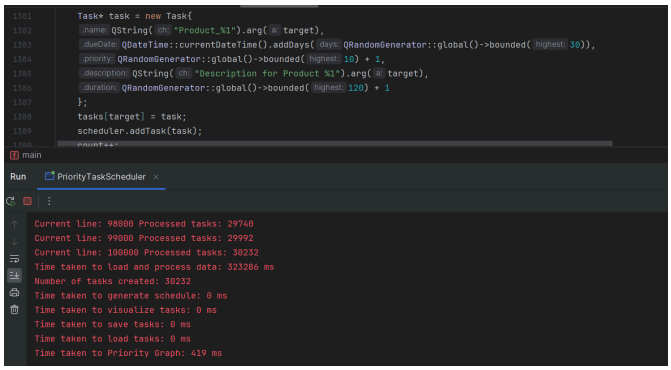
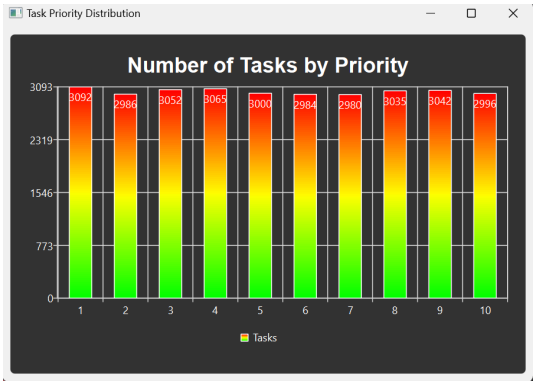
Team Name: Queue it up!
Team Members: Imaan Edhi, Nihitha Madem, Aarithi Rajendren
Github user names: imaanedhi, mademn123, aarithi123
Link to GitHub repo: https://github.com/mademn123/DSA-Project-3.git
Link to Video demo: https://youtu.be/Xlm-yD1hGi8
Problem: We are addressing challenges related to productivity and efficiency. For most people, they have to navigate through many tasks, meetings, and deadlines daily, which can often lead to important things being overlooked. Additionally, it's easy to lose track of which tasks should be prioritized, making it difficult to optimize time and ensure that all daily objectives are met.
Motivation: Students often feel overwhelmed by the thought of imminent deadlines. Creating a task scheduler will allow students to organize their thoughts, reduce stress, and enhance their time management skills. This would also help improve productivity and ensure tasks are completed in a timely manner.
Features: We will know we have solved the problem when tasks are accurately prioritized based on their due dates and the time required to complete them. Additionally, completed tasks should be removed automatically, and the priority levels of remaining tasks must be adjusted whenever a new task is added. This ensures that each task is always correctly prioritized, maintaining an efficient and up-to-date schedule.
Tools/Languages/APIs/Libraries used: C++, CLion, Qt
100,000 data points: https://snap.stanford.edu/data/amazon0302.html
The 100,000 data points can be found in the Amazon0302.txt file in our Github. The commented out code in our main.cpp is the code that uses the txt file and creates a priority graph based on those data points. The dataset is the Amazon product co-purchasing network, which provides a realistic graph/heap structure that can help evaluate the performance and functionality of our TaskScheduler implementation. Below is a screenshot of the output of the program using the 100,000 data points.





Data Structures/Algorithms implemented:

- Vector: The code uses std::vector to store tasks.
- Heaps: The code includes the <queue> header to implement the heap data structure

- Graphs. The code implements a graph structure using QGraphicsScene and QGraphicsEllipseItem to visualize tasks and their dependencies.
- Unordered map: The code uses std::unordered_map for tracking visited tasks in the depth-first search algorithm.
- Tree: The code uses QTreeWidget to display tasks in a hierarchical structure.
- Struct: A custom Task struct is defined to represent individual tasks with various attributes.

Additional Data Structures/Algorithms used:

- Topological Sort: The code uses a topological sorting algorithm (implemented via DFS) to order tasks based on their dependencies.
- Depth-First Search (DFS): The code implements a DFS algorithm to generate a schedule based on task dependencies.
- Linear Search: The code uses linear search to find tasks by name in the task list.
- Random Number Generation: The code uses QRandomGenerator to generate random positions for task items in the visualization.

Distribution of Responsibility and Roles: Who did what?

- Imaan: created a main user interface
- Nihitha: created heap data structure
- Aarithi: created graphs data structure
- Disclaimer: We all worked together on most of the code, the distribution of responsibility was to ensure that each person focused on learning all the necessary theory and implementation of the specific data structure/feature.

Any changes the group made after the proposal? The rationale behind the changes.
Mostly everything is based on our proposal, the only changes we made were the visuals. We changed the visuals to utilize more of the window effectively to implement more features.

Big O worst case time complexity analysis of the major functions/features you implemented:

- **generateSchedule()**: The time complexity of this function is O(log(v)) where v is the number of tasks. This is because the function performs dfs which has a time complexity of O(log(v)) since it is performing this traversal on a priority queue which is a tree. Therefore, the overall time complexity of this function is O(log(v)).
- **visualizeTasks()**: The time complexity of this function is O(v + e) where v is the number of tasks and e is the number of dependencies. This is because the function consists of a for loop that iterates through all the tasks in the graph and it iterates through v times and the other for loop also iterates through all the tasks in the graph but it iterates through the graph e times. The other operations are O(1) so the overall time complexity of this function is O(v + e).
- **addTask()**: The time complexity of this function is O(n) where n is the number of tasks currently in the task list. This is because the addTask() function consists of push_back()

into a vector which is O(1) and the addTaskToList() function which has a time complexity of O(n) since it calls the updateTaskColors() function which has a time complexity of O(n) due to the for loop it contains. That for loop iterates through all the elements that are currently in the task list. Therefore, the overall time complexity of this function is O(n).

- **addDependency()**: The time complexity of this function is O(nv + e) where n is the number of items in the taskList, v is the number of tasks, and e is the number of dependencies. The dominating term that contributes to this time complexity is when the updateDependenciesDisplay() is called since it has a time complexity of O(nv + e). This is because that function consists of two for loops: the first for loop iterates through the number of items in the taskList and then performs linear search on the graph which has v items in it and the second loop iterates over the number of dependencies. Therefore, the overall time complexity is O(nv + e).
- **isCircularDependency()**: The time complexity of this function is O(v + e) where v is the number of tasks and e is the number of dependencies. This is because the function performs dfs which has a time complexity of O(v + e) where v is the number of vertices and e is the number of edges. In this case, our graph consists of tasks which are vertices and dependencies which are edges. Therefore, the overall time complexity of this function is O(v + e).
- **loadtasks()**: The time complexity of this function is O(ve) where v is the number of tasks and e is the number of dependencies. This is because the nested for loop where the outer loop iterates over the number of tasks and the inner loop iterates over the number of dependencies is the dominating computation over the time complexity. Therefore, the overall time complexity is O(ve).
- **saveTasks()**: The time complexity of this function is O(v + e) where v is the number of tasks and e is the number of dependencies. This is because the function consists of a for loop that iterates through all the tasks in the graph and it iterates through v times and the other for loop also iterates through all the tasks in the graph but it iterates through the graph e times. The other operations are O(1) so the overall time complexity of this function is O(v + e).
- **removeTask()**: The time complexity of this function is O(nv + e) where n is the number of items in the taskList, v is the number of tasks, and e is the number of dependencies. The dominating term that contributes to this time complexity is when the updateDependenciesDisplay() is called since it has a time complexity of O(nv + e). This is because that function consists of two for loops: the first for loop iterates through the number of items in the taskList and then performs linear search on the graph which has v items in it and the second loop iterates over the number of dependencies. Therefore, the overall time complexity is O(nv + e).
- **removeDependency()**: The time complexity of this function is O(nv + e) where n is the number of items in the taskList, v is the number of tasks, and e is the number of

dependencies. The dominating term that contributes to this time complexity is when the updateDependenciesDisplay() is called since it has a time complexity of O(nv + e). This is because that function consists of two for loops: the first for loop iterates through the number of items in the taskList and then performs linear search on the graph which has v items in it and the second loop iterates over the number of dependencies. Therefore, the overall time complexity is O(nv + e).

- **sortTasksByDueDate()**: The time complexity of this function is O(nlog(n)) where n is the number of items in the taskList. This is because the function consists of a built-in sorting operation in Qt that has the same computational complexity as mergesort. Therefore, the time complexity of this function is O(nlog(n)).
- **sortTasksByPriority()**: The time complexity of this function is O(nlog(n)) where n is the number of items in the taskList. This is because the function consists of a built-in sorting operation in Qt that has the same computational complexity as mergesort. Therefore, the time complexity of this function is O(nlog(n)).
- **printAdjacencyList()**: The time complexity of this function is O(v + a) where v is the number of tasks and a is the number of adjacent edges. This is because there are two for loops where the outer loop is iterating through the number of tasks and the inner loop iterates through the number of dependencies but the time complexity is not O(va) since the inner loop does not iterate a times for each of the v tasks. Therefore, the overall time complexity of this function is O(v + a).
- **showPriorityGraph()**: The time complexity of this function is O(v*log(p) + p) where v is the number of tasks and p is the number of priority levels. This is because it takes O(v*log(p)) to count tasks by priority since in a QMap, inserting a value takes O(log(p)) times and this executes v times due to the for loop. It takes O(k) for the second for loop since it iterates through the priority count. Therefore, the overall time complexity is O(v*log(p) + p).
- Comparisons between two data structures: Heap vs. Graph
    - The heap includes a generateSchedule() function while the graph uses an isCircularDependency() function to use a DFS traversal to accomplish different tasks. The generateSchedule() function uses a DFS traversal to perform a topological sort to create a schedule without violating dependency constraints. This has a time complexity of O(log(v)) where v is the number of tasks since it is a heap. The isCircularDependency() function uses a DFS traversal to ensure adding a dependency would not create a cycle since dependencies require acyclic relationships. This has a time complexity of O(v + e) where v is the number of tasks and e is the number of dependencies and this is because it is a graph. These time complexities differ due to the fact that different data structures require different computational time to implement the same traversal. The generateSchedule() function is faster in terms of time complexity since log(v) is computationally faster than v + e. Therefore, the DFS traversal is faster for the

heap we implemented in our project compared to the graph we implemented in our project.

As a group, how was the overall experience for the project?

Working on the project as a group was a rewarding experience. Nihitha developed the graph structure, Aarithi implemented the heap, and Imaan focused on the user interface, ensuring smooth integration. We communicated effectively using GitHub and group chats, thorough scheduling and debugging together posed challenges.

Did you have any challenges? If so, describe.

We faced a few challenges during the project. Integrating the different components, such as the graph structure, heap, and user interface, required significant time and coordination. Debugging issues across roles was also tricky, as it often depended on multiple team members being available simultaneously. Additionally, managing our schedules to meet deadlines was sometimes difficult, especially with other commitments.

Despite these challenges, we tackled them through clear communication and collaboration, which ultimately strengthened our teamwork.

If you were to start once again as a group, any changes you would make to the project and/or workflow?

If we started over, we would integrate parts of the project earlier to catch issues faster. Setting clear deadlines and having regular meetings would help us stay on track. We'd also work on improving the user interface sooner and look for ways to make the task prioritization even better.

Comment on what each of the members learned through this process.

- Imaan: Developed skills in creating user-friendly interfaces and integrating multiple components, improving understanding of front-end and back-end coordination.
- Nihitha: Gained experience with heap structures, enhancing knowledge of efficient prioritization and resource management.
- Aarithi: Learned how to implement and visualize graph structures effectively, improving skills in data structure design and task dependency mapping.

References

- https://snap.stanford.edu/data/amazon0302.html
- https://doc.qt.io/qt-5/gettingstarted.html
- https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/
- https://doc.qt.io/qt-6/qtexamplesandtutorials.html
- https://www.boost.org/doc/libs/1_86_0/libs/graph/doc/
- https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
- https://wiki.qt.io/Qt_for_Beginners