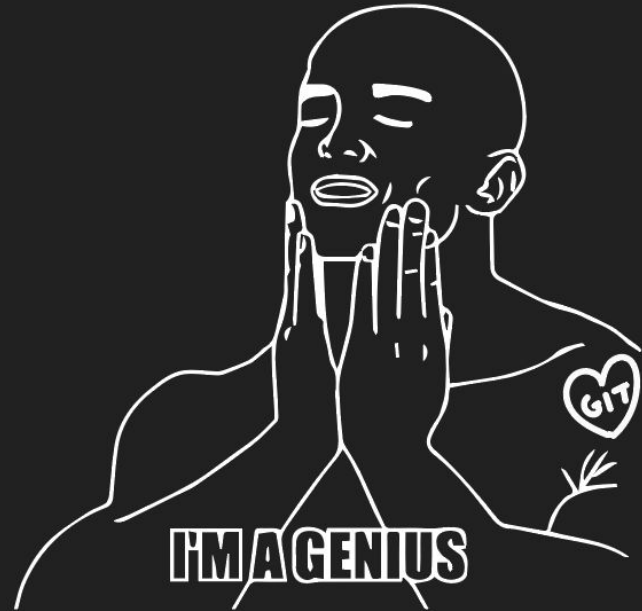


Atelier Git

08/11/2024

Le but de cet atelier



Déroulé de l'atelier

- Introduction
- Les différents concepts
 - commits
 - l'historique
 - branches
 - stash
 - dépôts distants
- Questions

- Les commandes
 - init
 - clone
 - add
 - restore
 - status
 - commit
 - **fetch**
 - branch
 - switch
 - **merge**
 - log
 - diff
 - **remote**
 - push
 - pull
- En pratique
- Questions

Basics

- Objectif : fonctionnalités basiques de Git
- Prenez des **notes**, pratiquez !



N'hésitez pas à **poser**
vos questions !

Qu'est-ce que git ?

- **Git** = Logiciel de gestion de version décentralisé
- **GitHub / GitLab** = Service web d'hébergement, basé sur git

Avantages :

Travailler hors connexion (on se connecte seulement pour push)

Donc très rapide !

Contrôle fin de la version finale du projet

Pas de gestion complexe des permissions : gros + en travail opensource

Possibilité de faire 1000 brouillons sans embêter les collaborateurs

Annexe : clé ssh

Pour partager des fichiers, par exemple sur GitHub ou la vogsphere, on peut utiliser une **clé SSH**. Elle permet l'authentification de votre machine.

Il faut en créer une, récupérer sa partie publique, et l'uploader sur l'hébergeur (GitHub, l'intra 42, ...).

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key ( path):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa
Your public key has been saved in id_rsa.pub
The key fingerprint is:
SHA256:IOxQADlMJRb2hieiYq8Vm5aKjPvBBCClZUD4P9xyRjA moi@z3r10p1.42lyon.fr
The key's randomart image is:
```

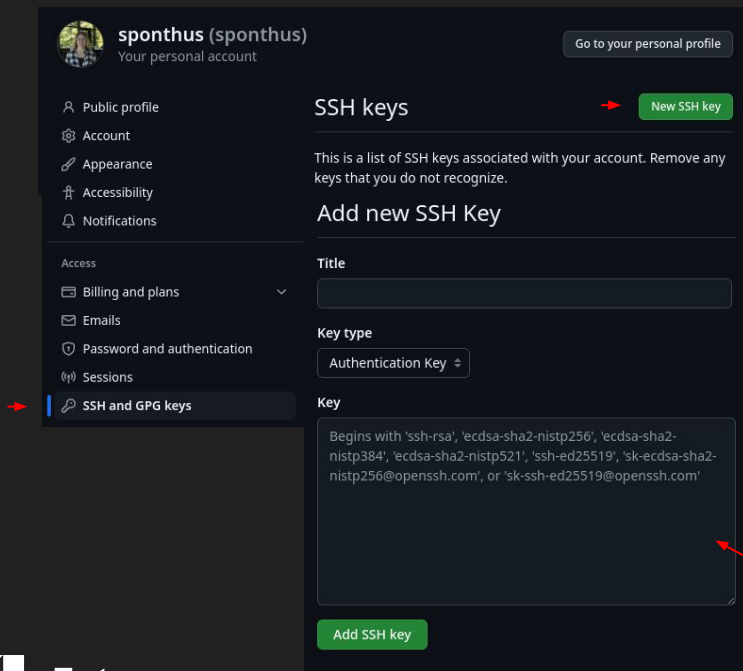
```
+---[RSA 2048]--+
|CX=uu.         |
|BBo*Ek         |
|+.=.oo         |
|..+...         |
|o.b.o S        |
|.. += +        |
|.o o*          |
|o= oo          |
|*+ooo          |
+----[SHA256]---+
```

```
$ cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQGDxgpJKsAG2Ek7Nq7XGoAtVVv
WVzJDpOEswI7XekukkkQNTxSWS50sGW3AWqAEUxez4A3FcRTM6HwHYM
vrACrAH0M+prjXZOi6mypoYtUD31nR8ThxoS9VqGISEV3mVRKhM5Ni
cAKiO6vhVePOz5A4N7EMmfQZ21mMn5WbEBFT61EEMfriW2x5IaU6pn
IvYrlceZ4ni66KIcdomqcaoRHY2clLEdLxuZjWeXxwkVUsvJ1PK/al
fJxH/ZPB2iGYr6uuBG6rkHO4NLe97fUJI/U4pwgz0RIQ3A0MBC6siY
PibwmxXF81+InQnhblJoH1057xk0M3ggdCyQAmueXrDa53cJG9wBOX
Z/5G+J/VImsqreQyWxoM3ph8XcvXbDUWBPW3zcm5DydOrf0G7H/p+W
J7uZrx23tuXaqRfcPLGDsM4JEQrRBc9tcgbJHNNu61150k+rKVdd+
L07aYI3iG9/bK8tkSGhzOfTRuukhdhN46sLAuNDbfHm4T8nYECnz8C
o0U=moi@z3r10p1.42lyon.fr
```

Ou uploader sa clé SSH

Sur GitHub

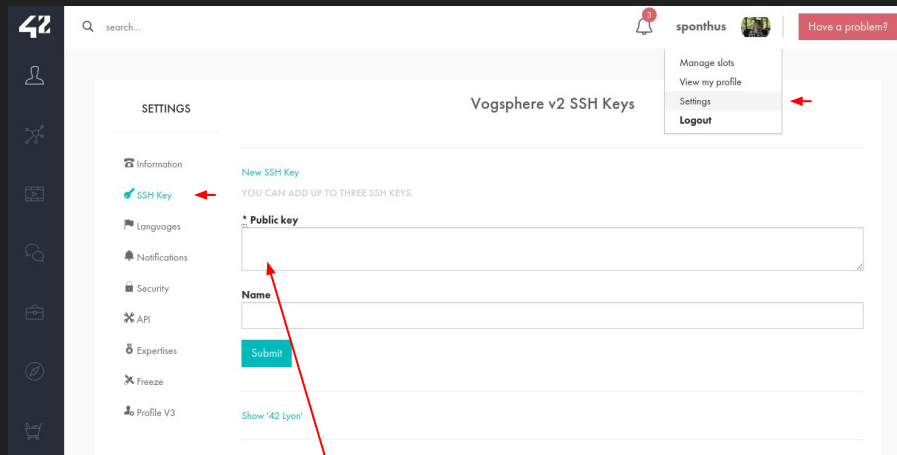
(repo git perso ou en groupe)



The screenshot shows the GitHub 'SSH keys' page for user 'sponthus'. The left sidebar contains navigation links: Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and plans, Emails, Password and authentication, Sessions, and SSH and GPG keys (highlighted with a red arrow). The main content area has a 'New SSH key' button and a list of existing keys. Below this is the 'Add new SSH Key' section with fields for Title, Key type (set to 'Authentication Key'), and Key. A red arrow points to the 'Key' field, which contains a long alphanumeric string. At the bottom is an 'Add SSH key' button.

Sur l'intranet

(attention : vous ne pourrez push que depuis l'école)



The screenshot shows the 'Vogsphere v2 SSH Keys' settings page. The left sidebar has a search bar and a list of settings: Information, SSH Key (highlighted with a red arrow), Languages, Notifications, Security, API, Expertises, Freeze, and Profile V3. The main content area has a 'New SSH Key' button and a form for adding a new key. The form has a 'Public key' label and a 'Name' field. A red arrow points to the 'Public key' field, which contains a long alphanumeric string. At the bottom is a 'Submit' button.

```
$ cat ~/.ssh/id_rsa.pub
```

```
ssh-rsa
AAAAAB3NzaC1yc2EAAAADAQABAAQDAxgxpJKsAG2Ek7Nq7XGoAtVVyWVzJDpOEswI7Xeku
kkQNTxSW5S50sGW3AWgaEUez4A3FcRTM6HwHYMrACrAH0M+prjXZ0i6mypoYtUD31nR8T
hxoS9VgGISEV3mVRKh5N1cAKiO6vhVePoZ5A4N7EMmfQZ21mMn5WbEBFT61EEMfriW2x5
IaU6pnIyYr1ceZ4ni66K1cdomgcaocRHY2c1LEdLxuzJWeXxwKVUsvJLPK/alfJxH/ZPB2i
GYr6uuB6GrKH04NLe97FUJI/U4pwwgzORIQ3A0Mbc6siYPIbwwmXF81+InQnhblJoH1057x
kOM3ggdCyQAmueXrDa53cuJG9wBOX2/5G+/V/ImsqreQyWxoM3ph8XcvXbDUWBPW3zcm5dy
dOrf0G7H/p+WJ7uZrx23tuXaqRFcPLDGdsM4JEqrRBC9tcbgJHNNu61150k+rkVdd+L07a
YI3iG9/bK8tkSGhzOfTRuukhdhN46sLAuNDbfHm4T8nYBCnz8CoOU=
moi@z3i0pl.42lyon.fr
```





C'est parti pour une plongée dans le monde
merveilleux de git !

Et on va découvrir un super site
pour visualiser tout ça :

<https://learngitbranching.js.org>

git add : la base de la gestion de l'index



Working Directory :

Cela représente l'état courant de votre dossier local.

L'index :

L'index, c'est ce que git a enregistré de votre travail. Si vous modifiez un fichier, git n'en saura rien tant que vous le l'avez pas informé via `git add`.

```
$ git status
On branch main
Your branch is up to date with
'origin/main'.
```

```
Changes to be committed:
  modified:   ft_atoi.c
  modified:   ft_strlen.c
```

} état dans l'index, différence avec HEAD

```
Changes not staged for commit:
  modified:   ft_atoi.c
```

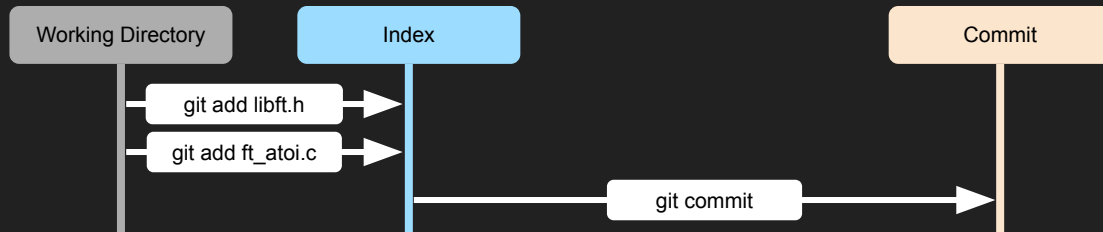
} dans le working directory, différence avec l'index

```
Untracked files:
  test.c
```



“HEAD” représente le commit sur lequel nos modifications se basent

git commit : le début de la gestion des versions



Vous êtes prêt à faire commit :

Quand vous avez informé git de tous les changements que vous souhaitez sauvegarder, vous êtes prêt à commit.

Le commit :

Un commit représente une sauvegarde de l'index à un instant T.

Chaque commit est **unique**, ne peut être modifié et possède un identifiant qui lui est propre.

En plus du contenu enregistré, il contient également l'identifiant de son parent.

Mais un commit peut aussi avoir plusieurs parents...

On va d'abord voir les branches 😊

git branch : vous inquiétez pas, ça mord pas

Une branche, c'est pas grand chose de plus qu'un pointeur vers un commit.

Elle ne "connaît" pas son historique, il est simplement retrouvé en liant chaque commit à son parent, récursivement.

On peut les créer différentes manière :

```
$ git branch <branche> # créé une branche, et c'est tout  
$ git switch -c <branche> # créé une branche et switch dessus instantanément
```

Et les supprimer, si on est pas dessus

```
$ git branch -d <branche> # supprime la branche si c'est "safe" de le faire  
$ git branch -D <branche> # supprime la branche
```



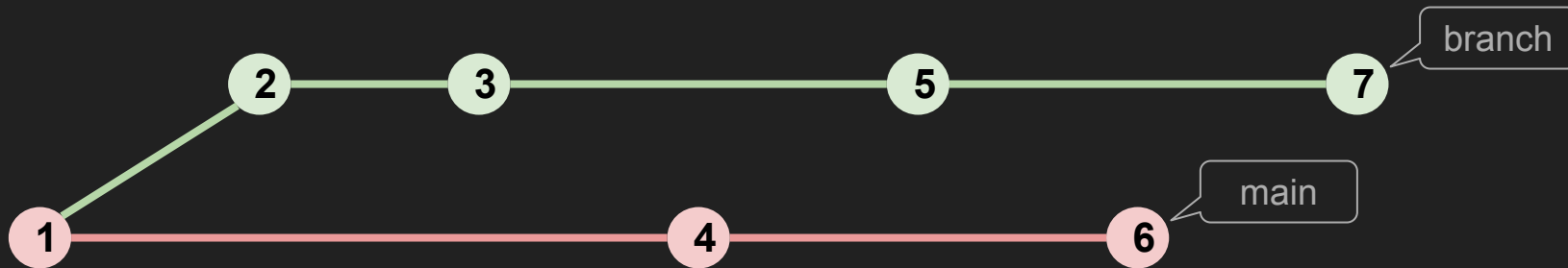
Je vous conseille d'utiliser -d en priorité, et vérifier pourquoi le mode "safe" échoue avant d'utiliser -D.

Pour passer d'une branche à une autre

```
$ git switch <branche>
```

git branch : vous inquiétez pas, ça mord pas

Voici un exemple de graph avec des branches :



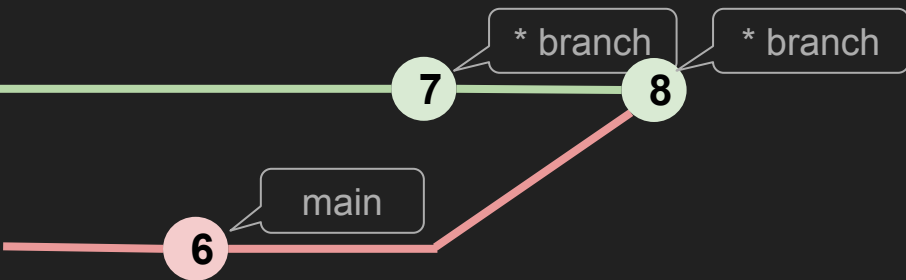
git merge : 2 en 1

Un `git merge` vous permet de créer un commit ayant deux parents.

Un commit peut avoir plus de deux parents, et `git merge` le permet, mais nous ne nous concentrerons pas sur cela.

Reprenons l'exemple précédent. Nous souhaitons récupérer les modifications de main (6) sur branch (7).

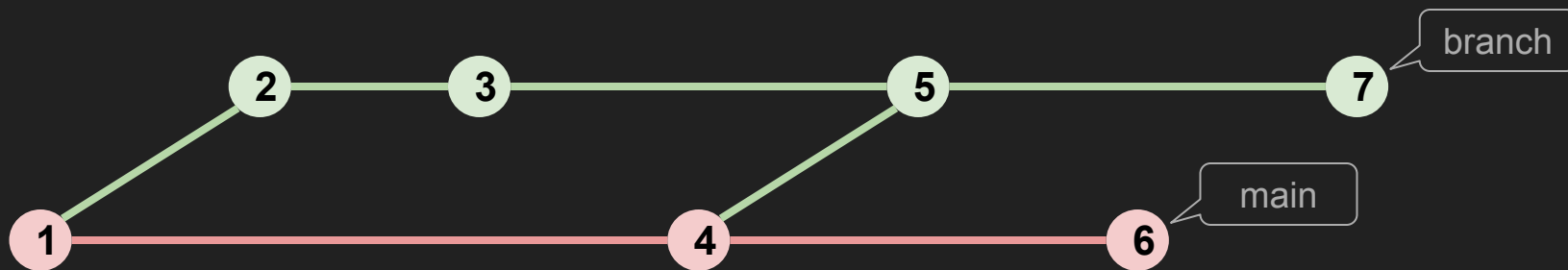
```
$ git merge main
```



Hop, on est à jour sur main 🎉

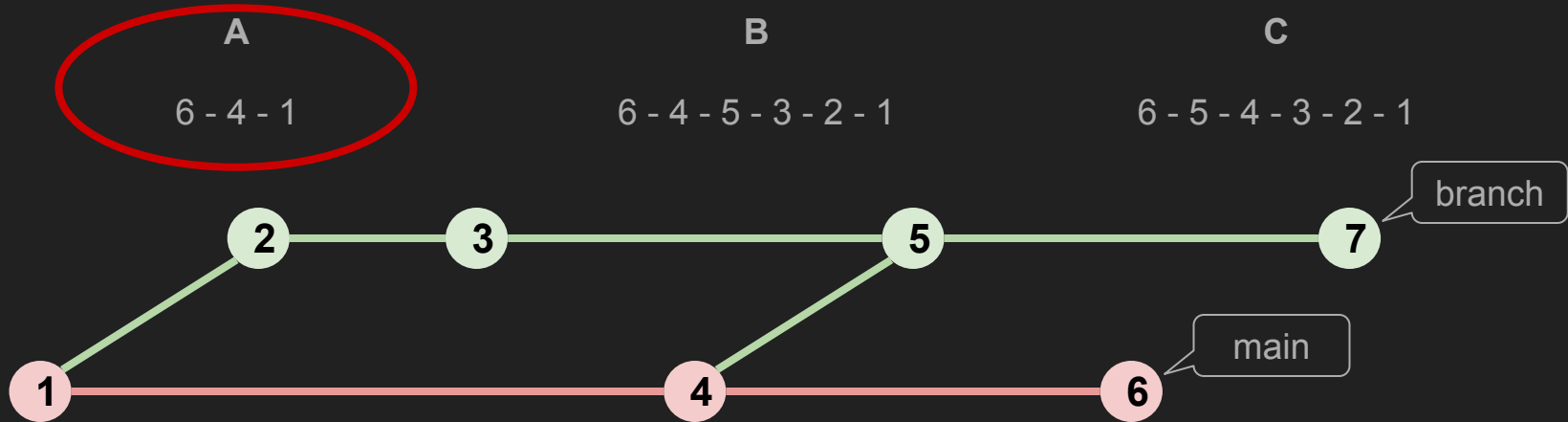
Quiz

Voici un graphique représentant un arbre Git



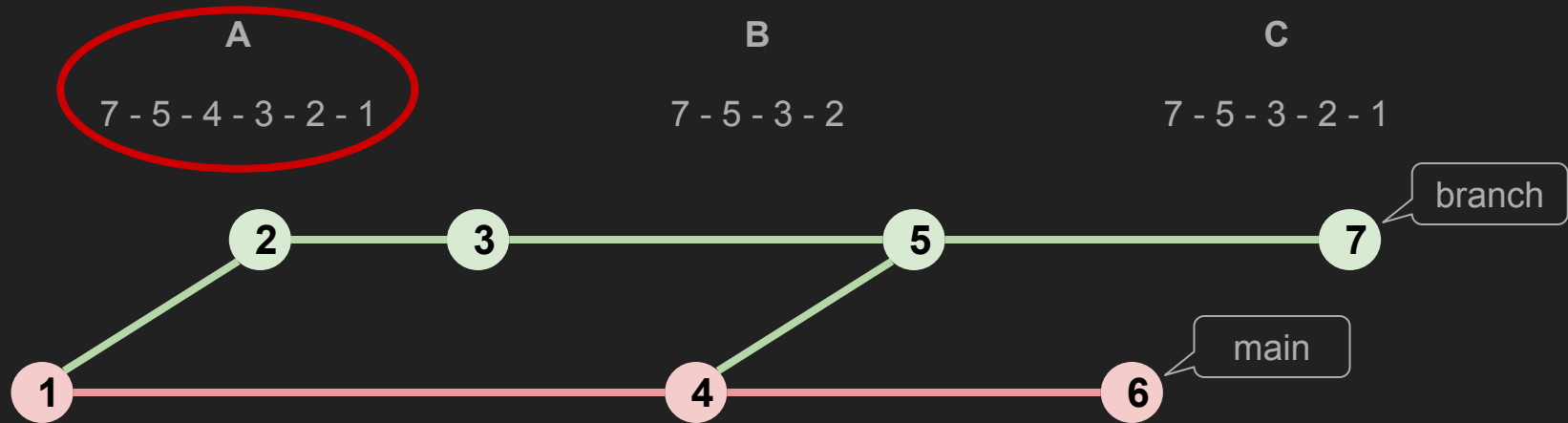
Quiz (1/2)

Dans ce graph, l'historique de "main" est :



Quiz (2/2)

Dans ce graph, l'historique de “branch” est :



Pour obtenir le résultat 7 - 5 - 3 - 2, le seul moyen est de soustraire l'historique de main à celui de branch.

git merge : quels commits ?

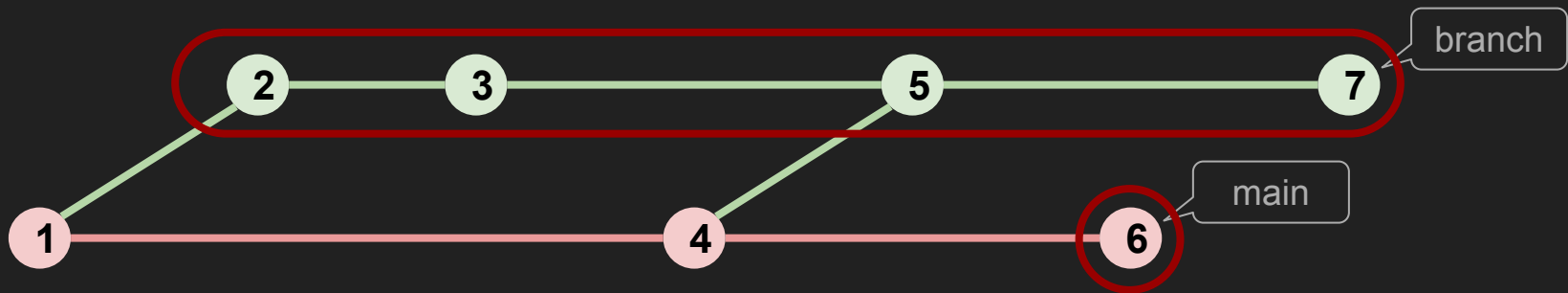
Il serait intéressant de savoir quels sont les commits qui seront impliqués dans le merge.

Pour les connaître, c'est exactement pareil que la dernière réponse du quiz : ce sont les commits de B - A.

Depuis "main", `git merge branch` incorporera les changements des commits 7, 5, 3 et 2.

Depuis "branch", `git merge main` incorporera le changement du commit 6, car 4 est déjà dans l'historique.

On peut les visualiser avec `git log ..<branche>`

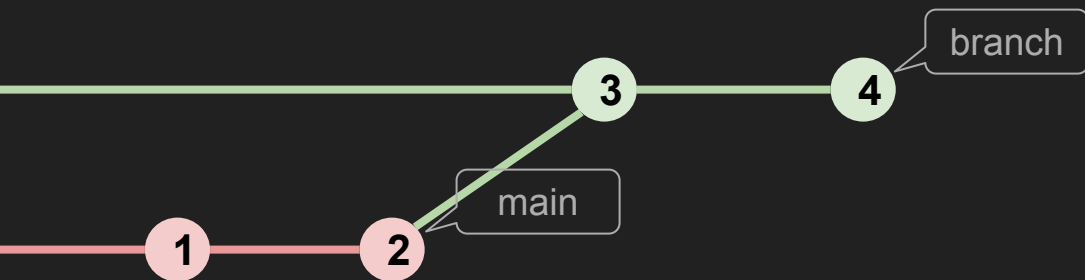


git merge : le fast-forward

Il y a un autre “type” de merge, c’est le fast-forward.

Il se produit lorsque la branche qu’on souhaite merger est uniquement en avance.

Voici un exemple :

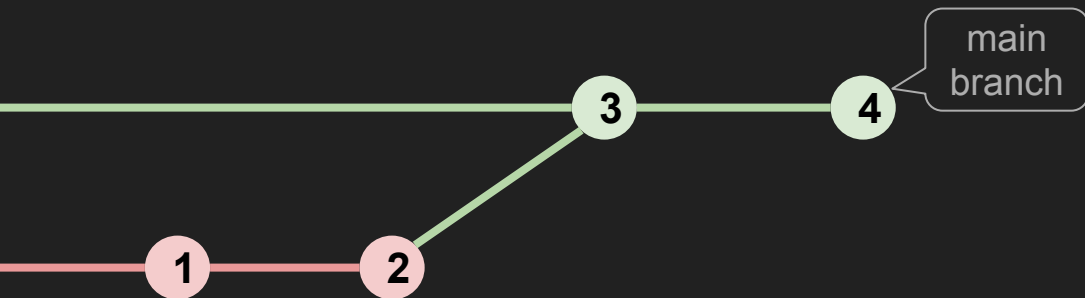


git merge : le fast-forward

Il y a un autre “type” de merge, c’est le fast-forward.

Il se produit lorsque la branche qu’on souhaite merger est uniquement en avance.

Voici un exemple :



Dans ce cas, au lieu de créer un nouveau commit, git fera simplement une avance rapide, en déplaçant main sur le commit 4.

Conflits : 🤖

Un conflit arrive pendant un merge, quand git ne sait pas quelle version doit être utilisée lors d'un merge.

Si une ligne est modifiée par deux personnes, quelle version on prend ?

Voici le message qui s'affiche et fait si peur :

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

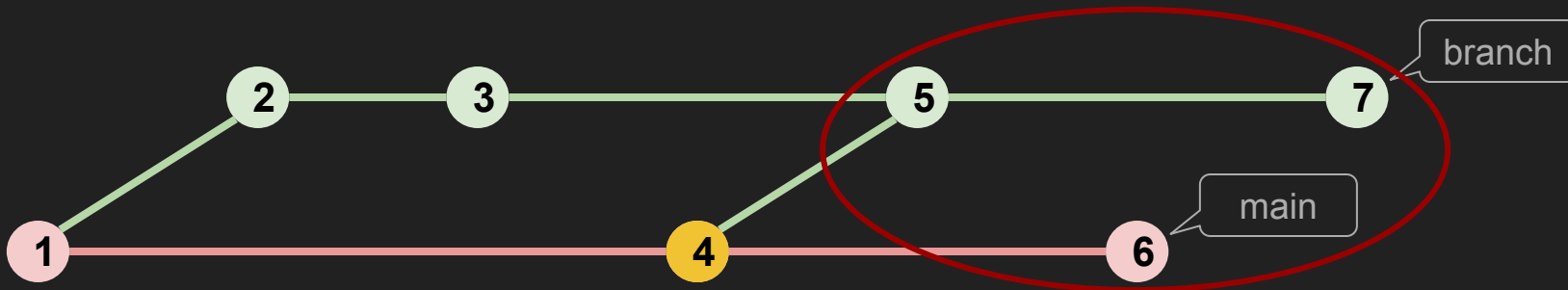
Conflits : comment ça marche, un merge ?

On va se pencher sur le “pourquoi”, avant de chercher comment on peut le résoudre.

Le conflit, comme on l’a dit, c’est quand git ne peut pas choisir quelle version d’une ligne prendre.

Mais quels commits sont sujets à avoir des conflits ? Et bien ce sont ceux qui se situent après la “merge-base”. Ici, le commit 4 est la “merge-base” car branch et main en héritent.

Si une ligne se trouve modifiée dans (5 ou 7) et 6, cela fera un conflit.



Conflits : Comment les résoudre

(nous utiliserons VSCode pendant la présentation)

```
<<<<<<< HEAD  
Hello 42  
=====  
Hello 43  
>>>>>>> branch
```

Voici un conflit.

Maintenant que vous savez pourquoi un conflit existe, et ce qu'est HEAD, il est bien plus simple de s'y retrouver.

Pour résoudre le conflit, il faut simplement retirer manuellement les lignes <<<, === et >>>, en sélectionnant un ou les deux parties encadrées.

Ou sur VSCode, cliquer sur l'option que l'on souhaite conserver.

Donc ici, on souhaite garder notre version (HEAD), on va donc retirer les marqueurs de conflit ainsi que "Hello 43".

On peut aussi choisir de garder les deux options si nécessaire.

git remote : c'est utile ça ?

Un remote est un répertoire commun, que les collaborateurs vont utiliser pour échanger leurs modifications. Le plus souvent, il est hébergé ailleurs : sur GitHub, ou sur un serveur local.

Lors d'un clone, l'url donnée en argument est définie comme remote par défaut, avec le nom "origin".

On peut visualiser la liste des remotes avec la commande `git remote` :

```
$ git clone git@vogsphere.42lyon.fr:vogsphere/intra-uuid-15934b7c-0822-471a-8e28-69bb046a8167-5290719-ibertran      atelier_git
$ cd atelier_git
$ git remote -v
origin  git@vogsphere.42lyon.fr:vogsphere/intra-uuid-15934b7c-0822-471a-8e28-69bb046a8167-5290719-ibertran  (fetch)
origin  git@vogsphere.42lyon.fr:vogsphere/intra-uuid-15934b7c-0822-471a-8e28-69bb046a8167-5290719-ibertran  (push)
```

Mais pourquoi on s'arrêterait à un seul repo ? 🤔

git remote add : oh ?

Le cas d'usage est simple : vous avez travaillé pendant trois semaines sur votre `push_swap`, en utilisant bien sûr le superbe testeur `./complexity`.

Mais vous n'avez utilisé que votre GitHub perso, et vous voulez le mettre sur la vogsphere.

Comment faire ? 🤔

```
$ rm -rf .git
$ git clone git@vogosphere.42lyon.fr:[...] ../push_to_vogsphere
$ mv * ../push_to_vogsphere
$ cd ../push_to_vogsphere
$ git add .
$ git commit -m "Push to vogsphere"
$ git push -f # bah oui, on force vu que c'est la troisième fois qu'on le fait
```

Non.

```
$ git remote add 42 git@vogsphere.42lyon.fr:vogsphere/[...]  
$ git remote -v  
42 git@vogsphere.42lyon.fr:vogsphere/[...] (fetch)  
42 git@vogsphere.42lyon.fr:vogsphere/[...] (push)  
origin https://github.com/ibtrd/42cursus-minishell.git (fetch)  
origin https://github.com/ibtrd/42cursus-minishell.git (push)  
$ git push 42
```


git remote add : que vient-il de se passer ?

Nous venons d'ajouter un remote nommé **42**. Grâce à ça, le code peut maintenant être envoyé sur différentes urls, sans avoir à être copié entièrement dans un nouveau git clone.

L'url peut être éditée via `git remote set-url <remote> <nouvelle-url>`.

Le remote peut aussi être supprimé avec `git remote remove <remote>`.

Si vous voulez **toujours envoyer** votre code sur GitHub **et** sur la vogsphere, vous pouvez utiliser

```
$ git remote set-url --add origin git@vogsphere.42lyon.fr:vogsphere/[...]
$ git remote -v
origin https://github.com/SimonCROS/push_swap_tester (fetch)
origin https://github.com/SimonCROS/push_swap_tester (push)
origin git@vogsphere.42lyon.fr:vogsphere/[...] (push)
```



La vogsphere sera seulement écrite, attention à ne pas push dessus manuellement.

Puis, une fois le projet verrouillé sur l'intra, il faudra supprimer l'url de la vogsphere via

```
$ git remote set-url --delete origin git@vogsphere # pas besoin de toute l'url, juste le début suffit
```

git clone : on connaît, mais comment ça marche ?

Pour finir cette présentation, on va parler de `git clone`.

Maintenant que nous avons une vue assez globale de git, il va être assez simple de décomposer ses actions avec d'autres commandes (le tout dans un nouveau dossier) :

```
$ git init
$ git remote add origin <url>
$ git fetch origin
$ git remote show origin
* remote origin
  Fetch URL: <url>
  Push URL: <url>
  HEAD branch: main
  [...]
$ git switch -c main origin/main
```

git push : Envoi vers un dépôt distant

Permet d'envoyer un commit a un dépôt distant.

```
$ git push <remote> <branch>
```

Si le dernier commit du dépôt distant n'est pas en avance sur le commit qui est push, le dépôt distant est mis à jour

```
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s,
done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:ibtrd/atelier_git.git
221564a..42b7d6a  main -> main
```



git push : Envoi vers un dépôt distant

Par contre, si le dépôt distant est en avance ... l'envoi est refusé.

```
$ git push origin main
To github.com:ibtrd/atelier_git.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'git@github.com:ibtrd/atelier_git.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```





La mise à jour doit donc être faite de notre côté! 

git fetch : git push mais dans l'autre sens

Permet de télécharger le commit le plus récent.

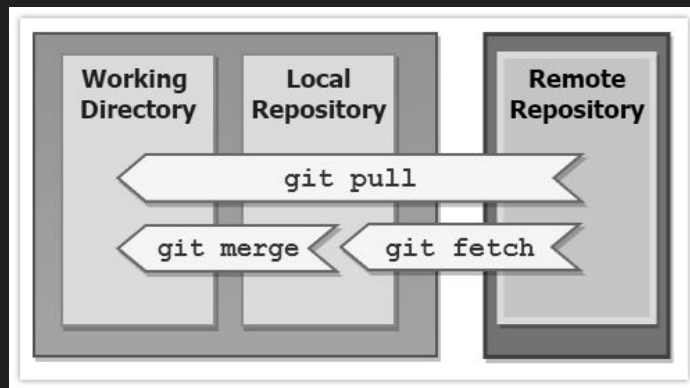
Celui ci est stocké dans le repo local, mais aucun changement n'est appliqué sur nos fichiers de travail.

Mais maintenant que celui ci est à notre disposition, il peut être merge!

git pull : dash 2en1

Succession d'un `git fetch` et d'un `git merge` du commit récupéré.

```
$ git pull origin main    =    $ git fetch origin main  
                               $ git merge origin/main
```



Vous pouvez utiliser git push sans arguments, et ça, c'est cool 😎 !

mais...

git push : The current branch has no upstream branch

Vous avez peut-être déjà eu ce problème, on va voir pourquoi ça arrive...

```
fatal: The current branch test has no upstream branch.  
To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin test
```

Quand on crée une nouvelle branche, git ne sait pas si elle doit être liée à votre remote.

La commande `git branch -vv` permet de visualiser les branches et leur upstream.

```
$ git branch -vv  
main d50101 [origin/main] Initial commit  
* test ea45308 new commit
```

Les commandes suivantes permettent de définir un nouvel upstream.

```
git branch -u <remote>/<branche>
```

```
git push -u <remote> <branche>
```

Exemple :

```
$ git push -u origin test  
$ git branch -vv  
main d581015 [origin/main] Initial commit  
* test ea45308 [origin/test] new commit
```

En pratique

Travailler sur un même repo git de partout !

- Avant de commencer à travailler :

```
$ git pull
```

Ou travailler sur une autre branche que celle qui a avancé à l'école, si rien n'a été push.
Ceci vous évitera de gros conflits !

- Donc plus simple ... Push sa progression avant de partir si on compte reprendre depuis chez soi !

```
$ git push
```

Et on n'oublie pas de nommer correctement ses commit, en cas de besoin de remonter en arrière si on a tout cassé !

Je souhaite renommer mon dernier commit (non push)

Je viens de faire un commit via

```
$ git commit -m "Added kibft.h"
```

mais je souhaite corriger la faute d'orthographe.

Pour renommer ce commit, une option `--amend` existe

```
$ git commit --amend -m "Added libft.h"
```

Un nouveau commit sera créé, remplaçant l'ancien.



Si vous avez ajouté des fichiers à l'index via `git add`, ils seront également ajoutés au commit.

Je souhaite modifier mon dernier commit (non push)

Je viens de faire un commit via

```
$ git commit -m "Added libft.h"
```

mais il y a une minuscule faute dans mon code 😭.

Pour ne pas ajouter de commit supplémentaire, je peux modifier le précédent après avoir corrigé le bug puis ajouté la modification à l'index :

```
$ git commit --amend -m "Added libft.h"
```

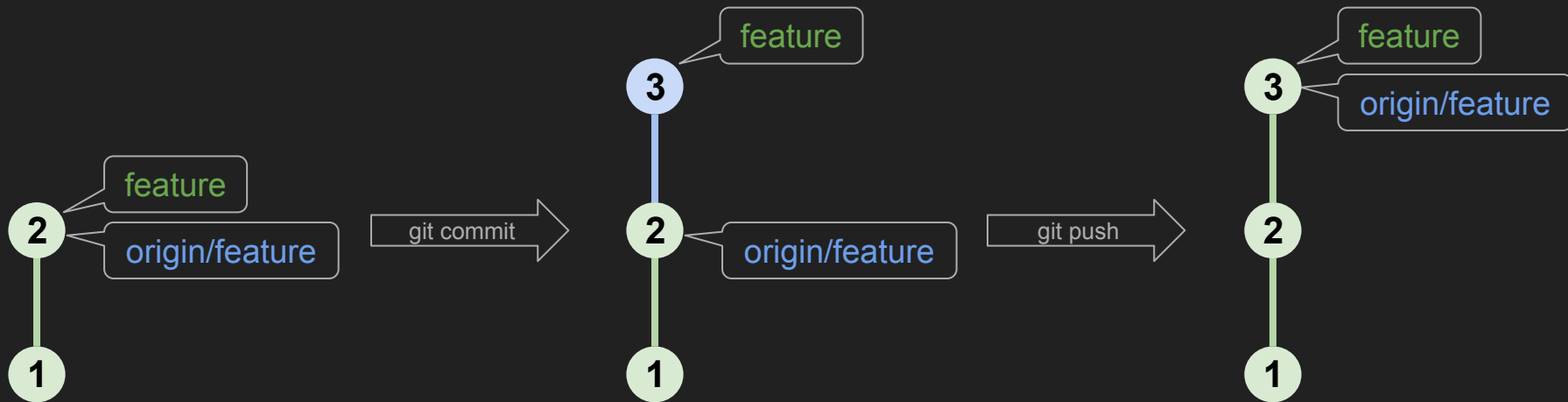
ou, si je veux garder le même message :

```
$ git commit --amend --no-edit
```

Je souhaite modifier mon dernier commit (déjà push)

Quand le commit que l'on souhaite modifier, quelque soit la manière de le faire, est déjà push, c'est un peu plus problématique.

Lors d'un push "standard", quand `origin/feature` est dans l'historique de `feature`, pas de problème, `origin/feature` est automatiquement déplacé sur `feature`.



Je souhaite modifier mon dernier commit (déjà push)

Le problème si on veut modifier un commit, pour le renommer, changer son contenu ou autre, c'est qu'on doit générer un nouveau commit. Souvenez-vous, un commit est **unique et non modifiable**.

On devra donc utiliser `git push --force` pour signifier qu'il faut déplacer **feature** où on lui demande.

`git commit --amend` modifie l'historique, ne l'utilisez pas sur des branches sur lesquelles vous travaillez à plusieurs !


Si vos collègues avaient récupéré l'ancien commit, le prochain `git pull` sera bien plus complexe !

Rassembler des branches sur Git : le conflit

J'ai fini d'implémenter une fonctionnalité sur `<ma_branche>`. Après l'avoir correctement testée, je souhaite l'ajouter sur `<main>`.

Probleme : j'avais déjà modifié le header via une autre fonctionnalité, et il y aura un conflit sur ce fichier !

```
$ git switch <main>
$ git merge <ma_branche>
Auto-merging header.h
CONFLICT (content): Merge conflict in header.h
Automatic merge failed; fix conflicts and then commit the result.
```



Rassembler des branches sur Git : le conflit

J'ai fini d'implémenter une fonctionnalité sur `<ma_branche>`. Après l'avoir correctement testée, je souhaite l'ajouter sur `<main>`.

Probleme : j'avais déjà modifié le header via une autre fonctionnalité, et il y aura un conflit sur ce fichier !

```
$ git switch <main>
$ git merge <ma_branche>
Auto-merging header.h
CONFLICT (content): Merge conflict in header.h
Automatic merge failed; fix conflicts and then commit the result.
```



→ Ouverture de mon éditeur de code pour régler ces conflits, et conserver toutes les modifications.

```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<< HEAD (Current Change)
int    abc
=====
float  var
>>>>>> me (Incoming Change)
```



```
int    abc
float  var
```

```
$ git add header.h
$ git commit -m "merge my functionality"
```

→ Petit check pour être sur :


```
$ git merge <ma_branche>
Already up to date.
```



Récupérer des modifications petit à petit

Le travail est toujours en cours sur `<ma_branche>`, mais `<main>` a tellement avancé que j'aimerais récupérer les dernières avancées.

```
$ git switch <ma_branche>
$ git merge <main>
Updating f6ec710..09621f1
Fast-forward
 bla | 0
 k   | 0
 z   | 6 ++++--
3 files changed, 4 insertions(+), 2 deletions(-)
create mode 100644 bla
create mode 100644 k
```



→ Ou gestion de conflits si il y en a, et je continue tranquillement

(et maintenant, vous savez faire)

J'ai ajouté une modification à l'index, mais je veux la retirer

C'est une question pour laquelle vous trouverez différentes réponses sur internet, selon l'année.

Il existe maintenant une commande nommée `git restore`.

Pour retirer un fichier de l'index, vous pouvez utiliser :

```
$ git restore -S <fichier|dossier> <fichier|dossier> ...
```

Et pour annuler une modification qui n'est pas dans l'index - **cela est non réversible** - :

```
$ git restore <fichier|dossier> <fichier|dossier> ...
```

Mon premier commit sur la vogsphere était sur la branche parsing, et maintenant, quand je clone je suis directement dessus !

Comment on répare ça 🙄 ?

On peut pas.

Il n'est pas possible de modifier cette information depuis un client.

Sur GitHub, GitLab ou autre concurrent, des boutons sont disponibles si vous êtes administrateur du repo pour modifier le nom de la branche utilisée par défaut pour le clone.


Sur l'intra de 42, il n'y a pas d'accès à un panneau d'administration d'un repo (en tant qu'étudiant en tout cas). Vous devrez donc faire en sorte de push votre code sur la branche qui sera clonée.


Pour ne pas impacter votre GitHub (ou autre), vous pouvez utiliser cette technique :

```
$ git push 42 main:parsing
```

Merci !



 “HEAD” représente le commit sur lequel nos modifications se basent

 “HEAD” représente le commit sur lequel nos modifications se basent