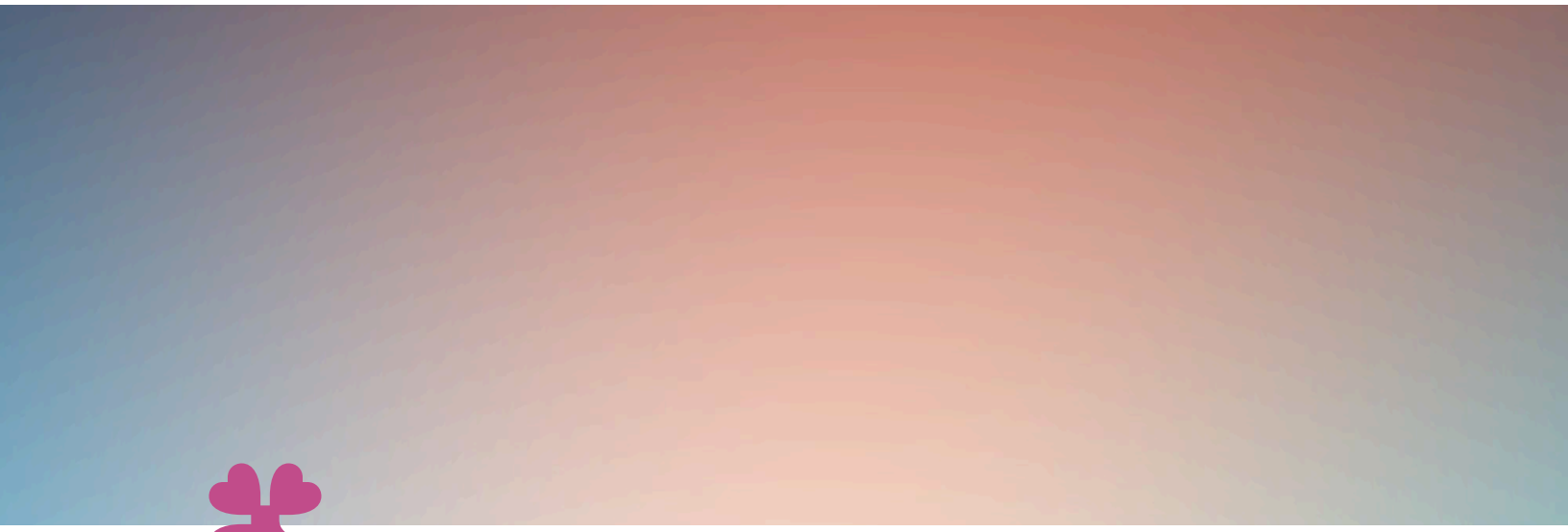

Internship Program Details

Evaluation Criteria

- **Technical Skills:** Code quality, problem-solving, and implementation.
- **Learning Ability:** How quickly and effectively you learn new topics.
- **Teamwork:** Contribution to team projects, code reviews, and communication.
- **Adaptability:** Ability to handle feedback and adapt to new challenges.

- **Commitment and Passion**: In general, how committed, passionate and interested you are for the opportunity that lies ahead.

Please note that evaluation sessions will be conducted every weekend for every student.

Curriculum Structure

The internship program is designed to run for a duration of 6 to 8 weeks, with flexibility to accommodate the academic calendar at MIT. The curriculum is tailored specifically to the backgrounds and skill levels of the interns, with distinct programs for CSE/IT students and EEE/ECE students.

Curriculum Differentiation

- **CSE/IT Students**:
  - **Programming Language**: Rust
  - **Curriculum Focus**: Given that CSE/IT students typically have a stronger foundation in software development, the curriculum is more advanced and challenging.
- **EEE/ECE Students**:
  - **Programming Language**: Python
  - **Curriculum Focus**: Recognizing that EEE/ECE students might have less experience in software development, the curriculum is structured to be slightly less demanding while still rigorous.

CSE/IT Curriculum

Week 1-3: Introduction & Core Go Programming Skills

- **Objective:**
  - **Week 1**:
    - Get familiar with the tools and environment you will be using, including setting up the development environment for Go and mastering the basics of version control with Git and GitHub.
    - Begin learning the fundamentals of Go programming, focusing on its simplicity, concurrency model, and type system.
  - **Week 2-3**:
    - Build on your Go fundamentals by applying them to a small project, reinforcing your understanding of the language's core concepts.
    - Test your problem-solving skills by tackling coding challenges that require applying Go's features to solve practical problems.

- **Topics to Learn:**

- **Week 1:**
  1. **Development Environment Setup**:
     - **Go Installation**:
       - Learn how to install Go on your machine. Understand the Go workspace, including the `GOPATH`, `GOBIN`, and `GOROOT` environment variables.
       - Set up a Go project using the Go module system, which manages dependencies. Learn how to initialize a Go module, add dependencies, and use the `go mod` command to manage versions.
     - **Version Control with Git**:
       - Learn the basics of Git: initializing a repository, staging changes, committing changes, and pushing to a remote repository on GitHub.
       - Understand branching and merging: how to create a new branch, switch between branches, merge branches, and resolve conflicts.
       - Explore collaborative workflows: learn about pull requests, code reviews, and collaboration on GitHub.

2. **Introduction to Go Programming**:
    - **Go Basics**:
        - Explore Go's type system, including primitive types (integers, floats, strings, booleans) and composite types (arrays, slices, maps, structs).
        - Learn how to define and use functions in Go, including the use of multiple return values, named return values, and variadic functions.
        - Understand Go's approach to error handling using return values rather than exceptions. Learn how to handle errors effectively and how to create custom error types.
    - **Concurrency in Go**:
        - Learn the basics of Go's concurrency model, which uses goroutines and channels. Understand how to create and manage goroutines for concurrent execution.
        - Explore how channels are used for communication between goroutines. Learn about buffered and unbuffered channels, and how to use `select` statements for multiplexing.
    - **Basic Data Structures**:
        - Learn about Go's built-in data structures: slices, maps, and structs. Understand how to create, modify, and iterate over these structures.
        - Explore the use of slices and maps for common tasks, such as dynamic arrays and key-value storage, respectively.
    - **Error Handling**:
        - Dive into Go's approach to error handling. Understand how to use the `error` type, handle errors returned from functions, and implement custom error handling logic.
        - Learn about Go's defer, panic, and recover mechanisms for handling unexpected errors and cleaning up resources.

- **Week 2-3:**
  1. **Project Implementation: To-Do List Application**:
     - **Project Structure**:
       - Set up a new Go project using the Go module system. Understand the project structure, including the `main` package and how to organize your code into multiple packages.
     - **Building the Application**:
       - **User Input Handling**: Implement basic functionality to accept user input from the command line to add, remove, and list tasks. Learn how to use the `fmt` package for input/output operations.
       - **Storing Tasks**: Use slices or maps to store tasks. Explore how Go's slices provide dynamic array functionality, and how maps can be used for quick lookups.
       - **Displaying Tasks**: Implement functionality to display tasks to the user, leveraging Go's string formatting capabilities and looping constructs.
       - **Error Handling**: Handle potential errors gracefully, such as invalid user input or empty task lists, using Go's error handling idioms.
     - **Testing the Application**:
       - Write unit tests for your To-Do list functions using Go's built-in testing framework. Learn how to run tests with the `go test` command and interpret test results.

2. **Coding Challenge**:
    - **Problem Statement**:
        - Solve a basic algorithmic problem such as finding the maximum value in a slice, checking for palindromes, or implementing a basic sorting algorithm (e.g., bubble sort).
    - **Go Implementation**:
        - Focus on implementing the solution in Go, paying attention to Go's idiomatic practices, such as the use of slices, maps, and error handling.
        - Write code that is clean and efficient, leveraging Go's features like goroutines for parallel processing if applicable.
    - **Performance and Optimization**:
        - Analyze the performance of your solution in terms of time complexity and memory usage. Learn basic profiling techniques in Go using the `pprof` package or `go test -bench`.
        - Refactor your solution if needed to improve performance, considering Go's strengths in concurrency and simplicity.
3. **Code Review and Feedback**:
    - **Review**:
        - Submit your To-Do list project and coding challenge solution for review. Participate in reviewing others' code, providing constructive feedback on code quality, performance, and adherence to Go best practices.
    - **Feedback**:
        - Receive feedback from evaluators on your project and coding challenge. Focus on areas where you can improve, such as handling edge cases, optimizing code, or adhering to Go idioms.

- **Tasks/Assignments:**
  - **Week 1:**
    1. **Development Environment Setup**:
       - Install Go on your machine. Verify the installation by compiling and running a basic Go program.
       - Set up a Git repository for your projects. Create a GitHub account if you don't have one and push your local repository to GitHub.
       - Practice branching and merging by creating a branch, making changes, and merging it back to the main branch. Handle a simulated merge conflict and resolve it.
    2. **Basic Go Programming**:
       - Write a simple Go program that prints "Hello, World!" to the console.
       - Extend the program to accept user input (e.g., a name) and print a personalized greeting.
       - Experiment with Go's data structures by creating functions that manipulate slices and maps. Practice error handling by writing functions that return errors for invalid input.

- **Week 2-3:**
  1. **To-Do List Application**:
     - Start by designing the To-Do list application, planning how you will structure the code and what data structures you will use.
     - Implement the basic functionality to add tasks to the list. Test your implementation by running the application and verifying that tasks are stored correctly.
     - Expand the application to support removing tasks and displaying all tasks. Focus on handling edge cases like trying to remove a task that doesn't exist.
     - Implement error handling throughout the application, ensuring that the program behaves predictably in response to user input errors.
  2. **Coding Challenge**:
     - Choose a simple algorithmic problem to solve using Go. Write a function to solve the problem and test it with various inputs.
     - After solving the problem, analyze your solution's time complexity. Consider if there are ways to optimize the solution without sacrificing readability.
     - Write unit tests for your solution to ensure it handles all possible cases, including edge cases and potential errors.
  3. **Code Review**:
     - Submit your To-Do list project and coding challenge solution for review by the evaluators. Review at least one other person's code, focusing on providing constructive feedback.
     - Reflect on the feedback you receive and identify areas for improvement in your next assignments.

Week 4-6: Algorithms & Data Structures

- **Objective:**
  - **Week 4**:
    - Develop a deep understanding of key algorithms and their applications, focusing on how they solve common computational problems.
    - Learn and implement advanced data structures, understanding their inner workings and when to use them.
  - **Week 5-6**:
    - Apply the learned algorithms and data structures to solve complex problems, emphasizing performance analysis and optimization.
    - Work collaboratively to solve challenging problems, fostering teamwork and communication skills.

- **Topics to Learn:**
  - **Week 4:**
    1. **Sorting Algorithms**:
       - **Overview of Sorting Algorithms**:
         - Learn about different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, QuickSort, and HeapSort.
         - Understand the time and space complexity of each algorithm, focusing on best, worst, and average-case scenarios.
         - Discuss the stability of sorting algorithms and when a stable sort is necessary.
       - **Implementation in Go**:
         - Implement basic sorting algorithms (Bubble Sort, Insertion Sort) to understand their simplicity and limitations.
         - Move on to more efficient algorithms like Merge Sort and QuickSort, focusing on recursive implementation, divide-and-conquer strategies, and the role of pivot selection in QuickSort.
         - Explore HeapSort by building a max-heap and understanding the heap property.
    2. **Searching Algorithms**:
       - **Linear and Binary Search**:
         - Compare linear search with binary search, understanding the trade-offs between them.
         - Implement binary search in Go, including handling edge cases such as searching in a list with duplicates or searching in an unsorted list.
         - Analyze the time complexity of binary search and understand its logarithmic nature.
       - **Advanced Searching Techniques**:
         - Introduce search trees (Binary Search Trees - BSTs) and how they allow for more efficient searching, insertion, and deletion.
         - Discuss balanced trees like AVL and Red-Black trees, explaining the importance of maintaining balance in BSTs.

- **Week 5**:
  1. **Recursion**:
     - **Understanding Recursion**:
       - Study the concept of recursion, where a function calls itself to solve smaller instances of a problem.
       - Learn about the base case and recursive case, which are essential for preventing infinite recursion and ensuring that the problem converges to a solution.
       - Explore the use of recursion in algorithms like Merge Sort, QuickSort, and solving problems such as calculating factorials, Fibonacci numbers, and solving the Tower of Hanoi.
     - **Recursion vs. Iteration**:
       - Compare recursion with iteration, understanding when one approach might be preferred over the other.
       - Implement recursive solutions and then convert them to iterative ones to observe the differences in performance and readability.

2. **Dynamic Programming (DP)**:
    - **Introduction to Dynamic Programming**:
        - Understand the concept of dynamic programming as an optimization technique for problems with overlapping subproblems and optimal substructure.
        - Learn about memoization (top-down approach) and tabulation (bottom-up approach), the two main strategies in DP.
        - Study classic DP problems such as the Fibonacci sequence, 0/1 Knapsack problem, and the Longest Common Subsequence (LCS).
    - **Implementing DP in Go**:
        - Implement the Fibonacci sequence using both memoization and tabulation.
        - Solve the 0/1 Knapsack problem using DP and analyze the space-time trade-offs.
        - Work on the LCS problem, implementing it in Go and understanding the recursive and DP-based solutions.

- **Week 6:**

1. **Advanced Data Structures**:
    - **Trees**:
        - **Binary Trees and Binary Search Trees (BSTs)**:
            - Understand the structure and properties of binary trees and BSTs. Learn how they differ from general trees.
            - Implement a binary tree and BST in Go, including insertion, deletion, and traversal operations (in-order, pre-order, post-order).
            - Explore the concept of self-balancing trees (e.g., AVL trees), understanding how rotations maintain tree balance.
        - **Tree Traversal Algorithms**:
            - Learn about different tree traversal techniques and their applications.
            - Implement in-order, pre-order, and post-order traversal algorithms in Go and discuss their use cases (e.g., in expression trees).
    - **Graphs**:
        - **Graph Basics**:
            - Understand the fundamental concepts of graphs, including nodes (vertices), edges, and graph representations (adjacency list and adjacency matrix).
            - Learn about different types of graphs: directed, undirected, weighted, unweighted, cyclic, and acyclic.
        - **Graph Traversal Algorithms**:
            - Study Breadth-First Search (BFS) and Depth-First Search (DFS), understanding how they explore nodes in a graph.
            - Implement BFS and DFS in Go, applying them to problems like finding connected components, detecting cycles, and solving mazes.
            - Explore shortest path algorithms like Dijkstra's algorithm for weighted graphs and understand its application in real-world scenarios.

- **Heaps**:
  - **Introduction to Heaps**:
    - Learn about binary heaps (min-heap and max-heap) and their properties.
    - Understand the heap operations: insertion, deletion, and heapify.
    - Discuss the role of heaps in implementing priority queues.
  - **Heap Implementation**:
    - Implement a binary heap in Go from scratch, focusing on how the heap property is maintained during insertion and deletion.
    - Apply heaps in solving problems like finding the k-th largest element in an array and heap sort.
- **Hash Tables**:
  - **Basics of Hashing**:
    - Understand the concept of hashing and how hash tables (or hash maps) allow for efficient data retrieval.
    - Learn about hash functions, collision resolution techniques (chaining, open addressing), and load factors.
  - **Hash Table Implementation**:
    - Implement a simple hash table in Go, including handling collisions using chaining.
    - Solve problems that involve fast lookups, such as checking for duplicates, frequency counting, and implementing a simple dictionary.

- **Tasks/Assignments:**
  - **Week 4:**
    1. **Sorting Algorithms**:
       - Implement the basic sorting algorithms (Bubble Sort, Insertion Sort) in Go. Analyze their performance on small and large datasets, noting the differences in time complexity.
       - Implement Merge Sort and QuickSort, paying attention to the recursive nature of these algorithms. Test these implementations on various datasets, including edge cases like already sorted and reverse sorted arrays.
       - Implement HeapSort by first building a max-heap. Analyze the space complexity and compare it with other sorting algorithms.
    2. **Searching Algorithms**:
       - Implement linear search and binary search in Go. Compare their performance and explain why binary search is more efficient for sorted datasets.
       - Implement a basic Binary Search Tree (BST) in Go. Perform insertion, deletion, and search operations, and analyze their time complexity.
       - Explore balancing techniques by implementing an AVL tree and observing how rotations maintain the tree's balance after insertions and deletions.

- **Week 5:**
  1. **Recursion**:
     - Implement the recursive solution to common problems like the factorial of a number, generating Fibonacci numbers, and solving the Tower of Hanoi.
     - Convert these recursive solutions to iterative ones and compare the pros and cons of each approach.
     - Work on a problem like solving mazes or generating permutations of a string using recursion.
  2. **Dynamic Programming (DP)**:
     - Implement the Fibonacci sequence using both memoization and tabulation. Analyze the time and space complexity of both approaches.
     - Solve the 0/1 Knapsack problem using dynamic programming, focusing on understanding the recursive relation and how it's translated into code.
     - Implement the Longest Common Subsequence (LCS) problem in Go using DP. Discuss the importance of DP in optimizing recursive solutions.

- ### Week 6:

1. **Advanced Data Structures**:
    - **Trees**:
        - Implement a binary tree and a BST in Go. Practice insertion, deletion, and traversal operations. Analyze the time complexity for each operation.
        - Implement a self-balancing tree (e.g., AVL tree) in Go. Focus on understanding how rotations work and why they are necessary to maintain balance.
        - Solve problems like finding the height of a tree, checking if a tree is balanced, and finding the lowest common ancestor (LCA) in a binary tree.
    - **Graphs**:
        - Implement graph representations (adjacency list and adjacency matrix) in Go. Understand the trade-offs between the two representations.
        - Implement BFS and DFS algorithms in Go. Apply these algorithms to solve problems like finding all connected components in an undirected graph and detecting cycles.
        - Work on a problem like finding the shortest path in a weighted graph using Dijkstra's algorithm.
    - **Heaps**:
        - Implement a binary heap (min-heap or max-heap) in Go. Practice insertion and deletion operations and understand how the heap property is maintained.
        - Solve problems like finding the k-th largest element in an array using a heap. Implement HeapSort and compare its performance with other sorting algorithms.

- **Hash Tables**:
  - Implement a hash table in Go, focusing on handling collisions using chaining.
  - Solve problems like checking for duplicates in an array, counting the frequency of elements, and implementing a simple dictionary.
  - Discuss the trade-offs of different collision resolution techniques and analyze the performance of your hash table implementation.

2. **Group Challenge**:
   - **Algorithmic Problem Solving**:
     - Work as a group to solve a series of algorithmic problems that require the application of sorting, searching, and dynamic programming techniques. Each group member can be responsible for a different aspect of the problem, such as implementing the algorithm, optimizing it, or testing.

Week 7-8: Coding Challenges

- **Objective:** Assess problem-solving skills, speed, and ability to apply concepts under pressure.
- **Topics to Focus On:**
  - Advanced problem-solving techniques through coding challenges.
  - Focused practice on popular LeetCode problems (easy to medium difficulty).
- **Tasks/Assignments:**
  - Daily coding challenges using platforms like LeetCode, HackerRank, or Codewars.
  - Solve a set of LeetCode problems that cover a range of topics such as arrays, strings, dynamic programming, and tree traversal.
  - Group discussion on problem-solving strategies and optimization techniques.

EEE/ECE Curriculum

Week 1-3: Introduction & Core Programming Skills

- **Objective:**
  - **Week 1**:
    - Get familiar with the tools and environment you will be using throughout the internship, including setting up Python, mastering version control with Git, and collaborating on GitHub.
    - Learn the fundamentals of Python programming, focusing on understanding data types, control structures, functions, and modules, which form the building blocks of Python development.
  - **Week 2-3**:
    - Apply the foundational Python skills to build a small, practical project that consolidates your understanding of core programming concepts.
    - Test and refine your problem-solving skills by completing a coding challenge, emphasizing the application of Python to algorithmic problems.

- **Topics to Learn:**

- **Week 1:**
  1. **Development Environment Setup**:
     - **Python Installation and Setup**:
       - Install the latest version of Python on your machine, ensuring compatibility with your operating system. Familiarize yourself with different ways to run Python code, including the Python interpreter, scripts, and integrated development environments (IDEs) like PyCharm, VS Code, or Jupyter Notebook.
       - Set up a virtual environment using `venv` or `virtualenv`. Understand the importance of virtual environments in managing dependencies and keeping your projects isolated.
       - Install essential Python packages using `pip`, the Python package installer. Learn how to manage packages in a virtual environment and explore the basics of `requirements.txt` for sharing dependencies.
     - **Version Control with Git**:
       - Learn the basics of Git, including initializing a repository, staging changes, committing, and pushing changes to a remote repository on GitHub.
       - Understand branching in Git, and how to create, switch between, and merge branches. Practice resolving merge conflicts to understand common issues that arise in collaborative development.
       - Familiarize yourself with GitHub: creating repositories, cloning them to your local machine, and using pull requests for code reviews and collaborative work.

2. **Introduction to Python Programming**:
    - **Python Basics**:
        - Explore Python's data types: integers, floats, strings, and booleans. Learn about type conversion and how Python handles different data types in operations.
        - Understand control structures in Python: conditionals (`if` , `elif` , `else`) and loops (`for` , `while`). Learn how to use control structures to make decisions and repeat actions.
        - Functions: Learn how to define functions in Python using the `def` keyword, including parameters, return values, and the scope of variables within functions. Understand the difference between positional and keyword arguments.
        - Modules: Learn how to organize your code into modules. Understand how to import and use built-in Python modules (e.g., `math` , `os` , `datetime`), as well as how to create and import your own modules.

3. **Basic Data Structures**:
    - **Lists**:
        - Learn how to create, modify, and access elements in a list. Understand list methods such as `append()`, `remove()`, `sort()`, and `reverse()`.
        - Explore list slicing and how to manipulate lists using slicing techniques.
        - Practice iterating over lists using loops, and learn how to use list comprehensions for concise and efficient list operations.
    - **Dictionaries**:
        - Understand the concept of key-value pairs in dictionaries. Learn how to create, modify, and access elements in a dictionary.
        - Explore dictionary methods such as `get()`, `keys()`, `values()`, and `items()`.
        - Learn how to iterate over dictionaries and handle cases where keys might not be present.
    - **Sets**:
        - Understand sets and their properties, such as uniqueness and unordered collection of elements.
        - Learn how to create and manipulate sets using methods like `add()`, `remove()`, `union()`, and `intersection()`.
        - Explore the use of sets for operations like removing duplicates from a list or finding common elements between collections.
    - **Using `try-except` Blocks**:
        - Understand the importance of error handling in writing robust

- **Week 2-3:**
    1. **Project Implementation: To-Do List Application**:
        - **Project Structure**:
            - Set up a new Python project using your IDE. Organize the project files into modules if necessary. Start with a basic structure that includes a main script and additional modules for different functionalities.
        - **Sets**:
            - Define the basic requirements for the To-Do list application: the ability to add tasks, remove tasks, and view the list of tasks.
        - **Building the Application**:
            - **User Input Handling**: Implement functionality to accept user input from the command line to add, remove, and list tasks. Use `input()` for capturing user input and explore basic input validation techniques.
        - **Storing Tasks**: Choose an appropriate data structure, such as a list or dictionary, to store tasks. Consider the pros and cons of each structure and how it might impact the application's functionality.
    4. **Error Handling**: