4. Complexity Analysis

This section analyzes the time and space complexity of the MiniGit system. The goal is not only to quantify performance but also to justify why the chosen design is suitable for an educational, lightweight version-control system. Wherever relevant, comparisons are made with real Git to help the reader understand trade-offs.

4.1 Understanding the Scope of Complexity

MiniGit is intentionally designed as a **single-user, local, file-based system**. Unlike real Git, it does not need to: - Handle millions of objects - Support distributed networking - Optimize for large repositories

Therefore, complexity analysis focuses on: - File size (number of tracked files) - Number of commits in a branch - Number of branches

Let: - **F** = number of files staged or tracked - **C** = number of commits in a branch - **B** = number of branches

---

4.2 Time Complexity of Commands

4.2.1 `init`

Operations performed: - Create directories (`.minigit`, `objects`, `commits`) - Create small metadata files (`HEAD`, `branches.txt`, `index.txt`)

Time Complexity: - O(1)

Explanation: The number of operations does not depend on repository size. Initialization is constant-time.

Comparison with Git: Real Git also performs initialization in constant time, though it creates more internal structures.

4.2.2 `add <file>`

Operations performed: - Read file content - Hash file content - Write object file - Append entry to index

Time Complexity: - O(S), where S is the size of the file

Explanation: - Reading and hashing a file is linear in file size - Writing the object is also linear

If multiple files are added: - $O(S_1 + S_2 + \ldots + S_n)$

Comparison with Git: Git also hashes file contents (SHA-1 / SHA-256). The conceptual complexity is identical.

4.2.3 `commit -m "message"`

Operations performed: - Read all staged files from index - Construct commit metadata - Hash combined data - Write commit file - Update branch head - Clear index

Time Complexity: - O(F)

Explanation: Each staged file entry is read once. Metadata handling is constant-time.

Comparison with Git: Git internally creates tree objects, blobs, and commit objects, making it more complex but still linear in number of staged files.

4.2.4 `log`

Operations performed: - Read branch head - Traverse commits using parent pointers - Read each commit file

Time Complexity: - O(C)

Explanation: Each commit is visited exactly once, similar to traversing a linked list.

Why this is acceptable: - Branch histories are typically linear - MiniGit avoids graph traversal complexity

Comparison with Git: Git performs graph traversal (DAG), which is more complex but optimized using commit graphs and caching.

4.2.5 `branch <name>`

Operations performed: - Check if branch exists - Append new branch entry

Time Complexity: - O(B)

Explanation: The branches file is scanned linearly to detect duplicates.

Comparison with Git: Git uses references stored as files or packed refs, enabling near O(1) access.

4.2.6 `checkout <branch>`

Operations performed: - Verify branch existence - Update HEAD file

Time Complexity: - O(B)

Explanation: Branch lookup is linear due to file-based storage.

Note: MiniGit does not restore working directory state (unlike real Git), keeping complexity minimal.

4.2.7 `merge <branch>`

Operations performed: - Read commit data from two branches - Store file mappings in unordered_map - Detect conflicts - Write merged index - Create merge commit

Time Complexity: - $O(F_1 + F_2)$

Explanation: Each file entry from both branches is processed once.

Why unordered_map matters: - Average O(1) lookup for conflict detection

Comparison with Git: Git performs a three-way merge using a common ancestor, which is significantly more complex.

4.3 Space Complexity

4.3.1 Object Storage

- Each unique file content is stored once
- Duplicate contents reuse the same hash

Space Complexity: - O(U), where U = number of unique file contents

This mimics Git's deduplication strategy.

4.3.2 Commit Storage

- Each commit stores metadata and file mappings

Space Complexity: - $O(C \times F)$

Trade-off: - Simple storage format - Easy to debug and inspect

---

4.3.3 In-Memory Structures

- `unordered_map` during merge
- Temporary strings during hashing

Space Complexity: - O(F)

Memory usage is minimal and released after command execution.

4.4 Why These Complexities Are Acceptable

MiniGit prioritizes: - Clarity over extreme optimization - Educational value - Deterministic behavior

For repositories with: - Tens or hundreds of files - Linear commit history

Performance is more than sufficient.

4.5 Comparison Summary with Real Git

Feature	MiniGit	Real Git
Commit traversal	Linked list	DAG
Merge	Two-way	Three-way
Branch lookup	Linear	Near O(1)
Storage	Plain files	Packed & compressed

4.6 Future Optimizations

If MiniGit were extended: - Replace branch file with hash map - Cache commit metadata - Introduce tree objects - Implement delta compression

These would reduce time and space complexity but at the cost of simplicity.

4.7 Conclusion

The complexity of MiniGit is intentionally simple, predictable, and aligned with its learning goals. While it does not match Git's industrial-scale optimizations, it faithfully demonstrates the **core algorithmic principles of version control systems**.