

Exercice

Créer le fichier index.html avec le contenu suivant:

```
<!DOCTYPE html>
<html>
<body>
<span>Initial version</span>
</body>
</html>
Copy
```

1. Lancer `git init` pour initialiser le dépôt.
2. Lancer `git status -s` pour obtenir le status.
3. Lancer `git add index.html` pour ajouter le fichier (ses modifications) au suivi.
4. Modifier le fichier `index.html` pour remplacer `Initial version` par `Seconde version`.
5. Lancer `git status` (et sa version courte) afin de constater l'état particulier du suivi.
6. Lancer `git commit -m « Initial version »` pour valider les modifications au dépôt.
7. Lancer à nouveau `git status -s`
8. Lancer `git diff` pour voir la différence entre le fichier et le dépôt
9. Lancer `git add index.html` pour ajouter les nouvelles modifications au suivi.
10. Lancer `git status -s`
11. Lancer `git reset` pour annuler le suivi des nouvelles modifications
12. Lancer `git status -s`
13. Lancer `git diff` pour vérifier que l'état du fichier n'a pas changé

Comme vous avez pu le constater, `git add` effectue une sorte de snapshot des modifications, et toutes celles apportées par la suite ne seront prise en compte au prochain commit que si un nouveau `git add` les y rajoute.

Et pour aller plus loin, revertons le fichier `index.html` avec:

```
git checkout HEAD -- index.html
Copy
```

Allons plus loin avec git rebase

Garder un historique propre est essentiel pour travailler en équipe ou même s'y retrouver plus tard.

Il est fréquent d'avoir besoin de changer de branche pour quelques raisons que ce soient (hotfix...) et plutôt que de stasher, il est plus pratique de faire un petit commit de backup (`git commit -n` quand le lintage le bloque 😊).

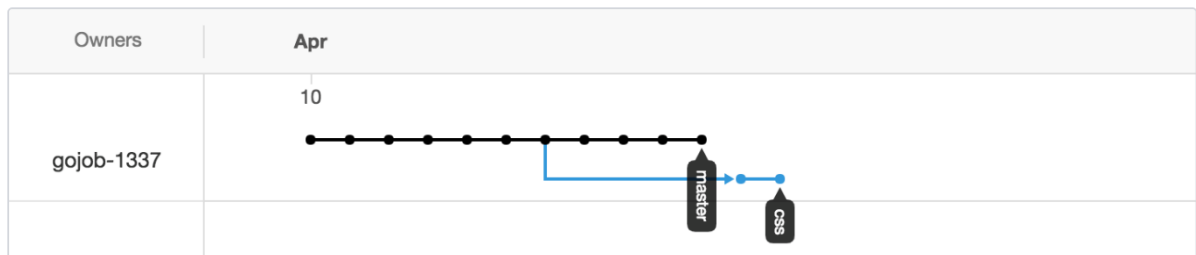
Ce commit n'a pas lieu de rester dans l'historique, car il n'apporte pas le projet à un état de consistance satisfaisant (feature incomplete, bug / todo divers...).

Cloner le dépôt servant de support.

```
git clone https://github.com/gojob-1337/workshop-2018-04-rebase.git
```

Copy

Ce dépôt contient deux branches **master** et **css**



Exercices

Partie 1

Sur la branche **master**, lors du commit libellé **append h3**, le bout de code **class = « »** a été laissé par erreur.

Pour l'exercice, nous allons le supprimer.

1. Récupérer le hash du commit (à savoir **a5c7315**)

```
2. ~/workshop$ git log --oneline
3. 20a65bd (HEAD -> master, origin/master) remove spaces
4. 69ea0a9 remove heading
5. 0e18763 append head
6. 3040084 append h6
7. ba65167 append h5
8. eafcab6 append h4
9. a5c7315 append h3
10. 282cc62 append h2
11. a1a2109 append h1
12. cabcc16 Initial commit
```

Copy

13. Lancer `git rebase -i a5c7315^` (Noter le ^, sinon, il aurait fallu prendre le hash

```
pick efc4790 append h3
pick 27f33cc append h4
pick f62b1ea append h5
pick 785c28e append h6
pick 2c1b1f8 append head
pick 92929a5 append intro
pick 3e48966 remove heading
pick 18c6189 remove spaces

# Rebase 282cc62..18c6189 onto 282cc62 (8 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
```

14. Remplacer le premier `pick` par `edit` ou `e`

```
e efc4790 append h3
pick 27f33cc append h4
pick f62b1ea append h5
pick 785c28e append h6
pick 2c1b1f8 append head
pick 92929a5 append intro
pick 3e48966 remove heading
pick 18c6189 remove spaces
```

Puis enregistrer.

15. Effectuer les modifications nécessaires sur le fichier `index.html`
16. Ajouter les modifications au suivi

```
git add index.html
Copy
```

17. Continuer le rebase

```
git rebase --continue
```

Copy

18. Certains commits ne vont pas poser de problème alors que d'autres seront en conflit car les mêmes lignes y sont modifiées. Corriger les conflits, add pour les valider et continuer le rebase jusqu'à ce qu'il soit terminé.

```
19. ~/workshop$ git rebase --continue
20. Successfully rebased and updated refs/heads/master.
```

Copy

21. Une fois le dernier rebase terminé, regarder l'historique pour constater les changements

Dans cet exercice, nous remontons dans le passé pour modifier un commit, il faut bien comprendre que le processus de rebase va rejouer tous les commits qui suivent celui modifié et qu'en cas de conflit, il peut falloir intervenir sur chaque commit.

Cela peut devenir fastidieux en cas de conflit assez massif. Si vous prévoyez de squasher un certain lot de commit, il sera alors plus judicieux de squasher d'abord puis modifier un ancien commit.

En cas d'erreur, il est possible d'annuler le rebase en court avec :

```
git rebase --abort
```

Copy

Petit plus:

1. Modifier le contenu du span.
2. Lancer

```
3. git add index.html
```

```
git commit --amend
```

Copy

4. Enregistrer et regarder l'historique

Cette commande très pratique permet de modifier le dernier commit simplement en ajoutant le paramètre **--amend** à la commande de commit.

Partie 2

Le but va être de déplacer la branche **css** à la fin de la branche **master**, afin de pouvoir la merger et obtenir un historique clair.

1. Passer sur la branche **css**

```
git checkout css
```

Copy

2. Rebaser sur **master** **css**

```
git rebase -i master
```

Copy

```
pick a5c7315 append h3
pick c5701b7 append h1 style
pick 7058f55 append h2 style
```

Dans ce cas, le commit nommé **append h3** va diverger de celui que l'on a modifié sur **master**, c'est pourquoi il apparaît ici.

3. Comme dans le premier exercice, alterner les résolutions de conflit jusqu'à la fin du rebase

Partie 3

Mergeons la branche **css** sur **master** et supprimons la.

```
git checkout master
git merge css
git branch -d css
```

Copy

Comme certains commits de la branche **css** ne sont pas mergés sur **master** (du fait des rebases, squash...), il faut passer par le **-D** majuscule.

```
git branch -D css
```

Copy

Regarder alors l'historique.

```
~/workshop$ git log --oneline
becce91 (HEAD -> master) append h2 style
a54c2f6 append h1 style
082e020 remove spaces
368e5da remove heading
4ff33dc append head
```

```
62976f2 append h6
a7226dd append h5
0c4aa48 append h4
9c5fad8 append h3
282cc62 append h2
a1a2109 append h1
cabcc16 Initial commit
Copy
```

Merger les commits par lot pour obtenir l'historique suivant:

```
~/workshop$ git log --oneline
af498bc (HEAD -> master) append style
38e2d86 replace content
31522ed append headings
cabcc16 Initial commit
Copy
```

Partie 4

Nous allons annuler les deux derniers commits et conserver les modifications afin de nous remettre en édition simple.

```
~/workshop$ git reset --soft HEAD~2
Copy
```

```

diff --git a/index.html b/index.html
index 0aac673..c944863 100644
--- a/index.html
+++ b/index.html
@@ -1,15 +1,17 @@
<!DOCTYPE html>
<html>
-
- <head></head>
-
+ <head>
+   <title>Git Example</title>
+   <style>
+     h1 {
+       color: red;
+     }
+     h2 {
+       color: green;
+     }
+   </style>
+ </head>
  <body>
- <h1>This is heading 1</h1>
- <h2>This is heading 2</h2>
- <h3>This is heading 3</h3>
- <h4>This is heading 4</h4>
- <h5>This is heading 5</h5>
- <h6>This is heading 6</h6>
+   <span>This page is just a demo</span>
  </body>
-
  </html>
\ No newline at end of file

```

Modifier le title avec **Git workshop** et commiter le tout.

Nous allons maintenant annuler le dernier commit et reverter les changements de celui-ci:

```

~/workshop$ git log --oneline
f691b73 (HEAD -> master) workshop
31522ed append headings
cabcc16 Initial commit

~/workshop$ git reset --hard HEAD~1
HEAD is now at 31522ed append headings

~/workshop$ git log --oneline
31522ed (HEAD -> master) append headings

```

```
cabcc16 Initial commit
```

```
~/workshop$ git status -s
```

Copy

Comme vous avez pu le voir dans cette partie, le reset permet d'annuler des commit en se déplacement par rapport à la HEAD (HEAD~1 pour se positionner à l'avant dernier commit, HEAD~2 à celui le précédent...) et l'option **--soft** permet de conserver les changements alors que **--hard** les supprime tout simplement.

Pensez avec la tête

Dans la partie précédente, nous avons utilisé les hash pour promener dans l'historique.

Une approche moins laborieuse consiste à utiliser la référence **HEAD** au lieu des hashes. **HEAD** est une référence qui pointe vers le commit le plus récent de la branche courante.

Nous pouvons donc nous déplacer de façon relative à **HEAD** (soit de façon relative au dernier commit).

HEAD => dernier commit

HEAD~1 => avant dernier commit

HEAD~2 => avant avant dernier commit

...

Example:

```
# git log --oneline
9ff4a63 (HEAD -> master) remove spaces
6482e4d remove heading
1ba8390 append head
f03f8a5 append h6
541cf6e append h5
3329866 append h4
31e2bd3 append h3
282cc62 append h2
a1a2109 append h1
cabcc16 Initial commit
```

```
git rebase -i HEAD~2
```

Copy


```
pick 6482e4d remove heading
pick 9ff4a63 remove spaces
```

Le HEAD~2 pointe l'avant avant dernier soit **1ba8390 append head**, donc le rebase pointe ceux après celui-ci.

Pour inclure le commit pointé, il est également possible d'associer le HEAD tildé au ^

Ainsi, **git rebase -i HEAD~2^**

```
pick 1ba8390 append head
pick 6482e4d remove heading
pick 9ff4a63 remove spaces
```

Use the force

Lorsque l'on modifie l'historique localement, il y a par définition une divergence avec la remote, il faut donc ne pas avoir peur de pousser ses modifications en utilisant la force.

Attention, il ne faut pas puller dans ce cas, sinon... vous rappatriez des commits dépréciés et vous n'avez pas fini de régler les conflits.

```
~/workshop$ git push --force-with-lease
Copy
```

Notez qu'au contraire de **force**, **force-with-lease** permet de ne pas écraser de commits poussés par d'autres pendant vos modifications.

Cette commande ne sera pas exécuté ici, car le workshop est protégé.

TP au Git

2.1 Création d'un nouveau dépôt

Nous allons d'abord nous intéresser à l'aspect gestionnaire de versions de Git : comment enregistrer l'historique des modifications apportées à un projet. Pour obtenir un dépôt Git sur lequel travailler, deux options sont possibles :

1. création d'un dépôt vide (typiquement utilisé pour commencer un nouveau projet de développement) ;

2. copie (clone dans le langage de Git) d'un dépôt existant pour travailler sur cette copie de travail (typiquement utilisé pour collaborer avec les développeurs d'un projet en cours).

Examinons la première option. Git a plusieurs interfaces utilisateur. La plus complète étant l'interface en ligne de commande (CLI), nous nous servons de celle-ci.

Pour créer un nouveau dépôt, on utilise la commande `git init monrepo`. Cette commande initialise un dépôt Git dans le répertoire `monrepo` (celui-ci est créé s'il n'existe pas). Ce répertoire contient alors à la fois une version de travail (dans `monrepo`) et un dépôt Git (dans `monrepo/.git`).

Question 2.1. Initialiser un nouveau dépôt Git dans un répertoire `sandwich`, et créez le fichier `burger.txt` qui contient la liste des ingrédients d'un burger, un ingrédient par ligne.

Steak salade 1 tomate cornichon fromage

2.2 Add et Commit

Source : <http://thomas.enix.org/pub/conf/git2011/presentation.pdf>

Figure 1 – git add commit workflow

Pour être intégrée dans l'historique des révisions du dépôt (pour être "commitée"), chaque modification doit suivre le workflow montré en Figure 1 :

1. la modification est d'abord effectuée sur la copie de travail ;
2. elle est ensuite mémorisée dans une aire temporaire nommée `index`, avec la commande `git add` ;
3. enfin, ce qui a été placé dans l'`index` peut être "commité" avec la commande `git commit`.

`git diff`, selon les paramètres d'appel peut être utilisé pour observer les différences entre les états en Figure 1 ; le format d'affichage est le même de la commande `diff -u`.

Question 2.2. Vérifiez avec `git status` l'état dans lequel se trouve votre dépôt. Vos modifications (l'ajout du fichier `burger.txt`) devraient être présentes seulement dans la copie de travail.

Question 2.3. Préparez `burger.txt` pour le commit avec `git add burger.txt`. Utilisez `git status` à nouveau pour vérifier que les modifications ont bien été placées dans l'`index`. Puis, utilisez `git diff --cached` pour observer les différences entre l'`index` est la dernière version présent dans l'historique de révision (qui est vide).

Question 2.4. Commitez votre modification avec `git commit -m "<votre_message_de_commit>"`. Le message entre guillemets doubles décrira la nature de votre modification (généralement ≤ 65 caractères).

Question 2.5. Exécutez à nouveau `git status`, pour vérifier que vos modifications ont bien été commitées.

Question 2.6. Essayez à présent la commande `git log` pour afficher la liste des changements effectués dans ce dépôt ; combien y en a-t-il ? Quel est le numéro (un hash cryptographique en format SHA1) du dernier commit effectué ?

Question 2.7. Créez quelques autres sandwiches `hot_dog.txt`, `jambon_beurre.txt`. . .et/ou modifiez les compositions de sandwiches déjà créés, en commitant chaque modification séparément. Chaque commit doit contenir une et une seule création ou modification de fichier. Effectuez au moins 5 modifications différentes (et donc 5 commits différents). À chaque étape essayez les commandes suivantes :

— `git diff` avant `git add` pour observer ce que vous allez ajouter à l'index ;

— `git diff --cached` après `git add` pour observer ce que vous allez committer.

Note : la commande `git commit <file>` a le même effet que `git add <file>` suivie de `git commit`.

2

Question 2.8. Regardez à nouveau l'historique des modifications avec `git log` et vérifiez avec `git status` que vous avez tout commité. Git offre plusieurs interfaces, graphiques ou non, pour afficher l'historique. Essayez les commandes suivantes (`gitg` et `gitk` ne sont pas forcément installés) :

— `git log`

— `git log --graph --pretty=short`

— `gitg`

— `gitk`

2.3 Voyage dans le temps

Question 2.9. Vous voulez changer d'avis entre les différents états de la Figure 1 ? Faites une modification d'un ou plusieurs sandwiches, ajoutez-la à l'index avec `git add` (vérifiez cet ajout avec `git status`), mais ne la commitez pas. Exécutez `git reset` sur le nom de fichier (ou les noms de fichiers) que vous avez préparés pour le commit ; vérifiez avec `git status` le résultat.

Question 2.10. Votre modification a été « retirée » de l'index. Vous pouvez maintenant la jeter à la poubelle avec la commande `git checkout` sur le ou les noms des fichiers modifiés, qui récupère dans l'historique leurs versions correspondant au tout dernier commit. Essayez cette commande, et vérifiez avec `git status` qu'il n'y a maintenant plus aucune modification à commiter.

`git checkout` est une commande très puissante. Elle vous permet de voyager entre différentes branches (voir plus loin) et aussi de revenir temporairement à une version précédente de votre copie de travail.

Question 2.11. Regardez l'historique de votre dépôt avec `git log` ; choisissez dans la liste un commit (autre que le dernier). Exécutez `git checkout COMMITID` où `COMMITID` est le numéro de commit que vous avez choisi.

Vérifiez que l'état de vos sandwiches est maintenant revenu en arrière, au moment du commit choisi. Que dit maintenant git status ?

git log n'affiche plus les commits postérieurs à l'état actuel, sauf si vous ajoutez l'option --all.

Attention, avec git checkout les fichiers de votre copie de travail sont modifiés directement par Git pour les remettre dans l'état que vous avez demandé. Si les fichiers modifiés sont ouverts par d'autres programmes (e.g. un éditeur de texte comme Emacs), il faudra les réouvrir pour observer les modifications.

Question 2.12. Vous pouvez retourner à la version plus récente de votre dépôt avec git checkout master.

Vérifiez que cela est bien le cas. Que dit maintenant git status ?

2.4 Clone

En faisant le devoir pour aujourd'hui, dont le sujet est disponible aussi sur <https://upsilon.cc/~zack/teaching/1314/ed6/tp-1-devoir.pdf> (1)

vous avez eu accès à la plate-forme d'hébergement GitLab

<http://moule.informatique.univ-paris-diderot.fr:8080/> (2) en utilisant le protocole ssh (authentification au moyen de votre clef publique), puis vous avez créé un projet sur GitLab (c'est-à-dire un dépôt Git) et vous avez récupéré une copie de ce projet sur votre répertoire personnel.

Assurez-vous d'avoir accompli toutes les tâches du devoir pour aujourd'hui (1).

Pour chaque séance de TP, vous pouvez créer un nouveau projet sur GitLab nommé tp1, tp2, etc. selon le numéro de séance, ou bien vous pouvez créer un seul projet pour tout le cours nommé Cproj et organisé avec des répertoires tp1, tp2, etc.

Question 2.13. Création et récupération d'un projet. Créez un nouveau projet sur GitLab (2) nommé tp1.

Récupérez une copie de ce projet sur votre répertoire personnel de votre machine au moyen de la commande `$ git clone git@moule.informatique.univ-paris-diderot.fr:votre_identif/votre_projet.git`

3 où votre_identif est votre nom utilisateur dans le compte LDAP et votre_projet est le nom que vous avez donné à votre projet, donc tp1 dans ce cas. Quel répertoire a été créé ? Que contient-il (y compris les répertoires cachés) ?

Question 2.14. Une première révision. Placez-vous dans le répertoire créé par la commande git clone cidessus (c'est le répertoire de travail de votre dépôt). Créez un fichier description.txt. Éditez-le pour y ajouter une courte description du fait que vous allez utiliser ce dépôt pour le TP1 du cours de Cproj. Ajoutez ce fichier à l'index du répertoire au moyen de la commande

`$ git add nom_fichier` où nom_fichier est le nom du fichier que vous venez de créer, donc description.txt dans ce cas.

Commitez ce changement au moyen de la commande

`$ git commit`

A l'absence du paramètre -m (2.4) Git va lancer pour vous un éditeur de texte (vi par défaut) pour entrer un message de commit approprié. Le format doit être celui que nous avons vu en cours, i.e. une ligne courte (≤ 65

caractères) de description générale puis, si nécessaire, une ligne vide et plusieurs lignes des description plus détaillé du révision.

Allez sur la page de votre projet tp1 dans GitLab (2). Y trouvez-vous le fichier description.txt est-il présent ? (vous pouvez voir les fichiers qui ont été envoyés – s'il y en a – sur le serveur de GitLab dans l'onglet Files dans la page de votre projet).

Question 2.15. Modifications. Éditez le fichier description pour y préciser votre nom et prénom.

Lancez Gitg depuis votre répertoire. Gitg a deux onglets History et Commit. Dans l'onglet Commit, on remarque 4 cadres :

- Unstaged qui contient la liste des modifications qui ont été apportées dans le dépôt et qui n'ont pas été sélectionnées pour être commitées,
- Staged qui contient la liste des modifications qui ont été apportées et qui ont été sélectionnées pour être commitées,
- Changes qui affiche une modification,
- Commit message qui contient le message du commit courant.

Question 2.16. Une seconde révision. Commit à l'aide de Gitg. À l'aide de Gitg, commitez votre modification.

Que se passe-t-il si vous commitez sans donner un message de commit ? Commitez avec le message "deuxième revision". Voyez-vous des changements sur GitLab (2) ?

Question 2.17. Seconde copie de travail. Dans un nouveau répertoire tmp, exécutez à nouveau la commande

```
$ git clone git@moule.informatique.univ-paris-diderot.fr:votre_identif/tp1.git
```

Que contient le répertoire créé ? Pourquoi ?

Question 2.18. Envoi des données sur le serveur de GitLab. Dans la copie de travail où vous avez créé et commité le fichier description.txt, exécutez la commande

```
$ git push origin master
```

pour envoyer pour la première fois au serveur GitLab les modification au projet que vous avez commité. 1 Cette fois, que s'est-il passé sur GitLab (2) ?

Question 2.19. Mise à jour de la deuxième copie de travail. Exécutez dans tmp/tp1 la commande

```
$ git pull
```

Que s'est-il passé dans le répertoire tmp/tp1 ? Dans le répertoire tmp/tp1, supprimez le fichier description.txt : comment pouvez-vous le récupérer depuis GitLab ? Sinon, essayez la commande `git checkout description.txt`.

Question 2.20. Historique du répertoire. Lancez Gitg. Comment pouvez-vous accéder dans Gitg au contenu du fichier description.txt juste après que vous ayez décompressé l'archive ? Et dans GitLab ?

1. Cette syntaxe longue est nécessaire la première fois que vous faites push pour indiquer où vous voulez envoyer vos changements.

Pour les fois suivantes, la syntaxe plus courte `git push` sera suffisante.

4

3 Git avancé

3.1 Clone et Remote

Très peu de logiciels sont développés aujourd'hui à parti de zéro. Le cas normal est de plus en plus de se baser sur un logiciel existant, et de chercher à faire remonter aux auteurs du logiciel les changements qui sont importants pour nous. Git rend tout cela très efficace pour les auteurs originels et pour les contributeurs.

Comme projet d'exemple, nous allons utiliser `purity`, un logiciel jouet qui estime la correspondance entre un joueur et un profil (e.g. `nerd`, `hacker`, etc).

Question 3.1. Récupérez le dépôt Git de `purity` avec la commande

```
git clone http://anonscm.debian.org/git/collab-maint/purity.git
```

Quel est le nom du repertoire que vous avez crée ? Lancez `Gitg` et observez l'histoire du repertoire (de la même

manière que dans la question 2.8).

3.2 Branches de développement

Nous allons maintenant nous concentrer sur la notion de branche. Lors du développement d'un projet, il peut arriver que l'on veuille introduire une nouvelle fonctionnalité dans le projet, sans « casser » le projet. Nous voudrions donc pouvoir basculer instantanément de la version stable du projet à sa version « en développement ».

C'est ce que nous permettent de faire les branches.

Plus précisément, vu la facilité avec laquelle Git permet de gérer les branches, en Git on a tendance à utiliser les branches pour chaque développement non trivial, e.g. :

— pour corriger un bug numéroté 1234, on procédera comme suit :

1. à partir du branche principale de développement (normalement nommé `master`) on créera une nouvelle branche nommée `bug/1234`
2. on changera la copie de travail pour être sur la branche `bug/1234`
3. on corrigera le bug, en faisant tous les commits nécessaires
4. on retournera sur la branche `master`
5. on fusionnera (merge) la branche `bug/1234` avec `master`

Question 3.2. Dans le dépôt de `purity` que vous avez cloné, exécutez `git branch` et vérifiez que, d'après lui, vous êtes bien sur la branche `master` : la branche actuelle est celui avec une étoile. « `master` » est le nom de la branche de développement principale, que Git à crée pour vous. Notez aussi que la sortie de `git status` contient le nom de la branche actuelle ; essayez !

On veut maintenant traduire en Français quelques-unes des questions du test pour vrais hacker de purity, et le faire selon le style propre de Git.

Question 3.3. Créez une nouvelle branche pour le développement à l'aide de `git branch NOM`. L'effet de la commande est de créer une nouvelle branche, nommée `NOM`, à partir de la branche courante. e.g. vous pouvez essayer avec `git branch fr-translation` pour la nommer « fr-translation ». Vérifiez les deux choses suivantes :

1) l'existence de la nouvelle branche (vous pouvez le faire à la fois avec `git branch` et avec `gitg`) ; et 2) le fait que vous êtes toujours sur la branche « master » (vous pouvez le faire avec `git status`, `git branch`, et autres outils).

Vous êtes maintenant prêts à développer vos traductions.

Question 3.4. Mettez-vous sur la branche de développement. Pour le faire il faut utiliser la commande `git checkout` comme nous l'avons fait dans la première partie du TP pour voyager dans le temps : il suffit de passer comme paramètre le nom du branche à la place de l'identifiant de commit, e.g. `git checkout fr-translation`. Vérifiez après avec `git branch` ou `git status` que vous êtes maintenant sur la branche désirée.

Question 3.5. Traduisez en français quelques questions du test de purity pour vrais hackers. Ouvrez le fichier `tests/hacker` et remplacez le texte en anglais de la première question avec sa traduction en français ; committez vos changements. Si vous avez des doutes sur le format du fichier, regardez `tests/format` qui en contient une description. Répétez l'opération pour au moins 2 autres questions.

Important : faites un commit pour chaque traduction. Pour traduire 3 questions, donc, il faudra faire 3 commits différents, avec les messages de commit appropriés.

Question 3.6. Exécutez `purity` sur le fichier que vous venez de traduire, pour vérifier que les questions que vous aviez touchées sont maintenant en français.

Question 3.7. Revenez maintenant sur la branche principale, `master`. Pour le faire exécutez `git checkout master`.

Vérifiez que vos changements, maintenant ne sont plus dans le fichier que vous avez touché. Néanmoins, ils n'ont pas été oubliés. Lancez `gitg` et vérifiez que les changements existent, même s'il ne sont pas partie de la branche sur laquelle vous êtes maintenant.

Question 3.8. Nous allons maintenant implémenter une autre fonctionnalité, le patching. Téléchargez le patch :

<http://upsilon.cc/zack/teaching/1213/ed6/tp5-adding-some-colors.patch>. Vérifiez que vous êtes sur la branche `master` et appliquez-le à `pt.c`. Committez les changements induits par ce patch (utilisez la commande `patch`).

Vérifiez avec `gitg` l'état de votre dépôt. À ce point, les 2 branches `master` et `fr-translation` sont parties dans 2 directions différentes : chacune a au moins un commit qui n'existe pas dans l'autre branche.

Vous êtes maintenant prêts à fusionner les changements effectués dans la branche `fr-translation` sur `master`. Notez que l'action de fusionner n'est pas une action symétrique : décider de fusionner le changement de `fr-translation` sur `master` n'est pas la même chose que fusionner les changements de `master` sur `fr-translation`. Dans le premier cas `master` sera changé pour contenir aussi les changements de `fr-translation` et celui-ci ne sera pas touché ; vice-versa dans le deuxième cas.

Question 3.9. Vérifiez que vous êtes toujours sur la branche master. Fusionnez-y les changements de la branche fr-translation avec la commande `git merge fr-translation`. Git vous demandera un message de commit, comme d'habitude, donnez une description de l'action que vous êtes en train de faire, e.g. « merge French translations of first N questions of the hacker test ».

Question 3.10. Regardez maintenant l'historique de votre dépôt avec `gitg`. Le « commit » que vous venez d'ajouter est un commit de type « merge » entre deux branches.

Jusqu'à présent, nous n'avons envisagé que des scénarios dans lesquels la fusion des branches est simple, mais il peut arriver qu'il y ait des conflits, par exemple un même bogue corrigé de manière sensiblement différente dans deux branches différentes.

Question 3.11. Que se passe-t-il dans ce cas-là ? Essayez d'implémenter ce scénario des conflits.

Par exemple : créez à partir de master une branche qui traduit en français les messages d'aide en ligne qui sont affichés lorsque vous appuyez sur 'h' pendant l'exécution. Avant de la fusionner avec master, créez une autre branche à partir de master et dans cette branche supprimez la description de l'option 'l' de l'aide en ligne.

Maintenant retournez sur master et effectuez la fusion des 2 branches.

Comment Git vous permet-il de résoudre les conflits ? Écrase-t-il unilatéralement les modifications effectuées dans une branche ?

Dorénavant, vous travaillerez systématiquement dans votre dépôt, en commitant (avec des messages explicatifs) de temps en temps afin de conserver un historique de votre travail. N'oubliez pas d'envoyer vos modifications sur [GitLab](#).

L'objectif de ce travail était de manipuler les outils de développement qui permettent la coexistence et l'interaction entre différentes versions d'un même fichier source. Au terme de cette séance, vous devrez savoir :

- Calculer la différence textuelle entre deux versions d'un fichier source et l'utiliser pour passer d'une version à l'autre.
- Calculer les différences textuelles concurrentes entre deux versions d'un fichier de référence.
- Comprendre la notion de conflit entre modifications concurrentes.

Pour récapituler, voici le tutoriel officiel du site web officiel de Git [5].

4 Git- Cheat sheet [1], [2]

Creation	Historique des commits
Cloner localement un dépôt distant <code>git clone https://github.com/<remote></code>	Afficher tous les commits commençant par les plus récents <code>git log</code>
Créer un nouveau dépôt local <code>git init</code>	Afficher les changements dans le temps pour un fichier spécifique <code>git log -p <fichier></code>
Changements locaux	Responsable du dernier changement d'un fichier <code>git blame <fichier></code>
Fichiers modifiés dans le répertoire de travail. À utiliser fréquemment! <code>git status</code>	Mettre à jour et publier
Intégrer les changements d'un fichier au prochain commit <code>git add <fichier></code>	Télécharger toutes les modifications depuis REMOTE et les fusionner / les intégrer à HEAD <code>git pull <remote> <branch></code>
Retirer un fichier du prochain commit <code>git rm <fichier></code>	Télécharger toutes les modifications depuis REMOTE, mais ne pas les intégrer à HEAD <code>git fetch <remote></code>
Envoyer les changements vers le dépôt <code>git commit</code>	Fusionner et Versionnage
	Fusionner une branche dans le HEAD actuelle <code>git merge <branch></code>

BONUS

A.1 Clone et Remote

Question A.1. Étudiez l'historique de votre nouveau dépôt à l'aide de `git log --graph` ou d'autres outils que nous avons déjà utilisés.

Question A.2. Compilez le projet avec la commande `make LIBDIR=tests/`, testez-le avec la commande `./purity sample`, regardez le code source de `purity` dans le fichier `pt.c`.

Avant de procéder à l'intégration de vos changements, vous voulez noter que la version de `purity` sur laquelle vous travaillez n'est pas la version officielle. Ajoutez donc un commentaire à la fin du fichier `README` pour clarifier que vous êtes en train de traduire les questions de tests/hacker en français. Commitez le changement au `README`.

A.2 Échange de patches

Rendre efficace l'échange des changements est un des objectifs principaux de Git. Une manière de le faire est d'avoir plusieurs dépôts et de permettre aux gérants de chacun d'envoyer des commits d'un dépôt à un autre (voir prochaine section). Une autre manière est d'échanger des patches qui sont générés par Git à partir des commits.

Le processus (workflow) habituel dans ce cas est donc le suivant :

- `git clone`
- développement / `git commit` / développement / `git commit` / . . .
- choix des commits à envoyer
- stérilisation des commits en format patch (avec la commande `git format-patch`)
- envoi des patches par mail
- intégrations des patches avec `git am` et/ou `git apply`

Question A.3. Travaillez à deux pour les exercices de cette section. À partir d'un dépôt Git (e.g. celui de `purity`) développez des changements indépendants (e.g. continuez la traduction de quelques questions dans des tests différents). Ensuite, identifiez avec `git log` les numéros du premier et dernier commit que vous voulez envoyer à votre binôme. Notez les quelque part.

Question A.4. Ce que vous devrez envoyer à votre binôme est une série de patches (patch series) à appliquer l'un après l'autre. Pour la produire exécutez la commande `git format-patch COMMIT1..COMMIT2` ou `COMMIT1` est l'identifiant du premier commit (le plus vieux) à envoyer et `COMMIT2` est l'identifiant du dernier (le plus récent).

Git va vous produire une liste de fichiers, un pour chaque commit, qui commence avec les préfixes `0000`, `0001`, `0002`, etc. Regardez le contenu de ces fichiers, en quel format sont ils écrits ?

Question A.5. Envoyez les fichiers à votre binôme (par mail ou autre moyen). Entre-temps, il aura fait la même chose avec vous.

Question A.6. Grâce à la commande `git am` vous pouvez appliquer un ou plusieurs patches dans votre dépôt et les committer atomiquement. En cas de conflits, Git vous demandera de les résoudre comme d'habitude.

Alternativement, `git apply` applique un patch à la copie de travail courante, mais n'effectue pas le commit. Essayez les deux commandes et intégrez les changements de votre binôme avec les vôtres.

Comme travailler avec des identifiants de commits très longs n'est pas aisé, Git offre plusieurs astuces. D'abord vous pouvez utiliser seulement un préfixe court d'un identifiant de commit, tant que le préfixe n'est pas ambigu dans un dépôt 2

Ensuite, vous avez des raccourcis, par exemple :

- `HEAD` correspond au dernier commit de la branche courante
- `HEAD^` correspond à l'avant-dernier commit de la branche
- `HEAD^^` correspond toujours à l'avant-avant-dernier commit de la branche (etc. pour autres ^)
- `COMMITID^` correspond au commit précédent le commit `COMMITID` ; donc `HEAD^` est un cas particulier de cette syntaxe
- `COMMITID~N` correspond au Nième avant-dernier commit du commit `COMMITID`