and dynamic) for input size vs run time (individually), to provide context that each scheduler is working based on the output generated shows different run-time scales for each plot.

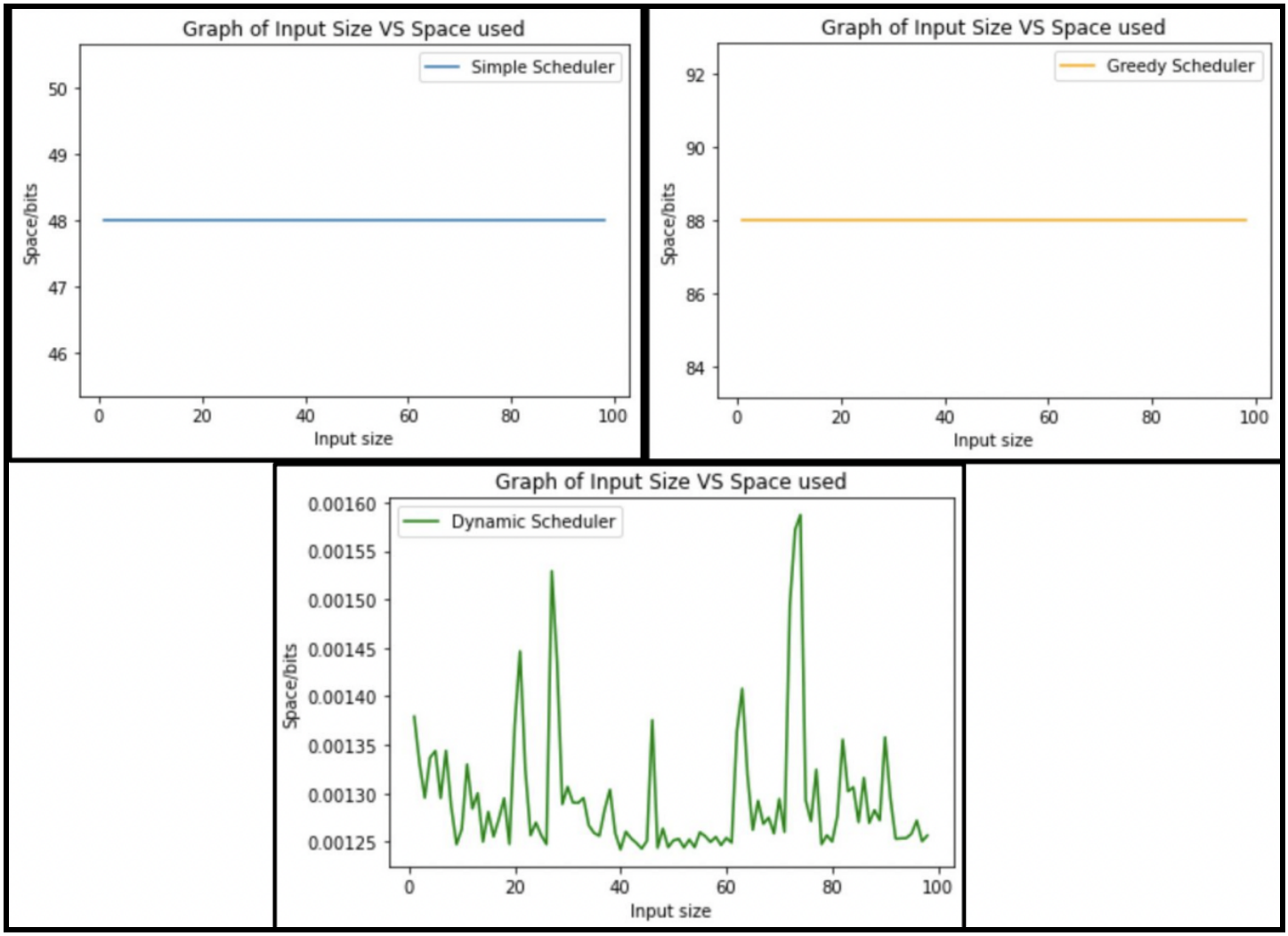

Fig. 6 shows the experimental plots of the three schedulers (simple, greedy and dynamic) for input size vs space used (individually), to provide context that each scheduler is working based on the output generated shows different run-time scales for each plot.

D: Greedy and DP Multi-tasking:

We believe that multitasking can be possible by creating a multitasking function beforehand in the simple task scheduler which can be used to create multi-tasks. Adding multi-tasks in the optimization approach would not make a difference but rather increase the time and space complexity of the algorithms. Therefore, building on the Multitasking section above, I will inherit those properties to use a multi-task scheduler instead of a simple scheduler for my optimization methods.

Q5 – Computational Critique

A:

No, the three implementations do not use the same data structure as the simple task scheduler uses heaps and priority queues while the optimization algorithms build on that scheduler and use lists to keep track of the tasks that are scheduled. DP goes a step further and uses the lists to make tables for finding optimal substructure for overlapping subproblems. The reason for the use of lists in these algorithms is discussed in the Algorithmic Strategy section above.

B:

We haven't implemented a multitasking method but the algorithms do utilize priority values in a better way. Although we do not have outputs for our optimization by modifying the simple scheduler we were able to see that our tasks scheduler was considering fixed tasks as well and can organize tasks up to 100 tasks which per day is quite a huge number but even for a week can be considered a decent input size. Other than that the optimations are not made to consider data stream and bulk data as we are only using one list, we have discussed this and the implementation in detail in the Multitasking section above.

However, in terms of the dynamic scheduler, there are some python implementations that can be improved that were not directly discussed in question 1 but rose through our Python Implementation section. We face the following problems. Firstly, the dynamic_scheduler function is trying to create a table of values using a nested list comprehension, but the resulting list is not a valid table because it does not have the same number of rows as the day's list. The dynamic_scheduler function tries to populate the table by iterating over the day's list and using the duration and profit attributes of the Task_profit objects, but these attributes are not accessed from Task_proftit as the task details are not working while run_task_scheduler operates. Therefore, to fix these issues, we will need to update the Task_profit class and the dynamic_scheduler function to use the correct attribute names and to implement a valid dynamic programming solution. Here is an example of how we could modify the Task_profit class and the dynamic_scheduler function to achieve this.

C:

Theoretically and experimentally, I used time and space complexities as metrics as shown in Fig. 7, 8 and 9. We expect to see O(N^2) scaling growth for the simple task scheduler and greedy and as we mentioned above dynamic would be between linear and quadratic O(M*N) as shown in Fig. 9. We are using these metrics as they cover the operations that these algorithms implement and by comparison we can check the efficiency of these algorithms. Furthermore, given that our constraint is time, using time would provide relevant context about whether we are optimizing time to complete tasks as well as reducing the time of implementing the algorithm. We are using space, especially to see dynamic space utilization as we are using tabulation to store multiple tasks and profit in the given time constraint. We can also use the profit that we

have been using as an output or our priority values to check which algorithm produces the better utility as the purpose of optimization is to find global solutions and make solutions more efficient and profit/priority values maximize that.

D:

Fig. 7 and 8 show scaling growth with respect to time ans space respectively. Whereas, Fig. 9 shows the expected scaling growth for schedulers with respect to time, theoretically. Based on our theoretical expectations, we expected greedy and dynamic schedulers to have similar time efficiency but greedy algorithms more efficient in terms of space. However, based on our experimental plots, we see greedy algorithms to have the highest efficiency constant scaling growth from both time and space perspectives O(1) and O(M+1) while dynamic has O(M+1) and O(1) time and space scaling growth. This contradicts our expectations as they are almost similar in terms of scaling growth with a slight difference in terms of greedy is taking more space while DP is taking more time so based on what the user prefers, either one can be a good fit. Lastly, the noise in the time_complexity could be due to our priority value calculation as we are using 10,000 as a scaling factor to get large enough priority values and the range for end_time is large which can cause more anomalies. Lastly, in Fig. 5 and 6 we can see the noise more closely which is caused by the random task generator that produces tasks that have a longer duration than the end_time so no task for that week can be completed which leads to fluctuation in scaling growth in run-time.

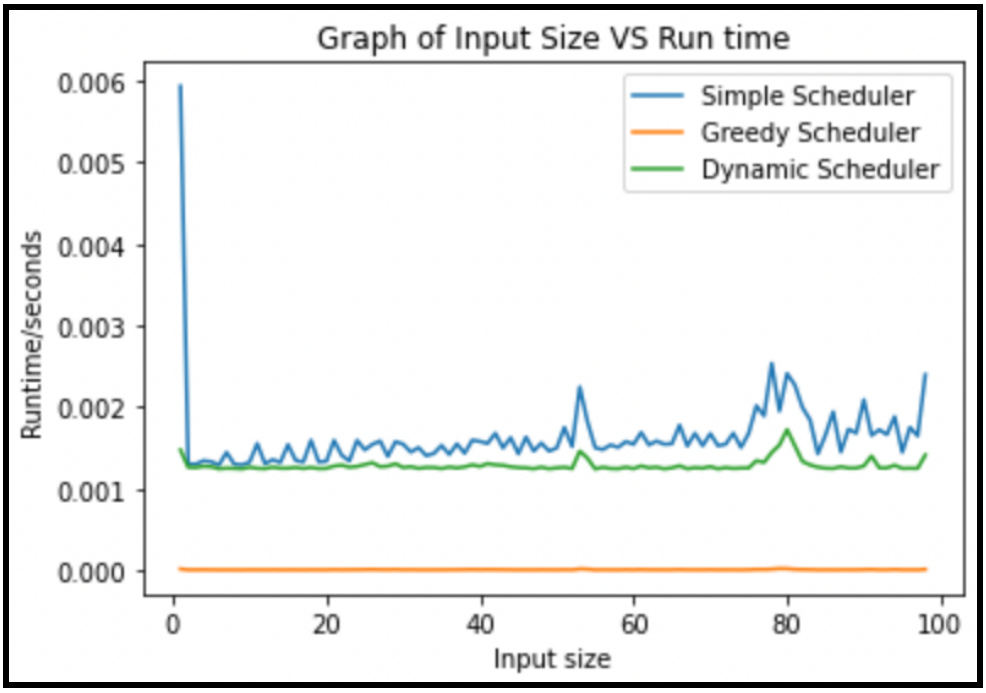

Fig. 7 shows experimental plots of the three schedulers with respect to input size vs run time (scaling growth based on time). It shows O(log(M+N)) scaling growth for simple task scheduler and O(1) for greedy and as we mentioned above that dynamic would be between linear and quadratic so O(M+1)

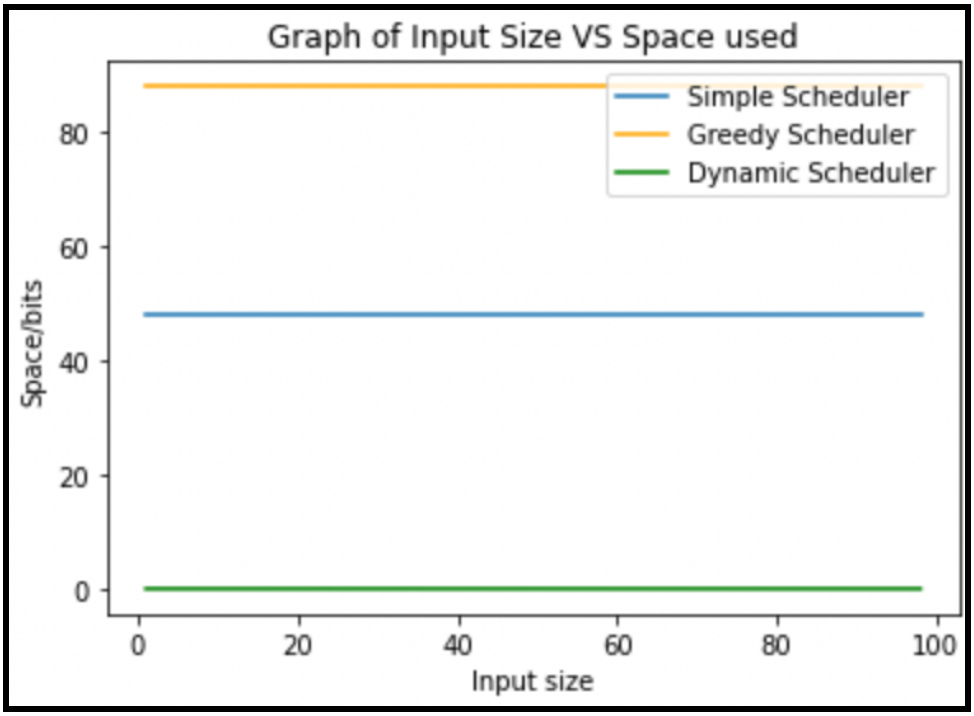

Fig. 7 shows experimental plots of the three schedulers. It shows O(1) scaling growth for the simple task, greedy and dynamic schedulers.

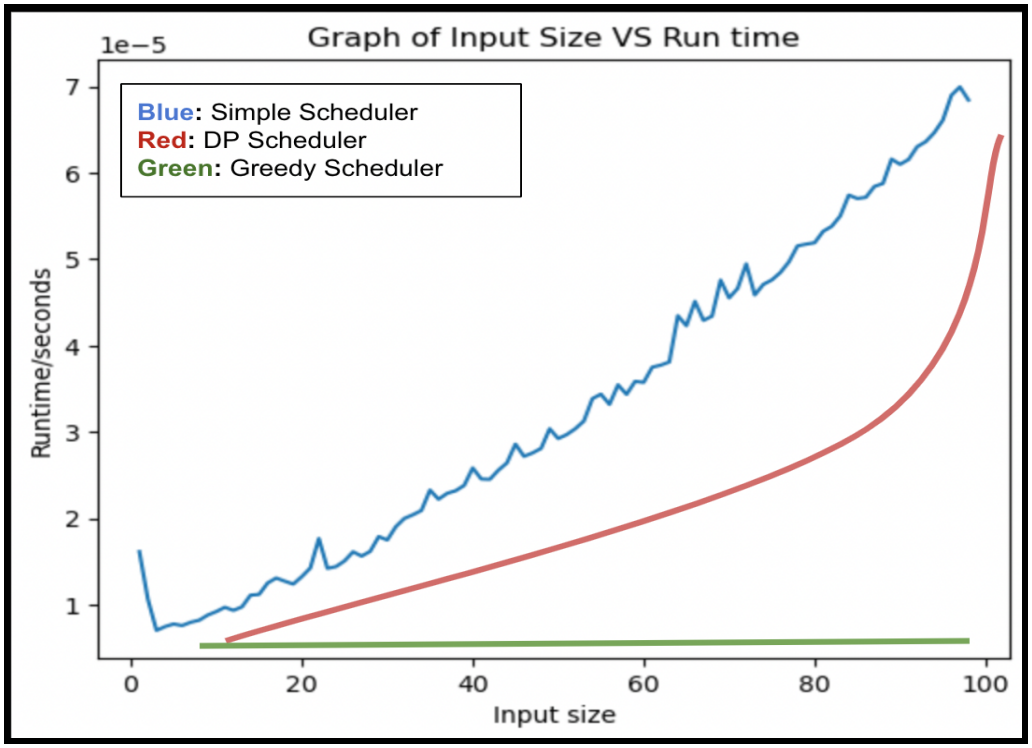

Fig. 9 shows our expectation of the algorithm using google draw to modify our python visualization.

E:

From a theoretical and experimental perspective, using the plots from Fig. 3 and 4. Even though the greedy algorithm approach does not give global optimal solutions on every run but as a user, my purpose is to find the profit of doing tasks in a certain way and what would be the best way of doing that. Just in terms of the optimal way of doing tasks to maximize dynamic programming would be better as although it might not be fast but it does provide the correct solution each time and is more consistent. Whereas, the greedy approach will be fast but not consistent We would not use a simple task scheduler as we expect that it will have the worst performance as stated in the previous assignment. We can use these assessments to critique the algorithms as well as by using a list and tabulation method for both the optimization methods we expect to see a better performance which proves our point from the last assignment that priority queues and heaps are not the best data structure for task scheduling as on asymptotic regime we will see more noise which indicates either loss of data or burden on the servers when considering data steam. Therefore, the best strategy depends on the user's preference but personally, we should aim for the most optimal solution to increase the profit as in real-life scenarios even for large data sets, one extra minute would not make as big a difference as finding an optimal path for doing tasks which can save hours. So the best strategy is dynamic programming, especially as we saw from experimental plots that both of the approaches have similar scaling growth.

Appendix Part I

Learning Objectives: (I have modified these descriptions from the LBA Part I assignment) (Bukhari, 2022)

1. **#cs110-PythonProgramming:**

I have modified code from the last class to consider priority values using durations and time limits compared to using dependencies before as assumed priority values for prioritizing the tasks. I have provided plausible test cases that might be needed in a practical situation while using the task scheduler and attempted to clear the bugs from my code. Lastly, I used run-time for my simple tasks scheduler as a metric to plot experimental plots to show the efficiency of my code while commenting on what improvement can be made and how. I still have some flaws in my DP approach but I have provided multiple methods to fix those problems with moret time in the Computational Critique section.

Word Count: 65 words

2. **#cs110-ComputationalCritique:**

I have analyzed run-time and space as metrics with respect to theoretical and experimental plots to comment on the efficiency of my code. I have theoretically explained the effect of higher input size on the run-time and space and assessed possible causes of the noise in the asymptotic regime. I have provided reasonable justifications for using time limits, duration and profits for the optimization of my code. Lastly, I have provided solutions for improving the code e..g adding more features like multitasking and recommended how to use other metrics to check the efficiency of the code e.g. priority values or profit.

Word Count: 62 words

3. **#cs110-CodeReadability:**

I have included detailed docstrings to inform the audience of the inputs, outputs, and purpose of a function. I added multiple inline comments to justify the implementation of the code and what we are trying to do. I have used clear, concise, and consistent variable names to provide ease when fixing or reusing code. Lastly, I have included multiple success and error messages that can be used to pinpoint the areas of improvement once reusing the code.

Word Count: 57 words

1. **#cs110-AlgoStratDataStruct:**

I have compared the use of heaps, priority queue, and lists in optimizing our task scheduler. Moreover, I have suggested possible improvements and the strategy to implement it using python e.g. multi-tasking feature using two priority queues and use analogies as a means to explain the algorithm e.g. swimming duck analogy. Lastly, I have used the past feedback to consider superseding effect using the above data structures and how to handle data stream and bulk data storage using our scheduler.

Word Count: 48 words

2. **#cs110-Professionalism:**

I have used the word "We" to refer to the author and the reader to improvise the interactivity of the reader with my text. I have explained in the comments above the code where I got the code from and which modifications I made to avoid plagiarism. I included all the code and plots that were required for the assignment. Lastly, I ensured to use a professional tone, and concise language to present my work with specific and direct arguments and added figure captions and headers at the top of the page.

Word Count: 45 words

3. **#cs110-ComplexityAnalysis:**

I have analyzed the time and space complexity analysis of the there algorithms theoretically and experimentally. I have utilized multiple lines of evidence to build on my intuitions and dissected operations to find the overall asymptotic growth of the algorithms. Lastly, I justified why we will observe these complete and how they relate to data structures and algorithms (based on how each scheduler will manage time and space) and can be improved by modifying the data structures.

Word Count: 51 words

HCs: (Bukhari, 2022)

1. **#algorithms:**

I have implemented complex algorithms using heaps, priority queues, classes and lists and tables. I have used small steps and comments with my code to show the implementation of the algorithms. Lastly, I have used multiple analogies, constraints, and requirements to explain the use of the dynamic and greedy methods and which one is better for building a task scheduler based on metrics like time and space complexity, profits, and practical reasoning.

Word Count: 48 words

2. **#constraints:**

I have identified multiple constraints for our algorithm. Firstly, I used the constraint of durations and time limits to modify my priority values. Secondly, I identified and provided critiques on constraints like limited space and how to use two priority queues to make a multi-task scheduler. Lastly, I overcame the constraint of no scaling growth in contradiction to my expectation but commented the reasoning behind the change and possible ways we use optimization on DP approach.

Word Count: 52 words

3. **#composition:**

I have used composition to present information in a parsimonious manner by using relevant analogies and examples to explain the data structures and the algorithm. I used comments and docstrings to compose code that is easy to understand and clear for all external users/audiences by keeping track of complicated operations and using one-line in-line comments. I have attempted to ensure that words are not repeated until necessary and simple language to convey the purpose of the data structures and algorithms.

Word Count: 58 words

Word Count for both Appendix: 622 words[1]

[1] Note: the word count within the panels/screenshots can be ignored as they were not updated with changes made in the text. Hence, the final word count of the appendix is provided.

Appendix Part II: A

Packages:

```
#import relevant packages
import pandas as pd
import numpy as np
import time
import matplotlib
from matplotlib import pyplot as plt
import random
```

Simple Task Scheduler:

```
#adapted from CS110 Session 13 - [7.2] Heaps and priority queues and LBA Part I Assignment

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 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.priority_value < self.heap[i]:
            raise ValueError('new key is smaller than the current key')
        self.heap[i] = key
        #switches parent and child if the paraent is smaller than the child
        while i > 0 and self.heap[self.parent(i)].priority_value <
self.heap[i].priority_value:
            j = self.parent(i)
            holder = self.heap[j]
            self.heap[j] = self.heap[i]
            self.heap[i] = holder
            #switches the current node to the index of the parent
```

```
            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
        """
        #compares the left and right tasks
        l = self.left(i)
        r = self.right(i)
        heap = self.heap
        #considers the largest task on left
        if l <= (self.heap_size-1) and heap[l]>heap[i]:
            largest = l
        else:
            largest = i
        #considers the largest task on right
        if r <= (self.heap_size-1) and heap[r] > heap[largest]:
            largest = r
        if largest != i:
            #Organizes the heap based on prioratized task
            heap[i], heap[largest] = heap[largest], heap[i]
            self.heapify(largest)

    def heappop(self):
        """
        returns the largest 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]
        #removes the task
        self.heap.pop()
        self.heap_size-=1
        self.heapify(0)
        return maxk

    def print_heap(self):
        """
        prints all the tasks with their descriptions and priority values in the heap

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

        Returns
        ----------
         """

        for i in self.heap:
            print("\n",f'Task: {i.description} and its priority value is
{10000/(i.end_time - i.duration)+1}')

#Code taken from Session 7.2: Heaps and priority queues
#The code I solved with my group was mainly utilized
#with attribute changes and changes to use MaxHeapq instead of heapq module.
```

```
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 preceed this task
    - status: Current status of the task

    """
    #Initializes an instance of Task
    def __init__(self, id, description, duration, dependencies, end_time,status="N"):
        """
        Parameters
        ----------
        None
        """
        self.id = id
        self.description = description
        self.duration = duration
        self.dependencies = dependencies
        self.status = status
        self.end_time = end_time
        self.priority_value = self.priority_calculator(self.end_time, self.duration)

    def  priority_calculator(self, end_time, duration):
        """
        returns the priority value of a task

        Parameters
        ----------
        end_time: int
                  the ending time of a task

        duration: int
                  the duration of a task

        Returns
        ----------
        int
```

```
            the priority value of the task
        """
        return 10/ (end_time - duration)+1

    def __lt__(self, other):
        """
        check if a task has a greater priority value than a different task

        Parameters
        ----------
        other
            class instance
            a different task

        Returns
        ----------
        Boolean

        """
        return self.priority_value < other.priority_value

#inherits the properties from MaxHeapq
class TaskScheduler(MaxHeapq):
    """
    A Simple Daily Task Scheduler Using Priority Queues
    """
    NOT_STARTED = 'N'
    IN_PRIORITY_QUEUE = 'I'
    COMPLETED = 'C'

    def __init__(self, tasks):
        self.tasks = tasks
        super().__init__()

    #prints all the tasks and their dependencies
    def print_self(self):
        """
        prints all the task 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!")

    #removes a particular task from the dependencies list of other tasks
    def remove_dependency(self, id):
        """
        removes a particular task from the dependencies list of other tasks

        Parameters
        ----------
        id
          task Id

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

        for t in self.tasks:
            if t.id != id and id in t.dependencies:
                t.dependencies.remove(id)

    #gets the tasks ready by checking for tasks that has not been started and without
dependencies and pushing it into the heap
    def get_tasks_ready(self):
        """
```

```
        gets the tasks ready by checking for tasks that has not been started and
without dependencies and pushing it into the heap

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

        Returns
        ----------
        None
        """
        for task in self.tasks:
            # If the task has no dependencies and is not yet in the queue
            if task.status == self.NOT_STARTED and not task.dependencies:
                # Update status of the task
                task.status = self.IN_PRIORITY_QUEUE
                # Push task into the priority queue
                self.heappush(task)


    #checks for unscheduled tasks (tasks whose started is "NOT_STARTED")
    def check_unscheduled_tasks(self):
        """
        returns the largest 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
        """
        for task in self.tasks:
            if task.status == self.NOT_STARTED:
                return True
        return False
```

```
def format_time(self, time):
    """
    returns the time in the hour minute format

    Parameters
    ----------
    time
        int
        time in minutes

    Returns
    ----------
    string
        time in the hour minute format
    """
    return f"{time//60}h{time%60:02d}"

#runs the task scheduler
def run_task_scheduler(self, starting_time):
    """
    runs the task scheduler by popping items from the heap and running
    the get task ready function and the remove dependency function

    Parameters
    ----------
    starting_time
                int
                starting time of the task scheduler

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

    descriptions=[]
    current_time = starting_time
    print("Running a simple scheduler:\n")
    while self.check_unscheduled_tasks() or self.heap:
```

```
            # Identify tasks that are ready to execute
            # (those without dependencies) and
            # push them into the priority queue
            if self.check_unscheduled_tasks():
              self.get_tasks_ready()
            #Check for tasks in the priority queue
            if len(self.heap) > 0 :
                # get the task on top of the priority queue
                task =  self.heappop()
                descriptions.append(task.description)
                print(f"⏰t={self.format_time(current_time)}")
                print(f"\tstarted '{task.description}' for {task.duration} mins...,
with a priority value of {round(10000/(task.end_time - task.duration)+1,1)}")
                current_time += task.duration
                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
        total_time = current_time - starting_time
        print(f"\n🏁 Completed all planned tasks in
{total_time//60}h{total_time%60:02d}min!")
        return descriptions
```

Test Case:

```
#I have removed some initial tasks and changed some dependencies.
#the code is working on by prioratizing dependencies
#dependencies act as the main priority which is also how I derived my priority values

start_scheduler = 8*60
#my tasks
task_0 = Task(id=0, description='8 am: Lab Work',
        duration=180, dependencies=[],end_time= 300)
task_1 = Task(id=1, description='4 pm: Take a tour of National Taiwan Museum ',
        duration=150, dependencies=[0],end_time= 480)
task_2 = Task(id=2, description='7 pm: Light Sky Lanterns at a festival',
        duration=160, dependencies=[0,1],end_time= 600)
```

```
task_3 = Task(id=3, description='9:30 pm: Visit Taipei Night Market for Scallion
pancakes',
        duration=75, dependencies=[0,1,2],end_time= 650)
task_4 = Task(id=4, description='10 pm: Returning to the res-hall via a bus',
        duration=30, dependencies=[0,1,3], end_time= 700 )
task_5 = Task(id=5, description='11 pm: Talking with Minervans ',
        duration=60, dependencies=[0,1,4], end_time= 750)
task_6 = Task(id=6, description='11pm: Laundry',
        duration=60, dependencies=[0,1,4], end_time= 800)

monday = [task_0 ,task_1, task_2, task_3,
        task_4, task_5, task_6]

task_scheduler = TaskScheduler(monday)
task_scheduler.run_task_scheduler(start_scheduler)
```

Appendix Part II: B

Dynamic Programming:

```
class Task_profit:
    """
    A shorter version of class Task but with profit for completing tasks.
    """

    def __init__(self, description, tasks, duration,profit):
        self.tasks = tasks
        self.description = description
        self.duration = duration
        self.profit = profit

    def task_details(self):
        '''
        It prints the deatils of the activities
        '''
        return (self.description, self.tasks, self.duration, self.profit)


def dynamic_scheduler(days, time_limit):
    """
    Dynamic programming alogrithm appraoch to return profit created by completing
maximum tasks in a given time limit.

    Parameter
    --------------
    days: list
        a list containing the task for each day that need to be completed
    time_limit: int
        an integar for time limit for the week.

    Returns:
    --------------
    list
        optimal list of tasks, time run and the profit generated
    """
```

```
    #time limit
    time = time_limit
    #profit for scheduler
    profit = 0
    #order tasks based on higher profit
    days = sorted(days, key = lambda i: i.profit, reverse = True)

    #create a table to store the results of subproblems
    table = [[0 for _ in range(time + 1)] for _ in range(len(days) + 1)]

    #iterate through each day
    for i in range(1, len(days) + 1):
        task_profit_obj = days[i - 1]
        curr_profit = task_profit_obj.profit
        curr_day_duration = 0

        #iterate through each task in the current day
        for j in task_profit_obj.tasks:
            task_obj = j
            curr_day_duration += task_obj.duration

        #populate the table with the maximum profit that can be achieved
        #for each time limit and for each day
        for t in range(1, time + 1):
            if curr_day_duration <= t:
                table[i][t] = max(table[i - 1][t], table[i - 1][t - curr_day_duration]
+ curr_profit)
            else:
                table[i][t] = table[i - 1][t]

    #reconstruct the solution by starting from the last day and working backwards
    i = len(days)
    t = time
    dynamic = []
    while i > 0 and t > 0:
        if table[i][t] != table[i - 1][t]:
            dynamic.append([days[i - 1], curr_day_duration, curr_profit])
            t -= curr_day_duration
```

```
        i -= 1

    return dynamic[::-1]
```

Inputs and Test Case:

```
#week's tasks:

monday = [Task(id=0, description='8 am: Lab Work',
        duration=100, dependencies=[],end_time= 200),
        Task(id=1, description='4 pm: Take a bus to Sky Lantern Festival',
        duration=50, dependencies=[0],end_time= 300),
        Task(id=2, description='7 pm: Light Sky Lanterns at a festival',
        duration=160, dependencies=[0,1],end_time= 400),
        Task(id=3, description='9:30 pm: Visit Taipei Night Market for Scallion
pancakes',
        duration=75, dependencies=[0,1,2],end_time= 450),
        Task(id=4, description='10 pm: Returning to the res-hall via a bus',
        duration=30, dependencies=[0,1,3], end_time= 450 )]

tuesday = [Task(id=0, description='8 am: Lab Work',
        duration=60, dependencies=[],end_time= 200),
        Task(id=1, description='4 pm: Take a tour of National Taiwan Museum ',
        duration=100, dependencies=[0],end_time= 250),
        Task(id=2, description='9:30 pm: Visit Taipei Night Market for Scallion
pancakes',
        duration=75, dependencies=[0,1],end_time= 300),
        Task(id=3, description='10 pm: Returning to the res-hall via a bus',
        duration=30, dependencies=[0,1,2], end_time= 340 ),
        Task(id=4, description='11 pm: Talking with Minervans ',
        duration=60, dependencies=[0,1,3], end_time= 400),
        Task(id=5, description='11pm: Laundry',
        duration=60, dependencies=[0,1,3], end_time= 400)]

wednesday = [Task(id=0, description='8 am: Lab Work',
        duration=100, dependencies=[],end_time= 200),
        Task(id=1, description='4 pm: Take a tour of National Taiwan Museum ',
        duration=50, dependencies=[0],end_time= 250),
```

```
        Task(id=2, description='7 pm: Light Sky Lanterns at a festival',
        duration=100, dependencies=[0,1],end_time= 250)]

thursday = [Task(id=0, description='8 am: Lab Work',
        duration=180, dependencies=[],end_time= 200),
        Task(id=1, description='11 pm: Talking with Minervans ',
        duration=60, dependencies=[0], end_time= 330),
        Task(id=2, description='11pm: Laundry',
        duration=60, dependencies=[0,1,], end_time= 350)]

friday = [Task(id=0, description='8 am: Lab Work',
        duration=180, dependencies=[],end_time= 300),
        Task(id=1, description='4 pm: Take a tour of National Taiwan Museum ',
        duration=150, dependencies=[0],end_time= 300),
        Task(id=2, description='11 pm: Talking with Minervans ',
        duration=60, dependencies=[0,1], end_time= 400),
        Task(id=3, description='11pm: Laundry',
        duration=60, dependencies=[0,1], end_time= 450)]

monday_tasks = Task_profit('Monday', monday,
TaskScheduler(monday).run_task_scheduler(start_scheduler),100)
tuesday_tasks = Task_profit('Tuesday', tuesday,
TaskScheduler(tuesday).run_task_scheduler(start_scheduler),150)
wednesday_tasks = Task_profit('Wednesday', wednesday,
TaskScheduler(wednesday).run_task_scheduler(start_scheduler),50)
thursday_tasks = Task_profit('Thursday', thursday,
TaskScheduler(thursday).run_task_scheduler(start_scheduler),250)
friday_tasks = Task_profit('Friday', friday,
TaskScheduler(friday).run_task_scheduler(start_scheduler),220)
optimized_week_tasks = [monday_tasks,tuesday_tasks, wednesday_tasks,
thursday_tasks,friday_tasks]
```

Appendix Part II: C

Greedy Algorithm:

```
def greedy_scheduler(days, time_limit):
   """
   Greedy alogrithm appraoch to return profit created by completing maximum tasks in a
given time limit.

   Parameter
   --------------
   days: list
       a list containing the task for each day that need to be completed
   time_limit: int
       an integar for time limit for the week.

   Returns:
   --------------
   list
       optimal list of tasks, time run and the profit generated
   """

   #time limit
   time = time_limit
   #profit for scheduler
   profit = 0
   #order tasks based on higher profit
   days = sorted(days, key = lambda i: i.profit, reverse = True)

   #greedy schedule
   greedy = []

   for i in range(len(days)):
     task_profit_obj = days[i]
     curr_profit = task_profit_obj.profit
     #keeping track of current day's duration
     curr_day_duration = 0

     #Reduce time from time_limt when current day's task is completed
```

```
    for j in task_profit_obj.tasks:
      task_obj = j
      curr_day_duration += task_obj.duration
      time -= task_obj.duration

    #If we can finish the tasks during the time limit,
    #then increase the profit
    if curr_day_duration <= time:
      profit += curr_profit
      greedy.append([task_profit_obj, curr_day_duration, curr_profit])
   return greedy
```

Test cases:

```
#test case 1: large time constraint to complete all tasks
#to make changes in the profit we need to change the profit and tasks in the input


optimal_profit_greedy = greedy_scheduler(optimized_week_tasks, 2500)
for i in range(len(optimal_profit_greedy)):
 print("Day:" + optimal_profit_greedy[i][0].description + " | " + 'Running Time:' +
str(optimal_profit_greedy[i][1]) + " | " + 'Profit:' +
str(optimal_profit_greedy[i][2]))
```

```
#test case 2: limited tasks completed based on small time constraint
optimal_profit_greedy = greedy_scheduler(optimized_week_tasks, 1500)
for i in range(len(optimal_profit_greedy)):
 print("Day:" + optimal_profit_greedy[i][0].description + " | " + 'Running Time:' +
str(optimal_profit_greedy[i][1]) + " | " + 'Profit:' +
str(optimal_profit_greedy[i][2]))
```

```
#test case 3: duplicate profits
```

```
monday_tasks = Task_profit('Monday', monday,
TaskScheduler(monday).run_task_scheduler(start_scheduler),100)
tuesday_tasks = Task_profit('Tuesday', tuesday,
TaskScheduler(tuesday).run_task_scheduler(start_scheduler),150)
wednesday_tasks = Task_profit('Wednesday', wednesday,
TaskScheduler(wednesday).run_task_scheduler(start_scheduler),50)
thursday_tasks = Task_profit('Thursday', thursday,
TaskScheduler(thursday).run_task_scheduler(start_scheduler),50)
friday_tasks = Task_profit('Friday', friday,
TaskScheduler(friday).run_task_scheduler(start_scheduler),220)
optimized_week_tasks = [monday_tasks,tuesday_tasks, wednesday_tasks,
thursday_tasks,friday_tasks]

optimal_profit_greedy = greedy_scheduler(optimized_week_tasks, 10)
for i in range(len(optimal_profit_greedy)):
 print("Day:" + optimal_profit_greedy[i][0].description + " | " + 'Running Time:' +
str(optimal_profit_greedy[i][1]) + " | " + 'Profit:' +
str(optimal_profit_greedy[i][2]))
```

Appendix Part II: D

Time Complexity:

```
#removing some print statements cause errors so I have kept them while making plots

import sys

space_used_1 = []
space_used_2= []
space_used_3 = []
tracker_2 = []
input_size_2 = []
experiments = 50

for i in range(2, 100):
   no_exp_2 = []
   for n in range(1,i):
       no_exp_2.append(Task(id = i-1, description = i, duration = random.randrange(50,
200), dependencies = [], end_time = random.randrange(1000, 10000)))
   tracker_2.append(no_exp_2)

for i in tracker_2:
   A_2 = []
   B_2 = []
   C_2 = []

   for x in range(experiments):
       #A_2
       # should give you the full object size
       tasks_scheduler = TaskScheduler(i)
       tasks_scheduler.run_task_scheduler(start_scheduler)
       size_1 = sys.getsizeof(tasks_scheduler)
       A_2.append(size_1)

       #B_2
       tasks_scheduler_2 = greedy_scheduler(optimized_week_tasks, 1000)
       size_2 = sys.getsizeof(tasks_scheduler_2)
       B_2.append(size_2)
```

```
        #C_2
        tasks_scheduler_3 = dynamic_scheduler(optimized_week_tasks, 1000)
        size_3 = sys.getsizeof(tasks_scheduler_3)
        C_2.append(size_3)

    input_size_2.append(len(i))

    space_used_1.append(sum(A_2)/experiments)
    space_used_2.append(sum(B_2)/experiments)
    space_used_3.append(sum(C_2)/experiments)

plt.plot( input_size_2, space_used_1, label='Simple Scheduler')
plt.plot( input_size_2, space_used_2, label='Greedy Scheduler',  color = 'orange')
plt.plot( input_size, run_time_3, label='Dynamic Scheduler', color = 'green')
plt.xlabel('Input size')
plt.ylabel('Space/bits')
plt.title('Graph of Input Size VS Space used')
plt.legend()
```

Space Complexity:

```
#removing some print statements cause errors so I have kept them while making plots

import sys

space_used_1 = []
space_used_2= []
space_used_3 = []
tracker_2 = []
input_size_2 = []
experiments = 50

for i in range(2, 100):
   no_exp_2 = []
   for n in range(1,i):
       no_exp_2.append(Task(id = i-1, description = i, duration = random.randrange(50,
200), dependencies = [], end_time = random.randrange(1000, 10000)))
   tracker_2.append(no_exp_2)

for i in tracker_2:
   A_2 = []
   B_2 = []
   C_2 = []

   for x in range(experiments):
       #A_2
       # should give you the full object size
       tasks_scheduler = TaskScheduler(i)
       tasks_scheduler.run_task_scheduler(start_scheduler)
       size_1 = sys.getsizeof(tasks_scheduler)
       A_2.append(size_1)

       #B_2
       tasks_scheduler_2 = greedy_scheduler(optimized_week_tasks, 1000)
       size_2 = sys.getsizeof(tasks_scheduler_2)
       B_2.append(size_2)

       #C_2
```

```
        tasks_scheduler_3 = dynamic_scheduler(optimized_week_tasks, 1000)
        size_3 = sys.getsizeof(tasks_scheduler_3)
        C_2.append(size_3)

    input_size_2.append(len(i))

    space_used_1.append(sum(A_2)/experiments)
    space_used_2.append(sum(B_2)/experiments)
    space_used_3.append(sum(C_2)/experiments)

plt.plot( input_size_2, space_used_1, label='Simple Scheduler')
plt.plot( input_size_2, space_used_2, label='Greedy Scheduler',  color = 'orange')
plt.plot( input_size, run_time_3, label='Dynamic Scheduler', color = 'green')
plt.xlabel('Input size')
plt.ylabel('Space/bits')
plt.title('Graph of Input Size VS Space used')
plt.legend()
```

References

AlgoDaily. (n.d.). *AlgoDaily - Daily coding interview questions. Full programming interview prep course and software career coaching.* Algodaily.com. Retrieved December 15, 2022, from https://algodaily.com/lessons/getting-to-know-greedy-algorithms-through-examples

Bukhari, S. H. (2022, November). *Project 1 - A Day in the Life of a Minervan (Part I)*.

Gray, N. (2019, July 24). *Data Structures Part 1: Bulk Data*. Game Developer. https://www.gamedeveloper.com/programming/data-structures-part-1-bulk-data

Heaps and priority queues [7.2]. Pre-Class Work. https://sle-collaboration.minervaproject.com/?id=06646410-f55c-4e96-87b9-668cab17cb69&userId=11949&name=Hassan+Bukhari&avatar=https%3A//s3.amazonaws.com/picasso.fixtures/Syed_Hassan%20Bukhari_11949_2021-08-30T02%3A58%3A10.807Z&iframed=1&readOnly=0&isInstructor=0&enableSavingIndicators=1&signature=f094960cb55702032ee41361b81df7e6e08db36cc16488f1201334a1ad034a15

Heaps and priority queues [7.2]. Breakout room. https://sle-collaboration.minervaproject.com/?id=14caae4d-9085-4ce0-9555-284834dc8e62&userId=11949&name=Hassan+Bukhari&avatar=https%3A//s3.amazonaws.com/picasso.fixtures/Syed_Hassan%20Bukhari_11949_2021-08-30T02%3A58%3A10.807Z&isInstructor=0&signature=20b021307bd97dd1e07ae5d1b197fd3e71b142f537a4923ea885c11b1e4366d0

Juniper Networks. (2021, January 13). *Strict-Priority Queue Overview*. Www.juniper.net; Juniper Networks. https://www.juniper.net/documentation/us/en/software/junos/cos-security-devices/topics/concept/cos-strict-priority-queue-security-overview.html

Kumar, N. (2017, April 13). *Tabulation vs Memoization*. GeeksforGeeks. https://www.geeksforgeeks.org/tabulation-vs-memoization/

Walker, A. (2020, February 6). *Greedy Algorithm with Examples*. Guru99. https://www.guru99.com/greedy-algorithm.html