ADDIS ABABA UNIVERSITY

COLLEGE OF TECHNOLOGY AND BUILT ENVIRONMENT

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

FUNDAMENTALS OF DATA STRUCTURE AND ALGORITHM ANALYSIS

PROJECT TITLE: MINIGIT: A CUSTOM VERSION CONTROL SYSTEM

COURSE CODE: SECT - 3091

SECTION 1

Name	ID No
1. Lemi Gobena	UGR/1589/16
2. Fita Alemayehu	UGR/7071/16
3. Olit Oljira	UGR/8925/16
4. Bekalu Addisu	UGR/9538/16
5. Misganaw Habtamu	UGR/1707/16

Submission date: **20/06/2025**
Submitted to: **Mr. Beakal**

MINIGIT: A CUSTOM VERSION CONTROL SYSTEM

1. Detailed Project Overview

The MiniGit project is an ambitious educational endeavor designed to immerse students in the foundational principles of version control systems by having them implement a lightweight, Git-like system from scratch. Far beyond just understanding how to use a version control tool, this project aims to demonstrate the complex internal workings of systems like Git, allowing students to grasp the underlying data structures and algorithms that power them.

Core Purpose and Simulation:

The central aim is to build a command-line interface (CLI) based system that simulates a local-only Git experience. This means all operations – from tracking changes and creating snapshots of your project to managing different lines of development (branches) and integrating changes (merges) – will occur directly on the user's computer, without relying on external libraries or network functionalities.

Educational Objectives and Skill Development:

Through this hands-on implementation, students are expected to achieve several key learning objectives:

- **Deepen Git Understanding:** Gain a comprehensive insight into "how Git works under the hood," rather than just memorizing commands.
- **DSA Application:** Learn to design and implement complex data structures, specifically applying concepts like Hashing for content addressing, Directed Acyclic Graphs (DAGs) for commit history, and Trees for representing file structures within commits.
- **File Management Proficiency**: Improve practical skills in handling file input/output (I/O) operations, which are crucial for persisting project state and retrieving file contents.
- **CLI Development:** Enhance capabilities in developing robust and user-friendly command-line interfaces.
- **Software Engineering Practices:** Practice essential software engineering concepts such as modularity, scalability, versioning, diffing (identifying changes), and merging (combining changes).

2. Data Structures Used

The MiniGit system leverages several core data structures to manage its version control functionalities:

Blobs:

- ✓ Description: Stores the exact content of a file at a specific point in time. Each unique file content is stored once and referenced by its hash.
- ✓ DSA Concepts: Hashing (for content addressable storage) and File I/O (for persistent storage on disk).

Commit Nodes:

- ✓ Description: Represents a snapshot of the project at a particular moment. A commit object includes metadata (timestamp, commit message, parent commit(s)), and a mapping of file paths to their corresponding blob hashes (representing the project's state).
- ✓ DSA Concepts: Effectively forms a Directed Acyclic Graph (DAG) where commits are nodes and parent pointers are edges. This structure also resembles a Linked List when traversing linearly (e.g., for *log* command). Internally, a hash map (*std::unordered_map*) is used to store file paths to blob hashes for the snapshot.

Branches:

- ✓ **Description:** Named references that point to a specific commit. They allow developers to maintain different lines of development.
- ✓ **DSA Concepts:** Essentially a Hash Map (*std::unordered_map* conceptually, or *std::map* as used in *findLCA*) where branch names are keys and commit hashes are values. These are persistently stored as files in *.minigit/refs/heads/.*

Staging Area (Index):

- ✓ **Description:** An intermediate area that holds a list of files and their versions (blob hashes) that are prepared to be included in the next commit.
- ✓ **DSA Concepts:** Implemented as a Hash Table *(std::unordered_map<std::string, std::string>*) mapping file paths to their blob hashes. Persisted to disk in *.minigit/index.*

Log History:

- ✓ **Description:** The sequential record of commits, typically traversed backward from the current HEAD.
- ✓ **DSA Concepts:** The traversal itself follows the Linked List structure inherent in the parent pointers of commit nodes.

3. Design Decisions

- **File-System Based Persistence:** All MiniGit data (objects, refs, index) is stored directly in the local file system within the hidden *.minigit/* directory. This design choice directly mirrors Git's object model, providing a concrete understanding of how version control data is physically stored without relying on external databases.
- **Content-Addressable Storage (Hashing):** Files (blobs) and commit objects are identified and stored based on the hash of their content. This ensures data integrity, de-duplication of identical content, and forms the basis for efficient change tracking. SHA-1 or a custom hash function is suggested for this purpose.
- **Immutable Objects:** Once created, blobs and commit objects are immutable. Any change results in a new hash and a new object being stored. This foundational principle simplifies data management and ensures historical integrity.
- **HEAD Pointer:** The HEAD file dynamically points to either a specific branch reference or directly to a commit hash (detached HEAD). This design allows flexible switching between branches and specific historical states.
- **Three-Way Merge Strategy:** The *merge* command implements a three-way merge, utilizing a Lowest Common Ancestor (LCA) to identify the common base from which two diverging branches originated. This enables intelligent merging of changes by comparing both branches against their common history, providing the ability to detect and mark conflicts automatically.
- **Line-by-Line Diffing:** The diff command provides a basic line-by-line comparison, indicating additions (+), deletions (-), and unchanged lines (). While not as sophisticated as Git's diff (which uses a Longest Common Subsequence algorithm), it effectively demonstrates the concept of identifying content differences.
- **Command-Line Interface (CLI):** The system operates entirely via command-line commands, providing a practical experience in building interactive console applications.

4. Limitations and Future Improvements

Current Limitations:

- ✓ **Local-Only Operation:** MiniGit currently only operates on a single local repository. It lacks networking capabilities to interact with remote repositories (e.g., *git clone, git push, git pull, git fetch*).
- ✓ **Simplified Merge Conflict Resolution:** While the merge command detects and marks conflicts, it does not offer automated conflict resolution strategies. Users must manually edit conflict markers and then re-add/commit.
- ✓ **Basic Diffing Algorithm:** The *diff* implementation is a simplified line-by-line comparison. It may not optimally detect moved lines, reordered blocks, or complex textual changes that a more advanced diff algorithm (like the Myers algorithm for LCS) would handle.
- ✓ **No Stashing:** There is no mechanism to temporarily save uncommitted changes (like *git stash*).

- ✓ **No Remote Tracking Branches:** The concept of remote tracking branches (e.g., *origin/master*) is not present.
- ✓ **Limited Error Handling and Edge Cases:** While basic error handling is present, a production-grade Git would have more robust handling for file permissions, corrupted objects, or concurrent access.
- ✓ **No Interactive Rebasing:** Complex history rewriting operations like rebase are not implemented.
- ✓ **No rm or mv Tracking:** Files moved or removed from the working directory are not explicitly tracked by a *minigit rm* or *minigit mv* command, relying solely on add to update the snapshot.
- ✓ **No Tagging:** The ability to mark specific commits with human-readable tags (e.g., for releases) is not implemented.

Future Improvements:

1. **Remote Repository Support:** Implement *clone, push, pull*, and *fetch* commands to enable collaboration and remote storage. This would involve adding network communication capabilities.
2. **Advanced Merge Strategies:** Implement more sophisticated merge algorithms, potentially with heuristics for auto-resolving simple conflicts (e.g., "theirs" or "ours" for non-conflicting lines).
3. **Improved Diff Algorithm:** Upgrade the *printDiff* function to use a more advanced algorithm (e.g., Longest Common Subsequence) for more accurate and readable diffs.
4. **Stashing Mechanism:** Add a *stash* command to temporarily save changes that are not ready to be committed.
5. **Garbage Collection:** Implement a mechanism to periodically clean up unreferenced objects to save disk space.
6. **Interactive Commands:** Introduce interactive versions of commands like *rebase* or a*dd -p* for more granular control.
7. **Graphical User Interface (GUI):** Develop a simple GUI for MiniGit for easier visual interaction.
8. **Performance Optimization:** For very large repositories, optimize file I/O and hashing operations.
9. **Reflog Implementation:** Track changes to HEAD and branch references for recovery purposes.

5. Challenges Faced During Development

Building MiniGit from the ground up, while incredibly rewarding, came with its own set of puzzles and tricky moments for our group. Here are some of the main hurdles we encountered and how we approached them:

- ✓ **Making MiniGit Remember Things (The "Forgetful" Program):**

One of the biggest early challenges was making our MiniGit "remember" anything between commands. When you run a program from the command line (like *main.exe add* or *main.exe commit*), it starts, does its job, and then completely shuts down, forgetting everything it just did.

This meant our "staging area" (where you mark files for your next commit) would be empty every time we tried to commit! We had to teach MiniGit to "write down" its memory – like a to-do list for the next command – by saving this staging information into a special file on your computer.

- ✓ **Playing Hide-and-Seek with Files (File System Quirks):**

Even after teaching MiniGit to save its memory, we ran into frustrating moments where our code would confidently say "I saved that file!" but then, a second later, another part of our MiniGit (or even us looking in the computer's folders) would say "That file doesn't exist!" This was often like playing hide-and-seek. We discovered that sometimes, other programs on the computer (like antivirus software or cloud syncing services like OneDrive) could secretly interfere, either making files disappear or preventing our MiniGit from seeing them properly. It required careful testing and sometimes moving our project to a 'quieter' spot on the computer.

- ✓ **Making Branches Come Together (The Merge Puzzle):**

Merging different versions of a project (like combining a 'new feature' branch back into the 'main' version) was a major challenge. It's not just about copying files; it's about figuring out what changed where, especially when two people changed the same part of a file in different ways. Our MiniGit had to be smart enough to detect these "conflicts" and put special markers in the file to show where the disagreement was, so a human could fix it. Building that "conflict detection" logic required a lot of careful thought and testing.

- ✓ **Understanding Confusing Error Messages (Debugging Headaches):**

When our code didn't work as expected, the computer's error messages could often be very cryptic – like reading a foreign language! Because our program runs so quickly from the command line, it was hard to see exactly where something went wrong. We had to learn to strategically place "debug messages" (little notes that print to the screen) throughout our code. This was like asking our MiniGit to "talk to us" and tell us what it was doing step-by-step, helping us narrow down where the bugs were hiding.

- ✓ **Working Together on a Shared Codebase :**

Even though our MiniGit itself is "local-only," as a group, we were developing it by sharing our code on a common platform like GitHub. This brought its own set of challenges. Sometimes, when one person finished their part and tried to share it with the others, it wouldn't fit perfectly with the latest changes made by someone else. This meant our shared code could become messy or even stop working until we carefully sorted out whose changes went where, which often involved fixing "merge conflicts" in our own MiniGit project's code! This taught us a lot about coordinating our work and making sure everyone was building on the most up-to-date version.