Duel Enrollment General Education Planner - Report

My application does is useful for students at my high school looking to find classes they can take for college credit.

3 algorithms as steps or a flowchart, and a snapshot of the algorithm's code.

ALGORITHMS

1. Linear Search

- **Where:**

In generatePLAN, when checking if a course has already been completed:

- **What it does:**

Iterates through the list of course options and checks if any are in the student's completed courses.

```
for (Course c : options) {
if (student.completedCourses.contains(c.name.toUpperCase())) {
// ...
}
}
```

```
for (Course c : options) {
    if (student.completedCourses.contains(c.name.toUpperCase())) {
        System.out.println("Skipping " + area + " (already completed via " + c.name + ")");
        alreadyCompleted = true;
        break;
    }
}
if (alreadyCompleted) continue;
```

Creation Process: One key part of the generatePLAN function is checking whether a student has already completed a course. To handle this, I used a linear search. It loops through each course

option and checks if the course name exists in the student's completed courses list. This ensures we don't recommend anything the student has already finished.

At first, I considered using a recursive approach, but it was overkill for this task. A simple loop was faster to write, easier to read, and did exactly what I needed. I convert course names to uppercase during the check to keep the comparison consistent.

I used ChatGPT during the early planning phase. It helped me think through the logic and structure, but I wrote all the core algorithms myself. ChatGPT was useful for troubleshooting and brainstorming, but not for writing full code. The linear search is simple, but it plays a big role in making the course plan accurate and helpful.

2. Recursive Algorithm

- **Where:**

In recursivePlanner:

- **What it does:**

Recursively assigns planned courses to semesters, printing out the plan semester by semester.

```
static void recursivePlanner(LinkedList<Course> courses, Queue<String> semesters) {
if (courses.isEmpty() || semesters.isEmpty()) return;
// ...
recursivePlanner(courses, semesters);
}
```

Creation Process: In the recursivePlanner method, I used a recursive algorithm to assign courses to semesters. It goes semester by semester, placing available courses into the plan. Once a semester is filled or no more courses are left, the method calls itself with the updated list until everything is assigned.

Recursion felt natural here because the process is repetitive and follows a clear pattern. Each call handles one semester, then passes the rest to the next. It keeps the code clean and avoids complex loops or manual tracking of state across semesters.

I used ChatGPT to help me talk through how to manage the recursive structure, especially when dealing with edge cases like an empty course list or semester queue. I wrote the full algorithm myself, but having a tool to bounce ideas off helped me stay focused. This recursive approach is what makes the planner feel intelligent and organized.

3. Stack-based Undo Algorithm

- **Where:**

In generatePLAN, when handling the "undo" command:

- **What it does:**

Uses a stack to allow the user to undo their last course selection (LIFO behavior).

```
if (input.equalsIgnoreCase("undo") && !undoStack.isEmpty()) {
Course removed = undoStack.pop();
plannedCourses.remove(removed);
System.out.println("Last course selection undone: " + removed.name);
continue;
```

}

Creation Process: In the generatePLAN function, I implemented a stack-based undo system. When a user types "undo", the most recent course they selected is removed. This works using a stack, which follows the Last-In, First-Out (LIFO) rule. The most recent course is popped from the stack and removed from the planned list.

I chose a stack because it naturally fits the behavior of undo actions. Users expect the last thing they did to be the first thing undone. Using undoStack.pop() makes this quick and efficient without needing to search or sort anything.

To figure out how best to implement this, I skimmed through a few threads on Stack Overflow. That helped me understand how others used stacks in similar situations. While I didn't copy any code, those examples gave me confidence in using the stack this way. It's a small feature, but it adds a lot of flexibility for users planning their courses.

Summary:

- Linear search for completed courses
- Recursive planner for semester assignment
- Stack-based undo for course selection

Big O time of these algorithms:

Each of the algorithms I used was chosen with performance and simplicity in mind. The linear search used to check completed courses runs in O(n) time, where n is the number of course options. Since the list of completed courses is typically short, this was acceptable.

The recursive planner distributes courses across semesters and has a time complexity of roughly O(n) as well, assuming each course is placed once. The recursion depth depends on the number of semesters and how courses are divided up.

The stack-based undo is highly efficient. Both push() and pop() operations on the stack are O(1), making the undo functionality fast no matter how many actions a user has taken.

Data structures:

LinkedList: I used this for the list of courses because I needed to frequently add and remove courses from various positions, and LinkedList allows this efficiently.

Queue (for semesters): A queue was perfect for managing the order of semesters—first-in, first-out (FIFO) behavior helped process semesters in a logical order.

Stack (for undo): A stack was the ideal choice for undo functionality due to its LIFO nature. It allows me to easily remove the last selected course with minimal overhead.

HashMap (for storing courses): Because we didn't need to have a data structure to store in a comparative order, the HashMap was better to use compared to a TreeMap. While a TreeMap would have a logarithmic time, a HashMap utilizes a hashing function and resizes by a constant factor, leading to O(1) storage and search time.

Explains a step in the design or development process where you encountered an opportunity and how you used this.

The idea came while testing the planner manually. I found myself wishing I could undo the last step instead of redoing everything. This led me to implement a stack-based undo system, where each selected course is pushed onto a stack. If the user types "undo", the last course is popped off the stack and removed from the current plan.

Explains a step in the design or development process where you encountered an error and how you resolved this.

One bug I hit early on was with the undo feature. After popping a course from the stack, I removed it from the plannedCourses list using .remove(removed), but it wasn't always getting removed. I realized that I needed to override the equals() method in the Course class so that course objects were compared by name instead of by memory reference. Once I fixed that, the removal worked consistently.

Next Version:

In the next version of the course planner, one major improvement I would make is allowing the course list to be loaded dynamically by parsing a text file. Instead of hardcoding courses into the program, users could edit a plain text file to update course offerings, names, and prerequisites. This would make the planner much more flexible and scalable, especially for use at different schools or departments. I would write a parser that reads each line of the file, constructs Course objects, and adds them to the system automatically.

Along with that, I plan to add support for prerequisite checking using a topological sorting algorithm. Topological sort is ideal for this because it helps order courses in a way that ensures all prerequisites are taken before a dependent course. I'd build a directed graph where each

course is a node and each prerequisite is a directed edge. Then, I'd perform the sort to generate a valid order of courses, respecting dependencies.

Together, these changes would make the planner much more intelligent and adaptable. Users would have control over the course data, and the app would be able to generate more realistic, accurate plans based on prerequisite chains.