

Les branches avec Git - Rebaser

Table des matières

I. Contexte	3
II. Différences entre merge et rebase	3
III. Exercice : Appliquez la notion	5
IV. Rebaser une branche par rapport à une autre	5
V. Exercice : Appliquez la notion	9
VI. Réécrire l'historique	9
VII. Exercice : Appliquez la notion	13
VIII. Auto-évaluation	13
A. Exercice final	13
B. Exercice : Défi	15
Solutions des exercices	16

I. Contexte

Durée : 1 h

Environnement de travail : GitBash / Terminal

Pré-requis : Connaître les bases de la gestion des branches

Contexte

Lorsque nous travaillons avec Git et son système de branches, il nous arrivera de vouloir récupérer des modifications d'une branche à une autre, et nous serons alors confrontés à deux cas : **la fusion de modifications** sur une branche commune (*merge*) et **la récupération de modifications** d'une branche commune sur une branche en cours (*rebase*). Nous allons détailler dans cette partie le fonctionnement du rebase.

II. Différences entre merge et rebase

Objectifs

- Qu'est-ce que le rebasage ?
- Comprendre la différence entre merge et rebase

Mise en situation

Avec Git, il est possible de récupérer des modifications d'une autre branche pour les positionner avant notre travail en cours. C'est ce que l'on appelle le **rebasage**, ou *rebase* en anglais. Cette opération n'est pas sans risques, mais elle permettra de faire des manipulations qu'il ne serait pas possible de faire avec des fusions.

Le système de branches de Git peut être considéré comme un arbre, la branche `master` étant le tronc duquel sont créées d'autres branches. L'une des possibilités de la commande `git rebase` va être de pouvoir couper certaines branches pour les greffer à d'autres.

Exemple

Nous avons commencé nos développements sur une nouvelle branche `bugfix`, qui part de `master`. Cependant, pendant nos développements, de nouveaux commits ont été apportés à `master` (C et D), corrigeant certains bugs que nous avons aussi sur notre branche. Nous pourrions avoir envie de récupérer ces modifications sur notre branche en cours.

Une des solutions serait de fusionner notre branche sur `master`, grâce à `git merge`, puis d'en créer une nouvelle. Mais cela n'est pas conseillé, car nos développements ne sont pas terminés.

```
1 A---B---C---D ← master
2      \
3      E---F ← bugfix
```

Notre branche `bugfix` comporte les commits A, B, E et F. Nous souhaiterions que son historique comporte les commits C et D.

Exemple Avec git merge

Partons du principe que les développements de notre branche `bugfix` ne sont pas terminés, mais, qu'en l'état, ils ne provoqueraient pas de bugs s'ils étaient mergés sur `master`. Voilà ce que nous pourrions faire :

```
1 git checkout master
2 git merge bugfix
3 git branch -D bugfix
4 git checkout -b bugfix
```

Quelques explications :

1. Nous nous sommes positionnés sur la branche `master`.
2. Nous avons mergé la branche `bugfix` sur `master`. Notre historique à ce moment est le suivant : A---B---C---D---E---F.
3. Nous supprimons la branche `bugfix` qui n'est plus utilisée.
4. Nous recréons la branche pour pouvoir continuer nos développements.

Voilà ce que nous obtenons à la fin :

```
1          master
2          ↓
3 A---B---C---D---E---F
4          ↑
5          bugfix
```

Cette méthode n'est pas idéale, puisque nos développements ne sont pas terminés et que les modifications apportées pourraient devoir être adaptées pour la suite de nos développements. Ce que nous souhaiterions idéalement, c'est pouvoir récupérer les modifications de `master` dans la branche `bugfix` sans avoir à merger des développements non terminés dans `master`.

Exemple Avec git rebase

Ce que nous aimerions plutôt obtenir est une structure de branches comme ceci :

```
1 A---B---C---D ← master
2          \
3          E---F ← bugfix
```

Grâce à `git rebase`, nous pouvons nous contenter de faire :

```
1 git checkout bugfix
2 git rebase master
```

Quelques explications :

1. Nous nous sommes positionnés sur la branche `bugfix`.
2. Nous avons rebasé la branche `bugfix` à partir de `master`.

Nous prévoyons ici d'avoir le résultat attendu. Dans la réalité, ce ne sera pas exactement le cas. Le résultat obtenu est le suivant :

```
1 A---B---C---D ← master
2          \
3          E'---F' ← bugfix
```

Remarque Pourquoi E' et F' ?

La commande `git rebase` permet de faire de la **réécriture d'historique**. Ici, on essaie de mentir à Git en lui disant que la branche `bugfix` est bien partie du commit D de `master`, alors que ce n'est pas le cas (nous étions partis de B). C'est pour cela que Git a besoin de modifier nos commits. Dans les faits, seuls les **hashs** de nos commits ont été modifiés, pas le contenu.

Attention La manipulation de l'historique

La réécriture d'historique est une opération très dangereuse. C'est la principale cause de perte de travail. Il faut donc bien veiller à pousser nos branches sur le serveur distant avant d'effectuer ces actions. Ainsi, si quelque chose se passe mal, nous pourrions revenir en arrière en récupérant la branche distante en local.

Syntaxe À retenir

- La commande `git merge` permet de fusionner nos commits à la suite des commits d'une branche commune.
- La commande `git rebase` permet de récupérer des commits d'une branche commune sur notre branche en cours.

Complément

Rebaser (Rebasing)¹

III. Exercice : Appliquez la notion

Question

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `README.md` avec le texte `Hello world\n`
2. Ajoutez le fichier et créez un commit nommé `Add README.md`
3. Créez une branche `bugfix` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour ajouter un point à la fin avec la commande `sed -i s/world/world./g README.md`
4. Créez un commit nommé `Add point at the end of README.md`
5. Placez-vous de nouveau sur la branche `master`, puis créez un nouveau fichier `DOCUMENTATION.md` avec la commande avec le texte `This is my documentation file\n`
6. Créez un commit nommé `Add DOCUMENTATION.md`
7. Rebaser la branche `bugfix` par rapport à la branche `master`
8. Vérifiez que vous avez bien trois commits sur votre branche `master`

```
1 4f76e0c (HEAD -> bugfix) Add point at the end of README.md
2 027282f (master) Add DOCUMENTATION.md
3 bc9535a Add README.md
```

Les *hashs* des commits peuvent changer sur votre environnement.

IV. Rebaser une branche par rapport à une autre

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Rebaser-Rebasing>

Objectif

- Rebaser une branche par rapport à une autre

Mise en situation

Récupérer un historique de commits en changeant la base d'une branche est une opération relativement aisée avec Git, même si elle peut être dangereuse si mal exécutée. Nous allons voir tout de suite comment l'effectuer en toute sécurité.

Exemple

Plaçons-nous dans l'exemple suivant :

```
1 A---B---C---D ← master
2     \
3     E---F ← bugfix
```

Nous avons créé une branche `bugfix` à partir du commit `B` de `master` et fait de nouveaux commits sur les deux branches. Nous souhaitons récupérer les commits `C` et `D` de `master` sur notre branche `bugfix`. Dans la pratique, nous aurions les historiques suivants :

Branche `master` :

```
1 5ff56e5 (HEAD -> master) Add reference to sources.md
2 f8fe732 Add sources.md
3 7b3e3ae Add description in readme
4 8c13dd2 Add README.md
```

Branche `bugfix` :

```
1 c2c8f01 (HEAD -> bugfix) Add todo
2 c02f52e Add more description in readme
3 7b3e3ae Add description in readme
4 8c13dd2 Add README.md
```

Utilisons la commande `git rebase` afin de récupérer les modifications de `master` sur notre branche `bugfix` :

```
1 git checkout bugfix
2 git rebase master
```

Si tout se passe sans conflit, voilà à quoi ressemblera l'historique de la branche `bugfix` :

```
1 625cabd (HEAD -> bugfix) Add todo
2 4c23da7 Add more description in readme
3 5ff56e5 (origin/master, master) Add reference to sources.md
4 f8fe732 Add sources.md
5 7b3e3ae Add description in readme
6 8c13dd2 Add README.md
```

Remarque

Quelques subtilités

Les commits des lignes 3 à 6 sont ceux de la branche `master`. On a donc bien récupéré les modifications de `master`.

On remarque également que, si l'on compare les commits des lignes 1 et 2, les intitulés sont les mêmes que ceux de la branche `bugfix`, mais les hashes ont été modifiés.

L'explication pour ce second point est simple. Lorsque Git rebase une branche par rapport à une autre, il suit le fonctionnement suivant :

1. Il trouve l'ancêtre commun aux deux branches (le dernier commit en commun),
2. Il revient à cet état et met de côté les commits "en trop",
3. Il met à jour la branche avec les commits de la branche à partir de laquelle on rebase,
4. Il applique l'un après l'autre les commits de la base qu'on veut rebaser.

Fondamental Le fonctionnement du rebase

Revenons à notre exemple. Ici, Git a suivi le fonctionnement suivant :

1. Il a trouvé l'ancêtre commun aux deux branches : le commit 7b3e3ae.
2. Il a mis les commits c02f52e et c2c8f01 de la branche `bugfix` de côté.
3. Il a récupéré les commits f8fe732 et 5ff56e5 de la branche `master`.
4. Il a réappliqué les modifications des commits c02f52e et c2c8f01 en créant respectivement les commits 4c23da7 et 625cabd.

Schématiquement, les branches ressemblent maintenant à cela :

```

1 A---B---C---D ← master
2           \
3           E'---F' ← bugfix

```

Méthode Gérer les conflits

Dans le cas où la partie 4 réappliquerait des commits sur des fichiers ayant été modifiés, des conflits pourraient survenir. Ces conflits doivent être résolus à la manière de ceux apparaissant dans les fusions de branches, à la différence des commandes à exécuter à la fin. En clair, il faut :

1. Résoudre les conflits à la main si Git n'a pas pu les résoudre lui-même
2. Ajouter les fichiers dont les conflits ont été résolus avec `git add`
3. Continuer le rebasage avec la commande `git rebase --continue`

Tout au long de la résolution de nos conflits, nous serons dans un état particulier dans lequel il est fortement déconseillé de faire autre chose que la résolution des conflits, tel qu'indiqué juste au-dessus.

Voilà un exemple de commande `git status` qui nous indique quoi faire pendant une résolution de conflit dans un rebasage :

```

1 $ git status
2 interactive rebase in progress; onto 5ff56e5
3 Last command done (1 command done):
4   pick c02f52e Add more description in readme
5 Next command to do (1 remaining command):
6   pick c2c8f01 Add todo
7   (use "git rebase --edit-todo" to view and edit)
8 You are currently rebasing branch 'bugfix' on '5ff56e5'.
9   (fix conflicts and then run "git rebase --continue")
10  (use "git rebase --skip" to skip this patch)
11  (use "git rebase --abort" to check out the original branch)
12
13 Unmerged paths:
14   (use "git restore --staged <file>..." to unstage)
15   (use "git add <file>..." to mark resolution)

```

16

both modified: README.md

Attention Mettre fin à un rebase

L'opération de rebasage est une opération dangereuse. Si, à un moment ou à un autre, nous souhaitons l'arrêter (par exemple parce que nous pensons avoir fait une bêtise dans la résolution d'un conflit), il est possible d'utiliser la commande `git rebase --abort`.

Méthode Mettre à jour la base distante

Une fois notre branche rebasée, nous pourrions mettre à jour notre branche distante.

Si nous tentons de pousser notre branche avec `git push`, nous aurons un message de ce genre :

```
1 $ git push
2 To github.com:<username>/exemple-rebase.git
3 ! [rejected]        bugfix -> bugfix (non-fast-forward)
4 error: failed to push some refs to 'git@github.com:<username>/exemple-rebase.git'
5 hint: Updates were rejected because the tip of your current branch is behind
6 hint: its remote counterpart. Integrate the remote changes (e.g.
7 hint: 'git pull ...') before pushing again.
8 hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Le dépôt distant refuse notre push parce que les branches distantes et locales diffèrent.

En effet, notre branche distante comporte les commits A---B---E---F, alors que notre branche locale possède les commits A---B---C---D---E'---F'.

Là où Git n'aurait aucun soucis pour ajouter des commits à notre branche distante, le problème se situe au niveau des commits E et F qui ont disparu. Le dépôt distant rejette donc notre push et nous indique que l'historique de notre branche locale est « derrière » celui de notre branche distante.

Attention Forcer la mise à jour du dépôt distant

Git nous indique, **à tort**, qu'il faudrait récupérer les commits manquants avec la commande `git pull`.

Si nous faisons cela, nous aurons les commits E et F en double, car ils sont respectivement déjà présents en tant que E' et F'.

Puisque nous sommes sûrs d'avoir correctement résolu nos conflits, nous pouvons utiliser la commande `git push --force-with-lease`.

L'option `--force-with-lease` permet de vérifier que l'*upstream* n'a pas été modifié entre le moment où nous poussons et la dernière fois que nous avons récupéré la branche. C'est assez rare, mais lorsqu'on réécrit de l'historique, deux précautions valent mieux qu'une.

Un simple `git push --force` supprimerait les commits d'autres personnes, là où l'option `--force-with-lease` fera la vérification et refusera le push le cas échéant.

Les opérations de réécriture d'historique sont à bannir sur les branches partagées par plusieurs développeurs.

Une fois notre branche distante mise à jour, nous avons terminé notre opération de rebasage. Même si l'opération peut sembler dangereuse, Git nous fournit tous les outils nécessaires pour l'effectuer dans les meilleures conditions.

Syntaxe **À retenir**

- La commande `git rebase` permet de rebase une branche, c'est-à-dire de la mettre à jour par rapport à une autre.
- Dans le cas de conflits, il suffit de les résoudre à la main, d'ajouter les fichiers concernés et d'exécuter la commande `git rebase --continue`.
- Cette opération réécrit l'historique et est donc potentiellement dangereuse, puisque vous pourriez perdre du code. Il faut bien pousser ses branches sur leur upstream avant d'effectuer ce genre d'opérations.

ComplémentRebaser (Rebasing)¹Documentation `git push --force-with-lease`²

V. Exercice : Appliquez la notion

Question

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `README.md` avec le texte `Hello world\n`
2. Ajoutez le fichier et créez un commit nommé `Add README.md`
3. Créez une branche `bugfix` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour ajouter un point à la fin avec la commande `sed -i s/world/world./g README.md`
4. Créez un commit nommé `Add point at the end of README.md`
5. Placez-vous de nouveau sur la branche `master` puis modifiez le fichier pour ajouter un point d'exclamation à la fin avec la commande `sed -i s/world/world\!/g README.md`
6. Créez un commit nommé `Add exclamation mark at the end of README.md`
7. Rebasez la branche `bugfix` par rapport à la branche `master` (vous devriez avoir des conflits)
8. Résolvez les conflits en gardant les modifications de la branche `bugfix` et terminez le rebasage

VI. Réécrire l'historique

Objectif

- Apprendre à réécrire son historique avec `rebase`

Mise en situation

Git ne prend connaissance de nos modifications que lorsque nous faisons des commits. Pour profiter de ses avantages, il est donc conseillé de faire des commits tôt, et de le faire souvent. Cela veut dire que, lorsque nous développons une nouvelle fonctionnalité sur une branche dédiée, nous devons faire un commit à chaque fois que notre travail est dans un état stable, même s'il n'est pas terminé. Ainsi, si nous cassons tout à la suite d'une modification, nous pouvons facilement revenir au commit précédent.

¹ <https://git-scm.com/book/fr/v2/Les-branches-avec-Git-Rebaser-Rebasing>² <https://git-scm.com/docs/git-push#Documentation/git-push.txt---no-force-with-lease>

L'effet indésirable est que, à la fin du développement, nous risquons de nous retrouver avec une multitude de commits pour une seule et même fonctionnalité. Cependant, la commande `git rebase` nous permet de réécrire l'historique de notre branche pour, par exemple, fusionner des commits entre eux, les renommer ou les supprimer.

Méthode Réécrire son historique

Nous avons actuellement deux branches : `master` et `feature`.

La branche `feature` comporte le développement d'une nouvelle fonctionnalité : la création d'une liste des utilisateurs. Celle-ci a été faite en plusieurs étapes : l'affichage HTML de la liste des utilisateurs (commit E), le rendu CSS de notre liste (commit F) et la correction d'un bug suite à notre validation interne (commit G).

```
1 A---B---C ← master
2       \
3       E---F---G ← feature
```

Aucun de ces commits n'a de sens sans les deux autres. Nous ne pouvons pas choisir de n'en garder qu'un, puisque les deux autres sont très fortement liés.

L'historique de la branche `feature` ressemble à ceci :

```
1 625cabd (HEAD -> feature, origin/feature) Fix design
2 4c23da7 Add user list design
3 5ff56e5 Add user list template
4 f8fe732 (origin/master, master) Add login page
5 7b3e3ae Add homepage
6 8c13dd2 Create project template
```

Dans notre cas, pour garder un historique clair, nous voudrions pouvoir fusionner ces trois commits en un seul.

Méthode Comment procéder ?

La commande `git rebase` nous permet de faire cela en réécrivant l'historique à partir de notre propre branche, et plus à partir d'une autre branche, grâce à l'option `-i`.

```
1 git rebase -i HEAD~3
```

L'option `-i`, ou `--interactive`, va nous permettre d'influer sur la liste des commits et les modifier.

Le paramètre `HEAD~3` est une notation relative signifiant *trois commits avant HEAD* (`HEAD` étant notre dernier commit). Dans notre cas, il s'agit du quatrième commit : `f8fe732`.

Remarque Faire référence à un commit

Il existe différentes manières de faire référence à un commit :

- La **notation absolue** qui est le fait de fournir son hash (court ou long), par exemple `f8fe732` ou `f8fe732d0d002beb5fdb0ed606226bc36cd51e05`
- La **notation relative**, par rapport à `HEAD` (le dernier commit de la branche) :
 - soit en spécifiant le nombre de commits avec `~`, par exemple `HEAD` (le dernier commit), `HEAD~1` (l'avant dernier commit), `HEAD~2` (le troisième commit)...
 - soit en spécifiant autant de `^` que de commits en arrière, par exemple `HEAD` (le dernier commit), `HEAD^` (l'avant dernier commit), `HEAD^^` (le troisième commit)...

Complément

Dans notre cas, la commande pourrait aussi s'écrire `git rebase -i HEAD^^^` ou `git rebase -i f8fe732`.

Méthode **L'interaction avec la commande**

Après avoir utilisé la commande `git rebase -i`, Git va nous récapituler la liste des commits en nous proposant des actions à leur appliquer.

Exemple

Dans le cas de notre exemple, voilà ce que nous obtenons avec la commande `git rebase -i HEAD~3`:

```

1 pick 625cabd Fix design
2 pick 4c23da7 Add user list design
3 pick 5ff56e5 Add user list template
4
5 # Rebase f8fe732..625cabd onto f8fe732 (3 commands)
6 #
7 # Commands:
8 # p, pick <commit> = use commit
9 # r, reword <commit> = use commit, but edit the commit message
10 # e, edit <commit> = use commit, but stop for amending
11 # s, squash <commit> = use commit, but meld into previous commit
12 # f, fixup <commit> = like "squash", but discard this commit's log message
13 # x, exec <command> = run command (the rest of the line) using shell
14 # b, break = stop here (continue rebase later with 'git rebase --continue')
15 # d, drop <commit> = remove commit
16 # l, label <label> = label current HEAD with a name
17 # t, reset <label> = reset HEAD to a label
18 # m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
19 # .      create a merge commit using the original merge commit's
20 # .      message (or the oneline, if no original merge commit was
21 # .      specified). Use -c <commit> to reword the commit message.
22 #
23 # These lines can be re-ordered; they are executed from top to bottom.
24 #
25 # If you remove a line here THAT COMMIT WILL BE LOST.
26 #
27 # However, if you remove everything, the rebase will be aborted.
28 #
29
```

Méthode **Choisir son mode de réécriture**

Ce que Git nous indique, c'est que nous pouvons préfixer nos lignes par un des choix possibles, les réordonner ou en supprimer. Parmi les choix qui s'offrent à nous, les plus courants sont :

- `pick` : utilise le commit tel quel, ne le modifie pas
- `reword` : utilise le commit tel quel, mais propose de modifier son message
- `edit` : applique le code, mais ne commit pas tout de suite, permettant de modifier le code ; `git commit` et `git rebase --continue` permettent de terminer l'opération
- `squash` : permet de fusionner le commit dans le commit précédent (celui du haut) et permet de modifier son message
- `fixup` : pareil que `squash`, mais garde seulement le message du commit précédent
- `drop` : supprime le commit

Exemple

Dans notre cas, puisque nous voulons fusionner des commits, nous allons utiliser le choix `squash`. Notre liste modifiée sera donc la suivante :

```
1 pick 625cabd Fix design
2 squash 4c23da7 Add user list design
3 squash 5ff56e5 Add user list template
```

Nous indiquons ici que nous voulons fusionner `625cabd` et `4c23da7` dans `5ff56e5`. Les fusions se font toujours du bas vers le haut. Si nous sauvegardons et quittons notre fichier, Git en ouvre un nouveau, nous invitant à définir le nouveau message de commit. Une fois le message indiqué et le fichier fermé, nous pouvons afficher de nouveau la liste de nos commits.

```
1 ac90ba9 (HEAD -> bugfix) Add users list
2 f8fe732 (origin/master, master) Add login page
3 7b3e3ae Add homepage
4 8c13dd2 Create project template
```

Nous obtenons donc le commit `ac90ba9` au lieu des trois autres précédents. Ce nouveau commit possède un nouveau hash.

Complément **git commit**

La commande `git commit` permet d'agir sur le dernier commit. Par exemple, si vous souhaitez modifier le dernier message de commit, vous pouvez utiliser :

```
1 git commit --amend
```

Attention **Le changement de hash**

La commande `git rebase` permet de faire de la réécriture d'historique. Lorsque nous modifions ou fusionnons des commits, leur hash change, ce qui peut provoquer des différences avec la branche distante que Git ne pourra pas résoudre convenablement.

Lorsque vous essaieriez de pousser votre branche sur l'upstream, Git vous indiquera à tort que des commits manquent. Dans ce cas, il faudra exécuter la commande `git push --force-with-lease`.

Les opérations de réécriture d'historique sont à bannir sur les branches partagées par plusieurs développeurs.

Complément **Les autres opérations**

Nous n'avons vu ici que l'opération consistant à fusionner des commits. Il en existe cependant plusieurs autres. Leur fonctionnement est soit plus simple, soit identique. Nous ne les couvrirons donc pas ici.

Syntaxe **À retenir**

- La commande `git rebase` et l'option `-i` permettent de réécrire l'historique Git d'une branche, permettant de réordonner, modifier ou fusionner des commits.
- Cette opération réécrit l'historique et est donc potentiellement dangereuse, puisqu'elle peut engendrer une perte de contenu.
- Il faut bien pousser ses branches sur leur upstream avant d'effectuer ce genre d'opérations.

ComplémentRéécrire l'historique¹**VII. Exercice : Appliquez la notion****Question**

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `README.md` avec le texte `Hello world\n`
2. Ajoutez le fichier et créez un commit nommé `Add README.md`
3. Créez une branche `bugfix` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour ajouter un point à la fin avec la commande `sed -i s/world/word./g README.md` (la faute est intentionnelle dans le cadre de l'exercice)
4. Créez un commit nommé `Add point at the end of README.md`
5. Corrigez la modification que vous venez de faire grâce à la commande `sed -i s/word/world/g README.md`
6. Créez un commit nommé `Fix typo in README.md`
7. Réécrivez l'historique de la branche `bugfix` afin de fusionner les deux commits de la branche en un (vous pouvez réutiliser le premier message)

VIII. Auto-évaluation**A. Exercice final****Exercice**

Exercice

La réécriture d'historique est une opération sans danger, parce que Git nous empêche de faire des erreurs.

- ☐ Vrai
- ☐ Faux

Exercice

Quelle commande permet d'ajouter les commits d'une branche après ceux d'une autre branche ?

Exercice

Quelle commande permet de récupérer les commits d'une branche avant ceux d'une branche en cours ?

Exercice

Quelle commande va me permettre de récupérer l'historique `A---B---C---D---E'---F'` sur la branche `bugfix` à partir de l'historique suivant ?

```

1 A---B---C---D ← master
2   \
3     E---F ← bugfix

```

¹ <https://git-scm.com/book/fr/v2/Utilitaires-Git-R%C3%A9%C3%A9crire-l%E2%80%99historique>

- ☐ `git checkout master && git rebase bugfix`
- ☐ `git checkout bugfix && git rebase master`

Exercice

Quelle commande va initialiser la réécriture de l'historique de la branche `bugfix` pour fusionner les deux derniers commits ?

```
1 A---B---C---D ← bugfix
```

- ☐ `git rebase -i HEAD~2`
- ☐ `git rebase -i HEAD^^`
- ☐ `git rebase -i C`
- ☐ `git rebase -i bugfix`

Exercice

Quelle commande permet d'annuler un rebasage en cours ?

Exercice

Lorsqu'on fait un rebasage interactif, que fait le choix `fixup` dans la liste ?

- ☐ Il fusionne le commit choisi dans le commit un cran plus haut dans la liste
- ☐ Il fusionne le commit choisi dans le commit un cran plus bas dans la liste
- ☐ Il demande de préciser le message de commit
- ☐ Il ne demande pas de préciser le message de commit

Exercice

Lorsqu'on fait un rebasage interactif, les commits sont affichés dans la liste (de haut en bas)...

- ☐ Du plus ancien au plus récent, comme pour `git log`
- ☐ Du plus récent au plus ancien, comme pour `git log`
- ☐ Du plus ancien au plus récent, à la différence de `git log`
- ☐ Du plus récent au plus ancien, à la différence de `git log`

Exercice

Que doit-on faire pour conclure un rebasage après avoir géré les conflits avec `git add` ?

- ☐ `git rebase --continue`
- ☐ `git merge`
- ☐ `git commit`

Exercice

Quelle commande doit-on utiliser après avoir fait le rebasage de commits déjà présents sur l'upstream ?

- ☐ `git push`
- ☐ `git push --force-with-lease`

B. Exercice : Défi

Dans cet exercice, nous allons appliquer l'ensemble de nos connaissances avec les rebasages pour travailler sur un fichier HTML comportant une liste de légumes.

Question

Exécutez les étapes suivantes :

1. Sur une branche `master` sans commits, créez un fichier `index.html` (cf. contenu en fin d'exercice)
2. Ajoutez le fichier et créez un commit nommé `Add vegetable list`
3. Poussez la branche `master` sur votre dépôt distant en définissant l'upstream après l'avoir créé sur GitHub
4. Créez une branche `fix-typos` depuis ce commit, placez-vous dessus, puis modifiez le texte du fichier pour corriger les fautes de frappe avec la commande `sed -i 's/Pome/Pomme/g; s/Carote/Carotte/g; s/Tommate/Tomate/g; s/Cerisse/Cerise/g' index.html`
5. Créez un commit nommé `Fix typos in vegetable list`, puis poussez-le sur GitHub
6. Créez une nouvelle branche `remove-fruits` depuis `master` puis modifiez le fichier pour corriger la liste en supprimant les fruits avec la commande `sed -i '/Pome/d; /Poire/d; /Cerisse/d' index.html`
7. Créez un commit nommé `Remove fruits from vegetable list`, puis poussez-le sur GitHub
8. Fusionnez la branche `fix-typos` dans la branche `master`, poussez `master` et supprimez la branche `fix-typos` (locale et distante)
9. Rebasez la branche `remove-fruits` à partir de la branche `master` (vous devriez avoir des conflits)
10. Résolez les conflits en gardant les corrections des deux branches et terminez la fusion, puis poussez `remove-fruits` de nouveau
11. Sur la branche `remove-fruits`, modifiez le fichier pour ajouter dans la liste le légume Oignon, puis créez un nouveau commit nommé `Add onion` et poussez la branche
12. Réécrivez l'historique de votre branche pour fusionner les deux derniers commits en un, nommé `Remove fruits from vegetable list and add onion`, puis poussez la branche
13. Fusionnez la branche `remove-fruits` dans la branche `master`, poussez `master` et supprimez la branche `remove-fruits` (locale et distante)

Fichier **index.html** du point 1 :

```

1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6       content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Document</title>
10  </head>
11  <body>
12    <h1>Liste des fruits :</h1>
13    <ul>
14      <li>Pome</li>
15      <li>Carote</li>
16      <li>Salade</li>
17      <li>Poire</li>
18      <li>Tommate</li>
19      <li>Cerisse</li>

```

```
19         </ul>  
20     </body>  
21 </html>
```

Solutions des exercices

Exercice p. Solution n°1

```
1 # 1
2 echo -e "Hello world\n" > README.md
3
4 # 2
5 git add .
6 git commit -m"Add README.md"
7
8 # 3
9 git checkout -b bugfix
10 sed -i s/world/world./g README.md
11
12 # 4
13 git commit -am"Add point at the end of README.md"
14
15 # 5
16 git checkout master
17 echo -e "This is my documentation file\n" > DOCUMENTATION.md
18
19 # 6
20 git add .
21 git commit -m"Add DOCUMENTATION.md"
22
23 # 7
24 git checkout bugfix
25 git rebase master
26 git log --oneline
```

Exercice p. Solution n°2

```
1 # 1
2 echo -e "Hello world\n" > README.md
3
4 # 2
5 git add .
6 git commit -m"Add README.md"
7
8 # 3
9 git checkout -b bugfix
10 sed -i s/world/world./g README.md
11
12 # 4
13 git commit -am"Add point at the end of README.md"
14
15 # 5
16 git checkout master
17 sed -i s/world/world\!/g README.md
18
19 # 6
20 git commit -am"Add exclamation mark at the end of README.md"
21
22 # 7
23 git checkout bugfix
24 git rebase master
```

```
25
26 # 8
27 vi README.md
```

README.md avec conflits :

```
1 <<<<<< HEAD
2 Hello world!
3
4 =====
5 Hello world.
6 >>>>>> 59bcfe8... Add point at the end of README.md
7
```

Ne gardez que les lignes 5 et 7 (bloc 59bcfe8).

README.md avec conflits corrigés :

```
1 Hello world.
2
1 # 8 bis
2 git add .
3 git rebase --continue
```

Exercice p. Solution n°3

```
1 # 1
2 echo -e "Hello world\n" > README.md
3
4 # 2
5 git add .
6 git commit -m"Add README.md"
7
8 # 3
9 git checkout -b bugfix
10 sed -i s/world/word./g README.md
11
12 # 4
13 git commit -am"Add point at the end of README.md"
14
15 # 5
16 sed -i s/word/world/g README.md
17
18 # 6
19 git commit -am"Fix typo in README.md"
20
21
```

Après l'étape 6, votre historique devrait ressembler à ceci :

```
1 ec9224c (HEAD -> bugfix) Fix typo in README.md
2 1b54de9 Add point at the end of README.md
3 c4753e3 (master) Add README.md
```

Continuez avec le rebase :

```
1 # 7
2 git rebase -i HEAD~2
3 # ou git rebase -i HEAD^^
```

Vous devriez avoir une liste de deux commits (les hashes peuvent changer dans votre environnement) :

```
1 pick 1b54de9 Add point at the end of README.md
2 pick ec9224c Fix typo in README.md
```

Modifiez-la pour fusionner le commit ligne 2 dans le commit ligne 1 (on utilise `fixup` au lieu de `squash` pour ne garder que le premier message) :

```
1 pick 1b54de9 Add point at the end of README.md
2 fixup ec9224c Fix typo in README.md
```

Sauvegardez puis quittez le fichier. Votre historique devrait ressembler à ceci :

```
1 220bd43 (HEAD -> bugfix) Add point at the end of README.md
2 c4753e3 (master) Add README.md
```

Exercice p. 13 Solution n°4

Exercice

La réécriture d'historique est une opération sans danger, parce que Git nous empêche de faire des erreurs.

☐ Vrai

☒ Faux



La réécriture d'historique peut faire perdre du code si vous supprimez des commits ou que vous résolvez mal vos conflits. Elle peut aussi poser des problèmes si vous réécrivez des commits d'une branche partagée avec d'autres développeurs.

Exercice

Quelle commande permet d'ajouter les commits d'une branche après ceux d'une autre branche ?

`git merge`

Exercice

Quelle commande permet de récupérer les commits d'une branche avant ceux d'une branche en cours ?

`git rebase`

Exercice

Quelle commande va me permettre de récupérer l'historique A---B---C---D---E'---F' sur la branche `bugfix` à partir de l'historique suivant ?

```
1 A---B---C---D ← master
2   \
3     E---F ← bugfix
```

☐ `git checkout master && git rebase bugfix`

☒ `git checkout bugfix && git rebase master`

Exercice

Quelle commande va initialiser la réécriture de l'historique de la branche `bugfix` pour fusionner les deux derniers commits ?

```
1 A---B---C---D ← bugfix
```

☒ `git rebase -i HEAD~2`

☒ `git rebase -i HEAD^^`

☐ `git rebase -i C`

☐ `git rebase -i bugfix`


Exercice

Quelle commande permet d'annuler un rebasage en cours ?

`git rebase --abort`

Exercice

Lorsqu'on fait un rebasage interactif, que fait le choix `fixup` dans la liste ?

- ☒ Il fusionne le commit choisi dans le commit un cran plus haut dans la liste
- ☐ Il fusionne le commit choisi dans le commit un cran plus bas dans la liste
- ☐ Il demande de préciser le message de commit
- ☒ Il ne demande pas de préciser le message de commit
-  Il fusionne le commit choisi dans le commit un cran plus haut dans la liste (le commit plus ancien chronologiquement). Il ne demande pas de préciser le message de commit, il ne garde que le message du commit dans lequel il est fusionné.

Exercice

Lorsqu'on fait un rebasage interactif, les commits sont affichés dans la liste (de haut en bas)...

- ☐ Du plus ancien au plus récent, comme pour `git log`
- ☐ Du plus récent au plus ancien, comme pour `git log`
- ☒ Du plus ancien au plus récent, à la différence de `git log`
- ☐ Du plus récent au plus ancien, à la différence de `git log`

Exercice

Que doit-on faire pour conclure un rebasage après avoir géré les conflits avec `git add` ?

- ☒ `git rebase --continue`
- ☐ `git merge`
- ☐ `git commit`

Exercice

Quelle commande doit-on utiliser après avoir fait le rebasage de commits déjà présents sur l'upstream ?

- ☐ `git push`
- ☒ `git push --force-with-lease`

Exercice p. Solution n°5

```
1 # 2
2 git add index.html
3 git commit -m "Add vegetable list"
4
5 # 3
6 git remote add origin git@github.com:<username>/defi-git-rebase.git
7 git push -u origin master
8
9 # 4
10 git checkout -b fix-typos
11 sed -i 's/Pome/Pomme/g; s/Carote/Carotte/g; s/Tommate/Tomate/g; s/Cerisse/Cerise/g' index.html
12
```

```

13 # 5
14 git add index.html
15 git commit -m"Fix typos in vegetable list"
16 git push -u origin fix-typos
17
18 # 6
19 git checkout -b remove-fruits master
20 sed -i '/Pome/d; /Poire/d; /Cerisse/d' index.html
21
22 # 7
23 git add index.html
24 git commit -m"Remove fruits from vegetable list"
25 git push -u origin remove-fruits
26
27 # 8
28 git checkout master
29 git merge fix-typos
30 git push
31 git push origin --delete fix-typos && git branch -D fix-typos
32
33 # 9
34 git checkout remove-fruits
35 git rebase master
36
37 # 10
38 # Corrigez les conflits (cf fichier plus bas)
39 git add index.html
40 git rebase --continue
41 git push --force-with-lease
42
43 # 11
44 sed -i '/Tomate/a\          <li>Oignon</li>' index.html
45 git commit -am"Add onion"
46 git push
47
48 # 12
49 git rebase -i HEAD^^
50 # Utilisez l'option "squash" sur le commit "Add onion" puis définissez le message de commit
51 git push --force-with-lease
52
53 # 13
54 git checkout master
55 git merge remove-fruits
56 git push
57 git push origin --delete remove-fruits && git branch -D remove-fruits

```

Vous devriez terminer avec 3 commits et une branche locale ainsi qu'une branche distante :

```

1 $ git log --oneline
2 352a942 (HEAD -> master, origin/master) Remove fruits from vegetable list and add onion
3 54b4db2 Fix typos in vegetable list
4 9f81d5f Add vegetable list
5
6 $ git branch -a
7 * master
8   remotes/origin/master

```

Fichier **index.html** avec conflits résolus (point 10) :

```

1 <!DOCTYPE html>

```

```

2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6       content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Document</title>
10  </head>
11  <body>
12    <h1>Liste des fruits :</h1>
13    <ul>
14      <li>Carotte</li>
15      <li>Salade</li>
16      <li>Tomate</li>
17    </ul>
18  </body>
19 </html>

```