CS 251 Outlab 2: Git and LaTeX

This Outlab is worth a total of 100 points. `git2.tar.gz` has resources for the second task, and `latex-resources.tar.gz` has resources for the LaTeX task.

Feel free to ask for clarifications on Piazza. There are no stupid questions :)
All updates from our discussions will be added in this colour.

Git (50 points)

[Regarding Q1] Please note that you don't need to register for any account on github or gitlab. Open https://git.cse.iitb.ac.in/ and use your cse username and password. Create a new project and start working.

Q1. Playing with Git [30 points]

(clone, add, commit, pull, push, tag, rebase)

Git offers various features to control and make easy collaboration across users, provided you know what features could be used for what kind of collaboration.

It is usual practice to have the implementation of basic APIs and code with final updates on the master branch, while having implementations on separate branches, evaluated, finalised and only then merged into the master branch. We shall attempt doing this for a basic searching library built in C++.

Note:
The person with the lexicographically smaller roll number is referred to as the *first user*. The other one as the *second user* .

Task1: **Basic API [7 points]**

1. Create a new private repository on git.cse.iitb.ac.in from the account of the *first user* Name the repository **<rollno1>-<rollno2>-git** If the name is not available, use random extra digits at the end (eg 180050001-180050002-git123)
2. Add the *second user* as a collaborator.

The following actions are to be performed by the first user.

1. Clone the repository.
2. Create a file **searching.h** that declares the following function (do not implement here):

bool search_custom(std::vector<int> , int);

The function takes a *vector<int> and int* as input and returns *true* if *int* is present in *vector<int>* else returns *false*

3. Create a file **main.cpp** that includes this header, reads in *n* elements and *num* from *stdin*, initialises a *vector<int>*, calls the function *search_custom* on the vector to search for *num* and prints out *"Found"* if *num* is present in vector, else prints out *"Not Found"* onto *stdout*.

Usage:
$ ./a.out < inp > out
inp:
```
5
1 2 6 7 10
2
```
out:
```
Found
```

4. Make a commit with the message "add API and data read".
5. Create a file **searching.cpp** with the implementation of the function declared in the header using a simple sequential search.
6. Compile and test your code. Add the executable file to .gitignore (read about it!)
7. Make a commit with the message "add basic implementation".
8. Now push your changes to the remote repository onto the master branch.

Task2: Branching [5 points]
The second user now wishes to improve upon the implementation of the *search_custom* function. He/she wants to work simultaneously without cluttering the first user's work with commits. So, he/she works on a different branch.

Now these next steps are to be performed by the second user in his/her **local** clone.

1. Clone the repository.
2. Create a branch named *binary-search* and checkout to it (check it out?).
3. Change the implementation of *search_custom* from sequential search to binary search. Assume array is sorted. Make a commit with the message "*change implementation to binary-search*".
4. Push your commit(s) to a remote branch *binary-search* (notice that the name of the remote branch could be different from the source branch).

Task3: Updating the API [3 points]
You are informed that the input vector will always be in sorted order. Also, there is a need to change the specification of the function to include searching a sub array.

Perform these tasks as the first user.

1. Add a comment in *searching.cpp* mentioning the new constraint. Make sure you write the comment inside the search_custom function.
   // input guaranteed to be in sorted order

2. Change the function declaration in **searching.h** to the following.

   bool search_custom(std::vector<int>, int num, int startidx, int endidx);

   The function is to take as input a *vector<int>* and returns *true* if *num* is present in the sub array [startidx, endidx) else returns *false Assume 0 <= startidx < endidx <= vector_length*

   Make a commit with the message “add constraint and update API to add subarray search”. ~~Do not make any changes to *searching.cpp*.~~ Push the changes to the remote branch master.

Task 4: Implementing the changes [10 points]

The first user asks the second user to update his/her implementation to reflect the API change.

The second user now has two options:

- Merge the API change from the master branch into the binary-search branch, and then make changes to the function and commit.
- Change the entire branch as though the branch originated, not from the commit with the message “add basic implementation” (as before) but from the latest commit itself (commit with message “add constraint and update API to add subarray search”).

To execute the second option, the second user has to perform a ***rebase***. This applies the changes in the commits originally in the branch to the new “base” and in the process, creates new commits to replace the old ones.

The following actions are to be performed by the second user on his/her local repository.

1. Pull all branches.
2. Checkout to the *binary-search* branch.
3. Perform a rebase to shift the base of the current branch to the latest commit on the master branch.
4. There will be merge conflicts. Resolve them and continue the git rebase.

   Note: In this particular scenario, the API change is not significant, so the utility of rebase may not be obvious. But if there is a subroutine that is being updated on master, the rebase helps in each commit reflecting the updated implementation of the sub-routine, which could be helpful for developers who want to branch out from such a commit.

5. Make the necessary updates to the implementation in searching.cpp to work according to the newer API. Make a commit with the message “update binary-search to reflect API update”.

   Try to push the changes to the remote *binary-search* branch. It fails. Try to understand why. Override using git push --force.

Task 5: Merging [5 points]

Now since the binary search is a more efficient implementation we would want to have that implementation reflected in our master branch.

The following actions are to be performed by the first user in his/her local repository on the master branch.

1. Tag the current implementation with tag “v0” (using git tag). Tagging is generally used to save a particular version for easy access.
2. Now *pull* the changes from the remote branch *binary-search* into the *master* branch.
    a. Make changes in the usage of the function in main.cpp entering the correct arguments to perform the same task. Commit your changes with the message “update usage in main.cpp”.
3. Push the updates.

Q2. Finding the culprit [20 points]

DISCLAIMER: The given repository in this question is in no way indicative of how an ideal repository must look like so please refrain from taking any inspirations from this while working on a collaborative project.

Finding and Restoring (10 points)

In this question you are provided with a compressed git repo **git2.tar.gz**. Extract it. This repo has been messed up by one of its developers under the alias Mr. X by modifying the bytecode of one of the binary files, you need to do some restoration to get the code in a working state. Do not forget to reconfigure the repository to reflect commits under your name (mention the full name or use the git cse username and email id pair) or roll no
Also record ALL the commands that you use since you need to submit that as well
(Use git cse username and email as above).

You’ll find 5 binary files a **main.sh** and some other text files in master branch later is extraneous**.** If everything is good, on executing **main.sh** (by running **bash main.sh** on terminal) it prints out **Everything is fine :)** else it would say **Something is wrong :(.**
All the dependencies are bytecodes so you cannot directly go about correcting the things inside the code, but following the steps below you can make this work.

1. One of the developers committed the code when it was working properly and with commit message “**Everything is working now**” but the since all commits were deleted by Mr. X from git log you need to find this commit in the full log history of the repository
2. Now checkout all files/(or the particular malicious file) from that commit to your current HEAD.
3. Check the output of “**bash main.sh**” now and make a commit **under your name** now with the message “**Everything is fine again**”
4. Now let us find out who Mr. X was, although the last commit he made was under his alias “Mr. X”, he did make one mistake which was to change the bytecode of the binary file and commit the change under his real name. Now see how one can display what was changed in each commit in full log history. Now copy the hash of

that commit where a binary file is modified and use git show/git log now with that hash to find out the culprit.

Note: This approach worked only because the collaborators had added a tar.gz file containing a git repo. If they had pushed the repo itself then .git does not get added to the remote server and therefore the earlier commits would not have been visible.

Removing the commit (10 points)

Make sure to keep the copy of your repo at this point on local machine
Now you want to get rid of the commit made by Mr. X. This can be done by a series of reset and rebase commands but let us do this with the help of interactive rebase using the following steps. (Don't do this using git reset, and if you do we'll always know it the way you found Mr. X ;))

1. The first thing we want to do is some series of false commits, this is because currently every file is added by Mr. X and so if we drop that commit the other files would go away with it. Do a "**rm -r ***" in your git repo (don't worry .git files won't be deleted) and commit this with the message "**deleted all files**".
2. Next checkout all the files from the commit with message "**Everything is fine again**", to the current head and recommit this with message "**Everything is fine again"**
3. Now open up an interactive git rebase session with the last 4 commits. (lookup on **git rebase -i**)
4. There are various operations that you can do with commits here, the one that we are interested in is drop. Drop all commits except for the last "**Everything is fine again**" commit that you made. A merge conflict would come, merge from the "**Everything is fine again**" and recommit with the same message again.
5. Everything should work now with Mr. X's commit deleted from git log.
6. Create a text file "culprit.txt" and mention the name of the culprit in that file. Commit the change with the message "**culprit info added**".

LaTeX (50 points)

You are provided a folder called `latex-resources`. You'll find `starter_pack.pdf` in it, along with a few `.png` images. Your task is to replicate this PDF as well as you can! For the pictures and tables, exact dimensions needn't match, but please try to stay faithful to the original. Replace "Your Team Name" by <rollno1>-<rollno2>

This task will be evaluated manually. A sincere attempt (i.e. you genuinely tried to demonstrate all the features from each section; if despite your best efforts you couldn't, you can add a line in bold explaining why) will earn you 35 points. If your source compiles without error (Overfull hbox warnings do not count), and you have reasonably demonstrated all the features in the original, you earn the full 50 points.

You can collaborate on Overleaf or git for this task. In the end, you will have to submit only the images we provided (don't change their names!) and the `.tex` and `.bib` files: call them `my_starter_pack.tex` and `my_starter_pack.bib`

Some key dimensions are specified in the document itself; for others, your version needn't be precise but try to come close to the original. The font and its size are the default, so you needn't specify anything. (Computer Modern, 10pt). You don't need to match the spacing down to the last line, but it shouldn't be too hard to keep the content on the same page as the original. Please see the list on Page 2. We're grading **manually**, and these are the points and features we're looking out for.

Submission Instructions

For the first question on git, add the users **sauravgarg** and **arpitag** to the repository on git.cse.iitb.ac.in with developer permissions. You don't need to make a separate submission on moodle. **We'll consider the date and time of the last commit for deadline purposes.**

For 2nd question just submit the final git2 folder along with the series of ALL commands that you used in the question in the same text file (culprit.txt) where you'll add Mr. X's name along with the latex work as mentioned below.

You have to turn your LaTeX work in on Moodle. Your submission directory must be called `<rollno1>-<rollno2>-outlab2`
Compress it into a tarball with the command:
`tar -zcvf <rollno1>-<rollno2>-outlab2.tar.gz <rollno1>-<rollno2>-outlab2`

Please maintain the following structure for your submission directory:

```
.
|-- aristotle.png
|-- ease-graph.png
|-- git2
|-- kripke.png
|-- my_starter_pack.bib
`-- my_starter_pack.tex
```

On Moodle, submit once per pair from the account of the student with the lexicographically smaller roll number.

Submission Deadline: Sunday August 30, 23:59. The link will be open for the next 6 hours. However if the submission is H hours late (here we take the ceiling function, even 2 minutes counts as 1 hour late), a penalty of 2^H % on your total marks will be applied. So post 6 hours of submission deadline (5:59 am onwards) your submission will not be considered.

(5:59 or 6? You don't really want to find out. Please be *within* time.)

Please do not consult other teams for exact blocks of LaTeX code. Would be a shame to copy an assignment for a *typesetting* language.