Version Control

We will be using Git, and GitLab, to manage the homework assignments, and we will later use GitHub to manage to final class project.

Version Control and Git

We will discuss key functionality of version control systems, mostly in the context of a particular such system called Git, which is used to maintain a huge number of popular projects. While most version control systems have similar capabilities, their terminology tends to vary.

A version control system typically offers the following functionalities:

1. You can identify a set of changes to your code as a single item, called a **commit**. This allows you to put together related changes into one.
2. Each commit incrementally builds upon some previous commit, creating a tree of commits. This is known as a **repository**. Many websites like GitHub, GitLab and others, allow you to store and manage these repositories online, allowing teams from all over the world to collaborate. They also on occasion offer automatic web-page hosting. In fact the page you ar reading right now is published via a GitHub repository.
3. The system comes built in with tools that allow us to do many things:
   - We can see when each commit was made.
   - We can who did the commit and what message they typed as a summary of the commit.
   - We can see line-by-line how the state of the repository on a particular commit differs from its state at some other commit.
   - We can see for each line of code precisely when it was changed, and by whom and for what reason.
4. A **branch** is a pointer to a particular leaf on this tree that carries a particular meaning to us. In effect, a branch points to a particular view of our codebase.

   We typically advance our code by making a new commit at a leaf, advancing the branch pointer at the same time. We can easily switch from one branch to another, completely changing the contents of our project directory.

   New branches are easy to create and delete, and can be used for many reasons:

   - A master branch typically represents the main working part of our code base. This is where stable additions can be made to our code.
   - An experimental branch can be used to try out a new idea, without destroying our main code base.

- A stable branch can contain the last stable release of our project, while we dedicate another "main" branch to work towards the new version of our application. If an urgent fix is needed, we can switch to this stable branch and apply the fix, then go back to working on the main branch.
- A deploy branch may contain specific setting used for deploying an application. We can **merge** the main branch into the deploy branch when we are ready to deploy our next release.
- A particular subteam may have their own branch where they do their work, unaffected by what other subteams are doing. When they are ready to share their work with the main team they can create a **pull request** to the team lead to merge their changes into the main branch.

5. Repositories are typically stored in a remote server. We create a **clone** of a repository on our computer in order to work on it. This clone contains both **local branches**, that we create to do our work before sharing it with the world, and also **remote branches** that are exact replicas of the branches in the remote repository.

   - We can **fetch** from the remote repository, which updates the remote branches in our computer with any changes that others did to the remote repository. We can then **merge** these changes into our local work, or we can **rebase** our local work so that it is as if it happened *after* all those changes.
   - The combination of fetching and merging/rebasing is often done with a single step, called a **pull**.
   - When we want to share our changes with the remote repository, we do a **push**.

6. Oftentimes a project deviates from its original. In our case, for the homework assignments, each student would be starting from an initial repository that the instructor creates, but then follow completely different paths with their solutions, that they each own and do not want to share with others. These are called forks, and the way to accomplish that is essentially for each student to create a copy of the repository on their own remote repository, that only they and the instructor have access to. They can then work at will with that repository and completely independently of the other students. So a **fork** is a new remote repository which was created as a spin-off of an existing remote repository in order to support independent development of the project in a different direction than the original.

Setting up Git locally

If you have not set up Git on your account before, you may need to do these steps:

1. First we should make sure the system knows your name and email, to use in commits. These would typically be set by running the following:

```
git config --global user.name "yournamehere"
git config --global user.email "youremailhere"
```

Make sure to use the same email that you plan to use for your online git repositories, that we will be creating shortly.

Using GitLab and other online project management systems

GitLab is an online hosting service for git repositories. It also offers project-management functionalities, for example you can organize your project work by creating issues, milestones, assigning tasks to users, labeling issues and so on.

1. The first thing you will need is a GitLab account. Go to https://www.gitlab.com and register a new account, if you do not already have one.
2. Sign in, and find the "functional-programming-assignments" project, which is at this link: https://gitlab.com/skiadas/functional-programming-assignments
3. In that page you should be seeing a link to "Fork" the project. Do so now, to create a fork of the project to your GitLab account. Make sure to set your project's visibility to Private under Settings -> Permissions (bottom of the left toolbar in GitLab).
4. Find the GitKraken application in the Developer section, and start it. Create a GitKraken account if you don't already have one. You can also create a GitHub account instead, and use that to log in.
5. From within GitKraken, open the Preferences menu from the far right. Choose Authentication, then GitLab. Choose "Connect to GitLab". Then log in to GitLab and select "Authorize".
6. Now we have linked GitKraken to GitLab. Exit the Preferences.

Further instructions are part of the second assignment.

Commit Workflow

We describe here the typical steps when making commits. A commits consists of two steps:

- **Staging** the changes that you want to commit.
- Forming a commit out of the staged changes.

1. Make sure all your files are saved, and that they are at a reasonably stable state (so one should be able to compile without errors, for example).

2. Open up GitKraken, and make sure the WIP section is the one highlighted. This section contains the changes that have not been committed yet. You see a quick summary in the line itself, including how many files are brand new (green), how many have changes (yellow) and how many have been deleted (red).

3. On the right side of the window, in the "Unstaged Files" section, you see the files in more detail. You can click on one file to see its contents and what changes there are in the file.

4. You can "stage" changes in three ways:

    - Stage all files at once via the "Stage all changes" button
    - Stage individual files by clicking the "Stage File" button to their right
    - Stage individual lines within a file by looking at the file contents and clicking the plus/minus symbols on the left of the lines.

5. You may have made more changes in your files than what this commit should contain. Pick carefully those changes that go together.

6. Once you have staged all the changes that should go together, enter a "Summary" for the commit message at the bottom right. The summary has a small length, and it should be a short and succint description of what the commit is about. If there is an issue associated with the commit, then the description should include that information and we will see how to do that in the future.

7. Click the "Stage files/changes to commit" button at the bottom.

8. If you left something out of the commit, you can follow the above steps but check the "Amend" checkbox at the bottom right. Then the changes will be added to the previous commit rather than creating a new commit. You should only do this if you have NOT pushed your changes yet.

9. When you are certain of your commits, use the "Push" button at the top to send your changes to the GitLab server.