Q1: Explain what Git is and its primary purpose.

Git is a distributed version control system widely used for tracking changes in source code during software development. Unlike other version control systems, Git stores data as a series of snapshots rather than as a list of changes. This approach allows developers to easily track the history of their project, revert to previous versions, and collaborate with others.

```
// Check the current Git status
$ git status


// View commit history
$ git log
```

Q2: Describe what a branch is in Git and its benefits.

A branch in Git is a pointer to a specific commit that allows you to work on different features or fixes independently from the main codebase. Branches enable you to isolate changes, experiment with new ideas, or develop features without affecting the stable version of your project. This makes it easier to manage development workflows, especially in collaborative environments.

```
// Create a new branch
$ git branch feature-branch


// Switch to the newly created branch
$ git checkout feature-branch
```

Branches are essential in Git for maintaining multiple lines of development, such as working on different features simultaneously or preparing releases.

Q3: What is a commit in Git and what information does it contain?

A commit in Git represents a snapshot of your project at a specific point in time. When you make a commit, Git records the current state of the files in your repository, along with metadata such as the author, commit message, timestamp, and a unique SHA-1 hash. This commit hash allows you to reference and revert to this specific state whenever needed.

```
// Create a commit with a message
$ git commit -m "Implement feature X"


// View details of a specific commit
$ git show <commit-hash>
```

Commits serve as milestones in your project's history, enabling you to track progress, undo changes, and collaborate with others effectively.

Q4: What is an untracked file in Git?

An untracked file in Git refers to a file that exists in your working directory but has not yet been added to the staging area. Git does not include these files in version control until you explicitly add them using the git add command.

```
// View untracked files
$ git status
```

```
// Add an untracked file to the staging area
$ git add filename.txt
```

Untracked files are useful for seeing which new files are present in your working directory but haven't yet been committed to the repository.

Q5: Explain what GitLab is and its primary features.

GitLab is a web-based DevOps platform that offers version control, continuous integration/continuous delivery (CI/CD) pipelines, and project management features. It provides a collaborative environment for development teams to manage code repositories, track issues, and automate the software development lifecycle.

- Version control with Git repositories
- Integrated CI/CD for automated testing and deployment
- Issue tracking and project management tools
- Code review and merge request capabilities
- Security and compliance management

GitLab is widely used in both open-source and enterprise environments to streamline software development and enhance collaboration among team members.

Q6: What are the three possible statuses for changes in Git?

The three possible statuses for changes in Git are:

- Modified: Files have been changed but not yet staged
- Staged: Changes have been added to the staging area and are ready to be committed
- Untracked: New files that have not yet been added to the staging area

```
// Check the status of files in your repository
$ git status
```

Understanding these statuses helps you manage your changes and prepare them for commits effectively.

Q7: How do you resolve conflicts while merging branches in Git?

To resolve conflicts during a Git merge, you need to manually edit the conflicting files to incorporate the necessary changes from both branches. You can do this directly in the command line or use graphical tools/IDEs that offer merge conflict resolution interfaces. After resolving the conflicts, stage and commit the resolved files.

```
// Start the merge process
$ git merge feature-branch

// If conflicts arise, edit the conflicting files and resolve them

// After resolving conflicts, add the resolved files to the staging area
$ git add resolved-file.txt

// Complete the merge by committing the changes
$ git commit -m "Resolve merge conflicts"
```

Proper conflict resolution ensures that your codebase remains functional and that all necessary changes are integrated correctly.

Q8: What is the purpose of a remote repository in Git?

A remote repository in Git is a version of your repository hosted on a server or a different machine, allowing multiple developers to collaborate on the same project. Remote repositories enable version control across different locations, making it easier to share changes and keep your project synchronized.

```
// Add a remote repository
$ git remote add origin https://github.com/user/repo.git

// Push changes to the remote repository
$ git push origin main
```

Remote repositories are essential for distributed development, allowing teams to collaborate effectively, share their work, and manage versions across different machines.

Q9: How do you create and switch to a new branch in Git?

To create a new branch in Git, use the git branch command followed by the branch name. You can then switch to the new branch using git checkout or git switch.

```
// Create a new branch
$ git branch feature-branch

// Switch to the new branch
$ git checkout feature-branch

// Alternatively, create and switch in one command
$ git switch -c feature-branch
```

Creating branches allows you to work on separate features or fixes without affecting the main codebase, making it easier to manage and merge changes later.

Q10: What is the purpose of rebasing in Git, and how is it different from merging?

Rebasing in Git is a way to apply changes from one branch onto another, resulting in a linear commit history. Unlike merging, which creates a new commit to represent the combination of changes, rebasing rewrites the commit history to incorporate changes from one branch onto another, making the history cleaner and easier to follow.

```
// Rebase the current branch onto main
$ git checkout feature-branch
$ git rebase main

// After resolving any conflicts, continue the rebase
$ git rebase --continue
```

Rebasing is particularly useful for keeping a project's commit history clean and for integrating the latest changes from the main branch into your feature branches without creating unnecessary merge commits.

Q11: What is the difference between git merge and git rebase?

Both git merge and git rebase are used to integrate changes from one branch into another, but they do so in different ways. git merge creates a new commit that combines the changes from both branches, preserving the history of both.

On the other hand, git rebase moves or applies your commits on top of another branch, creating a linear history by avoiding merge commits.

```
// Merging feature-branch into main
$ git checkout main
$ git merge feature-branch

// Rebasing feature-branch onto main
$ git checkout feature-branch
$ git rebase main
```

Use git merge when you want to preserve the history of all branches, and git rebase when you prefer a cleaner, linear commit history.

Q12: How do you revert a commit in Git?

To revert a commit in Git, you can use the git revert command, which creates a new commit that undoes the changes introduced by a previous commit. This method is preferred over git reset for undoing changes in a shared history because it doesn't alter the commit history.

```
// Revert a specific commit by its hash
$ git revert <commit-hash>

$ git revert 7a1f9d2
```

Reverting is a safe way to undo changes because it preserves the commit history and can be easily traced.

Q13: How do you stash changes in Git, and why is it useful?

Stashing in Git allows you to save your uncommitted changes without committing them, so you can switch branches or perform other operations without losing your work. The git stash command temporarily saves your changes and leaves your working directory clean.

```
// Stash your changes
$ git stash

// Apply the stashed changes
$ git stash apply

// List all stashes
$ git stash list
```

Stashing is particularly useful when you need to quickly switch contexts or when you are not ready to commit your changes but want to work on something else.

Q14: How do you clone a Git repository, and what does it do?

Cloning a Git repository means creating a copy of a remote repository on your local machine. The git clone command copies all the data, including the commit history, branches, and tags, from the remote repository to your local system.

```
// Clone a remote repository
$ git clone https://github.com/user/repo.git
```

```
// Clone with a specific branch
$ git clone -b branch-name https://github.com/user/repo.git
```

Cloning is the first step in contributing to an existing project, as it gives you a complete local version of the repository to work with.

Q15: What is the purpose of git pull, and how does it differ from git fetch?

git pull is a command that fetches changes from a remote repository and immediately tries to merge them into your current branch. It is a combination of git fetch, which downloads new data from the remote repository, and git merge, which integrates the changes into your working branch.

```
// Fetch changes from the remote repository
$ git fetch

// Merge the fetched changes into your current branch
$ git merge origin/main

// Pull in one step
$ git pull origin main
```

git fetch allows you to review changes before integrating them, while git pull automatically merges them, which can sometimes lead to conflicts. Use git pull when you want to directly update your branch with the latest changes from the remote repository.