1. HOW CAN WE SEE N MOST RECENT COMMITS IN GIT?
WE CAN USE GIT LOG COMMAND TO SEE THE LATEST COMMITS. TO SEE THE THREE MOST RECENT COMMITS WE USE FOLLOWING COMMAND:

GIT LOG -3

2. HOW CAN WE KNOW IF A BRANCH IS ALREADY MERGED INTO MASTER IN GIT?
WE CAN USE FOLLOWING COMMANDS FOR THIS PURPOSE:
GIT BRANCH --MERGED MASTER : THIS PRINTS THE BRANCHES MERGED INTO MASTER
GIT BRANCH --MERGED LISTS : THIS PRINTS THE BRANCHES MERGED INTO HEAD (I.E. TIP OF CURRENT BRANCH)
GIT BRANCH --NO-MERGED : THIS PRINTS THE BRANCHES THAT HAVE NOT BEEN MERGED
BY DEFAULT THIS APPLIES ONLY TO LOCAL BRANCHES.

WE CAN USE -A FLAG TO SHOW BOTH LOCAL AND REMOTE BRANCHES.
OR WE CAN USE -R FLAG TO SHOW ONLY THE REMOTE BRANCHES.

. WHAT IS THE PURPOSE OF GIT STASH DROP?
IN CASE WE DO NOT NEED A SPECIFIC STASH, WE USE GIT STASH DROP COMMAND TO REMOVE IT FROM THE LIST OF STASHES.
BY DEFAULT, THIS COMMAND REMOVES TO LATEST ADDED STASH
TO REMOVE A SPECIFIC STASH WE SPECIFY AS ARGUMENT IN THE GIT STASH DROP COMMAND.

4. WHAT IS THE HEAD IN GIT?
A HEAD IS A REFERENCE TO THE CURRENTLY CHECKED OUT COMMIT.
IT IS A SYMBOLIC REFERENCE TO THE BRANCH THAT WE HAVE CHECKED OUT.
AT ANY GIVEN TIME, ONE HEAD IS SELECTED AS THE ‘CURRENT HEAD’ THIS HEAD IS ALSO KNOWN AS HEAD (ALWAYS IN UPPERCASE).

5. WHAT IS THE MOST POPULAR BRANCHING STRATEGY IN GIT?
THERE ARE MANY WAYS TO DO BRANCHING IN GIT. ONE OF THE POPULAR WAYS IS TO MAINTAIN TWO BRANCHES:

MASTER: THIS BRANCH IS USED FOR PRODUCTION. IN THIS BRANCH HEAD IS ALWAYS IN PRODUCTION READY STATE.
DEVELOP: THIS BRANCH IS USED FOR DEVELOPMENT. IN THIS BRANCH WE STORE THE LATEST CODE DEVELOPED IN PROJECT. THIS IS WORK IN PROGRESS CODE.

6. WHAT IS SUBGIT?
SUBGIT IS SOFTWARE TOOL USED FOR MIGRATING SVN TO GIT. IT IS VERY EASY TO USE. BY USING THIS WE CAN CREATE A WRITABLE GIT MIRROR OF A SUBVERSION REPOSITORY.
IT CREATES A BI-DIRECTIONAL MIRROR THAT CAN BE USED FOR PUSHING TO GIT AS WELL AS COMMITTING TO SUBVERSION.
SUBGIT ALSO TAKES CARE OF SYNCHRONIZATION BETWEEN GIT AND SUBVERSION.

7. WHAT IS THE USE OF GIT INSTAWEB?
GIT-INSTAWEB IS A SCRIPT BY WHICH WE CAN BROWSE A GIT REPOSITORY IN A WEB BROWSER.
IT SETS UP THE GITWEB AND A WEB-SERVER THAT MAKES THE WORKING REPOSITORY AVAILABLE ONLINE.

8. WHAT ARE GIT HOOKS?
GIT HOOKS ARE SCRIPTS THAT CAN RUN AUTOMATICALLY ON THE OCCURRENCE OF AN EVENT IN A GIT REPOSITORY. THESE ARE USED FOR AUTOMATION OF WORKFLOW IN GIT.
GIT HOOKS ALSO HELP IN CUSTOMIZING THE INTERNAL BEHAVIOR OF GIT.
THESE ARE GENERALLY USED FOR ENFORCING A GIT COMMIT POLICY.

9. WHAT IS GIT?
GIT IS A MATURE DISTRIBUTED VERSION CONTROL SYSTEM (DVCS). IT IS USED FOR SOURCE CODE MANAGEMENT (SCM).
IT IS OPEN SOURCE SOFTWARE. IT WAS DEVELOPED BY LINUS TORVALDS, THE CREATOR OF LINUX OPERATING SYSTEM.
GIT WORKS WELL WITH A LARGE NUMBER OF IDES (INTEGRATED DEVELOPMENT ENVIRONMENTS) LIKE- ECLIPSE, INTELIJ ETC.
GIT CAN BE USED TO HANDLE SMALL AND LARGE PROJECTS.

10. WHAT IS A REPOSITORY IN GIT?
A REPOSITORY IN GIT IS THE PLACE IN WHICH WE STORE OUR SOFTWARE WORK.
IT CONTAINS A SUB-DIRECTORY CALLED .GIT. THERE IS ONLY ONE .GIT DIRECTORY IN THE ROOT OF THE PROJECT.
IN .GIT, GIT STORES ALL THE METADATA FOR THE REPOSITORY. THE CONTENTS OF .GIT DIRECTORY ARE OF INTERNAL USE TO GIT.

11. WHAT ARE THE MAIN BENEFITS OF GIT?
THERE ARE FOLLOWING MAIN BENEFITS OF GIT:

DISTRIBUTED SYSTEM: GIT IS A DISTRIBUTED VERSION CONTROL SYSTEM (DVCS). SO YOU CAN KEEP YOUR PRIVATE WORK IN VERSION CONTROL BUT COMPLETELY HIDDEN FROM OTHERS. YOU CAN WORK OFFLINE AS WELL.
FLEXIBLE WORKFLOW: GIT ALLOWS YOU TO CREATE YOUR OWN WORKFLOW. YOU CAN USE THE PROCESS THAT IS SUITABLE FOR YOUR PROJECT. YOU CAN GO FOR CENTRALIZED OR MASTER-SLAVE OR ANY
OTHER WORKFLOW.
FAST: GIT IS VERY FAST WHEN COMPARED TO OTHER VERSION CONTROL SYSTEMS.
DATA INTEGRITY: SINCE GIT USES SHA1, DATA IS NOT EASIER TO CORRUPT.
FREE: IT IS FREE FOR PERSONAL USE. SO MANY AMATEURS USE IT FOR THEIR INITIAL PROJECTS. IT ALSO WORKS VERY WELL WITH LARGE SIZE PROJECT.
COLLABORATION: GIT IS VERY EASY TO USE FOR PROJECTS IN WHICH COLLABORATION IS REQUIRED. MANY POPULAR OPEN SOURCE SOFTWARE ACROSS THE GLOBE USE GIT.

12. WHAT ARE THE DISADVANTAGES OF GIT?
GIT HAS VERY FEW DISADVANTAGES. THESE ARE THE SCENARIOS WHEN GIT IS DIFFICULT TO USE. SOME OF THESE ARE:

BINARY FILES: IF WE HAVE A LOT BINARY FILES (NON-TEXT) IN OUR PROJECT, THEN GIT BECOMES VERY SLOW. E.G. PROJECTS WITH A LOT OF IMAGES OR WORD DOCUMENTS.
STEEP LEARNING CURVE: IT TAKES SOME TIME FOR A NEWCOMER TO LEARN GIT. SOME OF THE GIT COMMANDS ARE NON-INTUITIVE TO A FRESHER.
SLOW REMOTE SPEED: SOMETIMES THE USE OF REMOTE REPOSITORIES IN SLOW DUE TO NETWORK LATENCY. STILL GIT IS BETTER THAN OTHER VCS IN SPEED.

13. WHAT ARE THE MAIN DIFFERENCES BETWEEN GIT AND SVN?
THE MAIN DIFFERENCES BETWEEN GIT AND SVN ARE:

DECENTRALIZED: GIT IS DECENTRALIZED. YOU HAVE A LOCAL COPY THAT IS A REPOSITORY IN WHICH YOU CAN COMMIT. IN SVN YOU HAVE TO ALWAYS CONNECT TO A CENTRAL REPOSITORY FOR CHECK-IN.
COMPLEX TO LEARN: GIT IS A BIT DIFFICULT TO LEARN FOR SOME DEVELOPERS. IT HAS MORE CONCEPTS AND COMMANDS TO LEARN. SVN IS MUCH EASIER TO LEARN.
UNABLE TO HANDLE BINARY FILES: GIT BECOMES SLOW WHEN IT DEALS WITH LARGE BINARY FILES THAT CHANGE FREQUENTLY. SVN CAN HANDLE LARGE BINARY FILES EASILY.
INTERNAL DIRECTORY: GIT CREATES ONLY .GIT DIRECTORY. SVN CREATES .SVN DIRECTORY IN EACH FOLDER.

USER INTERFACE: GIT DOES NOT HAVE GOOD UI. BUT SVN HAS GOOD USER INTERFACES.

14. HOW WILL YOU START GIT FOR YOUR PROJECT?
WE USE GIT INIT COMMAND IN AN EXISTING PROJECT DIRECTORY TO START VERSION CONTROL FOR OUR PROJECT.
AFTER THIS WE CAN USE GIT ADD AND GIT COMMIT COMMANDS TO ADD FILES TO OUR GIT REPOSITORY.

15. WHAT IS GIT CLONE IN GIT?
IN GIT, WE USE GIT CLONE COMMAND TO CREATE A COPY OF AN EXISTING GIT REPOSITORY IN OUR LOCAL.
THIS IS THE MOST POPULAR WAY TO CREATE A COPY OF THE REPOSITORY AMONG DEVELOPERS.
IT IS SIMILAR TO SVN CHECKOUT. BUT IN THIS CASE THE WORKING COPY IS A FULL-FLEDGED REPOSITORY.

16. HOW WILL YOU CREATE A REPOSITORY IN GIT?
TO CREATE A NEW REPOSITORY IN GIT, FIRST WE CREATE A DIRECTORY FOR THE PROJECT.
THEN WE RUN ‘GIT INIT’ COMMAND.
NOW, GIT CREATES .GIT DIRECTORY IN OUR PROJECT DIRECTORY. THIS IS HOW OUR NEW GIT REPOSITORY IS CREATED.

17. WHAT ARE THE DIFFERENT WAYS TO START WORK IN GIT?
WE CAN START WORK IN GIT IN FOLLOWING WAYS:

NEW PROJECT: TO CREATE A NEW REPOSITORY WE USE GIT INIT COMMAND.
EXISTING PROJECT: TO WORK ON AN EXISTING REPOSITORY WE USE GIT CLONE COMMAND.

18. GIT IS WRITTEN IN WHICH LANGUAGE?
MOST OF THE GIT DISTRIBUTIONS ARE WRITTEN IN C LANGUAGE WITH BOURNE SHELL. SOME OF THE COMMANDS ARE WRITTEN IN PERL LANGUAGE.

19. WHAT DOES ‘GIT PULL’ COMMAND IN GIT DO INTERNALLY?
IN GIT, GIT PULL INTERNALLY DOES A GIT FETCH FIRST AND THEN DOES A GIT MERGE.
SO PULL IS A COMBINATION OF TWO COMMANDS: FETCH AND MERGE.
WE USE GIT PULL COMMAND TO BRING OUR LOCAL BRANCH UP TO DATE WITH ITS REMOTE VERSION.

20. WHAT DOES ‘GIT PUSH’ COMMAND IN GIT DO INTERNALLY?
IN GIT, GIT PUSH COMMAND DOES FOLLOWING TWO COMMANDS:

FETCH: FIRST GIT, COPIES ALL THE EXTRA COMMITS FROM SERVER INTO LOCAL REPO AND MOVES ORIGIN/MASTER BRANCH POINTER TO THE END OF COMMIT CHAIN.
MERGE: THEN IT MERGES THE ORIGIN/MASTER BRANCH INTO THE MASTER BRANCH. NOW THE MASTER BRANCH POINTER MOVES TO THE NEWLY CREATED COMMIT. BUT THE ORIGIN/MASTER POINTER REMAINS THERE.

21. WHAT IS GIT STASH?
IN GIT, SOMETIMES WE DO NOT WANT TO COMMIT OUR CODE BUT WE DO NOT WANT TO LOSE ALSO THE UNFINISHED CODE. IN THIS CASE WE USE GIT STASH COMMAND TO RECORD THE CURRENT STATE OF THE WORKING DIRECTORY AND INDEX IN A STASH. THIS STORES THE UNFINISHED WORK IN A STASH, AND CLEANS THE CURRENT BRANCH FROM UNCOMMITTED CHANGES.
NOW WE CAN WORK ON A CLEAN WORKING DIRECTORY.
LATER WE CAN USE THE STASH AND APPLY THOSE CHANGES BACK TO OUR WORKING DIRECTORY.

AT TIMES WE ARE IN THE MIDDLE OF SOME WORK AND DO NOT WANT TO LOSE THE UNFINISHED WORK, WE USE GIT STASH COMMAND.

22. WHAT IS THE MEANING OF ‘STAGE’ IN GIT?
IN GIT, STAGE IS A STEP BEFORE COMMIT. TO STAGE MEANS THAT THE FILES ARE READY FOR COMMIT.
LET SAY, YOU ARE WORKING ON TWO FEATURES IN GIT. ONE OF THE FEATURES IS FINISHED AND THE OTHER IS NOT YET READY. YOU WANT TO COMMIT AND LEAVE FOR HOME IN THE EVENING. BUT YOU CAN COMMIT SINCE BOTH OF THEM ARE NOT FULLY READY. IN THIS CASE YOU CAN JUST STAGE THE FEATURE THAT IS READY AND COMMIT THAT PART. SECOND FEATURE WILL REMAIN AS WORK IN PROGRESS.

23. WHAT IS THE PURPOSE OF GIT CONFIG COMMAND?
WE CAN SET THE CONFIGURATION OPTIONS FOR GIT INSTALLATION BY USING GIT CONFIG COMMAND.

24. HOW CAN WE SEE THE CONFIGURATION SETTINGS OF GIT INSTALLATION?
WE CAN USE ‘GIT CONFIG --LIST’ COMMAND TO PRINT ALL THE GIT CONFIGURATION SETTINGS IN GIT INSTALLATION.

25. HOW WILL YOU WRITE A MESSAGE WITH COMMIT COMMAND IN GIT?
WE CALL FOLLOWING COMMAND FOR COMMIT WITH A MESSAGE:

$/> GIT COMMIT –M

26. WHAT IS STORED INSIDE A COMMIT OBJECT IN GIT?
GIT COMMIT OBJECT CONTAINS FOLLOWING INFORMATION:
SHA1 NAME: A 40 CHARACTER STRING TO IDENTIFY A COMMIT
FILES: LIST OF FILES THAT REPRESENT THE STATE OF A PROJECT AT A SPECIFIC POINT OF TIME
REFERENCE: ANY REFERENCE TO PARENT COMMIT OBJECTS

27. HOW MANY HEADS CAN YOU CREATE IN A GIT REPOSITORY?
THERE CAN BE ANY NUMBER OF HEADS IN A REPOSITORY.
BY DEFAULT THERE IS ONE HEAD KNOWN AS HEAD IN EACH REPOSITORY IN GIT.

28. WHY DO WE CREATE BRANCHES IN GIT?
IF WE ARE SIMULTANEOUSLY WORKING ON MULTIPLE TASKS, PROJECTS, DEFECTS OR FEATURES, WE NEED MULTIPLE BRANCHES. IN GIT WE CAN CREATE A SEPARATE BRANCH FOR EACH SEPARATE PURPOSE.
LET SAY WE ARE WORKING ON A FEATURE, WE CREATE A FEATURE BRANCH FOR THAT. IN BETWEEN WE GET A DEFECT TO WORK ON THEN WE CREATE ANOTHER BRANCH FOR DEFECT AND WORK ON IT. ONCE THE DEFECT WORK IS DONE, WE MERGE THAT BRANCH AND COME BACK TO WORK ON FEATURE BRANCH AGAIN.
SO WORKING ON MULTIPLE TASKS IS THE MAIN REASON FOR USING MULTIPLE BRANCHES.

29. WHAT ARE THE DIFFERENT KINDS OF BRANCHES THAT CAN BE CREATED IN GIT?
WE CAN CREATE DIFFERENT KINDS OF BRANCHES FOR FOLLOWING PURPOSES IN GIT:
FEATURE BRANCHES: THESE ARE USED FOR DEVELOPING A FEATURE.
RELEASE BRANCHES: THESE ARE USED FOR RELEASING CODE TO PRODUCTION.
HOTFIX BRANCHES: THESE ARE USED FOR RELEASING A HOTFIX TO PRODUCTION FOR A DEFECT OR EMERGENCY FIX.

30. HOW WILL YOU CREATE A NEW BRANCH IN GIT?

WE USE FOLLOWING COMMAND TO CREATE A NEW BRANCH IN GIT:

$/> GIT CHECKOUT –B

31. HOW WILL YOU ADD A NEW FEATURE TO THE MAIN BRANCH?
WE DO THE DEVELOPMENT WORK ON A FEATURE BRANCH THAT IS CREATED FROM MASTER BRANCH. ONCE THE DEVELOPMENT WORK IS READY WE USE GIT MERGE COMMAND TO MERGE IT INTO MASTER BRANCH.

32. WHAT IS A PULL REQUEST IN GIT?
A PULL REQUEST IN GIT IS THE LIST OF CHANGES THAT HAVE BEEN PUSHED TO GIT REPOSITORY. GENERALLY THESE CHANGES ARE PUSHED IN A FEATURE BRANCH OR HOTFIX BRANCH. AFTER PUSHING THESE CHANGES WE CREATE A PULL REQUEST THAT CONTAINS THE CHANGES BETWEEN MASTER AND OUR FEATURE BRANCH. THIS PULL REQUEST IS SENT TO REVIEWERS FOR REVIEWING THE CODE AND THEN MERGING IT INTO DEVELOP OR RELEASE BRANCH.

33. WHAT IS MERGE CONFLICT IN GIT?
A MERGE CONFLICT IN GIT IS THE RESULT OF MERGING TWO COMMITS.
SOMETIMES THE COMMIT TO BE MERGED AND CURRENT COMMIT HAVE CHANGES IN SAME LOCATION. IN THIS SCENARIO, GIT IS NOT ABLE TO DECIDE WHICH CHANGE IS MORE IMPORTANT. DUE TO THIS GIT REPORTS A MERGE CONFLICT. IT MEANS MERGE IS NOT SUCCESSFUL. WE MAY HAVE TO MANUALLY CHECK AND RESOLVE THE MERGE CONFLICT.

34. HOW CAN WE RESOLVE A MERGE CONFLICT IN GIT?
WHEN GIT REPORTS MERGE CONFLICT IN A FILE, IT MARKS THE LINES AS FOLLOWS:

EXAMPLE:

THE BUSINESS DAYS IN THIS WEEK ARE
<<<<<<< HEAD FIVE ======= SIX >>>>>>> BRANCH-FEATURE
TO RESOLVE THE MERGE CONFLICT IN A FILE, WE EDIT THE FILE AND FIX THE CONFLICTING CHANGE. IN ABOVE EXAMPLE WE CAN EITHER KEEP FIVE OR SIX.
AFTER EDITING THE FILE WE RUN GIT ADD COMMAND FOLLOWED BY GIT COMMIT COMMAND. SINCE GIT IS AWARE THAT IT WAS MERGE CONFLICT, IT LINKS THIS CHANGE TO THE CORRECT COMMIT.

35. WHAT COMMAND WILL YOU USE TO DELETE A BRANCH?
AFTER THE SUCCESSFUL MERGE OF FEATURE BRANCH IN MAIN BRANCH, WE DO NOT NEED THE FEATURE BRANCH.

TO DELETE AN UNWANTED BRANCH WE USE FOLLOWING COMMAND:

GIT BRANCH –D

36. WHAT COMMAND WILL YOU USE TO DELETE A BRANCH THAT HAS UNMERGED CHANGES?
TO FORCIBLY DELETE AN UNWANTED BRANCH WITH UNMERGED CHANGES, WE USE FOLLOWING COMMAND:

GIT BRANCH –D

37. WHAT IS THE ALTERNATIVE COMMAND TO MERGING IN GIT?
ANOTHER ALTERNATIVE OF MERGING IN GIT IS REBASING. IT IS DONE BY GIT REBASE COMMAND

38. WHAT IS REBASING IN GIT?
REBASING IS THE PROCESS OF MOVING A BRANCH TO A NEW BASE COMMIT.

IT IS LIKE REWRITING THE HISTORY OF A BRANCH.
IN REBASING, WE MOVE A BRANCH FROM ONE COMMIT TO ANOTHER. BY THIS WE CAN MAINTAIN LINEAR PROJECT HISTORY.
ONCE THE COMMITS ARE PUSHED TO A PUBLIC REPOSITORY, IT IS NOT A GOOD PRACTICE TO USE REBASING.

39. WHAT IS THE ‘GOLDEN RULE OF REBASING’ IN GIT?
THE GOLDEN RULE OF REBASING IS THAT WE SHOULD NEVER USE GIT REBASE ON PUBLIC BRANCHES. IF OTHER PEOPLE ARE USING THE SAME BRANCH THEN THEY MAY GET CONFUSED BY LOOKING AT THE CHANGES IN MASTER BRANCH AFTER GIT REBASING.
THEREFORE, IT IS NOT RECOMMENDED TO DO REBASING ON A PUBLIC BRANCH THAT IS ALSO USED BY OTHER COLLABORATORS.

40. WHY DO WE USE INTERACTIVE REBASING IN PLACE OF AUTO REBASING?
BY USING INTERACTIVE REBASING WE CAN ALTER THE COMMITS BEFORE MOVING THEM TO A NEW BRANCH.
THIS IS MORE POWERFUL THAN AN AUTOMATED REBASE. IT GIVES US COMPLETE CONTROL OVER THE BRANCH’S COMMIT HISTORY.
GENERALLY, WE USE INTERACTIVE REBASING TO CLEAN UP THE MESSY HISTORY OF COMMITS JUST BEFORE MERGING A FEATURE BRANCH INTO MASTER.

41. WHAT IS THE COMMAND FOR REBASING IN GIT?
GIT COMMAND FOR REBASING IS:

GIT REBASE

42. WHAT IS THE MAIN DIFFERENCE BETWEEN GIT CLONE AND GIT REMOTE?
THE MAIN DIFFERENCE BETWEEN GIT CLONE AND GIT REMOTE IS THAT GIT CLONE IS USED TO CREATE A NEW LOCAL REPOSITORY WHEREAS GIT REMOTE IS USED IN AN EXISTING REPOSITORY. GIT REMOTE ADDS A NEW REFERENCE TO EXISTING REMOTE REPOSITORY FOR TRACKING FURTHER CHANGES.
GIT CLONE CREATES A NEW LOCAL REPOSITORY BY COPYING ANOTHER REPOSITORY FROM A URL.

43. WHAT IS GIT VERSION CONTROL?
GIT VERSION CONTROL HELPS US IN MANAGING THE CHANGES TO SOURCE CODE OVER TIME BY A SOFTWARE TEAM. IT KEEPS TRACK OF ALL THE CHANGES IN A SPECIAL KIND OF DATABASE. IF WE MAKE A MISTAKE, WE CAN GO BACK IN TIME AND SEE PREVIOUS CHANGES TO FIX THE MISTAKE.
GIT VERSION CONTROL HELPS THE TEAM IN COLLABORATING ON DEVELOPING A SOFTWARE AND WORK EFFICIENTLY. EVERY ONE CAN MERGE THE CHANGES WITH CONFIDENCE THAT EVERYTHING IS TRACKED AND REMAINS INTACT IN GIT VERSION CONTROL. ANY BUG INTRODUCED BY A CHANGE CAN BE DISCOVERED AND REVERTED BACK BY GOING BACK TO A WORKING VERSION.

44. WHAT GUI DO YOU USE FOR WORKING ON GIT?
THERE ARE MANY GUI FOR GIT THAT WE CAN USE. SOME OF THESE ARE:

GITHUB DESKTOP
GITX-DEV
GITBOX
GIT-COLA
SOURCETREE
GIT EXTENSIONS
SMARTGIT

GITUP

45. WHAT IS THE USE OF GIT DIFF COMMAND IN GIT?
IN GIT, GIT DIFF COMMAND IS USED TO DISPLAY THE DIFFERENCES BETWEEN 2 VERSIONS, OR BETWEEN WORKING DIRECTORY AND AN INDEX, OR BETWEEN INDEX AND MOST RECENT COMMIT.
IT CAN ALSO DISPLAY CHANGES BETWEEN TWO BLOB OBJECTS, OR BETWEEN TWO FILES ON DISK IN GIT.
IT HELPS IN FINDING THE CHANGES THAT CAN BE USED FOR CODE REVIEW FOR A FEATURE OR BUG FIX.

46. WHAT IS GIT RERERE?
IN GIT, RERERE IS A HIDDEN FEATURE. THE FULL FORM OF RERERE IS “REUSE RECORDED RESOLUTION”.
BY USING RERERE, GIT REMEMBERS HOW WE’VE RESOLVED A HUNK CONFLICT. THE NEXT TIME GIT SEES THE SAME CONFLICT, IT CAN AUTOMATICALLY RESOLVE IT FOR US.

47. WHAT ARE THE THREE MOST POPULAR VERSION OF GIT DIFF COMMAND?
THREE MOST POPULAR GIT DIFF COMMANDS ARE AS FOLLOWS:

GIT DIFF: IT DISPLAYS THE DIFFERENCES BETWEEN WORKING DIRECTORY AND THE INDEX.
GIT DIFF –CACHED: IT DISPLAYS THE DIFFERENCES BETWEEN THE INDEX AND THE MOST RECENT COMMIT.
GIT DIFF HEAD: IT DISPLAYS THE DIFFERENCES BETWEEN WORKING DIRECTORY AND THE MOST RECENT COMMIT

48. WHAT IS THE USE OF GIT STATUS COMMAND?
IN GIT, GIT STATUS COMMAND MAINLY SHOWS THE STATUS OF WORKING TREE.

IT SHOWS FOLLOWING ITEMS:

THE PATHS THAT HAVE DIFFERENCES BETWEEN THE INDEX FILE AND THE CURRENT HEAD COMMIT.
THE PATHS THAT HAVE DIFFERENCES BETWEEN THE WORKING TREE AND THE INDEX FILE
THE PATHS IN THE WORKING TREE THAT ARE NOT TRACKED BY GIT.
AMONG THE ABOVE THREE ITEMS, FIRST ITEM IS THE ONE THAT WE COMMIT BY USING GIT COMMIT COMMAND. ITEM TWO AND THREE CAN BE COMMITTED ONLY AFTER RUNNING GIT ADD COMMAND.

49. WHAT IS THE MAIN DIFFERENCE BETWEEN GIT DIFF AND GIT STATUS?
IN GIT, GIT DIFF SHOWS THE DIFFERENCES BETWEEN DIFFERENT COMMITS OR BETWEEN THE WORKING DIRECTORY AND INDEX.
WHEREAS, GIT STATUS COMMAND JUST SHOWS THE CURRENT STATUS OF WORKING TREE.

50. WHAT IS THE USE OF GIT RM COMMAND IN GIT?
IN GIT, GIT RM COMMAND IS USED FOR REMOVING A FILE FROM THE WORKING TREE AND THE INDEX.
WE USE GIT RM –R TO RECURSIVELY REMOVE ALL FILES FROM A LEADING DIRECTORY.

51. WHAT IS THE COMMAND TO APPLY A STASH?
SOMETIMES WE WANT TO SAVE OUR UNFINISHED WORK. FOR THIS PURPOSE WE USE GIT STASH COMMAND. ONCE WE WANT TO COME BACK AND CONTINUE WORKING FROM THE LAST PLACE WHERE WE LEFT, WE USE GIT STASH APPLY COMMAND TO BRING BACK THE UNFINISHED WORK.

SO THE COMMAND TO APPLY A STASH IS:

GIT STASH APPLY
OR WE CAN USE

GIT STASH APPLY

52. WHY DO WE USE GIT LOG COMMAND?
WE USE GIT LOG COMMAND TO SEARCH FOR SPECIFIC COMMITS IN PROJECT HISTORY.
WE CAN SEARCH GIT HISTORY BY AUTHOR, DATE OR CONTENT. IT CAN EVEN LIST THE COMMITS THAT WERE DONE X DAYS BEFORE OR AFTER A SPECIFIC DATE.

53. WHY DO WE NEED GIT ADD COMMAND IN GIT?
GIT GIVES US A VERY GOOD FEATURE OF STAGING OUR CHANGES BEFORE COMMIT. TO STAGE THE CHANGES WE USE GIT ADD COMMAND. THIS ADDS OUR CHANGES FROM WORKING DIRECTORY TO THE INDEX.
WHEN WE ARE WORKING ON MULTIPLE TASKS AND WE WANT TO JUST COMMIT THE FINISHED TASKS, WE FIRST ADD FINISHED CHANGES TO STAGING AREA AND THEN COMMIT IT. AT THIS TIME GIT ADD COMMAND IS VERY HELPFUL.

54. WHY DO WE USE GIT RESET COMMAND?
WE USE GIT RESET COMMAND TO RESET CURRENT HEAD TO A SPECIFIC STATE.
BY DEFAULT IT REVERSES THE ACTION OF GIT ADD COMMAND.
SO WE USE GIT RESET COMMAND TO UNDO THE CHANGES OF GIT ADD COMMAND.

55. WHAT DOES A COMMIT OBJECT CONTAIN?
WHENEVER WE DO A COMMIT IN GIT BY USING GIT COMMIT COMMAND, GIT CREATES A NEW COMMIT OBJECT. THIS COMMIT OBJECTS IS SAVED TO GIT REPOSITORY.

THE COMMIT OBJECT CONTAINS FOLLOWING INFORMATION:

HASH: THE SHA1 HASH OF THE GIT TREE THAT REFERS TO THE STATE OF INDEX AT COMMIT TIME.
COMMIT AUTHOR: THE NAME OF PERSON/PROCESS DOING THE COMMIT AND DATE/TIME.
COMMENT: SOME TEXT MESSAGES THAT CONTAINS THE REASON FOR THE COMMIT .

56. HOW CAN WE CONVERT GIT LOG MESSAGES TO A DIFFERENT FORMAT?
WE CAN USE PRETTY OPTION IN GIT LOG COMMAND FOR THIS.

GIT LOG – PRETTY
THIS OPTION CONVERTS THE OUTPUT FORMAT FROM DEFAULT TO OTHER FORMATS.

THERE ARE PRE-BUILT FORMATS AVAILABLE FOR OUR USE.

GIT LOG –PRETTY=ONELINE
FOR EXAMPLE:

GIT LOG --PRETTY=FORMAT:"%H - %AN, %AR : %S"
BA72A6C - DAVE ADAMS, 3 YEARS AGO : CHANGED THE VERSION NUMBER

57. WHAT ARE THE PROGRAMMING LANGUAGES IN WHICH GIT HOOKS CAN BE WRITTEN?
GIT HOOKS ARE GENERALLY WRITTEN IN SHELL AND PERL SCRIPTS. BUT THESE CAN BE WRITTEN IN ANY OTHER LANGUAGE AS LONG AS IT HAS AN EXECUTABLE.
GIT HOOKS CAN ALSO BE WRITTEN IN PYTHON SCRIPT.

58. WHAT IS A COMMIT MESSAGE IN GIT?
A COMMIT MESSAGE IS A COMMENT THAT WE ADD TO A COMMIT. WE CAN PROVIDE MEANINGFUL INFORMATION ABOUT THE REASON FOR COMMIT BY USING A COMMIT MESSAGE.
IN MOST OF THE ORGANIZATIONS, IT IS MANDATORY TO PUT A COMMIT MESSAGE ALONG WITH EACH COMMIT.
OFTEN, COMMIT MESSAGES CONTAIN JIRA TICKET, BUG ID, DEFECT ID ETC. FOR A PROJECT.

59. HOW GIT PROTECTS THE CODE IN A REPOSITORY?
GIT IS MADE VERY SECURE SINCE IT CONTAINS THE SOURCE CODE OF AN ORGANIZATION. ALL THE OBJECTS IN A GIT REPOSITORY ARE ENCRYPTED WITH A HASHING ALGORITHM CALLED SHA1. THIS ALGORITHM IS QUITE STRONG AND FAST. IT PROTECTS SOURCE CODE AND OTHER CONTENTS OF REPOSITORY AGAINST THE POSSIBLE MALICIOUS ATTACKS.
THIS ALGORITHM ALSO MAINTAINS THE INTEGRITY OF GIT REPOSITORY BY PROTECTING THE CHANGE HISTORY AGAINST ACCIDENTAL CHANGES.

60. HOW GIT PROVIDES FLEXIBILITY IN VERSION CONTROL?
GIT IS VERY FLEXIBLE VERSION CONTROL SYSTEM. IT SUPPORTS NON-LINEAR DEVELOPMENT WORKFLOWS. IT SUPPORTS FLOWS THAT ARE COMPATIBLE WITH EXTERNAL PROTOCOLS AND EXISTING SYSTEMS.
GIT ALSO SUPPORTS BOTH BRANCHING AND TAGGING THAT PROMOTES MULTIPLE KINDS OF WORKFLOWS IN VERSION CONTROL.

61. HOW CAN WE CHANGE A COMMIT MESSAGE IN GIT?
IF A COMMIT HAS NOT BEEN PUSHED TO GITHUB, WE CAN USE GIT COMMIT --AMMEND COMMAND TO CHANGE THE COMMIT MESSAGE.
WHEN WE PUSH THE COMMIT, A NEW MESSAGE APPEARS ON GITHUB.

62. WHY IS IT ADVISABLE TO CREATE AN ADDITIONAL COMMIT INSTEAD OF AMENDING AN EXISTING COMMIT?
GIT AMEND INTERNALLY CREATES A NEW COMMIT AND REPLACES THE OLD COMMIT. IF COMMITS HAVE ALREADY BEEN PUSHED TO CENTRAL REPOSITORY, IT SHOULD NOT BE USED TO MODIFY THE PREVIOUS COMMITS.
IT SHOULD BE GENERALLY USED FOR ONLY AMENDING THE GIT COMMENT.

63. WHAT IS A BARE REPOSITORY IN GIT?
A REPOSITORY CREATED WITH GIT INIT –BARE COMMAND IS A BARE REPOSITORY IN GIT.
THE BARE REPOSITORY DOES NOT CONTAIN ANY WORKING OR CHECKED OUT COPY OF SOURCE FILES. A BARE REPOSITORY STORES GIT REVISION HISTORY IN THE ROOT FOLDER OF REPOSITORY INSTEAD OF IN A .GIT SUBFOLDER.
IT IS MAINLY USED FOR SHARING AND COLLABORATING WITH OTHER DEVELOPERS.
WE CAN CREATE A BARE REPOSITORY IN WHICH ALL DEVELOPERS CAN PUSH THEIR CODE.
THERE IS NO WORKING TREE IN BARE REPOSITORY, SINCE NO ONE DIRECTLY EDITS FILES IN A BARE REPOSITORY.

64. HOW DO WE PUT A LOCAL REPOSITORY ON GITHUB SERVER?
TO PUT A LOCAL REPOSITORY ON GITHUB, WE FIRST ADD ALL THE FILES OF WORKING DIRECTORY INTO LOCAL REPOSITORY AND COMMIT THE CHANGES.
AFTER THAT WE CALL GIT REMOTE ADD COMMAND TO ADD THE LOCAL REPOSITORY ON GITHUB SERVER.
ONCE IT IS ADDED, WE USE GIT PUSH COMMAND TO PUSH THE CONTENTS OF LOCAL REPOSITORY TO REMOTE GITHUB SERVER.

65. HOW WILL YOU DELETE A BRANCH IN GIT?
WE USE GIT BRANCH –D COMMAND TO DELETE A BRANCH IN GIT.
IN CASE A LOCAL BRANCH IS NOT FULLY MERGED, BUT WE WANT TO DELETE IT BY FORCE, THEN WE USE GIT BRANCH –D COMMAND.

66. HOW CAN WE SET UP A GIT REPOSITORY TO RUN CODE SANITY CHECKS AND UAT TESTS JUST BEFORE A COMMIT?
WE CAN USE GIT HOOKS FOR THIS KIND OF PURPOSE. WE CAN WRITE THE CODE SANITY CHECKS IN SCRIPT. THIS SCRIPT CAN BE CALLED BY PRE-COMMIT HOOK OF THE REPOSITORY.
IF THIS HOOK PASSES, THEN ONLY COMMIT WILL BE SUCCESSFUL.

67. HOW CAN WE REVERT A COMMIT THAT WAS PUSHED EARLIER AND IS PUBLIC NOW?
WE CAN USE GIT REVERT COMMAND FOR THIS PURPOSE.
INTERNALLY, GIT REVERT COMMAND CREATES A NEW COMMIT WITH PATCHES THAT REVERSE THE CHANGES DONE IN PREVIOUS COMMITS.
THE OTHER OPTION IS TO CHECKOUT A PREVIOUS COMMIT VERSION AND THEN COMMIT IT AS A NEW COMMIT.

68. IN GIT, HOW WILL YOU COMPRESS LAST N COMMITS INTO A SINGLE COMMIT?
TOM COMPRESS LAST N COMMITS A SINGLE COMMIT, WE USE GIT REBASE COMMAND. THIS COMMAND COMPRESSES MULTIPLE COMMITS AND CREATES A NEW COMMIT. IT OVERWRITES THE HISTORY OF COMMITS.
IT SHOULD BE DONE CAREFULLY, SINCE IT CAN LEAD TO UNEXPECTED RESULTS.

69. HOW WILL YOU SWITCH FROM ONE BRANCH TO A NEW BRANCH IN GIT?
IN GIT, WE CAN USE GIT CHECKOUT COMMAND TO SWITCH TO A NEW BRANCH.

70. HOW CAN WE CLEAN UNWANTED FILES FROM OUR WORKING DIRECTORY IN GIT?
GIT PROVIDES GIT CLEAN COMMAND TO RECURSIVELY CLEAN THE WORKING TREE. IT REMOVES THE FILES THAT ARE NOT UNDER VERSION CONTROL IN GIT.
IF WE USE GIT CLEAN –X, THEN IGNORED FILES ARE ALSO REMOVED.

71. WHAT IS THE PURPOSE OF GIT TAG COMMAND?
WE USE GIT TAG COMMAND TO ADD, DELETE, LIST OR VERIFY A TAG OBJECT IN GIT.
TAG OBJECTS CREATED WITH OPTIONS –A, -S, -U ARE ALSO KNOWN AS ANNOTATED TAGS.
ANNOTATED TAGS ARE GENERALLY USED FOR RELEASE.

72. WHAT IS CHERRY-PICK IN GIT?
A GIT CHERRY-PICK IS A VERY USEFUL FEATURE IN GIT. BY USING THIS COMMAND WE CAN SELECTIVELY APPLY THE CHANGES DONE BY EXISTING COMMITS.
IN CASE WE WANT TO SELECTIVELY RELEASE A FEATURE, WE CAN REMOVE THE UNWANTED FILES AND APPLY ONLY SELECTED COMMITS.

73. WHAT IS SHORTLOG IN GIT?
A SHORTLOG IN GIT IS A COMMAND THAT SUMMARIZES THE GIT LOG OUTPUT.
THE OUTPUT OF GIT SHORTLOG IS IN A FORMAT SUITABLE FOR RELEASE ANNOUNCEMENTS.

74. HOW CAN YOU FIND THE NAMES OF FILES THAT WERE CHANGED IN A SPECIFIC COMMIT?

EVERY COMMIT IN GIT HAS A HASH CODE. THIS HASH CODE UNIQUELY REPRESENTS THE GIT COMMIT OBJECT.
WE CAN USE GIT DIFF-TREE COMMAND TO LIST THE NAME OF FILES THAT WERE CHANGED IN A COMMIT.

THE COMMAND WILL BE AS FOLLOWS:

GIT DIFF-TREE -R
BY USING -R FLAG, WE JUST GET THE LIST OF INDIVIDUAL FILES.

75. HOW CAN WE ATTACH AN AUTOMATED SCRIPT TO RUN ON THE EVENT OF A NEW COMMIT BY PUSH COMMAND?
IN GIT WE CAN USE A HOOK TO RUN AN AUTOMATED SCRIPT ON A SPECIFIC EVENT. WE CAN CHOOSE BETWEEN PRE-RECEIVE, UPDATE OR POST-RECEIVE HOOK AND ATTACH OUR SCRIPT ON ANY OF THESE HOOKS.
GIT WILL AUTOMATICALLY RUN THE SCRIPT ON THE EVENT OF ANY OF THESE HOOKS.

76. WHAT IS THE DIFFERENCE BETWEEN PRE-RECEIVE, UPDATE AND POST-RECEIVE HOOKS IN GIT?
PRE-RECEIVE HOOK IS INVOKED WHEN A COMMIT IS PUSHED TO A DESTINATION REPOSITORY. ANY SCRIPT ATTACHED TO THIS HOOK IS EXECUTED BEFORE UPDATING ANY REFERENCE. THIS IS MAINLY USED TO ENFORCE DEVELOPMENT BEST PRACTICES AND POLICIES.
UPDATE HOOK IS SIMILAR TO PRE-RECEIVE HOOK. IT IS TRIGGERED JUST BEFORE ANY UPDATES ARE DONE. THIS HOOK IS INVOKED ONCE FOR EVERY COMMIT THAT IS PUSHED TO A DESTINATION REPOSITORY.
POST-RECEIVE HOOK IS INVOKED AFTER THE UPDATES HAVE BEEN DONE AND ACCEPTED BY A DESTINATION REPOSITORY. THIS IS MAINLY USED TO CONFIGURE DEPLOYMENT SCRIPTS. IT CAN ALSO INVOKE CONTINUOUS
INTEGRATION (CI) SYSTEMS AND SEND NOTIFICATION EMAILS TO RELEVANT PARTIES OF A REPOSITORY.

77. DO WE HAVE TO STORE SCRIPTS FOR GIT HOOKS WITHIN SAME REPOSITORY?
A HOOK IS LOCAL TO A GIT REPOSITORY. BUT THE SCRIPT ATTACHED TO A HOOK CAN BE CREATED EITHER INSIDE THE HOOKS DIRECTORY OR IT CAN BE STORED IN A SEPARATE REPOSITORY. BUT WE HAVE TO LINK THE SCRIPT TO A HOOK IN OUR LOCAL REPOSITORY.
IN THIS WAY WE CAN MAINTAIN VERSIONS OF A SCRIPT IN A SEPARATE REPOSITORY, BUT USE THEM IN OUR REPOSITORY WHERE HOOKS ARE STORED.
ALSO WHEN WE STORE SCRIPTS IN A SEPARATE COMMON REPOSITORY, WE CAN REUSE SAME SCRIPTS FOR DIFFERENT PURPOSES IN MULTIPLE REPOSITORIES.

78. HOW CAN WE DETERMINE THE COMMIT THAT IS THE SOURCE OF A BUG IN GIT?
IN GIT WE CAN USE GIT BISECT COMMAND TO FIND THE COMMIT THAT HAS INTRODUCED A BUG IN THE SYSTEM.
GIT BISECT COMMAND INTERNALLY USES BINARY SEARCH ALGORITHM TO FIND THE COMMIT THAT INTRODUCED A BUG.
WE FIRST TELL A BAD COMMIT THAT CONTAINS THE BUG AND A GOOD COMMIT THAT WAS PRESENT BEFORE THE BUG WAS INTRODUCED.
THEN GIT BISECT PICKS A COMMIT BETWEEN THOSE TWO ENDPOINTS AND ASKS US WHETHER THE SELECTED COMMIT IS GOOD OR BAD.
IT CONTINUES TO NARROW DOWN THE RANGE UNTIL IT DISCOVERS THE EXACT COMMIT RESPONSIBLE FOR INTRODUCING THE BUG.

79. HOW CAN WE SEE DIFFERENCES BETWEEN TWO COMMITS IN GIT?

WE CAN USE GIT DIFF COMMAND TO SEE THE DIFFERENCES BETWEEN TWO COMMITS. THE SYNTAX FOR A SIMPLE GIT DIFF COMMAND TO COMPARE TWO COMMITS IS:

GIT DIFF <COMMIT#1> <COMMIT#2>

80. WHAT ARE THE DIFFERENT WAYS TO IDENTIFY A COMMIT IN GIT?
EACH COMMIT OBJECT IN GIT HAS A UNIQUE HASH. THIS HASH IS A 40 CHARACTERS CHECKSUM HASH. IT IS BASED ON SHA1 HASHING ALGORITHM.
WE CAN USE A HASH TO UNIQUELY IDENTIFY A GIT COMMIT.
GIT ALSO PROVIDES SUPPORT FOR CREATING AN ALIAS FOR A COMMIT. THIS ALIAS IS KNOWN AS REFS. EVERY TAG IN GIT IS A REF. THESE REFS CAN ALSO BE USED TO IDENTIFY A COMMIT. SOME OF THE SPECIAL TAGS IN GIT ARE HEAD, FETCH_HEAD AND MERGE_HEAD.

81. WHEN WE RUN GIT BRANCH , HOW DOES GIT KNOW THE SHA-1 OF THE LAST COMMIT?
GIT USES THE REFERENCE NAMED HEAD FOR THIS PURPOSE. THE HEAD FILE IN GIT IS A SYMBOLIC REFERENCE TO THE CURRENT BRANCH WE ARE WORKING ON.
A SYMBOLIC REFERENCE IS NOT A NORMAL REFERENCE THAT CONTAINS A SHA-1 VALUE. A SYMBOLIC REFERENCE CONTAINS A POINTER TO ANOTHER REFERENCE.

WHEN WE OPEN HEAD FILE WE SEE:

$ CAT .GIT/HEAD
REF: REFS/HEADS/MASTER
IF WE RUN GIT CHECKOUT BRANCHA, GIT UPDATES THE FILE TO LOOK LIKE THIS:

$ CAT .GIT/HEAD
REF: REFS/HEADS/BRANCHA

82. WHAT ARE THE DIFFERENT TYPES OF TAGS YOU CAN CREATE IN GIT?
IN GIT, WE CAN CREATE TWO TYPES OF TAGS.

LIGHTWEIGHT TAG: A LIGHTWEIGHT TAG IS A REFERENCE THAT NEVER MOVES. WE CAN MAKE A LIGHTWEIGHT TAG BY RUNNING A COMMAND SIMILAR TO FOLLOWING:

$ GIT UPDATE-REF REFS/TAGS/V1.0
DAD0DAB538C970E37EA1E769CBBDE608743BC96D
ANNOTATED TAG: AN ANNOTATED TAG IS MORE COMPLEX OBJECT IN GIT. WHEN WE CREATE AN ANNOTATED TAG, GIT CREATES A TAG OBJECT AND WRITES A REFERENCE TO POINT TO IT RATHER THAN DIRECTLY TO THE COMMIT.
WE CAN CREATE AN ANNOTATED TAG AS FOLLOWS:

$ GIT TAG -A V1.1 1D410EABC13591CB07496601EBC7C059DD55BFE9 -M 'TEST TAG'

83. HOW CAN WE RENAME A REMOTE REPOSITORY?
WE CAN USE COMMAND GIT REMOTE RENAME FOR CHANGING THE NAME OF A REMOTE REPOSITORY. THIS CHANGES THE SHORT NAME ASSOCIATED WITH A REMOTE REPOSITORY IN YOUR LOCAL. COMMAND WOULD LOOK AS FOLLOWS:

GIT REMOTE RENAME REPOOLDNAME REPONEWNAME

84. SOME PEOPLE USE GIT CHECKOUT AND SOME USE GIT CO FOR CHECKOUT. HOW IS THAT POSSIBLE?
WE CAN CREATE ALIASES IN GIT FOR COMMANDS BY MODIFYING THE GIT CONFIGURATION.

IN CASE OF CALLING GIT CO INSTEAD OF GIT CHECKOUT WE CAN RUN FOLLOWING COMMAND:

GIT CONFIG --GLOBAL ALIAS.CO CHECKOUT
SO THE PEOPLE USING GIT CO HAVE MADE THE ALIAS FOR GIT CHECKOUT IN THEIR OWN ENVIRONMENT.

85. HOW CAN WE SEE THE LAST COMMIT ON EACH OF OUR BRANCH IN GIT?
WHEN WE RUN GIT BRANCH COMMAND, IT LISTS ALL THE BRANCHES IN OUR LOCAL REPOSITORY.
TO SEE THE LATEST COMMIT ASSOCIATED WITH EACH BRANCH, WE USE OPTION –V.

EXACT COMMAND FOR THIS IS AS FOLLOWS:

GIT BRANCH –V

IT LISTS BRANCHES AS:

ISSUE75 83B576C FIX ISSUE
* MASTER 7B96605 MERGE BRANCH 'ISSUE75'
TESTING 972AC34 ADD DAVE TO THE DEVELOPER LIST

86. IS ORIGIN A SPECIAL BRANCH IN GIT?
NO, ORIGIN IS NOT A SPECIAL BRANCH IN GIT.
BRANCH ORIGIN IS SIMILAR TO BRANCH MASTER. IT DOES NOT HAVE ANY SPECIAL MEANING IN GIT.
MASTER IS THE DEFAULT NAME FOR A STARTING BRANCH WHEN WE RUN GIT INIT COMMAND.
ORIGIN IS THE DEFAULT NAME FOR A REMOTE WHEN WE RUN GIT CLONE COMMAND.
IF WE RUN GIT CLONE -O MYORIGIN INSTEAD, THEN WE WILL HAVE MYORIGIN/MASTER AS OUR DEFAULT REMOTE BRANCH.

87. HOW CAN WE CONFIGURE GIT TO NOT ASK FOR PASSWORD EVERY TIME?
WHEN WE USE HTTPS URL TO PUSH, THE GIT SERVER ASKS FOR USERNAME AND PASSWORD FOR AUTHENTICATION. IT PROMPTS US ON THE TERMINAL FOR THIS INFORMATION.
IF WE DON’T WANT TO TYPE USERNAME/PASSWORD WITH EVERY SINGLE TIME PUSH, WE CAN SET UP A “CREDENTIAL CACHE”.
IT IS KEPT IN MEMORY FOR A FEW MINUTES. WE CAN SET IT BY RUNNING:
GIT CONFIG --GLOBAL CREDENTIAL.HELPER CACHE

88. WHAT ARE THE FOUR MAJOR PROTOCOLS USED BY GIT FOR DATA TRANSFER?
GIT USES FOLLOWING MAJOR PROTOCOLS FOR DATA TRANSFER:

LOCAL
HTTP
SECURE SHELL (SSH)
GIT

89. WHAT IS GIT PROTOCOL?
GIT PROTOCOL IS A MECHANISM FOR TRANSFERRING DATA IN GIT. IT IS A SPECIAL DAEMON. IT COMES PRE-PACKAGED WITH GIT. IT LISTENS ON A DEDICATED PORT 9418. IT PROVIDES SERVICES SIMILAR TO SSH PROTOCOL.
BUT GIT PROTOCOL DOES NOT SUPPORT ANY AUTHENTICATION.
SO ON PLUS SIDE, THIS IS A VERY FAST NETWORK TRANSFER PROTOCOL. BUT IT LACKS AUTHENTICATION.

90. HOW CAN WE WORK ON A PROJECT WHERE WE DO NOT HAVE PUSH ACCESS?
IN CASE OF PROJECTS WHERE WE DO NOT HAVE PUSH ACCESS, WE CAN JUST FORK THE REPOSITORY. BY RUNNING GIT FORK COMMAND, GIT WILL CREATE A PERSONAL COPY OF THE REPOSITORY IN OUR NAMESPACE. ONCE OUR WORK IS DONE, WE CAN CREATE A PULL REQUEST TO MERGE OUR CHANGES ON THE REAL PROJECT.

91. WHAT IS GIT GREP?
GIT IS SHIPPED ALONG WITH A GREP COMMAND THAT ALLOWS US TO SEARCH FOR A STRING OR REGULAR EXPRESSION IN ANY COMMITTED TREE OR THE WORKING DIRECTORY.
BY DEFAULT, IT WORKS ON THE FILES IN YOUR CURRENT WORKING DIRECTORY.

92. HOW CAN YOUR REORDER COMMITS IN GIT?
WE CAN USE GIT REBASE COMMAND TO REORDER COMMITS IN GIT. IT CAN WORK INTERACTIVELY AND YOU CAN ALSO SELECT THE ORDERING OF COMMITS.

93. HOW WILL YOU SPLIT A COMMIT INTO MULTIPLE COMMITS?
TO SPLIT A COMMIT, WE HAVE TO USE GIT REBASE COMMAND IN INTERACTIVE MODE. ONCE WE REACH THE COMMIT THAT NEEDS TO BE SPLIT, WE RESET THAT COMMIT AND TAKE THE CHANGES THAT HAVE BEEN RESET. NOW WE CAN CREATE MULTIPLE COMMITS OUT OF THAT.

94. WHAT IS FILTER-BRANCH IN GIT?
IN GIT, FILTER-BRANCH IS ANOTHER OPTION TO REWRITE HISTORY. IT CAN SCRUB THE ENTIRE HISTORY. WHEN WE HAVE LARGE NUMBER OF COMMITS, WE CAN USE THIS TOOL.
IT GIVES MANY OPTIONS LIKE REMOVING THE COMMIT RELATED CHANGES TO A SPECIFIC FILE FROM HISTORY.
YOU CAN EVEN SET YOU NAME AND EMAIL IN THE COMMIT HISTORY BY USING FILTER-BRANCH.

95. WHAT ARE THE THREE MAIN TREES MAINTAINED BY GIT?
GIT MAINTAINS FOLLOWING THREE TREES:

HEAD: THIS IS THE LAST COMMIT SNAPSHOT.
INDEX: THIS IS THE PROPOSED NEXT COMMIT SNAPSHOT.
WORKING DIRECTORY: THIS IS THE SANDBOX FOR DOING CHANGES.

96. WHAT ARE THE THREE MAIN STEPS OF WORKING GIT?
GIT HAS FOLLOWING THREE MAIN STEPS IN A SIMPLE WORKFLOW:

CHECKOUT THE PROJECT FROM HEAD TO WORKING DIRECTORY.
STAGE THE FILES FROM WORKING DIRECTORY TO INDEX.
COMMIT THE CHANGES FROM INDEX TO HEAD.

97. WHAT ARE OURS AND THEIRS MERGE OPTIONS IN GIT?
IN GIT, WE GET TWO SIMPLE OPTIONS FOR RESOLVING MERGE CONFLICTS: OURS AND THEIRS THESE OPTIONS TELL THE GIT WHICH SIDE TO FAVOR IN MERGE CONFLICTS.
IN OURS, WE RUN A COMMAND LIKE GIT MERGE -XOURS BRANCHA
AS THE NAME SUGGESTS, IN OURS, THE CHANGES IN OUR BRANCH ARE FAVORED OVER THE OTHER BRANCH DURING A MERGE CONFLICT.

98. HOW CAN WE IGNORE MERGE CONFLICTS DUE TO WHITESPACE?
GIT PROVIDES AN OPTION IGNORE-SPACE-CHANGE IN GIT MERGE COMMAND TO IGNORE THE CONFLICTS RELATED TO WHITESPACES.

THE COMMAND TO DO SO IS AS FOLLOWS:

GIT MERGE -XIGNORE-SPACE-CHANGE WHITESPACE

99. WHAT IS GIT BLAME?
IN GIT, GIT BLAME IS A VERY GOOD OPTION TO FIND THE PERSON WHO CHANGED A SPECIFIC LINE. WHEN WE CALL GIT BLAME ON A FILE, IT DISPLAYS THE COMMIT AND NAME OF A PERSON RESPONSIBLE FOR MAKING CHANGE IN THAT LINE.

FOLLOWING IS A SAMPLE:

```
$ GIT BLAME -L 12,19 HELLOWORLD.JAVA
^1822FE2 (DAVE ADAMS 2016-03-15 10:31:28 -0700 12) PUBLIC CLASS HELLOWORLD {
^1822FE2 (DAVE ADAMS 2016-03-15 10:31:28 -0700 13)
^1822FE2 (DAVE ADAMS 2016-03-15 10:31:28 -0700 14) PUBLIC STATIC VOID MAIN(STRING[]
ARGS) {
AF6560E4 (DAVE ADAMS 2016-03-17 21:52:20 -0700 16) // PRINTS "HELLO, WORLD" TO THE
TERMINAL WINDOW.
A9EAF55D (DAVE ADAMS 2016-04-06 10:15:08 -0700 17) SYSTEM.OUT.PRINTLN("HELLO,
WORLD");
AF6560E4 (DAVE ADAMS 2016-03-17 21:52:20 -0700 18) }
AF6560E4 (DAVE ADAMS 2016-03-17 21:52:20 -0700 19) }
```

100. WHAT IS A SUBMODULE IN GIT?
IN GIT, WE CAN CREATE SUB MODULES INSIDE A REPOSITORY BY USING GIT SUBMODULE COMMAND.
BY USING SUBMODULE COMMAND, WE CAN KEEP A GIT REPOSITORY AS A SUBDIRECTORY OF ANOTHER GIT REPOSITORY.
IT ALLOWS US TO KEEP OUR COMMITS TO SUBMODULE SEPARATE FROM THE COMMITS TO MAIN GIT REPOSITORY.