Introduction to Git

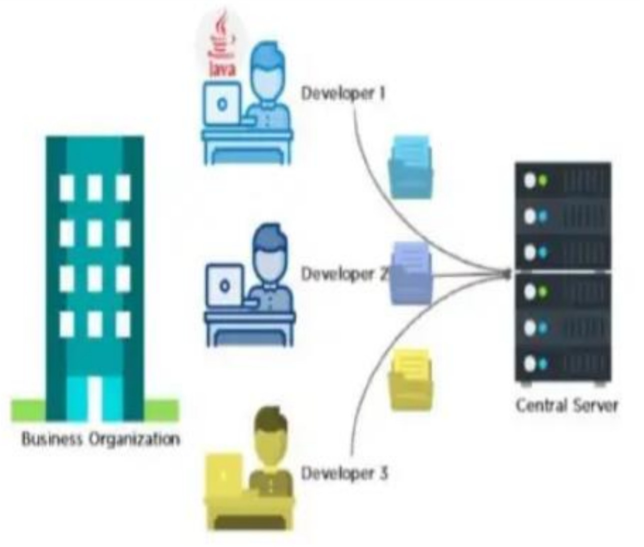
Aim:
To learn the basic concepts and commands of Git for version control and project management.

What is Git?

- Git is a version control system which lets you track changes you make to your files over time.
- With Git, you can revert to various states of your files.
- You can also make a copy of your file, make changes to that copy, and then merge these changes to the original copy.

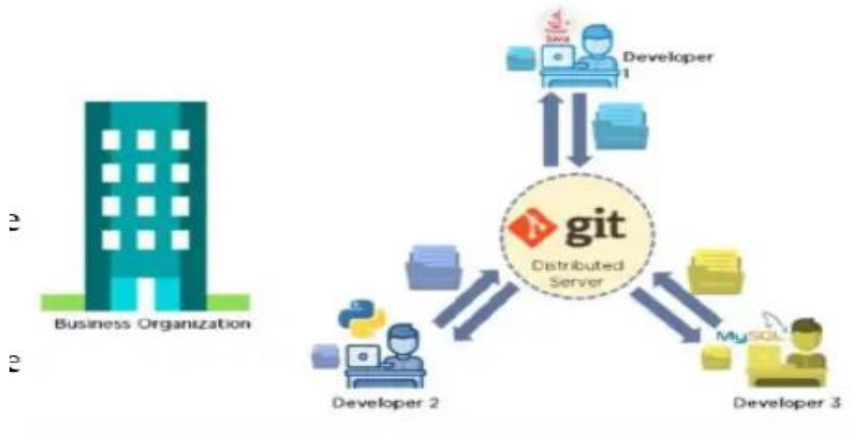
Introduction to GIT
Git is a distributed version control system (VCS) that is widely used for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control in the software development industry.

Before going through Git



- Developers used to submit their codes to the central server without having copies of their central server without having copies of their own.
- Any changes made to the source code were unknown to the other developers
- There was no communication between any of the developers.

After pass through the Git



- Every developer has an entire copy of the code on their local systems
- Any changes made to the source code can be tracked by others.
- There is regular communication between the developers

Introduction to Git

- **Repository (Repo):** A Git repository is a directory or storage location where your project's files and version history are stored. There can be a local repository on your computer and remote repositories on servers.
- **Commits:** In Git, a commit is a snapshot of your project at a particular point in time. Each commit includes a unique identifier, a message describing the changes, and a reference to the previous commit.
- **Branches:** Branches in Git allow you to work on different features or parts of your project simultaneously without affecting the main development line (usually called the "master" branch). Branches make it easy to experiment, develop new features, and merge changes back into the main branch when they are ready.
- **Pull Requests (PRs):** In Git-based collaboration workflows, such as GitHub or GitLab, pull requests are a way for developers to propose changes and have them reviewed by their peers. This is a common practice for open-source and team-based projects.
- **Merging:** Merging involves combining changes from one branch (or multiple branches) into another. When a branch's changes ae ready to be incorporated into the main branch, you can merge them.
- **Remote Repositories:** Remote repositories are copies of your project stored on a different server. Developers can collaborate by pushing their changes to a remote repository and pulling changes from it. Common remote repository hosting services include GitHub, GitLab, and Bitbucket.
- **Cloning:** Cloning is the process of creating a copy of a remote repository of your local machine. This allows you to work on the project and make changes locally.
- **Forking:** Forking is a way to create your copy of a repository, typically on a hosting platform like GitHub. You can make changes to your fork without affecting the original

project and later create pull requests to contribute your changes back to the original repository.

Features to GIT

- Version Control
- Collaboration
- Branching
- Distributed Development
- Backup and Recovery
- Code Review
- Open Source and Community Development

How to Configure Git

- To view the version of Git which is installed in the system: **$git –version**
- To initialize Git Repository: **$git init**
- To set username, type and execute these commands: **$git config –global user.name "gitlab"**
- The user name get overwrites as your emailed as user name: **$git config –global user.mail** **abcd_cs@tjohngroup.com**
- Open gitconfig using notepad once and you can able to view in the master username. You can check in C:\Program Files\Git\etc
- This command is used to **view all the Git configuration settings** currently in effect on your system, including those set at the system, global, and local levels: **$git config –list**
  - ✓ Displays a list of key-value pairs showing all configuration settings Git is using.
  - ✓ Includes settings like user name, email, default editor, aliases, merge tools, etc.
- To list the total files in current directory : **$ls**
- To list the hidden files in current directory: **$ls -a**
- **mkdir:** To create directory
- **cd:** To get into directory
- **status:** Git can notify if your folder has some modified files

GitHub

GitHub is a cloud-based platform that hosts Git repositories and enables developers to store, share, and collaborate on code projects. It provides tools for version control, branching, merging, and teamwork, allowing multiple users to work on the same project simultaneously without overwriting each other's work. GitHub acts as a central repository where developers can push their local code, track changes, review updates through pull requests, and manage issues efficiently, making it one of the most popular platforms for collaborative software development.

Result:

Successfully understood how to create a repository, add files, commit changes, and manage versions using Git.

Experiment 1: Setting Up and Basic Commands
Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Aim:
To initialize a new Git repository, create a file, add it to the staging area, and commit the changes using basic Git commands.

Algorithm / Procedure:

Step1: Open Git Bash
open Git Bash and navigate to your desired directory using cd command.
Step 2: Initialize a New Git Repository
Command: git init
This creates a hidden .git folder and initializes the directory as a Git repository.
Step 3: Create a New File
You can use touch to create a file.
Command: touch hello.txt
This creates an empty file named hello.txt.
Step 4: Add Some Content to the File
Use a text editor or echo command
Command: echo "This is my first Git file." > hello.txt
Step 5: Check the Status
Command: git status
This shows the untracked file hello.txt.
Step 6: Add the File to the Staging Area
Command: git add hello.txt
Step 7: Commit the Changes
Command: git commit -m "Initial commit: Added hello.txt"

Summary of Commands:
```
git init
touch hello.txt
echo "This is my first Git file." > hello.txt
git status
git add hello.txt
git commit -m "Initial commit: Added hello.txt”
```

Result:
A new Git repository was successfully created, the file hello.txt was added to the staging area, and the first commit was made successfully using Git commands.

Experiment 2: Creating and Managing Branches
Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "Feature-branch" into "master."

Aim:
To create a new branch, switch between branches, and merge the feature branch into the master branch using Git.

Algorithm / Procedure:

Step 1: Open Git Bash and navigate to your project directory
Command: cd /path/to/your/project
Step 2: Create a new branch named feature-branch
Command: git branch feature-branch

- This creates a new branch called feature-branch, but **does not switch to it**.

git branch feature-branch
Step 3: Switch to feature-branch
Command: git checkout feature-branch

- Now you're working in feature-branch. You can make changes, commit, etc

Step 4: Switch back to the master branch
Command: git checkout master
Step 5: Merge feature-branch into master
Command: git merge feature-branch

- This will apply the changes from feature-branch into master.

(Optional) Step 6: Delete the feature-branch after merging

- git branch -d feature-branch

Summary of Commands:
```
cd /path/to/your/project
git branch feature-branch
git checkout feature-branch
echo "This is content from feature-branch" >> hello.txt
git add hello.txt
git commit -m "Updated hello.txt in feature-branch"
git checkout master
git merge feature-branch
```

Delete a File in Git Bash
To delete a .txt file in an existing directory using **Git Bash**, follow these steps:
1. Navigate to the Directory
If you're not already in the correct directory:
Command: cd path/to/your/folder
2. Delete the File

Use the rm (remove) command:
Command: rm filename.txt
Example:
rm hello.txt
If You're Using Git with This Project
After deleting the file:
3. Check the Git Status
Command: git status
It will show that hello.txt has been deleted.
4. Stage the Deletion
Command: git rm filename.txt
OR you can just use git add . to stage all changes including deletions.
5. Commit the Deletion
Command: git commit -m "Deleted hello.txt"

Command Summary:
```
rm hello.txt         # Deletes the file
git status           # See the change
git rm hello.txt     # Tell Git the file was deleted (optional if using rm first)
git commit -m "Deleted hello.tx
```

Result:
A new branch named feature-branch was successfully created, switched, updated, and merged into the master branch, demonstrating basic branch management in Git.

Experiment 3: Creating and Managing Branches
Write the commands to stash your changes, switch branches, and then apply the stashed changes.

Aim:
To temporarily save uncommitted changes using Git stash, switch to another branch, and reapply the stashed changes.

Algorithm / Procedure:

Step 1: **Open Git Bash and Navigate to Project Directory**

- Move to your project folder.
  Command: cd /path/to/your/project

Step 2: **Make Some Changes**

- Modify or edit an existing file (for example):
  Command: echo "Temporary changes before switching branch" >> hello.txt

Step 3: **Stash Your Changes**

- Save uncommitted changes temporarily and clean the working directory.
  Command: git stash

Step 4: **Switch to Another Branch**

- Move to a different branch (e.g., feature-branch).
  Command: git checkout feature-branch

Step 5: **Apply the Stashed Changes**

- Reapply the latest stashed changes without removing them from stash history.
  Command: git stash apply

Step 6: **(Optional) Apply and Remove the Stash**

- If you want to apply and delete the stash entry in one step:
  Command: git stash pop

Command Summary:

```
git stash
git checkout feature-branch
git stash apply
```

Result:
The uncommitted changes were successfully stashed, the branch was switched, and the stashed changes were reapplied, demonstrating how to use Git stash for managing temporary work.

Experiment 4: Collaboration and Remote Repositories.
Clone a remote Git repository to your local machine.

Aim:
To create a remote repository on GitHub and clone it to the local machine using Git commands.

Algorithm / Procedure:
Step 1: Create a GitHub Account and Repository

1. Open https://github.com and sign up or log in.
2. Click on **New Repository** → Enter a name (example: git-prog4).
3. Add a description and select **Public**.
4. Tick **Initialize this repository with a README file**.
5. Click **Create Repository**.
6. To edit README:
   - Click on the **README.md** file → **Edit (pencil icon)** → modify text → **Commit changes**.
7. Verify the changes in the **Code** tab.

Step 2: Clone the Remote Repository to Local Machine

1. **Copy Repository URL**
   - On your GitHub repository page, click **Code** → copy the **HTTPS** URL (example: https://github.com/username/git-prog4.git)
2. **Open Git Bash and Navigate to Desired Folder**
   Command: cd /path/to/your/local/folder
3. **Clone the Repository**
   Use the git clone command to copy the remote repository to your local system.
   Command: git clone https://github.com/username/git-prog4.git
4. **Verify Cloning**
   - A new folder (git-prog4) will be created containing all the files and commit history.
     Command: cd git-prog4
     Command: ls

Step 3: Pull the Latest Changes from Remote

1. To update your local copy with the latest changes from GitHub:
   Command: git pull
2. To fetch changes first and then merge manually:
   Command: git fetch origin
   Command: git merge origin/main

Summary of Commands:
```
cd /path/to/your/local/folder
git clone https://github.com/username/git-prog4.git
```

```
cd git-prog4
git pull
git fetch origin
git merge origin/main
```

Result:
A remote repository was successfully created on GitHub, cloned to the local machine, and updated using Git pull and fetch commands, demonstrating collaboration between local and remote repositories.

Experiment 5: Collaboration and Remote Repositories:
Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Aim:
To fetch the latest changes from a remote repository and rebase the local branch onto the updated remote branch.

Algorithm / Procedure:

Step 1: Check your current branch
  Command: git branch
Step 2: Fetch changes from the remote repository
  Command: git fetch origin

  This command fetches updates from the origin (remote) without changing your working directory.
Step 3: Rebase your current branch onto the updated remote branch

- ✓ First, switch to feature-xyz (if you're not on it):
  Command: git checkout feature-xyz
  Now rebase onto origin/main:
  Command: git rebase origin/main
  Explanation:
- ✓ Git rewinds your local changes on feature-xyz.
- Applies the changes from origin/main.
- Reapplies your changes on top of them.

Example:

- ✓ Remote main has commits: A --- B --- C
- ✓ Your local feature-xyz has: A --- B --- C --- D --- E
- ✓ Remote main updates to: A --- B --- C --- F --- G (after fetch)
  You rebase your local branch:
  Command: git rebase origin/main
- ✓ Now your branch looks like:
  A --- B --- C --- F --- G --- D' --- E'
  (D' and E' are rebased versions of your local commits)

Step 4: Push your rebased branch (force push required)
  Since rebase changes history, you must force push:
  Command: git push origin feature-xyz –force

Summary of Commands

```
git checkout feature-xyz      # Switch to your working branch
git fetch origin              # Get latest from remote
git rebase origin/main        # Reapply your changes on top of latest main
# Resolve conflicts if needed
```

git push origin feature-xyz --force   # Update the remote branch

Difference between git **pull and git push** commands

- ✓ **git pull = download changes** from the remote repo to your local repo.
- ✓ **git push = upload your changes** from the local repo to the remote repo.

Result:

The latest changes from the remote repository were successfully fetched, and the local branch was rebased onto the updated remote branch, demonstrating effective synchronization between local and remote repositories.

Experiment 6: Collaboration and Remote Repositories

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge

Aim:
To merge a feature branch (feature-branch1) into the main branch (master) using Git, provide a custom merge commit message, and finally push the updated master branch to a remote repository.

Explanation: Suppose you're working on a project and have the following branches:

- master (main branch)
- feature-branch (where you've added new functionality)

Algorithm / Procedure:

Step 1: Initialize Git and create a new project folder
```
git init my-project
cd my-project
```

Step 2: Create a file and make the first commit in the master branch
```
echo "Master version" > welcome.txt
git add welcome.txt
git commit -m "master commit"
```

Step 3: Create and switch to a new branch named *feature-branch1*
```
git checkout -b feature-branch1
```
Make changes to the file:
```
echo "Added login feature" >> welcome.txt
git add welcome.txt
git commit -m "branch commit"
```

Step 4: Switch back to the master branch
```
git checkout master
```

Step 5: Merge the feature branch into master with a **custom commit message**
```
git merge feature-branch1 --no-ff -m "Merge feature-branch1: implemented login functionality"
```

Step 6: Add a remote repository and push the updated master branch
```
git remote add origin https://github.com/yourusername/my-project.git
git push origin master
```

Step 7 (Optional): Delete the feature branch after successful merge
```
git branch -d feature-branch1
```

Result after merging:

- The file welcome.txt now contains both “Master version” and “Added login feature”.
- A new merge commit is created in the master branch with your custom message.

Summary of Commands:

```
git init my-project
cd my-project
echo "Master version" > welcome.txt
git add welcome.txt
git commit -m "master commit"
git checkout -b feature-branch1
echo "Added login feature" >> welcome.txt
git add welcome.txt
git commit -m "branch commit"
git checkout master
git merge feature-branch1 --no-ff -m "Merge feature-branch1: implemented login
functionality"
git branch -d feature-branch1
git remote add origin https://github.com/yourusername/my-project.git
git push origin master
```

Result:

A new branch named **feature-branch1** was successfully created and merged into the **master** branch with a **custom commit message**.

The updated master branch was then **pushed to a remote repository**, demonstrating collaboration and remote repository management using Git.

PROGRAM 7: GIT TAGS AND RELEASES

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

AIM:

To create a **lightweight Git tag** named **"v1.0"** for a commit in the local repository and understand the purpose of Git tags.

EXPLANATION:

A **Git Tag** is a marker used to label a specific commit in the repository with a meaningful name (e.g., v1.0, v2.0).

Tags are commonly used for:

- Software releases
- Version management
- Marking stable points in the project

Types of Tags

1. **Lightweight Tag**
    - Simple pointer to a commit
    - No metadata (author, date, message)
    - Command: **git tag v1.0**
2. **Annotated Tag** (Not required here)
    - Stores metadata and message
    - Command: **git tag -a v1.0 -m "Release version 1.0"**

What is a Lightweight Tag?

A lightweight tag is like a branch that **does not move**.

It simply names a specific commit.

Algorithm/ Procedure:

Step 1: Initialize a Repository

- Command: git init myproject
- Command: cd myproject

Step 2: Create a File & Commit

- echo "Version 1 of my project" > file.txt
- git add file.txt
- git commit -m "Initial commit"

Step 3: Create a Lightweight Tag

- git tag v1.0
- This tags the latest commit as v1.0.

Step 4: Verify Tags

- git tag

Output:

- v1.0

Step 5: View Tagged Commit

git show v1.0

Step 6: Suppose you want to tag an older commit:
Command: git log --oneline
Step 7: Push Tags to Remote (Optional)
Command: git remote add origin https://github.com/yourname/git-tag-demo.git
Command: git push origin v1.0
(Or) to push all tags
Command: git push origin –tags

PROGRAM / COMMANDS:

```
# Step 1: Initialize a Repository
git init myproject
cd myproject

# Step 2: Create a File & Commit
echo "Version 1 of my project" > file.txt
git add file.txt
git commit -m "Initial commit"

# Step 3: Create a Lightweight Tag
git tag v1.0

# Step 4: Verify Tags
git tag
# Output: v1.0

# Step 5: View Tagged Commit
git show v1.0

# Step 6: Tag an Older Commit (optional)
git log --oneline
git tag v1.0 <commit-hash>

# Step 7: Push Tag to Remote (optional)
git remote add origin https://github.com/yourname/git-tag-demo.git
git push origin v1.0

# OR push all tags
git push origin –tags
```

SUMMARY OF COMMANDS :

- Create a lightweight tag:
  git tag v1.0
- Create a tag for a specific commit:
  git tag v1.0 <commit-hash>

- List all tags:
  git tag
- View the commit associated with the tag:
  git show v1.0
- Push a single tag to remote:
  git push origin v1.0
- Push all tags to remote:
  git push origin --tags

RESULT:
The lightweight Git tag **"v1.0"** was successfully created and applied to the commit. Tag details were verified, and the process of tagging specific commits and pushing tags to a remote repository was demonstrated.

Program 8: Advanced Git Operations

Write the command to cherry-pick a range of commits from “source-branch” to the current branch.

AIM:

To cherry-pick a **range of commits** from a source branch to the current branch using Git, and to understand the continuation and abort options during conflicts.

EXPLANATION:

Cherry-picking in Git allows you to **selectively apply specific commits** from one branch onto another.

What is Cherry-Picking?

- It copies changes from a selected commit (or range of commits) into your current branch.
- Git creates **new commit IDs** when cherry-picking.
- Useful for applying necessary fixes or features without merging the entire branch.

Cherry-Picking a Range

To apply multiple commits between two commit hashes:

git cherry-pick <start-commit>^..<end-commit>

- <start-commit> → First commit in the range
- <end-commit> → Last commit to include
- ^ excludes the starting commit itself
- Git applies all commits between the start and end

Handling Conflicts

- If a conflict occurs, Git stops and lets you fix the files.
- After fixing, continue with:
  git cherry-pick --continue
- To stop and undo cherry-picking:
  git cherry-pick –abort

Algorithm:

Step 1: Set your directory path (branch)
Step 2: Create Files, add, and commit the messages
Step 3: Use git log to explore the commit id that you would be applying to your custom feature branch
Command: git log (branch-name) –oneline
Step 4: Check out to master
Step 5: Once you have the commit you want to cherry-pick, copy the first few characters in the commit hash
Command: git cherry-pick <start-commit>^..<end-commit>
Step 6: Execute

PROGRAM / COMMANDS:

```
# Step 1: Check your current branch
git branch

# Step 2: Create files, add, and commit changes
echo "Line 1" > doc1.txt
git add file1.txt
git commit -m "Commit A"

echo "Line 2" > doc2.txt
git add file2.txt
git commit -m "Commit B"

echo "Line 3" > doc3.txt
git add file3.txt
git commit -m "Commit C"

# Step 3: Explore commit IDs in source branch
git log source-branch --oneline

# Step 4: Switch to master
git checkout master

# Step 5: Cherry-pick the required commit range
git cherry-pick <start-commit>^..<end-commit>

# If conflicts occur:
git cherry-pick --continue

# To abort cherry-pick:
git cherry-pick --abort
```

SUMMARY OF COMMANDS :

- **View commits in a branch:**
  git log branch-name --oneline
- **Cherry-pick a commit range:**
  git cherry-pick <start-commit>^..<end-commit>
- **Continue cherry-pick after resolving conflicts:**
  git cherry-pick --continue
- **Abort cherry-pick process:**
  git cherry-pick --abort
- **Switch to a branch:**
  git checkout branch-name

RESULT:
A range of commits from the source branch was successfully cherry-picked and applied to the current branch. The process of resolving conflicts, continuing the cherry-pick, and aborting when required was demonstrated.

Experiment 9: Analyzing and Changing Git History:
Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

Aim:
To view the complete details of a specific Git commit (such as author name, date, and commit message) using a given commit ID.

Explanation
Git stores the history of changes in a repository as a series of commits. Each commit has a unique commit ID (hash) that identifies it.
Using Git commands, we can:

- Inspect the details of any specific commit
- View metadata such as author, commit date, and commit message
- Analyze changes introduced in that commit

This is useful for:

- Debugging issues
- Reviewing changes made by a contributor
- Understanding project history

Algorithm / Steps

1. Open the Git Bash / Terminal.
2. Navigate to the required Git repository.
3. Identify or copy the commit ID you want to inspect.
4. Use the appropriate Git command to display commit details.
5. Observe the author name, date, commit message, and file changes.

Commands:

- To view the details of a specific commit, including the author, date, and commit message, you can use the git show or git log command with the commit ID. Here are both options:

1. Using git show:
Command: git show <commit-ID>

- Replace <commit-ID> with the actual commit ID you want to view. This command will display detailed information about the specified commit, including the commit message, author, date, and the changes introduced by that commit.

For example:
Command: $ git show abc123

2. Using git log:
Command: $ git log -n 1 <commit-ID>

- The -n 1 option tells Git to show only one commit. Replace <commit-ID> with the actual commit ID. This command will display a condensed view of the specified commit, including its commit message, author, date, and commit ID
- If you only want a more concise summary of the commit information (without the changes), you can use:

Command: git log -n 1 --pretty=format:"%h - %an, %ar : %s" <commit-id>

- This command displays a one-line summary of the commit, showing the abbreviated commit hash (%h), author name (%an), relative author date (%ar), and commit message (%s).

- Remember to replace <commit-id> with the actual commit hash or reference you want to inspect.

View all commits
$ git log
This shows:
- Commit ID (hash)
- Author
- Date
- Commit message

Example output:
```
commit a1b2c3d4e5f6g7h8
Author: Ravi Kumar
Date:   Mon Dec 11 10:45:30 2025 +0530

    Added login validation
```
Here,
a1b2c3d4e5f6g7h8 is the commit ID

Short and simple commit list
$ git log --oneline
Example:
```
a1b2c3d Added login validation
f4e5d6a Fixed UI issue
9c8b7a6 Initial commit
```
You can use the first 6–7 characters (a1b2c3d) as the commit ID.

Result
Thus, the details of the given commit ID including author, date, commit message, and changes made were successfully viewed using the git show command.

Experiment 10: Analyzing and Changing Git History

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."

Aim

To list all Git commits made by the author “JohnDoe” between the dates 2023-01-01 and 2023-12-31.

Explanation

Git allows us to filter commit history using different options such as:

- Author name
- Date range
- Commit message

By using the git log command with:

- --author → filters commits by author name
- --since → shows commits after a specific date
- --until → shows commits before a specific date

we can list only the commits made by a particular author within a given time period.
Dates should be given in YYYY-MM-DD format for clarity.

Algorithm / Steps

1. Open Git Bash / Terminal.
2. Navigate to the required Git repository.
3. Use the git log command with author and date filters.
4. View the list of commits that match the given conditions.

Command Used

Basic command

```
$ git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"
```

Command Explanation

- git log → Displays commit history
- --author="JohnDoe" → Shows commits made only by JohnDoe
- --since="2023-01-01" → Includes commits from this date onward
- --until="2023-12-31" → Includes commits up to this date

Command: One-line Format

```
$ git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31" --oneline
```

Sample Output

```
a1b2c3d Fixed login validation bug
d4e5f6a Added user profile page
9c8b7a6 Updated README file
```

Here:

- a1b2c3d, d4e5f6a, 9c8b7a6 → Commit IDs
- The messages show work done by JohnDoe in 2023

Result:
Thus, all commits made by the author JohnDoe between 2023-01-01 and 2023-12-31 were successfully listed using the `git log` command with author and date filters.

Program 11: Analyzing and Changing Git History

Write the command to display the last five commits in the repository's history.

Aim

To display the last five commits in the repository's history using Git.

Explanation

Git stores all changes to a project as commits.
The git log command is used to view the commit history.
By default, git log shows all commits.
To limit the output to a specific number of recent commits, we use the option -n, where n represents the number of commits.

Command Used

```
$ git log -5
```

Command Explanation

- git log → Displays commit history
- -5 → Limits the output to the last 5 commits

Sample Output

```
commit a1b2c3d4e5
Author: Sudhakar S <sudhakars106@gmail.com>
Date:   Mon Dec 15 09:45:30 2025 +0530
 Updated README file
commit f6e7d8c9b0
Author: Sudhakar S <sudhakars106@gmail.com>
Date:   Sun Dec 14 14:10:12 2025 +0530
 Fixed login issue
```

Alternative Simple View

```
$ git log --oneline -5
```

Example output:

```
a1b2c3d Updated README file
f6e7d8c Fixed login issue
```

Result

Thus, the last five commits in the repository's history were successfully displayed using the git log -5 command.

Experiment 12: Analyzing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".

Aim
To undo the changes introduced by a specific commit with the commit ID abc123.

Explanation
In Git, undoing a commit can be done in different ways depending on whether:

- You want to keep the commit history safe, or
- You want to remove the commit completely.

For most cases (especially in shared repositories), Git recommends reverting a commit instead of deleting it.

Algorithm (Including Getting Commit ID)

1. Open **Git Bash / Terminal**.
2. Create a new project directory and move into it.
3. Initialize the Git repository.
4. Create a new file and add content to it.
5. Add the file to the staging area.
6. Commit the file to the repository.
7. Display the commit history to **get the commit ID**.
8. Undo the changes introduced by the required commit ID (abc123).
9. Verify that the commit has been reverted.

Detailed Commands
Step 1: Create project folder and initialize Git
```
$ mkdir git-history-demo
$ cd git-history-demo
$ git init
```

Step 2: Create a file
```
$ touch sample.txt
$ echo "Git history example" > sample.txt
```

Step 3: Add file to staging
```
$ git add sample.txt
```

Step 4: Commit the file
```
$ git commit -m "Added sample.txt file"
```

Step 5: Get the commit ID
```
$ git log --oneline
```

Sample output:

```
abc123 Added sample.txt file
```
Here, **abc123 is the commit ID**

Step 6: Undo the changes introduced by commit abc123

```
$ git revert abc123
```

Step 7: Verify the result

```
$ git log --oneline
```
You will see a new commit like:
```
def456 Revert "Added sample.txt file"
abc123 Added sample.txt file
```

Main Command

$ git revert abc123

- Creates a new commit
- This new commit reverses the changes made by abc123
- Original commit remains in history (safe for teamwork)

Sample Output

```
Revert "Fixed login bug"
This reverts commit abc123.
```

Result

Thus, the changes introduced by the commit with ID abc123 were successfully undone using the appropriate Git command.