Université Paris Diderot
L3 Informatique

Cproj - Conduite de Projet
Année 2017-2018

TP Git

Version 0.9

1 Introduction

> Git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. En 2016, il s'agit du logiciel de gestion de versions le plus populaire qui est utilisé par plus de douze millions de personnes.
>
> — https://fr.wikipedia.org/wiki/Git

Lors de cette première séance de TP, nous allons nous initier à l'utilisation du Git et puis à la maintenance d'un répertoire de travail de manière structurée et ordonnée.

2 Premiers pas avec Git

Git, UN LOGICIEL DE CONTRÔLE DE VERSIONS

Git [4] est un logiciel de contrôle de versions, il permet de sauvegarder l'historique du contenu d'un répertoire de travail. Pour ce faire l'utilisateur doit régulièrement enregistrer (en créant une révision ou *commit*) les modifications apportées au répertoire, il pourra ensuite accéder à l'historique de toute les modifications et inspecter l'état du dossier à chaque révision.
Git a la particularité de permettre de créer une copie d'un répertoire de travail, *working copy*, et de synchroniser entre eux plusieurs copies du même répertoire, permettant la décentralisation du travail.
De plus, Git permet d'utiliser une ou plusieurs *branches de développement* et de fusionner entre elles ces branches.

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 servirons de celle-ci.

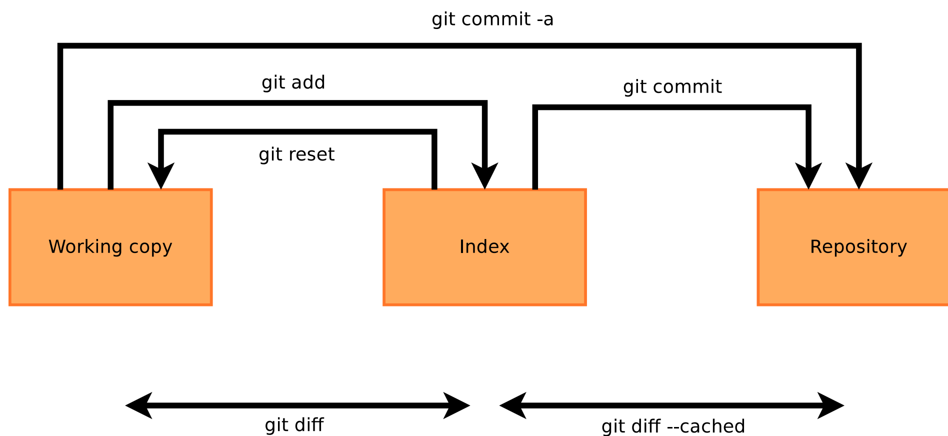
Pour créer un nouveau dépôt, on utilise la commande `git init monrepo`. Cette commande initialise un dépôt Git dans le répertoire `monrepo` (celui-ci est créé s'il n'existe pas). Ce repertoire contient alors à la fois une version de travail (dans `monrepo`) et un dépôt Git (dans `monrepo/.git`).

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
```

```
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 command `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 *characteurs).*

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

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`.

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/1617/cproj/TP-git.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
```

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 à été créé ? Que contient-il (y compris les répertoires cachés) ?*

Question 2.14. ***Une première révision.*** *Placez-vous dans le répertoire crée par la commande* git clone *ci-dessus (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 aller 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 ficher 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.

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 répertoire (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 ; commitez*

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`*. Commitez 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 lequels 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 confits ? 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	
Cloner localement un dépôt distant	git clone https://github.com/<remote>
Créer un nouveau dépôt local	git init
Changements locaux	
Fichiers modifiés dans le répertoire de travail. À utiliser fréquemment !	git status
Intégrer les changements d'un fichier au prochain commit	git add <fichier>
Retirer un fichier du prochain commit	git rm <fichier>
Envoyer les changements vers le dépôt	git commit

Historique des commits	
Afficher tous les commits commençant par les plus récents	git log
Afficher les changements dans le temps pour un fichier spécifique	git log -p <fichier>
Responsable du dernier changement d'un fichier	git blame <fichier>
Mettre à jour et publier	
Télécharger toutes les modifications depuis REMOTE et les fusionner / les intégrer à HEAD	git pull <remote> <branch>
Télécharger toutes les modifications depuis REMOTE, mais ne pas les intégrer à HEAD	git fetch <remote>
Fusionner et Versionnage	
Fusionner une branche dans le HEAD actuelle	git merge <branch>

Références

[1] git cheat sheet. https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf.

[2] git cheat sheet interactif. http://ndpsoftware.com/git-cheatsheet.html.

[3] git livre. https://git-scm.com/book/en/v2.

[4] git page d'accueil. https://git-scm.com/.

[5] git tutoriel. https://git-scm.com/docs/gittutorial.

A Bonus [3]

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 patchs

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 patchs 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 patchs par mail
- intégrations des patchs 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 patchs *(*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 patchs 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 *N*ième *avant*-dernier commit du commit `COMMITID`.

2. Dans un dépôt contenant 256 commits, il y a une probabilité de 50% qu'il existe 2 commits avec le même prefixe de 4 lettres. Il y a une chance sur deux que dans un dépôt de 65.000 commits il y en ait 2 qui aient le même prefixe de 8 lettres.