**GIT ASSIGNMENT 3(HotWax)**  
MUDIT THAKRE

Q1. Why do we create branches in a repository and why do we create pull requests instead of merging directly?

Ans. Making branches in a repository and using pull requests are key practices in today's version control. These methods help teams work better together and keep code quality high. Branches let developers work on new features or fixes without messing up the main code. This means people can work at the same time try new things , and keep the main branch stable and tidy. Pull requests, on the other hand, make it easy to review code and talk about changes. This ensures team members check all changes before they're added to the main code. They also allow for automatic testing and keep a record of all changes, which helps control the process of adding new code. Together, these practices help teams handle complex code more and lead to a better end product.

Q2. What is the difference between git add . and git add <filename>? What will we use when we have changes in multiples but we are not required to add some files?

Ans. The "git add ." command stages all changes (new, modified, and deleted files) in the current directory and its subdirectories. It's a fast way to add everything at once, but you can't pick and choose. The "git add <filename>" command stages only the specified file. It gives you exact control, so you can add changes to certain files without affecting others.

When you have changes in multiple files but do not want to stage all of them, you should use “git add <filename>” for each file you want to stage selectively. This approach ensures that only the desired changes are staged for the next commit, allowing for more granular control over what gets committed.

Q3. What is the difference between git fetch and git pull?

Ans. **git fetch**: Downloads changes from the remote repository to our local repository but does not merge them into your current branch. It updates your remote tracking branches. **git pull**: Downloads changes from the remote repository and automatically merges them into your current branch. It combines git fetch with git merge.

Q4. What is a head in a repository and what does it do?

Ans. HEAD in a Git repository is a reference to the last commit of the currently checked-out branch, indicating the most recent state of the code. It moves with each new commit and changes when switching branches, allowing Git to track the current working snapshot and branch context.

Q5. What is the .git folder in a repository?  
  
Ans. The .git folder in a repository is a hidden directory where Git stores all the information and objects necessary to manage the version control aspects of that repository. This includes configurations, logs, references to commits, branches, the staging area, and more. Essentially, it contains the entire history and configuration settings of the repository.

Q6. What are commit hashes and its use cases?  
  
Ans. Commit hashes in Git are unique identifiers for each commit. They are 40-character strings created by hashing the contents of the commit (including changes, author, date, and parent commit information) using the SHA-1 hash function.

Use Cases:  
Identifying Commits: Commit hashes uniquely identify each commit in the repository's history, allowing precise reference to specific changes.

Ensuring Integrity: The hash ensures the integrity of the commit history. Any change to a commit's content changes its hash, indicating alterations or corruption.

Navigating History: Hashes are used to navigate the commit history, check out specific commits, or compare changes between commits.

Branching and Merging: They are crucial in branching and merging operations, helping Git to track parent-child relationships between commits across different branches.

Tagging Releases: Commit hashes are used when tagging specific commits for releases, ensuring the exact code state is retrievable.

Automation and Scripting: In automated workflows, commit hashes are used to trigger builds, deployments, or other CI/CD processes based on specific commits.  
  
  
  
Q7. Different ways of syncing a branch with origin.  
  
Ans. Syncing a branch with its remote origin can be achieved through many Git commands such as

1. git fetch + git merge:  
   git fetch origin <branch>: Fetches updates from the remote branch but doesn't merge them.
2. git merge origin/<branch>: Merges the fetched changes into your current branch. This approach allows you to review changes before merging.  
   git pull:
3. git pull origin <branch>: A shorthand for git fetch followed by git merge. It fetches the remote branch and immediately merges the changes.
4. git rebase: After fetching with git fetch origin <branch>, use git rebase origin/<branch> to reapply your local branch's commits on top of the fetched branch commits. This is useful for maintaining a linear project history.
5. Using Git GUIs or IDEs: Many graphical user interfaces (GUIs) for Git and integrated development environments (IDEs) provide tools to sync branches with their remotes. These tools often abstract the commands into buttons or menu actions.